A centralised data management system for the South African mining and industrial sectors

J Herman

orcid.org 0000-0002-6634-7838

Mini-dissertation submitted in fulfilment of the requirements for the degree *Master of Engineering in Computer and Electronic Engineering* at the North-West University

Supervisor: Dr JC Vosloo

Graduation May 2018

Student number: 28344650

PREFACE

I thank our Father, who art in heaven, for blessing me with abilities and talents to make it this far in life. Through His strength, I was able to complete this study.

I thank the following people and organisations who supported the conclusion of this study:

- My loving mother for her care and support during my time of study and throughout my life.
- My brother for his motivation and example to sustain my efforts.
- TEMM International (Pty) Ltd and ETA Operations who funded the study and allowed for the implementation of the centralised data management system.
- Dr J. C. Vosloo and S. W. van Heerden for their comments, suggestions, and assistance.

The reader should note that a journal article was written by me, and is given as an extension of this study. This article is provided in Appendix 4, but must be considered for marking with this dissertation.

ABSTRACT

The digital universe is expanding at an exponential rate, and there is a greater variety of data. This phenomenon, known as big data, is making an impact in all industries – specifically in organisations. Organisations can become more efficient, profitable, and deliver better products and services by adopting big data. The traditional relational data systems of organisations are, however, inadequate for big data. Furthermore, improper data management may have caused inconsistent and duplicate data, known as data silos. This coupled with the fact that big data adoption is still in its infancy, created the need for this study.

Research in big data is still ongoing, and this study, therefore, adds to the body of knowledge. This study's objective is to illustrate the value of big data to increase its adoption, and specifically in the South African mining and industrial sectors. Therefore, research was performed regarding the design, development, and implementation of a centralised data management system. This system is at its core a big data system aimed at improving the efficiency of organisations. Security was also identified as a research area, which necessitated its inclusion as a system design objective. Through the design and development of the system, a practical framework is provided to assist organisations in employing big data.

The literature study investigated NoSQL data stores for use in big data systems. Big data system architectures were discovered as used in system design. Next, industry experience was sought to make a (big data) system's functionalities available to users and systems. From this industry knowledge, microservices and containers were identified and studied. The final part of the literature study evaluated NoSQL software to be used in the proposed centralised data management system. This evaluation led to the decision to use MongoDB as the data store in the proposed system.

The architecture of the system consisted of three layers, namely, the resource, service, and interface layers. In the resource layer, Mesosphere DC/OS was used to create a cluster, thereby providing computing resources to the other layers. The service layer used MongoDB, Apache Spark, and the Python programming language to provide the various (micro) services of the system. Interaction with the system was done through the interface layer. Thus, the technologies of the interface layer were web service software, namely, Apache Zeppelin and a Windows Communication Foundation web service.

The system was successfully implemented at an engineering services company with multiple clients in the South African mining and industrial sectors. The system either supported more users, or a quicker performance for the same number of users than the company's previous system. For the same number of users, the system achieved at least a 24.72% performance increase. Most importantly, the system used transport layer security (TLS) 1.2 with user authentication and message integrity. Further validation of the system was provided in a journal article that forms part of this dissertation and was written by the author. This journal article is given in Appendix 4.

The case study proved that implementing big data improves organisational efficiency. The privacy and security of the organisation’s data were ensured. Two other benefits of the system were its support for structured, unstructured, and partially structured data, as well as the volume of big data. The developed system can be extended to other industries to increase efficiency and productivity. Future organisational big data projects can be initiated by using the created system as a starting point.

Key terms: big data, NoSQL, microservices, containers, web service, organisational efficiency

TABLE OF CONTENTS

PREFACE ..... **I**

ABSTRACT ..... **II**

CHAPTER 1 BIG DATA IN ORGANISATIONS **3**
1.1 Big data and its challenges in organisations 3
1.2 Problem statement 6
1.3 Research questions 7
1.4 Research process 8

CHAPTER 2 STUDY OF DATA SYSTEM DESIGNS AND IMPLEMENTATIONS **11**
2.1 Preamble 11
2.2 Brief history of databases 11
2.3 Big data system architectures 20
2.4 Services to present system functionality 34
2.5 Big data software evaluation 41
2.6 Conclusion 44

CHAPTER 3 DATA MANAGEMENT SYSTEM DESIGN **46**
3.1 Preamble 46
3.2 Requirements analysis 46
3.3 Experimentation of database systems 47
3.4 System design 52
3.5 Conclusion 69

CHAPTER 4 RESULTS OF SYSTEM USAGE **72**
4.1 Preamble 72
4.2 System implementation on a case study 72
4.3 Verification of system operation 74
4.4 System operation validation 100
4.5 Research results 119
4.6 Conclusion 122

CHAPTER 5 CONCLUSION **124**
5.1 Summary 124
5.2 Study closure 128
5.3 Recommendations for future work 128

BIBLIOGRAPHY **130**

APPENDIX 1 DATABASE SYSTEM EVALUATION RESULTS **137**

APPENDIX 2 PYTHON VERIFICATION PROGRAM CODE **142**

APPENDIX 3 APACHE SPARK FULL VERIFICATION LOGS **154**

APPENDIX 4 JOURNAL ARTICLE OF STUDY **161**

LIST OF TABLES

Table 1: Comparison of data store system features 43
Table 2: Comparison of best performing NoSQL data store 52
Table 3: Specifications of each cluster computer 56
Table 4: Specifications of each Mesosphere DC/OS master VM 56
Table 5: Specifications of each Mesosphere DC/OS agent VM 56
Table 6: Details on the information stored in a tag document 66
Table 7: Details on the information stored in a tag value document 67
Table 8: Summary of system specifications 69
Table 9: Specifications and the components related to them 74
Table 10: Example value data set with expected results for three aggregation operations 90
Table 11: Expected and computed aggregation results of the example value data set 90
Table 12: Messages sent between client and server for the TLS 1.2 handshake protocol 97
Table 13: Cost comparison between developed system and three cloud services 100
Table 14: Theoretical distribution of queries for a day 109
Table 15: Summary of research questions as answered in this study 121
Table 16: Database system execution times in seconds for part of the tests 139
Table 17: Database system execution times in seconds for second part of the tests 140
Table 18: Database system execution times in seconds for final part of the tests 141

LIST OF FIGURES

Figure 1: Design science research process used in the study 10
Figure 2: Two relations that illustrate relational database concepts 13
Figure 3: JSON document example showing data storage as a key-value pair 16
Figure 4: Graph representation of a family 17
Figure 5: Data storage in a columnar data store compared with a relational DB 18
Figure 6: CAP theorem trade-off choices 22
Figure 7: Lambda architecture layers (recreated from [10, Sec. 1.7]) 27
Figure 8: Comparison of the execution performances of the systems for Query 1 48
Figure 9: Comparison of the execution performances of the systems for Query 2 49
Figure 10: Comparison of the execution performances of the systems for Query 3 50
Figure 11: Comparison of the execution performances of the systems for Query 4 51
Figure 12: Top-level system architecture 53
Figure 13: Resource layer final system diagram 57
Figure 14: Data storage subsystem design with details on its usage 60
Figure 15: Data processing subsystem design 62
Figure 16: Service layer final system diagram 63
Figure 17: Interface layer final system diagram 65
Figure 18: Case study organisation's systems overview 72
Figure 19: Location of the proposed system in the case study organisation 73
Figure 20: Screenshot of Exhibitor website showing each Mesosphere master node's status 75
Figure 21: Mesosphere DC/OS web interface after a successful login 77
Figure 22: Component page showing the health of all Mesosphere DC/OS components 78
Figure 23: User access control page showing an authorised email address 80
Figure 24: First MongoDB log file excerpt for the shard role showing the software activation 82
Figure 25: Second MongoDB log file excerpt for the shard role showing the start of the sharding process 83
Figure 26: Third MongoDB log file excerpt for the shard role showing the start of the replication process 84
Figure 27: Final MongoDB log file excerpt for the shard role showing the completion of the replication process 85
Figure 28: First MongoDB log file excerpt for the config server role showing the software activation 86
Figure 29: Second MongoDB log file excerpt for the config server role showing the start of the replication process 87
Figure 30: Final MongoDB log file excerpt for the config server role showing the completion of the replication process 87
Figure 31: First MongoDB log file excerpt for the router role showing the software activation 88

Figure 32: Final MongoDB log file excerpt for the router role showing the operation of the software 89
Figure 33: Apache Spark master web interface showing cluster information 92
Figure 34: Excerpt of Spark PythonPi program output 93
Figure 35: Apache Spark master web interface showing completed Python program 94
Figure 36: Excerpt of MongoDB Python program output 95
Figure 37: Wireshark capture of TLS messages exchanged between client and server 96
Figure 38: Apache Zeppelin web interface welcome screen 98
Figure 39: Apache Zeppelin notebook with Python test program output 99
Figure 40: Execution time of systems on a logarithmic axis for the single-day single-tag test 102
Figure 41: Execution time of systems on a logarithmic axis for the single-day all-tags test 103
Figure 42: Execution time of systems on a logarithmic axis for the all-data single-tag test 103
Figure 43: Execution time of systems on a logarithmic axis for the all-data all-tags test 104
Figure 44: Execution time of systems on a logarithmic axis for the day-average single-tag test 105
Figure 45: Execution time of systems on a logarithmic axis for the day-average all-tags test 106
Figure 46: Execution time of systems on a logarithmic axis for the all-days-average single-tag test 107
Figure 47: Execution time of systems on a logarithmic axis for the all-days-average all-tags test 107
Figure 48: Completion time of systems for a theoretical set of queries 110
Figure 49: Median execution time of systems for the varying-days single-tag test 111
Figure 50: Median execution time of systems for the varying-days all-tags test 112
Figure 51: Median execution time of systems for the varying-days day-average single-tag test 113
Figure 52: Enlarged region of Figure 51 for the MySQL and developed systems 113
Figure 53: Median execution time of systems for the varying-days day-average all-tags test 114
Figure 54: Median execution time of systems for the varying-users single-day single-tag test 116
Figure 55: Enlarged region of Figure 54 for the MySQL and developed systems 116
Figure 56: Median execution time of systems for the varying-users single-day all-tags test 117
Figure 57: Enlarged region of Figure 56 for the case study and developed systems 117

LIST OF ABBREVIATIONS

AABA	Architecture-centric Agile Big Data Analytics
ACID	atomicity, consistency, isolation, and durability
AP	availability and partition-resilience
API	application programming interface
AWS	Amazon Web Services
BASE	basic availability, soft state, and eventual consistency
BDD	big data design
BSON	binary JavaScript Object Notation
CA	consistency and availability
CAP	consistency, availability, and partition-resilience
CP	consistency and partition-resilience
CPU	central processing unit
CQL	Cassandra Query Language
CSV	comma separated value
DBMS	database management system
DevOps	development and operations
EPCIS	Electronic Product Code Information Services
ETL	extract, transfer, and load
GOOSSDM	Graph Object-oriented Semi-structured Data Model
GPU	graphics processing unit
HARNESS	Hardware- and Network-Enhanced Software Systems for Cloud Computing
HQL	Hive Query Language
HR	human resources
IoT	Internet of Things
IPv4	Internet Protocol version 4
JDBC	Java database connectivity
JSON	JavaScript Object Notation
LVC	lightweight virtualisation cluster
LXC	Linux Container
MDM	master data management
NoSQL	Not Only SQL *or* No to SQL
OCI	Open Container Initiative
ODBC	open database connectivity
OLTP	online transaction processing
PaaS	Platform-as-a-Service

RBAC	role-based access control
RDBMS	relational database management system
REST	representational state transfer
RFID	radio-frequency identification
SOA	service-oriented architecture
SOAP	simple object access protocol
SQL	structured query language
SSL	secure sockets layer
TCO	total cost of ownership
TLS	transport layer security
TOSCA	Topology and Orchestration Specification for Cloud Applications
URL	uniform resource locator
VM	virtual machine
WCF	Windows Communication Foundation
XML	extensible markup language

CHAPTER 1 BIG DATA IN ORGANISATIONS

1.1 Big data and its challenges in organisations

Big data. No concept has had such a significant impact on the data landscape of our modern time [1]. Big data has changed the way organisations perceive and use data. It has unknowingly improved our lives with more and better services [2]. Many sectors such as healthcare, transportation, energy, and the Internet of Things (IoT) have benefitted and are still benefitting from big data. In the following paragraphs, big data will be defined as it is one of the central concepts in this study. The challenges of big data within its context of organisations will be discussed. Data silos, a problem exacerbated by big data, will be introduced. This section will conclude with an outlook on adopting big data. This outlook sets the tone for the problem statement given in Section 1.2. A more detailed discussion of the technical aspects in this section will follow in Chapter 2.

What is big data? A common misconception is that big data only refers to data with an enormous size. However, an early definition of big data was that of data described in terms of three Vs [1, pp. 2–3], [2, p. 1]. The Vs are volume, variety, and velocity. Volume refers to the physical size of the data. It can be measured by the storage capacity required to store the data. Typically, the volume of big data could range from terabytes[1] to zettabytes[2]. Variety is a measure of unique or different data types. Variety is also classified in terms of structured, partially structured, and unstructured data. For example, data could consist of product information (partially structured), web pages (unstructured), and customer contact information (structured) [2, p. 96]. Velocity is the speed or rate at which data is generated. Velocity, therefore, dictates the speed at which data needs to be ingested into a system.

As the big data field grows, additional Vs have been added to the three-V definition. Three additional Vs that have been added are veracity, variability, and value [3]. Veracity indicates the uncertainty or unreliability of data. For example, data with a high veracity would have greater uncertainty and lower quality. Variability refers to the variation of the data velocity. This characteristic recognises that the flow rate of data fluctuates with possible peaks and troughs. Value refers to the usefulness of data, which is the impact of data [2, p. 1]. An example of the impact of data is that it could allow the identification of previously undiscovered patterns, thereby leading to new products or services. In [3], big data is said to have a low value relative to its volume, i.e. a low value-density. What is clear, however, is that the reason for processing and analysing big data is to obtain a high value.

How did big data emerge? It may seem that big data has only existed for the past decade. However, big data has been a challenge in some industries for longer than the existence of the term “big data” [1, p. 3].

[1] 1 terabyte = 10^{12} bytes
[2] 1 zettabyte = 10^{21} bytes

Industries in areas such as meteorology, genomics, banking, and insurance have experienced big data problems related to volume. In these cases, custom programs and systems were created to address their needs. However, as is the case with custom-developed software, the cost is immense. Here "cost" refers to the total cost of ownership (TCO) for these custom systems to these industries. TCO means all the costs of a system over its entire lifetime. Thus, maintenance, upgrades, electricity, labour, acquisition, and many more costs are included. The high TCO of custom systems prevents smaller businesses and start-up companies from capitalising on their data. As will be shown in Chapter 2, one of big data's impacts was tearing down the TCO barrier.

Why is big data important? Global data production is increasing at an exponential rate [4]. In 2013, 90% of the world's data had been generated in the previous two years [5]. In 2014, the International Data Corporation estimated that the digital universe doubles in size every two years [6]. Some of the main factors that cause this exponential growth of data are:

- Increasing use of social media such as Twitter, Facebook, and Instagram.
- The ubiquity of IoT devices that create and send data unlike ever before.
- Affordable sensors in any item to measure various phenomena.

For big data to be of value, it needs to be stored, processed, and analysed. If done correctly, big data can deliver business and competitive advantages [2, p. 132]. Organisations can gain and are already gaining valuable insights from big data [4]. By leveraging big data, an organisation can create data transparency, improve performance, customise services for specific segments, automate decision-making, innovate, and create [1, pp. 5–11]. Simply stated, by supporting and using big data correctly, an organisation can become more efficient and more successful.

Big data has had an advantageous impact in many sectors [1, Ch. 1]. Search engine companies such as Google, Yahoo!, and Bing have revolutionised searching large collections of information [1, p. 4]. Their successes were enabled by advancing natural language processing and semantic analysis technologies. These technologies are directly linked to big data.

Walmart analysts used big data to discover insights into product purchases with the issuing of hurricane warnings [1, pp. 5–6]. They found increased sales of expected items, such as batteries and flashlights, but also unexpected items, such as strawberry Pop Tart pastries and beer. These findings caused Walmart to stock extras of these items, such as for hurricane Frances during 2004.

At the University of Ontario, big data analytics is decreasing premature infant mortality rates [1, p. 6]. Dr Carolyn McGregor uses big data technology to collect real-time data streams, such as respiration and heart rate, from premature babies. Analysing these data streams enables the early detection of life-threatening infections.

As organisations store more data from more sources, data islands may start to form [1, p. 42]. These data islands form because of each business application storing and managing its data in an application database[3]. Application databases allow applications to be responsible for managing their data. Thus, whenever changes are made to some aspect of an application's database, other applications are not affected. When there is no sharing of information between application databases, the term "data islands" or "silos" are used.

The consequences of data silos are data inconsistency and data duplication. To illustrate data silos, consider this example. An organisation's human resources (HR) department may have an employee database. The same organisation also has a finance department with a payroll database. Imagine if the financial department had their own set of employee data instead of referring to the HR department's employee data. If the HR department changes any employee data, the financial department will end up using outdated data.

Data silos decrease productivity and the effectiveness of analytics. These implications were discussed by Li et al. in [7]. They stated that nearly half of a knowledge worker's time is wasted on unproductive tasks. These tasks include gathering of and searching for information, recreating existing information, and converting between data formats. A similar situation was discussed by Piedra et al. [8]. They specifically mentioned the increased difficulty of accessing data within multiple silos and the consequently reduced interoperability. In particular, they stated that data silos impede the free flow of information. In their case, data silos resulted in time being wasted by unnecessary searching for resources. Furthermore, time could also be wasted when making decisions based on insufficient information.

Organisations' traditional data systems are unable to support big data [1, p. 1]. Traditional technology is capable of supporting structured data, but not the variety of big data [2, p. 19], [9, pp. 33–39]. The creation and processing rates of big data are too great for traditional data systems. Another problem of traditional data systems is their inability to scale with an increase in data volume [10, Ch. 1]. The traditional way of increasing deteriorating data system performance was procuring a bigger computer [11, p. 8]. There are, however, limits to how powerful a computer one can procure. Not to mention the greater cost associated with bigger computers.

Adoption of big data is still in its infancy. Research by Gartner showed that of 199 companies, 48% invested in big data during 2016 [12]. Furthermore, only 15% of the respondents had deployed their big data projects to production. In 2015, the percentage of production big data projects was 14%. Thus, there was only a 1% increase in the number of production big data projects. Many companies are, however, experimenting with production big data projects. In a white paper by Knowledgent, 25% of a 100 respondents indicated that they had implemented a big data solution [13]. A final indication of big data adoption is given in a report by the International Institute for Analytics [14]. Their research on 194 USA and international companies

[3] A database is an ordered collection of related data stored on a computer. A database is organised in such a way as to allow swift retrieval and searching. Databases provide the capabilities to store, retrieve, modify, and delete data.

showed that 29% of companies have an operational big data system. A further 31% were busy implementing a big data system.

1.2 Problem statement

As was shown in the previous section, big data adoption is still in its infancy globally. Organisations are, therefore, not able to reap the benefits of big data. Furthermore, the traditional database systems of organisations are incapable of supporting big data. A further brief glimpse into the organisational data system landscape revealed the existence of data silos. Data silos can, however, prevent a successful big data system and negate the benefits of big data.

These previous observations are even more applicable to South Africa. Van Zyl reported on a survey of 50 South African businesses that 54% are not performing big data mining and analysis [15]. In the South African mining sector, Dennis Gibson, chief technical officer at Black and Veatch, identified big data as the big intervention from technology [16]. Furthermore, he predicted that big data and the analysis thereof would have a pronounced improvement in operational efficiency.

Despite the research already done in the big data field, many aspects remain unexplored while others are covered insufficiently [17]–[23]. Jin et al. stated that, "There are many challenges in harnessing the potential of big data today, ranging from the design of processing systems at the lower layer to analysis means at the higher layer …" [17]. Abbasi et al. asked the question, "… how can we use approaches such as action design research (ADR) to guide the development and harnessing of big data IT artefacts in organisational settings?" [18]. Abbasi et al. considered security a top priority by describing that, "In particular, privacy, security, and ethics of big data have significant implications and, hence, deserve special attention."

Zillner et al. continued identifying research directions specifically for industrial organisations. They noted that, "Within all industrial sectors it became clear that it was not the availability of technology, but the lack of business cases and business models that is hindering the implementation of big data" [19]. Günther et al. realised that, "Our review shows that the current literature is still at a nascent stage in terms of explaining how organisations realise value from big data" [20]. Thus, the reason for organisations' lack of big data adoption stems from a need to illustrate the value of big data. Sectors such as the mining and industrial sectors lack practical implementations that would aid in the adoption and acceptance of big data.

The literary works in this section all express a requirement for more big data research regarding systems and case studies in insufficiently explored industries. In summary, there is a need to perform big data research globally, but more specifically in the South African mining and industrial sectors. To assist with big data adoption, research should illustrate the value of big data to organisations. Hence, practical research must be performed within organisations of the mining and industrial sectors. Practical research needs to

focus on big data system implementation and solve the data silos problem. Finally, research should suggest how a big data system should be constructed and developed as an example to organisations.

1.3 Research questions

The solution proposed by this study is to design, develop, and implement a centralised data management system to realise some of big data's benefits. Specifically, the need for improved efficiency will be addressed by the proposed system. The proposed system is centralised in the sense that it would indirectly prevent data silos from forming. This prevention means it would support the data storage needs of various business systems and processes. The system is a data management system in the sense that it would not be a simple implementation of a big data supported database. Instead, it would provide data storage and processing, and allow for additional analytics. Such a system should support big data and allow for future big data projects. The future support of big data projects is important, because the big data field is ever expanding and improving.

This study's principal objective was to design and implement a centralised data management system. Thus, research into the big data field was performed to achieve this study objective. A literature review was conducted on the various aspects that constitute a centralised data management system. The knowledge gained from the literature review allowed for the design of the proposed system. Following a successful system design, the study proceeded with system development, verification, and implementation. Implementation was followed by an evaluation to validate the system and identify areas for improvement.

The questions that directed this research are as follows:

1. How should the data model for a centralised data management system be structured?
2. What system architecture could be used to design a centralised data management system?
3. Which big data platforms can be used to develop a centralised data management system?
4. Which type of service can be used to make the functionality of the centralised data management system available to systems and users?
5. What impact does the centralised data management system have on the efficiency of users and systems within the context of a case study?

The research questions lead to the objectives of this research, of which the main objective is stated below.

Main objective:

To design, develop, and implement a centralised data management system to improve organisational efficiency using big data.

Specific research objectives were identified to achieve the main objective. These specific objectives were grouped into literature and empirical objectives in the subsections that follow.

Literature objectives:

1. Investigate the characteristics of big data systems compared with traditional data systems.
2. Discover the architectures used in the design and development of big data systems in industrial settings.
3. Determine the methods and best practices used in industry to make the functionalities of big data systems available.
4. Evaluate the software to be used in the system for storing, processing, and managing big data.

Empirical objectives:

1. Evaluate the financial feasibility of the developed system, especially compared with the alternatives.
2. Compare the time performance of the developed system against an organisation's (traditional) data system.
3. Investigate the impact of increasing data volume on the time performance of the developed system.
4. Test the ability of the developed system to support multiple simultaneous users, which is a typical situation in an organisation.
5. Identify the impact of security on the developed system.

1.4 Research process

This study used design science research to structure the research process. The design science research methodology that was used subscribes to the design science research methodology of Peffers et al. [24] and the software engineering research framework of Uysal [25]. While the main research process was that of design science, attributes of case study research were also present in this study [26]. In general, the nature of the research was exploratory and improving. It was exploratory due to the lack of implementations within the study context. Therefore, this study will apply new (big data) principles to new organisational problems. The research was an improvement due to the research objective, which is for organisations to support big data and become more efficient. Therefore, the research was aimed at organisational data management improvement.

A detailed discussion of design science is beyond the scope of this study. Various literary works do, however, provide such a discussion, which formed the basis for this section [24], [25], [27]–[29]. What follows is a brief discussion of design science and how it aligned with the rest of the dissertation.

Design science research comprise three cycles [24], [27], [28]:

- Relevance cycle: In the first part of this cycle, problems and opportunities within a chosen application domain are identified. This application domain or context defines the requirements for the research, as well as the result acceptance criteria. The second part of this cycle is performed at the end of the design science research process. In this part, field testing of the designed artefact, i.e. the proposed system, is performed.

- Rigour cycle: This cycle provides the knowledge base to conduct design science research. Thorough research of scientific theories, engineering methods, expertise, and experience, as well as existing artefacts, should be performed. This thorough research is required to ensure innovation and definite research contributions. As part of the rigour cycle, any research contributions should be added to the knowledge base.
- Design cycle: It is the core of design science research where most of the actual work is done. This cycle iterates over the activities of designing, developing, and evaluating the artefact. The feedback from the evaluation of the artefact dictates if further iterations and refinement are required. The design cycle uses the outputs of the previous two cycles. The relevance cycle provides the requirements of the artefact. The rigour cycle provides the theories and methods used within the design cycle. Completion of the design cycle releases the artefact to the relevance cycle where it is field tested.

It is important to note that the design science cycles are iterative. For example, if field testing of the artefact during the relevance cycle is not satisfactory, a new iteration of all three cycles is initiated.

As stated earlier, this study followed a design science research process similar to that described in [24] and [25]. This design science research process is illustrated in Figure 1. In the figure, the process started with problem identification and finished with evaluation. At the verification and evaluation stages, it was possible to iterate back to previous stages, as indicated by the dashed lines. This dissertation followed the same outline as given in Figure 1. The exact meaning and location of each stage in the design science research process will be described briefly in the following paragraphs.

The identification of the need for big data system implementations in the South African mining and industrial sectors corresponds to the first part of the relevance cycle. During this stage, a specific problem in the context of organisational big data management was identified. The objectives and research questions that directed the research were defined. Chapter 1 of the dissertation is dedicated to this stage.

The investigation stage corresponds to the first part of the rigour cycle. An investigation was completed in the form of a literature study. In this stage, various literary works were studied to identify methods, expertise, and theories that could aid in the design of the proposed big data system. An evaluation of big data software was also performed, thereby identifying potential software for use in the design stage. This stage is given in Chapter 2 of the document.

Technical work was performed in the design stage, which is one part of the design cycle. This stage used the work from the preceding stages to create designs for the proposed system. Trade-off analysis was used to compare different design alternatives. This stage was visited several times to refine and improve the system's design. A part of Chapter 3 was employed for this stage.

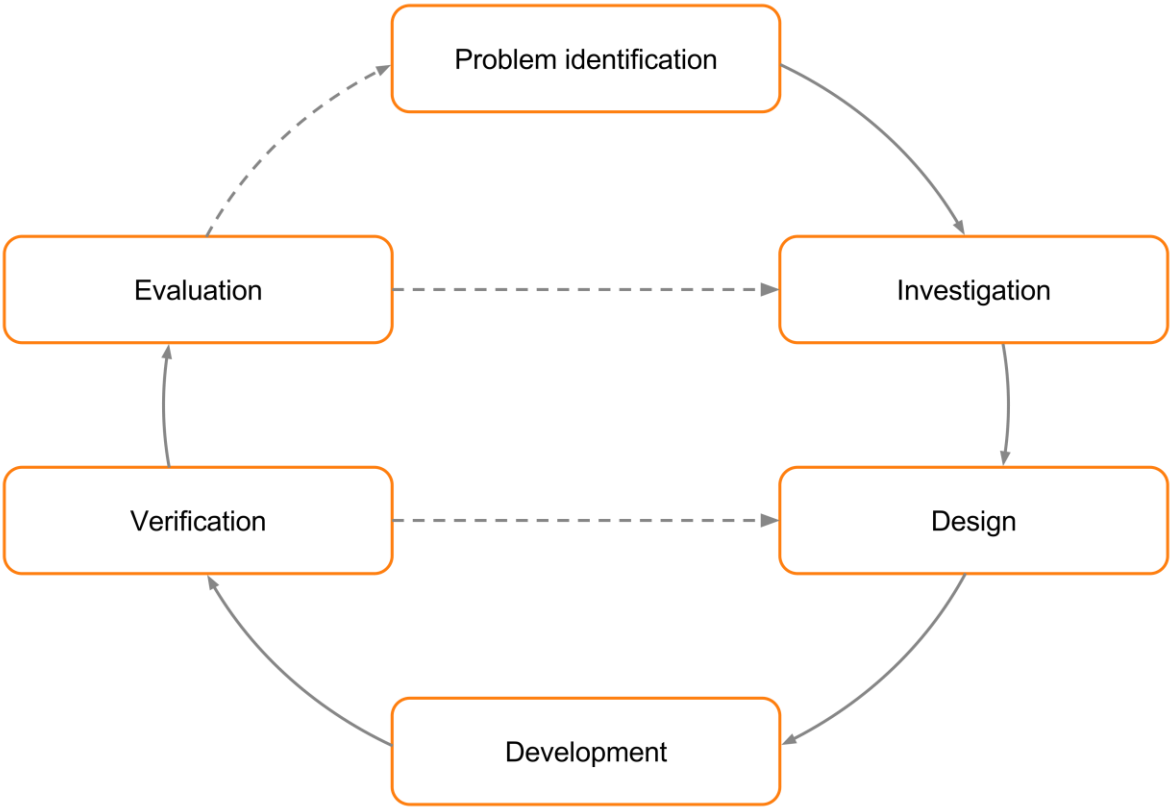


Figure 1: Design science research process used in the study

The development stage consisted of activities such as coding, installing, connecting, and setting up to create a functioning system. This stage is another part of the design cycle. It was more technical than the design stage and, therefore, only some part of it was included in this dissertation. The final part of Chapter 3 was dedicated to this stage.

Verification is the stage in which the functioning system was tested for correct operation. The specifications created in the design stage provided the verification criteria. This stage decided if additional iterations of the design cycle were necessary. Therefore, it is the final part of the design cycle. This stage is described in Chapter 4.

The final stage, evaluation, was used to implement the system on a case study. The case study was used to validate the system as specified in the problem identification stage. If the system did not achieve the research objective, further investigation or problem identification could have been done. Conclusions and contributions were made based on the validation results. This stage is part of both the relevance and rigour cycles. The remainder of Chapter 4 and the entirety of Chapter 5 are dedicated to this stage.

CHAPTER 2 STUDY OF DATA SYSTEM DESIGNS AND IMPLEMENTATIONS

2.1 Preamble

In this chapter, the research questions given in Section 1.3 were expanded to provide a knowledge base. This knowledge base, as discussed in Section 1.4, was instrumental in the design and development of the proposed system. This chapter starts by providing a brief history of the development of data systems, and in particular, database systems. The concepts and explanations provided are required to grasp the technical discussions in the rest of the chapter. Section 2.3 describes existing big data architectures for system development. This section relates directly to Research Question 2. In Section 2.4, some of the ways in which a system's functionality can be made available are discussed. Therefore, Section 2.4 refers to Research Question 4. Section 2.5 details an evaluation of data store software that was considered for the development of the system. Finally, a conclusion of this chapter is provided in Section 2.6.

2.2 Brief history of databases

Data systems are used to organise data [30]. The first of these data systems were manual records [30, p. 4]. With manual records, data was stored on physical files such as invoices and logs. Accounting and administrative personnel performed calculations and filing using these physical files. Some improvement was made with the introduction of early computation machines [30, p. 8]. These machines required data to be stored on paper tapes and keypunch cards, which are termed storage media. Keypunch operators were responsible for organising and creating the data on the storage media from physical files.

With the introduction of the minicomputer in the late 1950s and early 1960s [31], manual files were replaced with computer files [30, p. 8]. Widespread usage of computers and new storage media allowed for sequential file data systems to develop. As the name indicates, data was organised as sequential or flat files on magnetic tapes and hard disk drives. Only the same type of data could be stored in a file. When stored in a file, the data was sorted sequentially based on the creation order.

As technology progressed, it became possible to access data even faster and in a random order, and to store more data [30, pp. 6–8]. These advancements allowed for database systems to be developed. As indicated by the name, data is stored in a database. Databases allow access to data in a random or sequential order. Database systems kept evolving since their introduction in the 1970s. Since that time, two phenomena have led to the development of progressively better database systems: the demand for information and the growth of computing technology [30, p. 5]. For the remainder of the study, it is important to remember that a database system is one type of data system.

The demand for information is driven by organisations growing to establish a global presence [30, pp. 8–9]. This organisational growth has caused relentless competitive pressure. Consequently, information has become more important to make a profit and compete globally. The information demand is also driven by the need for organisations to improve customer service, to plan, and to increase awareness of organisational functioning [30, Ch. 4]. Different people in an organisation require access to the information they need to accomplish tasks. Therefore, there is a necessity for greater amounts of information of different types for different purposes [30, p. 9]. What is more, information can also be integrated with other information. However, without quick access to, sharing of, and storage of this information, it would be useless. Therefore, there is a need for progressively better database systems.

The growth of computing technology is a direct result of humanity's endless quest for technological advancement [30, pp. 4–6]. Technological growth is visible in the following examples [30, p. 6], [31, p. 52]:

- Computers can now store more, calculate faster, use less energy, and cost less than in the preceding decades.
- Computer networks have become widespread and connect almost all computers to the internet.
- Software development and human-computer interfaces have become more complex and can do more than during any previous time.

The demand for information has fuelled the growth of computing technology. As more information is required at a faster pace, technology was created or improved to satisfy the demand. Conversely, better storage and processing technologies created an opportunity to store and process larger amounts of increasingly complex information. Naturally, computing technology growth has improved many areas related to database systems.

Before continuing, it is necessary to define the different terms that will be used subsequently. A database management system (DBMS) is specialised software that provides various database capabilities. These capabilities include creating, using, managing, and protecting databases [30, pp. 42–43]. Some examples of DBMSs are Oracle Database, IBM DB2, and Oracle MySQL. Synonyms for DBMS are database software and database system. Thus, a database contains data and structural definitions of how data is stored [30, p. 21]. In common practice, the terms "database" and "DBMS" are used interchangeably, which is incorrect [30, p. 42]. This document will use the terms correctly.

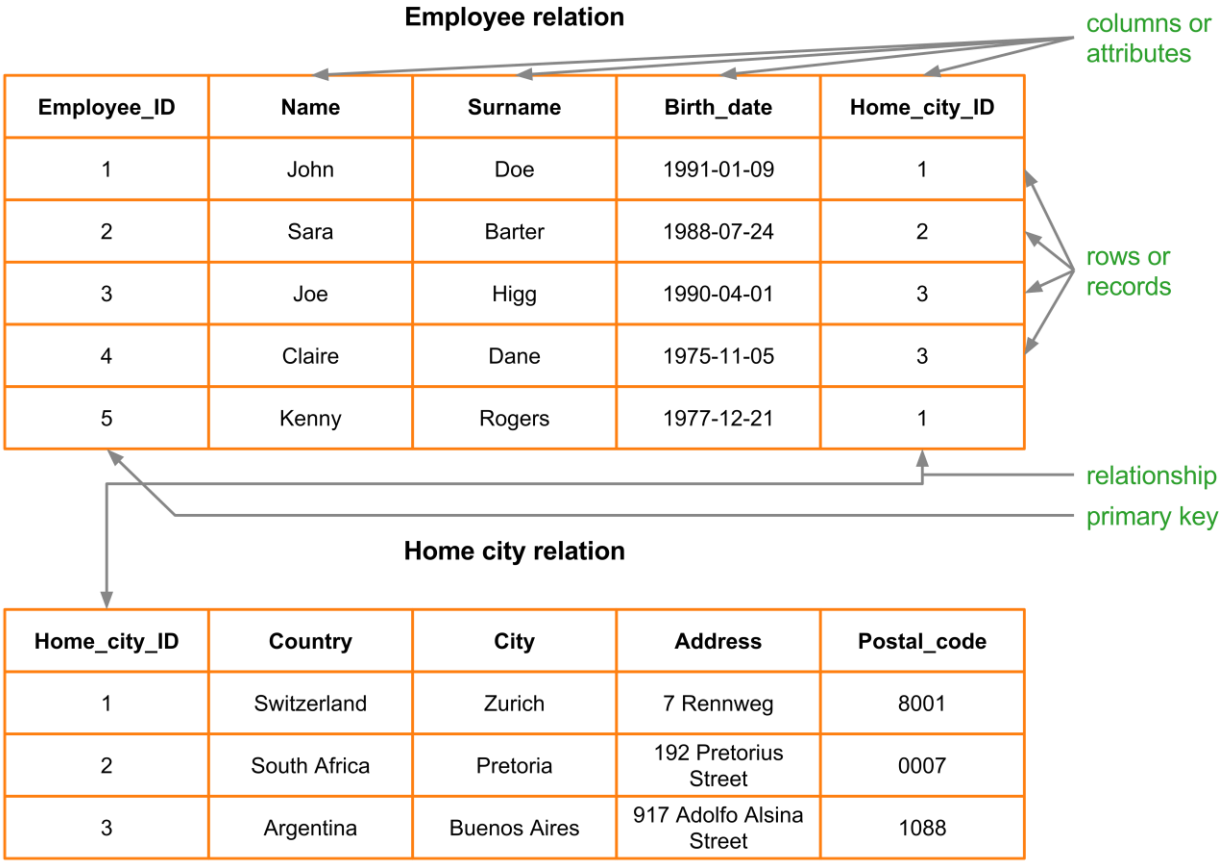
2.2.1 Relational database systems

The evolution of databases started with the creation of the relational database management system (RDBMS) [1, Ch. 4]. To access a database within an RDBMS requires using a structured query language (SQL) or a derivative thereof. The primary workloads for which an RDBMS was designed are online transaction processing (OLTP) and business intelligence. A database created by an RDBMS is termed a relational database. A relational database stores data in strictly defined and formatted tables [30,

pp. 27–28]. Relational databases and indeed RDBMSs were founded on the relational model, which was created by Dr E. F. Codd [30, p. 27].

The exact details of the relational model and relational database tables are discussed in [30, Ch. 8] and [32, Ch. 3]. From there, a relational database consists of one or more relational database tables. A relational database table is a two-dimensional object with rows and columns. In the relational model, a table is referred to as a relation. Each row of a relation is known as a record, instance, or tuple. The columns of a relation are referred to as attributes, fields, elements, or properties. Each record has an attribute, known as a primary key, that uniquely identifies it in the relation.

A relation stores data about an entity. An entity is an object or concept about which data is stored. Some examples of entities are a university course, an employee, or a purchase order. A database can contain an index, which is created from one or more unique attributes. The index can then be used to retrieve records quickly as opposed to searching through attributes. A final important aspect is the fact that relations can have relationships between them. An example of these relationships, as well as the details discussed in this paragraph, are shown in Figure 2.



Employee relation

Employee_ID	Name	Surname	Birth_date	Home_city_ID
1	John	Doe	1991-01-09	1
2	Sara	Barter	1988-07-24	2
3	Joe	Higg	1990-04-01	3
4	Claire	Dane	1975-11-05	3
5	Kenny	Rogers	1977-12-21	1

Home city relation

Home_city_ID	Country	City	Address	Postal_code
1	Switzerland	Zurich	7 Rennweg	8001
2	South Africa	Pretoria	192 Pretorius Street	0007
3	Argentina	Buenos Aires	917 Adolfo Alsina Street	1088

Figure 2: Two relations that illustrate relational database concepts

Figure 2 shows two relations. The first relation contains data of employees. The second relation contains all the possible home cities of the employees. Each employee record has a relationship with one home city.

The benefit of using a relationship as opposed to simply having home city attributes in the employee relation, is as follows: As more employee data is stored in the employee relation, there will be a duplication of home city information. By moving the home city information to a separate relation, one can refer to all home city information with its primary key. The result is that the home city information is independent of the employee information, and there is no unnecessary duplication of data. This process is referred to as normalisation and is discussed in [30, Ch. 10] and [32, Ch. 4].

Data stored in a relational database is structured as is apparent in the usage of an SQL and the relational model [1, pp. 77–78], [30, p. 27]. Hence, a relational database can be represented with three levels, which are known as schemas [30, pp. 129–131], [32, pp. 15–17]. The external schema comprises all the user views of the database. A user view contains some set of data elements from the database. When interacting with a database, users would only see the data elements of the view they are using. Therefore, user views allow a user only to see the data in which they are interested.

The conceptual schema is a representation of the entire database content. The conceptual schema can be thought of as an aggregation of all user views. This schema is more technical as it defines details about the storage and management of data in the database. Some of the conceptual schema details are the database structure, security constraints, and integrity checks. The final schema is the internal schema, also known as the physical schema. It defines the low-level details of the database, such as index files, storage of records, and data access.

In [1, Ch. 4], details are provided on the set-up of RDBMSs. The original set-up for an RDBMS was a single large server. Initial improvements were made in the area of parallel workload processing. These created set-ups where multiple servers were interconnected to form a cluster, which gave rise to cluster-based architecture. The advantages that cluster-based architectures offered compared with single servers are greater parallelism, high performance, and failover mechanisms. Failover, in this context, refers to the ability of a system to continue operating when it encounters a fault or error. The advancement of network technology allowed ever larger clusters of servers, which is referred to as grid computing.

2.2.2 NoSQL data stores

The late 2000s saw a revolution in the big data field [11, Sec. 1.5]. This revolution provided an alternative to relational databases and the aforementioned high TCO of custom big data systems. The revolution was the rise of NoSQL. NoSQL is characterised by its lack of a clear meaning [11, p. 10]. Two of the popular NoSQL meanings are “Not Only SQL” and “No to SQL”. In this study, NoSQL was taken to refer to data stores[4] with the following characteristics [1, Ch. 4], [2, Ch. 3], [9, Sec. 3.3], [11, pp. 10–12]:

[4] Data store is used as the NoSQL equivalent of a database system. The reason being that some NoSQL systems do not use databases in the traditional relational sense. The term is also self-explanatory as it refers to some system that stores data.

- Do not use SQL for data access or other tasks.
- Are generally open source projects.
- Operate with a dynamic schema or without a schema, i.e. schemaless.
- Have the ability to scale horizontally, except for graph databases, which are discussed later.
- Support higher data volumes than RDBMSs.
- Can accommodate different data types, unlike RDBMSs that only support structured data. This support for different data types means that NoSQL data stores do not have a relational data model.

NoSQL data stores can also be referred to as non-relational databases [9, p. 38]. In general, there are four types of NoSQL data store as distinguished from their data model [1, pp. 78–80, 170], [11, pp. xvii–xviii], [33, Sec. 4.3], [34, Sec. 4]:

- Key-value data store
- Document data store
- Graph data store
- Columnar data store

Key-value data stores, as discussed in [1, pp. 81, 166], [33, Sec. 4.3.1] and [34, Sec. 4.1], store data as key-value pairs. A key-value pair consists of two fields, namely, key and value. The key field must be unique as it is used to access a specific value. Therefore, a data search can only be performed on a key. A key-value pair can almost be thought of as a two-column table. The first column contains the key data and the second column contains the value data. The difference, however, is that there is no constraint on the value's data. This lack of a constraint means that different key-value pairs can store different data types and structures within the value field.

A document data store, as discussed in [1, pp. 166, 170–171], [33, Sec. 4.3.1] and [34, Sec. 4.2], stores documents. These documents are similar to key-value pairs. The key must be unique, but the value field can store data with greater complexity. Typically, the value field is stored in standard data formats. Examples of these are the extensible markup language (XML), JavaScript Object Notation (JSON), or binary JSON (BSON). These standard data formats are data structures with attribute name and value pairs, similar to key-value pairs. Thus, a document store allows searches on a document's value field and not just the key. An example of a JSON document is shown in Figure 3.

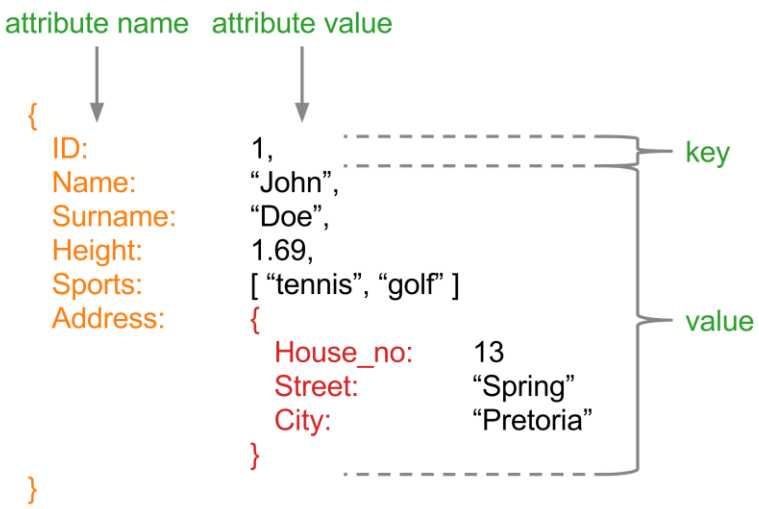


Figure 3: JSON document example showing data storage as a key-value pair

Figure 3 shows how to interpret a JSON document as a conceptual key-value pair. The document’s key and value comprise attribute name and value pairs. The flexibility of the data stored within an attribute value is evident. As shown, attribute values can be strings, integers, floating-point values, or collections. The address attribute value contains an entire document. In such a situation, the document is said to be nested or embedded in the outer document. The embedded document is referred to as a subdocument.

Graph data stores are discussed in [1, pp. 166, 171], [34, Sec. 1] and [35, Ch. 1]. A graph data store uses graph theory for its data model. As a result, it is more focused on relationships and visual data representations. Data is stored as a node (also known as a vertex), node relationship (also known as an edge), or node property. Nodes are conceptual objects and are the main entities about which relationships are stored. Nodes are described by node properties, which are key-value pairs. Figure 4 illustrates these concepts.

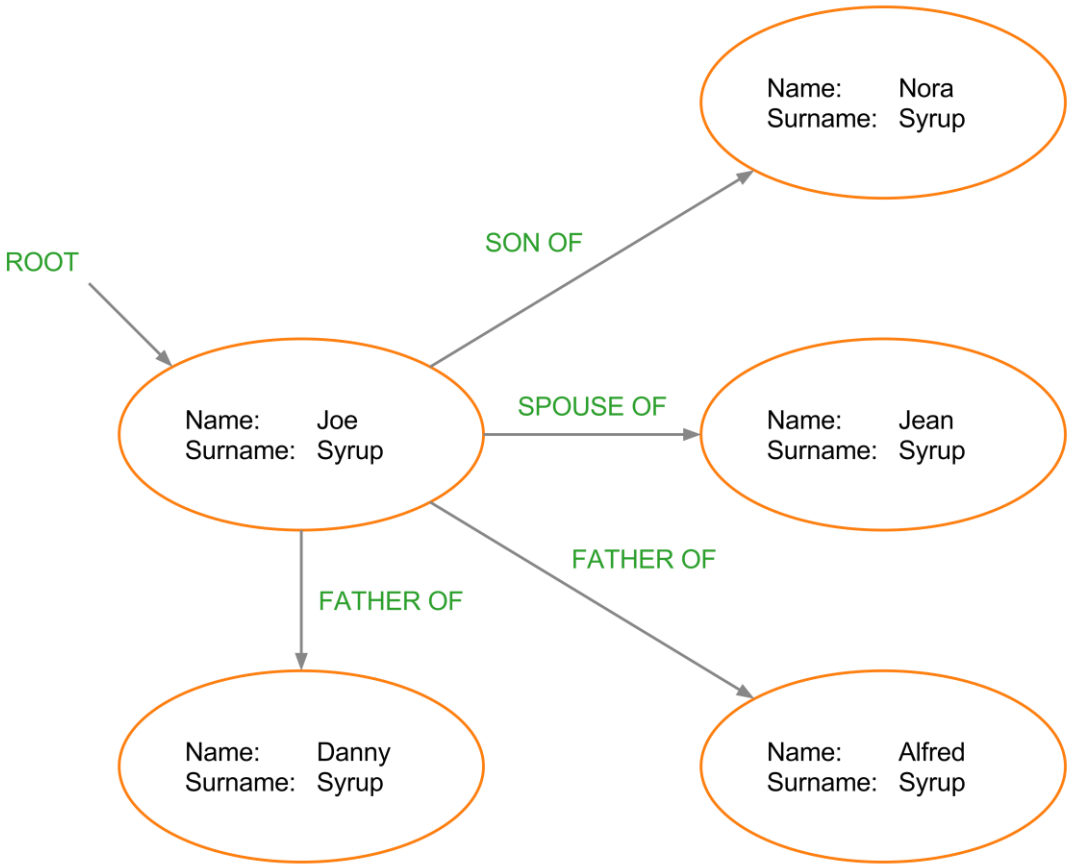
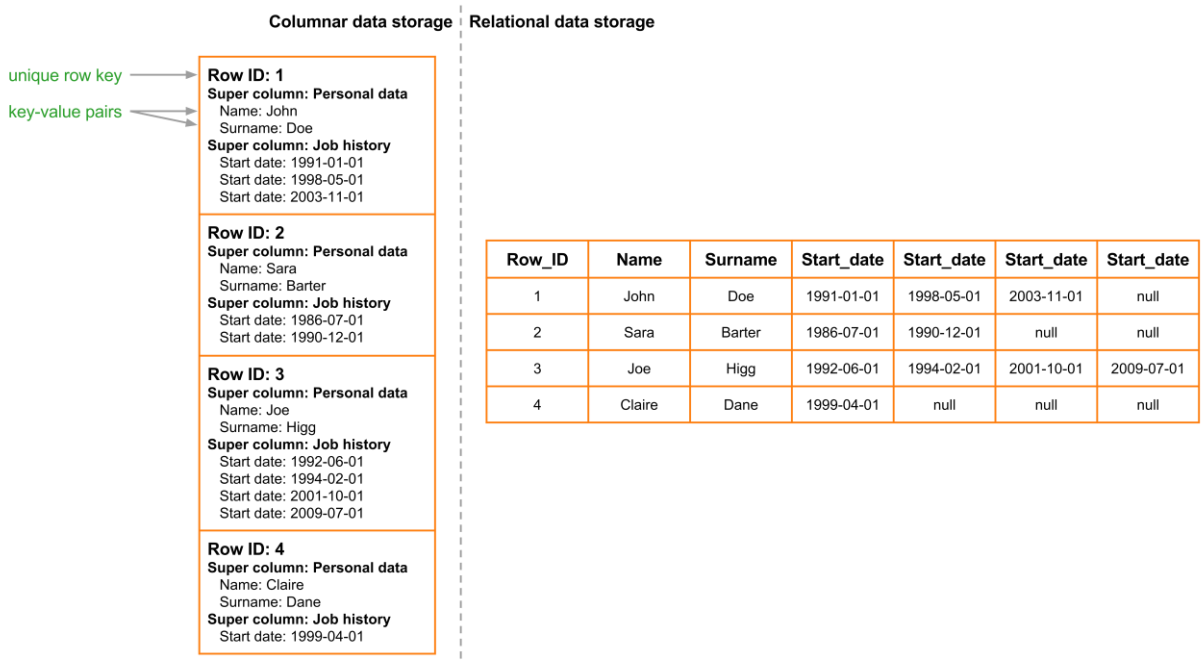


Figure 4: Graph representation of a family

In Figure 4, a family network is represented as a graph. In the figure, the nodes are people, and the edges are the human relationships between them. For simplicity, the focus of the graph is on Joe Syrup, who is the root node of the graph. All relationships in the graph are those that Joe has toward other people. For example, Joe Syrup is the son of Nora Syrup. It is also true that Nora Syrup is the mother of Joe Syrup. Such relationships would, however, have cluttered the figure with too many edges between people.

Columnar or column-oriented data stores, as described in [1, pp. 172–176, 94–95], [33, Sec. 4.3.1] and [34, Sec. 4.3], store data column-wise. This column-wise storage means that there are no required columns for each row. Two rows may have different columns and any number of columns. Columnar data stores have structured data models, similar to RDBMSs. There is, however, no concept of tables as with the relational model. Most columnar data stores were established from Google's Bigtable data system. To clarify the storage of columnar data, refer to Figure 5.



Row_ID	Name	Surname	Start_date	Start_date	Start_date	Start_date
1	John	Doe	1991-01-01	1998-05-01	2003-11-01	null
2	Sara	Barter	1986-07-01	1990-12-01	null	null
3	Joe	Higg	1992-06-01	1994-02-01	2001-10-01	2009-07-01
4	Claire	Dane	1999-04-01	null	null	null

Figure 5: Data storage in a columnar data store compared with a relational DB

Figure 5 shows an example of people's job history when stored in a columnar data store. The corresponding relational counterpart is also shown. Again, key-value pairs are used to store data, although here it is in a columnar form. The super column is simply a grouping of one or more columns. The figure shows only a simplified view of columnar data storage. Greater detail can be found in this study's references, for example, [1, pp. 172–176, 94–95], [33, Sec. 4.3.1] and [34, Sec. 4.3].

There are various advantages to using NoSQL data stores [1, pp. 75–77]. Many data stores have a distributed architecture making them highly parallel and partition-tolerant. Partition-tolerance is the ability to recover from the loss of a part (partition) of a data set. The distributed architecture of NoSQL data stores is also referred to as scale-out architecture. Scaling out or horizontal scaling refers to the usage of many computers to store data [1, pp. 75, 111]. Using many computers also distributes the processing of data, because queries can be satisfied by some set of the computers. Concurrency is increased because computers that are not affected by a query can execute other queries.

Scaling can also be performed in a vertical direction, also known as scaling up [1, pp. 91–93, 111]. As an example, consider a single server that is running an RDBMS. One would scale up by either upgrading the server or commissioning a more powerful server. It should be clear that scaling up becomes infeasible at some point [1, pp. 91–92]. There is a physical limit to the computing power of a single computer. Furthermore, more powerful computers become progressively more expensive.

It was mentioned in Section 1.1 that traditional data systems, i.e. RDBMSs, scale vertically. It is also true that RDBMSs can be extended to clusters. Therefore, it may seem that RDBMSs can also scale horizontally. Sadalage and Fowler, however, discussed this question in [11, Sec. 1.4]. They concluded that sharding, as

discussed later, an RDBMS is not the same as sharding a NoSQL data store. In effect, RDBMS sharding creates separate databases. Separate databases mean that it is left to the user or application to know in which database data resides. Thus, the consensus is that RDBMSs can only scale vertically.

Two mechanisms can be used to achieve scale-out architecture replication and sharding [1, pp. 75–77]. Before explaining the two mechanisms, one needs to understand database workloads [1, pp. 82–86]. A read-intensive workload means that a database performs many data-reading operations. Similarly, write-intensive workloads have a high number of write operations. Therefore, a mechanism that provides read scalability can support greater read-intensive workloads by scaling the data store. In the same way, write scalability refers to data store scaling for greater write-intensive workloads.

Replication refers to copying the same data to multiple computers, which are called nodes [1, p. 76]. There are two types of replication, namely, master-slave and peer-to-peer replication. Master-slave replication provides read scalability. One node is designated as the master. The master contains the most accurate and up-to-date data set. All other nodes are designated as slaves. Each slave has a copy of the master's data set. Updates are only performed on the master's data set. Each slave periodically synchronises to the master. Read scalability can be improved by adding more slaves. The problem, however, is that intensive write operations degrade performance due to too many synchronisation processes.

Peer-to-peer replication [1, p. 76] provides write scalability because there is no master, i.e. each node is equal. Each node can accept write operations, and each node has a copy of the data set. The disadvantage, however, is that a node might not have the most up-to-date data set. Thus, nodes have different versions of the same data set, which is known as data inconsistency. The following scenario further complicates data consistency: Imagine if two users try to update the same record with different values. This update happens at the same time, relatively speaking, on different nodes. The question is which value the updated record should have. There are techniques to resolve this conflict, but they are beyond the current scope.

Sharding, also known as a shared-nothing architecture, is when a data set is selectively divided into parts [1, p. 77]. These parts are called shards. Each shard is distributed to a different node. Therefore, each node has a part of the data set, and no node has the entire data set. The advantage is a rapid response to queries that operate only on a single shard. This data distribution means that given the constraint, both write and read scalability are provided. Problems arise, however, when a query references all data in a data set. Consequently, careful planning is required to create shards. Usually, optimal shards are created when they contain data that will most likely be accessed together.

The best scale-out approach is to combine master-slave replication with sharding [1, p. 77]. This approach means that each shard is copied to some number of nodes. This set of nodes has one master with multiple slaves. If a shard is destroyed, it can simply be replaced by a copy from another node. Therefore, one retains

the scalability afforded by sharding with partition-tolerance provided by master-slave replication. This approach was used as a criterion to evaluate possible data stores for the proposed system.

2.3 Big data system architectures

The previous section investigated the traditional use of database systems. The development of NoSQL data stores was briefly described. It was said that NoSQL data stores were developed based on some of the inadequacies of traditional relational database systems. The new paradigm in big data systems is thus built on the usage of NoSQL data stores. This section starts with the characteristics of database and distributed systems. Next, big data systems as implemented in various industries were investigated. The investigation was then narrowed down to the design and architecture of big data systems.

Relational database systems operate based on four principles: atomicity, consistency, isolation, and durability [1, pp. 111–112], [30, Ch. 15]. These four principles are referred to as the ACID properties. Each of the ACID properties will be discussed subsequently.

Atomicity. Relational database systems use a transaction as the fundamental unit of work [1, pp. 111–112], [30, pp. 447–449]. Atomicity guarantees that transactions are atomic. A transaction is atomic, i.e. indivisible, if it completes either all or none of its work. Atomicity means that a database would always be in a consistent state. This consistency is extremely important to ensure data integrity. For example, consider situations such as power failures or system crashes. In these situations, it is certain that data would not be corrupted or "damaged" through incomplete transactions. Any transactions that are completed before these situations can be undone, or the missing transactions can be performed.

Consistency. This property guarantees that the completion of a transaction will leave the database in a new consistent state [1, p. 112], [30, p. 450]. If the transaction has not been completed successfully, the database would remain in its old consistent state. A database is in a consistent state if all data is valid or conformal. Data is valid if it complies with all the database constraints and rules. Thus, consistency is achieved by applying the other ACID properties.

Isolation. For performance reasons, databases execute many transactions concurrently [30, p. 450]. From a user's point of view, a transaction should be completed without the interference of other transactions. At the same time, consistency must be ensured. Isolation allows transactions to be executed simultaneously with the same result as if they were executed sequentially [1, p. 112], [30, p. 450]. Each transaction is thus executed in isolation as though it were the only transaction. Intermediate results of transactions are hidden from other transactions and users until all the transactions are complete.

Durability. Durability guarantees that the results of all successful transactions are permanently stored in the database [1, p. 112], [30, p. 450]. If there is a database crash or failure, the changes of completed

transactions will remain in the database. Therefore, resumption of the system would show a consistent database state.

The ACID properties came at the cost of decreased scalability and reduced support for large volumes of data. These shortcomings were identified when the ACID properties were extended to large distributed databases. It is due to the strong ACID guarantees that traditional RDBMSs are only vertically scalable [1, pp. 39, 112], [34, p. 3], [36, p. 24]. NoSQL and newer systems had to be designed with other characteristics. Consequently, the CAP theorem was created to define what is possible in distributed systems.

CAP stands for strong consistency, high availability, and partition-resilience [1, pp. 112–113], [33, Sec. 4.2], [34, Ch. 3], [36]. The theorem was first published by Fox and Brewer [37]. Brewer discussed the CAP theorem in 2000 [38], which gave it the alternative name Brewer's theorem. The CAP theorem describes the trade-off between strong consistency, high availability, and partition-resilience in distributed systems. Originally, one could choose at most two of these properties to be satisfied in distributed systems.

Strong consistency. Multiple nodes in a cluster may have different versions of the same data set. Strong consistency, however, states that all nodes will have the same data set. Thus, strong consistency is equivalent to a single global data set. Therefore, different clients connecting to different cluster nodes will see the same data at the same time.

High availability. Replication or redundancy is used to provide high availability. Data is highly available if one can still access, i.e. read or write, it despite node failures. A client would, therefore, experience no loss of service when node failures or cluster-related faults occur.

Partition-resilience. A cluster is created by interconnecting computers via a network infrastructure. When problems arise in the connection between cluster nodes, communication between nodes could be lost. The cluster and network are then said to be partitioned. Logically, one could think of the cluster being divided into different sets of nodes. Each set has communication between the members of the set but not to other sets. A system that continues to operate despite the existence of partitions is partition-resilient or partition-tolerant.

The CAP theorem is graphically illustrated in Figure 6. The figure shows that one can only choose consistency and availability (CA); availability and partition-resilience (AP); or consistency and partition-resilience (CP). This choice of two characteristics has been the accepted way of thinking about the CAP theorem. Advancement of technology has, however, changed the rules of the CAP theorem.

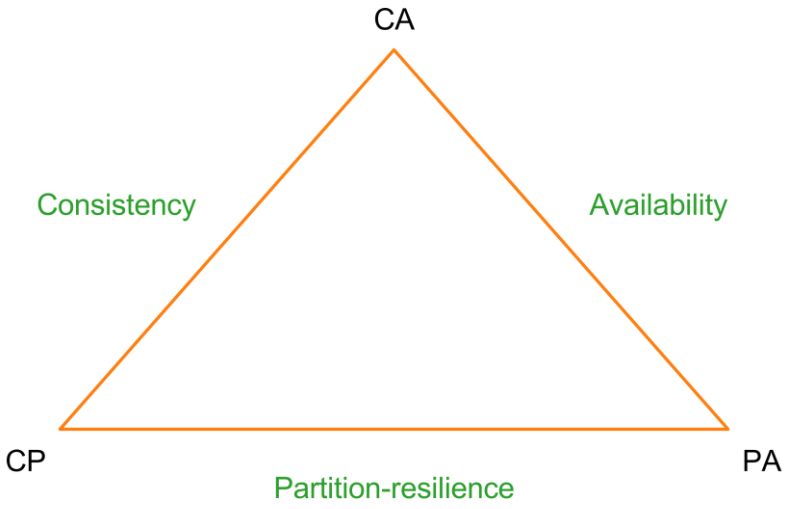


Figure 6: CAP theorem trade-off choices

In their original article, Fox and Brewer already mentioned a Weak CAP theorem [37]. With it, they were hinting at the possibility of possessing some degree of all the properties. They did, however, not exactly define the Weak CAP theorem.

Is the CAP theorem void? No. Brewer revisited the CAP theorem 12 years after its creation to address this very question [36]. The CAP theorem only prevents highly available and strongly consistent systems in the presence of network partitions. Brewer conceded that network partitions are rare. There is, therefore, no more choice of two out of three properties. The reason being that a system without partitions can support both consistency and availability. When partitions do occur, the system designer has a choice between consistency and availability. Herein lies the fundamental difference between NoSQL systems and RDBMSs. An ACID system has, in some way, chosen consistency first and availability second. NoSQL systems are precisely the opposite with availability first.

NoSQL systems are characterised by the BASE properties [1, p. 113], [34, p. 4], [36, pp. 23–25]. They are basic availability, soft state, and eventual consistency. Incidentally, BASE was first proposed by Fox, Brewer, and their colleagues [39].

Basic availability. Basic availability corresponds to the CAP theorem's availability. All parts of a system may not always be available. To the user, however, it appears as if the system is always available. As mentioned previously, this availability is guaranteed first, which is then followed by consistency.

Soft state. In the original article, Fox et al. defined soft state to be one that can easily be recreated [39]. In this state, data is not durable. The not-durable part seems problematic because data storage should be durable. Some explanation was, however, given by Brewer in [36]. He states that during recovery from partitions, durable operations in the ACID sense can be undone. This rollback is required if the operation

was invalid, such as withdrawing money from an empty bank account. Therefore, soft state means that the system state changes, unlike ACID that always has a consistent and durable state.

Eventual consistency. Eventual consistency is a weaker form of the CAP consistency. Besides the focus on availability, there are physical limits to consistency. In a distributed context, there is a time difference between the writing of data and its reception by every node. This time difference is called latency and implies that there is a time of inconsistency. Eventually, all nodes of the system will be in the same consistent state. Until that time, however, the system's state changes, hence its soft state. It is, therefore, possible for a user to read data that is out of date, i.e. stale.

It is important to remember that ACID and BASE systems are on the extremes of a spectrum [36, p. 24], [37, pp. 174–175], [39, Sec. 5.3]. One can interpret this spectrum informally as follows: The spectrum is a continuous scale from ACID to BASE. A system can be at any point on this scale. The point where the system is, represents the degree to which the CAP properties are present. For example, a system can be 80% available and 20% consistent. One can take this notion further to the subsystem level. Different subsystems within a system can implement CAP to a different degree. This different alignment is useful when different data requires different guarantees. For example, online orders must be consistent, whereas shopping cart information should be available.

The rest of this section will investigate various big data systems. A variety of systems were found in various contexts. In [40], Lee et al. developed a flexible and scalable repository for use in big sensor data. Their system archives different types of sensor data with the use of lossy coding, i.e. compression of data with a loss in accuracy. Data ageing was also introduced, which means that the older the sensor data, the less space it occupies. Therefore, the older the data, the lower its accuracy would be. The repository they created did not use a database. Instead, raw sensor data blocks were stored in specific clusters. Each succeeding cluster would store increasingly less accurate data. Data blocks would get moved to a cluster based on its age, i.e. temporal attribute.

Big data has found its way to wireless networks. One of the uses of big data in wireless networks is caching, as was shown by Zeydan et al. [41]. They used big data analytics and machine learning to proactively cache content in fifth generation (5G) wireless networks. The architecture that they proposed cached content based on its popularity to decrease traffic over the core of the wireless network. While their solution does not use data store software, it shows a viable strategy for increasing user satisfaction and decreasing the response times of user requests.

Various data management systems use cloud services for storage and processing. Cloud services are essentially resources and services that are obtained over the internet [42], [43]. In general, cloud services are meant to refer to public cloud services. They are public in the sense that a paying customer can use their services. Public cloud providers own the infrastructure that is leased to cloud consumers. On the other side,

private clouds are also possible but are for the exclusive use of organisations. Additional information can be found in [42] and [43].

A cloud data management system was developed by Jiang et al. [44]. Their system provides the ability for an enterprise to create a private cloud storage system. The system can also interact with public cloud providers to expand the enterprise's private cloud. Other features of their system include multitenancy, load balancing, and the usage of both MySQL and MongoDB. Multitenancy, with regard to software, means that one software instance is executed on a server, and this instance serves multiple clients.

To support multitenancy and load balancing, Jiang et al. used various caching and buffering techniques. Apache ActiveMQ was used as a buffer to queue user requests. This buffer prevented their server from being overloaded with too many concurrent requests. Furthermore, a first-move first-failure strategy was used to randomly return a direct failure response on some of the user requests. Again, this strategy alleviated a situation when too many requests were received simultaneously. Using the first-move first-failure strategy did, however, require a retry mechanism so that failed requests could obtain service.

The IoT field is also experiencing big data challenges. Kang et al. addressed the IoT big data challenges by designing and implementing a MongoDB-based repository [45]. Their research was limited to radio-frequency identification (RFID)/sensor data, which is rarely updated once it has been saved. The design of their system used an open-source Electronic Product Code Information Services (EPCIS) database schema.

To design their system, Kang et al. modelled the EPCIS schema into a MongoDB data model. The authors mentioned two design patterns for MongoDB: embedding and referencing. These design patterns are informally discussed in [46]–[48]. Embedding was used to create their data model because of the rare data updates.

After Kang et al. developed their data repository, it was tested and compared with a similar MySQL database. Their results showed that their MongoDB system achieved better query performance in two of three tests. In the third test, the MySQL system obtained better query performance.

The final two contributions by Kang et al. were, firstly, the choice of MongoDB's sharding key. Secondly, they investigated the impact of using more than one shard. They found that the sharding key must be chosen such that small chunks are created. Over time, these chunks will be distributed evenly over the shards. Further tests determined the impact of multiple shards on their MongoDB system. Their tests revealed that increasing the number of shards increases performance. This performance increase is of interest when the number of simultaneous clients is many, and the data search space is large.

All the systems that were discussed previously were created for specific contexts. It is clear that there is no general one-size-fits-all system that was created. Instead, each system used various technologies and was created for a specific purpose. A process or architecture was not provided for the creation of the previous

systems. Therefore, the study focused firstly on design and creation followed by implementations, as will be discussed further.

Design processes for big data are not as established as those for traditional relational databases. Banerjee et al. [49] specifically addressed the lack of conceptual and logical data modelling. Furthermore, they investigated the design of big data systems.

Banerjee et al. used the Graph Object-oriented Semi-structured Data Model (GOOSSDM) [50]. They created transformation rules that transform a GOOSSDM conceptual data model into a JSON logical data model. The logical data model was transformed into a physical data model by implementing the JSON structure in MongoDB. Banerjee et al. showed the shortcomings of GOOSSDM in their research, chief of which is its inability to represent schemaless data. Their research succeeded in designing and implementing a semi-structured schema-based system. There is, however, still a need for a schemaless big data design (BDD) and implementation.

Design principles for the architecture of big data are given by Begoli and Horey [51]. The design principles that they gave are based on the authors' experience working at the Oak Ridge National Laboratory. The projects they describe focus on the knowledge discovery, i.e. analysis, of big data. Despite this, they take a holistic view. The authors recognise that to perform data analysis, one needs to consider storage, processing, and results. The three main principles they identified are as follows:

- A variety of analysis methods should be available. This variety implies that software should be available to support statistical analysis, data mining, machine learning, and visual analysis. In the authors' system architecture, they provided analytical tools such as R, SAS, and Python for statisticians. SQL was provided for analysts. Finally, MADLib and Mahout were available for machine learning and data mining.
- Various data processing and storage systems should be employed to ensure flexibility. This flexibility is only available when not just one system is used to provide all the required functions. Instead, multiple systems should be used with each system specialising in a particular purpose. For example, using a graph database, relational database, column-oriented database, and key-value store for different types of data set.
- The results or end products should be easily accessible to users. Here, "easily" refers to the use of web interfaces and web services to deliver data. Accessibility can also be improved by using popular open standards, such as representational state transfer (REST), open database connectivity (ODBC)/Java database connectivity (JDBC), and JSON.

Chen et al. provide a practical methodology for designing big data analytics systems [52]. Their Architecture-centric Agile Big Data Analytics (AABA) methodology was verified and developed using 10 case studies. The AABA methodology consists of two principal parts: BDD and architecture-supported

agile analytics with automation. With the AABA methodology, the first focus is on creating a software architecture through BDD.

Chen et al. described why an architecture design is of the utmost importance for complex big data systems. The reason is that attributes such as performance, modifiability, and security are aspects of a system's architecture. This fact is also accepted in the industry, which is why there are usually architecture-centric approaches.

Looking back on traditional data systems, Chen et al. made the following observations: Traditional data system design is well established and has placed little to no emphasis on architecture design. The reason is that traditional data system architecture is rigid. This rigidity leads to a wealth of industry knowledge and accepted design patterns.

Chen et al. noted that big data systems require an architecture design different from the established designs. Furthermore, the architecture of a big data system will determine two things. The first is if it could support fast data from new uncleansed data sources. The second is, its ability to integrate old systems.

The second focus of the AABA methodology is agile analytics development with development and operations (DevOps) practices [53]–[55]. This part focuses on employing agile practices in the development of the system. During the development of their system, Chen et al. used analytical techniques to monitor and evaluate the developing system [52]. They noted that analytics could aid in the discovery of system aspects, such as faults or efficiency improvements. These aspects might not have been visible otherwise, thus aiding in system development. DevOps practices that were mentioned include activities such as continuous build integration, automatic test execution, and operational support. These activities allow for continuous improvement and continuous delivery of the system.

In the development of the AABA methodology, Chen et al. investigated three big data reference architectures: the data lake, polyglot persistence, and Lambda architectures. Chen et al. favoured the Lambda architecture in most of the systems that were designed. The three big data reference architectures were investigated and are further discussed subsequently.

2.3.1 Lambda architecture

The Lambda architecture was created by Marz and Warren [10] as a new paradigm for big data. It was developed from first principles with the objectives of scalability and simplicity. The overriding principle is using a variety of systems. Together, the systems should enable real-time computation of arbitrary functions on a given data set.

The essence of the Lambda architecture is shown in Figure 7.

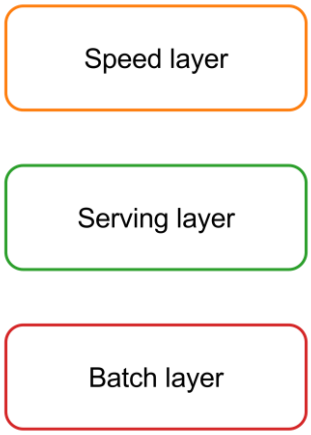


Figure 7: Lambda architecture layers (recreated from [10, Sec. 1.7])

From Figure 7, there are three layers: the batch, serving, and speed layers. The organisation of the three layers is deliberate. The top layers employ the functionality provided by the layers below. Each layer provides a specific part of the system's attributes, thereby implicitly reducing complexity.

The batch layer continuously creates batch views by precomputing a function on all the data. Consequently, it stores the master data set and updates it when necessary. The original Lambda architecture requires the batch layer to perform two functions. The first is to store an unmodifiable ever-growing master data set. The second is to compute any function on the master data set. It would, however, be unwise to have an immutable master data set when there are frequent changes. The reason being that frequent changes would lead to the creation of an entirely new master data set with each change.

The serving layer consists of a distributed database that stores the batch views created by the batch layer. By storing the batch views, the serving layer can perform random or sequential reads on the views. Upper layers can, therefore, use the serving layer to execute queries on the batch views. The serving layer must be able to support batch updates, but not random writes. This distinction is important because random writes increase the complexity of databases. Thus, the serving layer can use simplified database software. Batch updates are required because as the batch layer creates updated views, the serving layer needs to update them.

The speed layer is required because as the batch layer is precomputing, more data may arrive. Consequently, the newly arrived data will only be used in the next precomputation stage. Therefore, the speed layer minimises the time between the reception of new data and its availability to users. In essence, the speed layer is similar to the batch layer: it creates views from incoming data. The difference, however, is that the speed layer uses recent data. In contrast, the batch layer uses all historical data.

The speed layer does not use all recent data at once. Instead, it updates real-time views as it receives data, which is known as incremental computation. Incremental computation keeps the latency of new data usage as small as possible. Eventually, the data used by the speed layer will be visible in the batch views. This

eventuality means that the corresponding part of the real-time view can be discarded. The ephemeral nature of the real-time views decreases the complexity of the speed layer. The speed layer is, however, the most complex of the Lambda architecture layers.

To summarise, the most complex layer is the top (speed) layer. The least complex is the bottom (batch) layer. This configuration provides the Lambda architecture with the property of complexity isolation. Complexity isolation means the results of the most complex layer are temporary. The advantage of complexity isolation is if anything were to happen with the most complex layer, its results could be discarded. Discarding the most complex layer's data causes a rapid return of full service.

Kiran et al. used the Lambda architecture to implement a cost-effective data processing system on a cloud service platform [56]. Their system was implemented within the context of smart cities, which are analogous to large sensor networks. To realise a cost-effective system, the authors used the elasticity of cloud services, multitenancy, and pay-as-you-use cost model. Their choice of cloud service provider was Amazon Web Services (AWS). Each part of the system was implemented using various AWS products. Thus, there was no usage of third-party software such as Apache Hive or MongoDB, which were considered.

The system of Kiran et al. was validated with a case study on the Energy Sciences Network router production network. Over a three-month period, they were able to reduce the monthly cost by more than half. The cost reduction was only possible due to the flexibility of cloud services. They achieved this cost reduction by dynamically attaining different machines to optimise performance and cost.

In the spirit of the Lambda architecture, Ranjan [57] provided a high-level architecture design for big data processing services. This architecture consists of three layers: applications, a big data analytics ecosystem, and a resource layer. The big data analytics ecosystem consists of its own three layers: a data ingestion, data analytics, and data storage layer. Here the focus is on the big data analytics ecosystem, which relates to the subject of this study. The functions of each layer within the big data analytics ecosystem are as follows:

- Data ingestion: Is implemented by a queueing system, and it is the central collection point of data from all sources.
- Data analytics: Can consist of multiple systems that process and analyse data. The data can originate from the data ingestion or data storage layers. Ranjan gives three examples of processing and analysis systems: stream processing, batch processing, and data mining.
- Data storage: Consists of the data stores that manage and store final and intermediate data sets.

Ranjan further elaborated on big data stream processing and its deployment to cloud services. Ranjan states two technological advancements that have enabled big data stream processing. The first advancement is

data mining and application programming frameworks. These frameworks provide a basis that enables the creation of big data analytics applications. Four types of framework were mentioned:

- Large-scale data mining: This framework enables the parallel mining of data sets and application of data mining algorithms. Examples of such frameworks are Apache Mahout, Apache PredictionIO, Weka, and Apache Spark's MLlib. Most of these example frameworks use machine learning to perform data mining.
- Distributed message queueing: It is a system that queues real-time data streams. This queueing is required because modern systems, especially affordable ones, do not function perfectly in real time. Furthermore, it is usually more efficient to processes small batches or windows of data than single data points. Some examples of these systems are Apache Kafka, RabbitMQ, Redis, and Apache ActiveMQ.
- Data application programming: This type of framework allows for big data processing applications to be developed. These applications require data and tasks to be distributed over a cluster, with tasks processing in parallel. Tasks also require the ability to communicate, i.e. interprocess communication, and need rescheduling after faults. The framework automatically handles such management intricacies over distributed systems. Two examples of these frameworks are Apache Hadoop and Apache Storm.
- NoSQL database frameworks: With the limitations of relational database systems, a scalable, highly available, and less structured solution was needed for the big data era. A NoSQL data store is this solution, with multitudes of different types being available today. Some examples of NoSQL data stores include MongoDB, Apache Hive, Riak, Hypertable, and Amazon Dynamo.

According to Ranjan, the second advancement that enabled big data stream processing is data centre cloud services. Data centre cloud services provide flexible on-demand provisioning of computing resources, such as memory, central processing units (CPUs), graphics processing units (GPUs), and storage. These clouds are the most suitable for big data processing as one can scale any processing task to the desired level. By scaling processing tasks, one can realise the desired level of service quality. There are, however, challenges when deploying a dynamic big data system to a cloud service. Some of these challenges are orchestrating cloud resources, automatic deployment, intelligent monitoring, and quality of service assurance.

2.3.2 Polyglot persistence

Polyglot persistence was first described by Sadalage and Fowler, as discussed in [9], [11] and [58]. The need for polyglot persistence arose due to the dominance of relational databases. One must remember that relational databases dominated the data management landscape until the early 2000s. Enterprises tried to satisfy all their data storage needs, also known as persistence, with a single type of database. Even though enterprise applications interact with various data types, all data was forced into a relational schema. Sadalage and Fowler consequently derived the polyglot persistence concept, based on polyglot programming [11].

Polyglot programming refers to the usage of different programming languages when developing an application. Therefore, applications could be created using multiple programming languages. In the same way, data management systems could use multiple data stores. Furthermore, polyglot persistence requires one to use data stores, which are suited to the specific storage requirements. Sadalage and Fowler predicted increased performance when using different data stores for different purposes. This prediction is a natural result of using a data store for its intended and specific purpose. In contrast, using one data store for all one's needs would be ineffective.

Another advantage of polyglot persistence is avoiding impedance mismatch. Impedance mismatch is a phenomenon originally related to relational databases. It describes a situation where the structure of data in an application differs from its stored structure. Application data is not always used in a relational form as stored in an RDBMS. Therefore, impedance mismatch results in additional translation to and from the application and relational data structures. Polyglot persistence avoids impedance mismatch due to its affinity toward NoSQL data stores.

Kaur and Rani designed a polyglot persistence healthcare information system, named PolyglotHIS [59], [60]. They recognised the need for healthcare information systems to store data in its most complete form, thereby ensuring greater insights and discoveries. As Kaur and Rani found, however, organisations tend to mould databases for particular data. This specific moulding leads to multiple disparate databases with complicated or no communication between them. What was discussed here corresponds to the statements made in Section 1.1 regarding data silos. Thus, confirming the need for this study.

The polyglot persistence concept provides an alternative to multiple disparate databases. Kaur and Rani, therefore, decided upon polyglot persistence to develop PolyglotHIS. Their system addressed the need for storing all types of healthcare information. Examples include hospital payroll, active bills, billing history, laboratory reports, and blood relations among patients. Storage of the healthcare information was provided by PostgreSQL, MongoDB and Neo4j.

In compliance with polyglot persistence, PolyglotHIS unified all the data stores' data through a query administrator. The query administrator provided a REST application programming interface (API). This API was presented to users in the form of a query interface.

The advantage of PolyglotHIS, and indeed any polyglot persistence system, is undoubtedly the unification of heterogeneous data stores. Unifying heterogeneous data stores further increases user satisfaction. User satisfaction is increased by providing one interface with access to all data. Technical details, such as different query languages, inter-data store result merging and data store locations, are all hidden from the user.

The disadvantage of polyglot persistent systems is the increased complexity of interfacing and connecting different data stores. Kaur and Rani have, however, simplified the programming model of their system

through their architecture design decision. Two options were available for architectural design: global schema and query translation.

The global schema design creates a record of the content of all the data stores. A user then queries this record. The challenge of the global schema design is incorporating changes in the data sources. It must be able to reconfigure itself as data sources are updated and removed. Kaur and Rani found that the global schema design was not scalable and too strenuous to implement. Consequently, the authors chose the query translation design due to its scalability and flexibility.

A final aspect is that PolyglotHIS makes innovative use of translators to abstract the underlying data stores. The result is that new data stores can be added easily by only creating a new translator. This support for new data stores results in a modular, i.e. flexible, system. Consequently, the system is not tied to specific data stores. This data store independence means future data stores, with possibly better features or performance, are effortless to support.

To validate PolyglotHIS, the creators compared its performance with the performance of each of its data stores. What the results showed was the degree of overhead or performance penalty incurred by PolyglotHIS. As is to be expected, there was some difference between the performance of PolyglotHIS and its separate data stores. However, as the data set size increased, the performance level of PolyglotHIS converged to that of its data stores.

Prasad and Avinash applied the polyglot persistence concept to energy data management systems [61]. Their focus was on the energy utility industry where smart grids and meters are the main data sources. In the design of their system, they focused on obtaining both ACID and CAP characteristics. An RDBMS provided the ACID characteristics. For the CAP characteristics, they produced a list of data stores. For each data store, they noted the two CAP properties that each favoured.

With their data store list, they could choose a data store for each system module. Each module was satisfied according to the CAP support it required. For example, a module that required consistency and partition-tolerance could be implemented with Bigtable, HBase, or MongoDB. Another module that required consistency and availability could, for example, be implemented with MySQL, PostgreSQL, DB2, or Oracle.

Prasad and Avinash designed their system with the following software: ACID was provided by MySQL. MongoDB was used for consistency and partition-tolerance of customer profiles. HBase was also used for consistency and partition-tolerance to store current and historical bills. Real-time smart meter data was stored in HBase. HBase also performed analytical processing on its data sets. The output of the analytical processing was stored in OpenTSDB. Prasad and Avinash's system was developed for a specific organisation. However, they stated the possibility of extending their model to any organisation within the electrical energy supply industry.

Another polyglot persistence healthcare information system was implemented by Prasad and Sha in [62]. The methodology they followed was similar to Kaur and Rani's [59], [60], as was explained previously. The system that Prasad and Sha implemented used more than twice the number of data stores used by Kaur and Rani.

Prasad and Sha chose a MySQL system to store transactional data [62]. Maintenance and user session data was stored with the use of a key-value store, namely Redis. Recommendations, which are mainly relational data, was stored in the Neo4j graph data store. Patient profile data was stored by CouchDB, which is a document store. Logs of event data were stored using a columnar data store, namely, Cassandra. Inventory data was also stored in a key-value store, but here Riak was used instead of Redis. Finally, pharmaceutical product catalogues were stored in MongoDB.

Prasad and Sha provided reasons for choosing each of the specific data stores. There was, however, no mention of the performance of their completed system. Their work highlights a possible concern with polyglot persistence, which is the number of data stores to use. It is clear that polyglot persistence recommends a data store for each specific purpose. It is, however, unclear how far this principle must be taken. It can be concluded that the greater the number of data stores, the greater the complexity of the system. Therefore, there is a danger of designing a system that is unnecessarily complex.

A final concern regarding polyglot systems is the ability to add new data stores with little effort. This concern must already be addressed during the design process. At the very least, this would entail a system that does not make rigid decisions on the data stores that will be used. Prasad and Sha have addressed this concern by wrapping each of the data stores in a web service. These web services are implemented either with REST or simple object access protocol (SOAP). By using web services to expose the functionality of data stores, the authors have abstracted all data stores. The advantage is that there are no assumptions regarding the data stores.

2.3.3 Data lake

The data lake architecture was created by James Dixon of the Pentaho Corporation, as discussed in [63] and [64]. It was defined in contrast to data warehouses and data marts. To explain these terms, Dixon used an analogy of bottled water. A data mart or data warehouse would be a store containing bottles of water. The bottles of water have been prepared and structured for effortless consumption. In contrast, the data lake would be in a more natural state such as a lake. The lake is filled with content (water) by one incoming stream. Various users can make use of the lake by diving in, examining it, and taking samples.

In database terminology, the data warehouse can also refer to as a decision-support data warehouse. The data warehouse would extract, transfer, and load (ETL) data into it from an operational database. Therefore,

there is a time difference between the storage of data in an operational database and its appearance in a data warehouse.

A data warehouse is typically created for a specific group of users who benefit economically from its existence. This economic motivation is important due to the data warehouse's high cost. Furthermore, a data warehouse is designed for prestructured queries; thereby helping its users focus on specific objectives. The data within a data warehouse is, in general, organised as a relational database. Therefore, a data warehouse or data mart does not subscribe to big data's variety and NoSQL's scalability characteristics.

The data lake architecture, as one would expect, was created for the big data era. Data within a data lake is stored in a raw, i.e. unprocessed, form. Additionally, all the data of a source is stored within the lake, in contrast to only selected data. The reason for the raw data form is that it allows one to perform arbitrary aggregations at a later stage. Coupled with the fact that all the data is available, provides one with complete information certainty. This certainty is of fundamental importance because if one aggregates data using pre-aggregated data, incorrect results might be obtained. Consequently, if one makes decisions or assumptions based on incorrect results, the consequences could be dire. The additional advantages of the data lake are as follows:

- Arbitrary queries that are defined when needed. A user can create various queries and is not limited to predefined queries.
- User-defined focus, i.e. there is no predefined focus. The user can use the data in the lake for any purpose.
- Open to any user. Users from any group can use the data lake services because they are not targeted at a specific user group.
- Real-time data usability. Data stored within the data lake is directly usable and does not require ETL. Therefore, data availability is more timely.
- Structured, unstructured, and partially structured data are supported because of the use of a NoSQL data store.

There is contradicting literature regarding data lake architecture. Some contradictions were addressed by James Dixon in [65]. In particular, some sources, such as [66] and [67], define a data lake as having many sources. This distinction is a minor change, but it creates additional challenges over a single-source data lake. Henceforth, a data lake with one source will be referred to as a data lake. The modified data lake will be referred to as a multisource data lake.

There are some potential problems with multisource data lakes as discussed in [66] and [67]. One problem is that care should be taken when storing a source's data. One would not want to merely store all data for the chance that it may be used. If the data is not used, time, storage, and effort would have been wasted. Thus, a clear understanding of the eventual data usage is needed to prevent the unnecessary storage of data.

Both single-source and multisource data lakes could use disparate data sources. These data sources may duplicate data, increase data redundancy (in a negative way), and lead to inconsistent data. Consequently, one would have to implement a strategy to integrate and manage the data sources. One possible strategy to resolve disparate data sources is master data management (MDM). Another strategy, as discussed by O'Leary, is to use artificial intelligence and crowdsourcing [66].

An important challenge, not only for data lakes and warehouses, is data silos [67], [68]. Data silos are somewhat related to disparate data sources within data lakes. It is, however, also true that unmanaged homogeneous data sources could result in data silos. Data silos were discussed in Section 1.1. Here a second confirmation is given for the need of the study.

As alluded to above, MDM is meant to reconcile data silos [1, p. 33]. Only vital core business data is part of the master data set [1, pp. 127–140]. The goal of MDM is to create a single source of truth for master data. MDM is closely related to data quality. For data to become part of the master data set, it must undergo a strictly governed process. This process contains tasks such as collection, filtering, integration, consolidation, and quality assurance. Without such a strictly governed process, master data would not be unique, consistent, and trusted. MDM is one of the main focus points of this study. The focus on MDM led to using the word "centralise" in the proposed system name.

2.4 Services to present system functionality

For a system to be used by other systems and applications, it needs to provide an interface. In the software industry, the interface through which a system's functionality can be accessed is called an API. The focus in this section is not on how to implement an API, but rather on the techniques available to allow users to interact with a system's API. There are many techniques to expose system functionality [69]. To avoid irrelevant literature, however, only the technique that is used in most modern systems will be discussed [70]. The remainder of this section will discuss service-oriented architecture (SOA). Microservices and its relation to SOA will also be discussed.

2.4.1 Service-oriented architecture

With SOA, the focus is on creating discrete software units. Each software unit provides a specific functionality [70]. Other applications and systems can reuse the software units to obtain the desired functionality. There are various advantages to using an SOA approach, but the most relevant are:

- Complex systems are modularised in a simplified way. This simplicity is achieved by integrating services that are independent of platform and technology.
- Loose coupling is promoted, which means that applications or systems are less interdependent.
- Standards-based interoperability between systems is ensured.

- Access to data is provided in a location-independent way. Therefore, devices such as smartphones and tablets can natively access data.
- Scalability and flexibility are improved because services can be developed more easily by integrating pre-existing applications.

Serrano et al. state that the simplest approach to implement an SOA is using web services [70]. They mention two main technologies that are used for web services: SOAP and REST. Simply stated, these two technologies define the rules that govern the exchange of information when accessing web services.

SOAP is an older protocol [70], [71]. It is more flexible regarding the various data transport protocols that it supports. The data format used by SOAP is XML. XML is more robust but slower to process than other data formats.

REST is a newer protocol than SOAP [70]. It supports various data formats including JSON and XML. The rules imposed by REST are less rigid, which results in its greater implementation flexibility. REST uses specific uniform resource locators (URLs) as its communication mechanism. Although REST is the most popular web service technology, SOAP was chosen due to its greater security. Greater security was required due to the privacy requirements of organisational data. If organisations store client data, then increased security is even more important. Therefore, it was of the utmost importance to ensure confidentiality and integrity when interacting with the web service.

The advantages of an SOA approach were the reason for choosing a web service for the system. The dominant factor was the fact that most systems and users can use web services. This versatility is due to the success of the internet, for which web services were first developed. In fact, most programming languages and applications now have some way of interacting with web services.

2.4.2 Microservices

The latest trend in software architecture is using a microservices pattern [72], [73]. Salah et al. depict microservices as the evolution of SOA [69]. Others such as Pautasso et al., however, classify microservices as a special approach to, or a subset of SOA [74]. In essence, microservices is a service-based architecture, and so too is SOA [75, Ch. 1].

Thönes [72] defines a microservice as a small application with a single responsibility. This small application can be scaled, tested, and deployed independently. As previously mentioned, SOA was defined in terms of software units. It is, therefore, clear that microservices share some of the characteristics of SOA.

To define the difference between SOA and microservices one could look to Richards [75, Ch. 3]. He states that microservices is an architecture where components share as little as possible. He contrasts this with SOA, which shares as much as possible. Another difference between SOA and microservices was

mentioned in [74] and [75]. SOA places business logic in a service bus. This service bus is in the centre of the system architecture. It connects consumers with the desired services. Microservices, however, place business logic within the microservice in what is known as an endpoint. Other services connect to this endpoint responsible for communication and integration.

The differences between microservices and SOA have been defined. Why microservices is frequently used has not been addressed. In [76], Singleton thoroughly described the trade-offs and typical usage of microservices. He stated that microservices are optimal when used to simplify large systems. Microservices simplify testing, increase the speed of releasing new software, and improve quality. A system's quality is improved by reusing proven services. Additionally, the reliability of a system is improved due to the redundancy and scalability of microservices. The increased reliability and scalability comes from the ability to deploy more instances of a microservice. By deploying more microservice instances, one alleviates service congestion.

A final advantage of microservices is reusing services to create many products [76]. For example, if a team of 20 developers create one monolithic system, they will have a single product. A monolithic system is a single large system, also known as a monolith. In such a system, all its functionalities are closely integrated and coupled [77, p. 94]. If, however, the developers divide into groups of five, each group could create a microservice. The four microservices could then be combined into possibly 15 different products.

The advantages of microservices are clear, but many challenges need to be addressed [76], [78], [79]. Some of these challenges are unique to microservices due to their distributed nature and independence. The challenges are:

- Additional code is required to establish communication between the microservices [76]. This additional code includes concepts such as service catalogues, and messaging and queueing software to discover microservice instances. This discovery of microservices is known as service discovery. Service discovery is required for other services and applications to communicate with the correct microservice instance.
- A system to monitor and control microservices is needed [76]. Such a system should be able to ensure service operation by handling microservices' errors. The need for a monitoring system comes from the multitude of microservices that must be operational for the conglomerate system to function [79]. If some system is not able to start or restart microservices, their scalability would be for nought.
- Choosing an appropriate data exchange format and messaging standard is of the utmost importance. Sill discussed various data formats and messaging standards in [78]. The choice of data exchange format and the messaging standard is a design decision. If this design decision is made incorrectly, it could lead to interoperability problems.

- The boundaries of microservices need to be decided [76], [79]. For example, one needs to choose when a new microservice is created as opposed to including functions in an existing microservice. A logical trade-off would be that too few microservices could lead to overly complex microservices. On the other side, too many microservices, while very simple, could lead to complex management and communication. There is, however, no methodology that can be applied when creating microservices, which was discussed by Linthicum [79].

2.4.3 Containers and orchestration

A final aspect of microservices that needs to be discussed is containers. Containers are analogous to shipping containers. To demonstrate the usefulness of containers, one needs to understand the usage of shipping containers. When storing content in a shipping container, one is assured that there will be no interaction between the stored content and another shipping container's content. One is not concerned with the details of the transportation of the shipping container. The reason being, it is guaranteed that any cargo ship can transport the shipping container. Finally, one will have confidence in the fact that the content that was stored in the shipping container, will be the content that is retrieved when opening the shipping container.

The characteristics above are retained by software containers, as discussed in various sources [79]–[81]. Note that henceforth software containers will be referred to as containers. As a shipping container isolates its content, so too does a container isolate software from the operating system. The software within the container is not dependent on or influenced by any software outside the container.

When creating and testing a container, the same environment is reproduced on a destination system. This environment reproduction means that the container is portable to any platform that can run containers. As a result of container portability, one can deploy containers to many systems simultaneously. Software within a container will function the same way on testing, development, and production systems – meaning it will be reproducible. Lastly, by using containers, one achieves extra security because of a container's isolation.

Based on the short description of containers, it is clear they are closely aligned with microservices. Many have, erroneously, considered containers to imply microservices [78], [79]. There are containerless ways of implementing microservices as discussed in [76] and [78]. Containers are, however, the most popular, if not the best, way of implementing microservices [78]–[80]. Consequently, it was decided that containers would be used to create the proposed system's microservices.

There are many container platforms as discussed by Pahl in [80]. Some examples are Docker, Rocket (also known as rkt), OpenVZ, and Linux Container (LXC). Current practices and popularity have, however, suggested that Docker should be used to create and run containers [80], [82]. The reason is that Docker

partly created and follows the Open Container Initiative (OCI) standard [78], [81]. The OCI is a set of open industry standards that define a container format and runtime [83]. Docker's use of OCI has led to its adoption and support by many large organisations such as IBM, Google, and Microsoft [82].

A possible alternative to Docker would have been Rocket. Rocket also supports the OCI standard [84]. In addition, its native container image format and runtime environment was created from another open specification, namely, App Container [85]. Thus, Rocket can run both App Container Images and Docker Images.

Using containers creates similar challenges as was discussed for microservices. Pahl [80] discusses these challenges in detail, but only one challenge is mentioned here. The challenge expands on the microservice challenge of a system to monitor and control microservices. In the context of containers, the challenge is container orchestration and clustering. Essentially, this challenge relates to the creation of a cluster and the deployment of containers on it.

A cluster is a group of hosts, which can either be physical computers or virtual machines (VMs) [42]. Typically, a cluster would comprise commodity hardware, in contrast to specialised computers and servers [11, p. 8]. Each host functions as a node in the cluster. The cluster manages all nodes together as if they were a pool of resources. Resources include memory, storage, and processing. In essence, a cluster provides an abstraction of computing resources giving the impression of a single virtual computer.

Container orchestration software aims to create a cluster. Literature often refers to orchestration software as Platform-as-a-Service (PaaS) software [80], [86]. Hence, orchestration software can be used to create a private PaaS cloud.

Pahl discussed the features and advantages of container orchestration [80]. The advantages are:

- Containers can be hosted easily in the cluster with secure communication between them.
- The ability to autoscale and load-balance is provided. Autoscaling means that more containers are executed when there is an increase in service usage. Load-balancing equalises the usage of each container. That is, each container will receive approximately the same amount of traffic. Without load-balancing, one container may be used too much, while others are not used at all.
- Service discovery and orchestration are provided in a scalable and distributed way. This scalability makes it easier for containers to be distributed and discovered.
- Microservices can be transferred easily and migrated between clusters.

The alternative to orchestration would be to construct a cluster manually. Doing so would, however, result in none of these advantages.

Pahl discussed some container orchestration software for consideration, namely, Kubernetes, Mesos, and Cloudify [80]. More products were mentioned, but they are paid-for services. Kratzke also discussed the

orchestration of containers and mentioned OpenShift and CoreOS [86]. CoreOS has, however, indicated that it would no longer support its orchestration software, called fleet [87]. Instead, CoreOS has created a project based on Kubernetes. It should be noted that of these projects, Cloudify is based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [80], [88], [89]. This standardisation is of interest when one requires a highly standardised solution.

Why the review of containers and orchestration? It has been shown that big data systems, such as the proposed system, are complex. A large amount of effort and time would go into creating a big data system. Therefore, one would like the system to have a long life expectancy and be sustainable. Otherwise, all effort and time would have been wasted. One does not want to create a big data system only to start replacing it after completion. For this reason, this section investigated more in-depth architectures and concepts for big data systems.

To achieve this goal of sustainability and life expectancy, one would need to create a flexible and moveable system. For example, a system should be as independent as possible from its hardware. Kratzke discussed a similar train of thought [86]. He discussed in depth how the application of a lightweight virtualisation cluster (LVC) approach prevents vendor lock-in. He focused on preventing vendor lock-in when deploying systems using cloud services, such as AWS and Google Cloud Platform. Despite Kratzke's focus on cloud services, his approach would apply to private, i.e. an organisation's own, hardware. Simply stated, Kratzke's LVC approach necessitates the use of containers and orchestration software.

Liu et al. also discussed the importance of containers and orchestration [90]. They wanted to create a computing platform for scientific workflows. This platform had to be easy to deploy on any cloud service. They achieved their objective by using Docker containers and orchestration software. Although their system was not a big data system, their requirements were similar. They achieved reproducibility, ease of installation, and increased utilisation by applying the container-based system approach. Additionally, their system was considered "general" in that it could run tasks other than scientific workflows.

Stillwell and Coutinho described, among other concepts, the application of containers and orchestration to the Hardware- and Network-Enhanced Software Systems for Cloud Computing (HARNESS) [91]. Docker containers, the OpenStack software project, and microservices were used to create HARNESS. The system had the broad objective of running applications by allocating various resources. In fact, HARNESS was a cloud service providing research-oriented resources in addition to the traditional processors, memory, and storage.

HARNESS is almost a generalisation of what is desired in this study's proposed system. However, it is almost because the system fulfils the functions provided by orchestration software. The extent to which the HARNESS systems uses Docker for its critical components is also unclear. The proposed system is at a

higher level of abstraction that makes use of a HARNESS type of system. Furthermore, Docker was also used in all the proposed system's critical components.

Cao et al. provided insights into the construction of a private cloud [92]. They addressed problems related to providing support for any software framework. They discussed the deployment of big data processing frameworks such as Apache Spark and Hadoop. The problems they addressed were related to the design challenges for the proposed system. Their system, Docklet, did not address all the aspects of the proposed system. Docklet was meant to provide virtual cluster environments to users. Consequently, their experience would only be useful at a lower level of the proposed system.

Cao et al. developed Docklet using containers. However, they created their system from lower level components. For example, instead of using container orchestration software, they created their own. Consequently, their system is somewhat dependent on Linux computers and the existence of certain software. Docklet could also prove difficult to move to cloud services if, for example, no cloud service provides their specialised environment.

Fazio et al. created a system that focuses on big data storage, similar to the proposed system [93]. Their system, however, used cloud services for its implementation. MongoDB and OpenStack's Swift project were used for document and object storage respectively. There was no design and implementation based on Docker and microservices. Therefore, the system of Fazio et al. is not a complete solution as was sought in this study.

A final system that was found using microservices is described by Le et al. [94]. Le et al. document their redesigning of the Nevada Research Data Center using a microservice architecture. Both the proposed system and their system are data systems. Further, this study also proposes the use of microservices, as did their study. The objective of Le et al. was to improve the scalability, flexibility, reliability, and maintainability of the previous monolithic system. These characteristics are the same that were sought for the proposed system, hence the use of microservices.

The differences between this study and that of Le et al. are as follows:

- Their system uses a traditional database system.
- Their system uses a REST web service for an API.
- Their design does not use containers.
- The application context was that of research into alternative energy sources and conservation of natural resources.

In their recommendations for future work, Le et al. stated the need to use big data solutions together with cloud services.

2.5 Big data software evaluation

The data management landscape is filled with many data storage systems. Some were mentioned in the previous sections, but even more are mentioned in [2, Ch. 2] and [33, Sec. 4.3.1]. New data storage systems are also developed continually. This section will evaluate some data storage systems for use in the proposed system. It is, however, impossible to evaluate all possible data storage systems. Therefore, the evaluation will be focused on software mentioned in literature and based on industry popularity [95].

The data storage systems were chosen based on two criteria. The first criterion was support for big data. Secondly, the software had to be open source or have a free version, which keeps costs down. Keeping costs down allows for experimentation as was done in Section 3.3. Consequently, commercial products such as those from Amazon were not considered.

Couchbase. Couchbase Server Community Edition [96] is a NoSQL document data store with a distributed architecture. It can be deployed as a document store, a key-value store, or a distributed cache. The query language it uses, N1QL, is based on SQL. There is a shared-nothing architecture, which prevents a single point of failure. High availability is provided through features such as replication and automatic failover. Various programming languages, such as C, C++, .Net, PHP, and Java, are supported. Security features such as the secure sockets layer (SSL) and TLS encryption, auditing, and role-based access control (RBAC) are only provided with the commercial product [97].

HBase. Apache HBase [98] is a distributed and scalable big data store. It supports very large tables, for example, tables with billions of rows and millions of columns. HBase is modelled on Google's Bigtable, making it a column-oriented data store. Features similar to RDBMSs are provided, such as strongly consistent reads and writes. It provides automatic failover and databases are automatically sharded. The native programming language to access HBase is Java. External APIs are provided for Thrift and REST, but the C/C++ client could not be located. Security mechanisms are also provided, such as Kerberos authentication, secure web access, and encryption [99].

MongoDB. MongoDB Community Server [100] is an open source NoSQL document data store. It uses a flexible data model for its documents, which enables dynamic schemas. It has some features of relational databases, such as secondary indexes and strong consistency. MongoDB can accommodate multisite database architectures. High availability is provided through automated failover and master-slave replication. Programming languages such as C, C++11, .Net, Java, and PHP are officially supported. MongoDB provides security measures such as TLS encryption and RBAC.

Cassandra. Apache Cassandra [101] is a distributed NoSQL data store that stores data as a partitioned row store [102]. A row store stores data in the same way a relational database system does. Although Cassandra is a row store, some literature sources use it as a column-oriented data store. The Cassandra Query Language (CQL), which is based on SQL, is used to access data. Cassandra scales linearly. For example, if five servers

can execute 10 000 transactions per second, then 10 servers will be able to execute double the number of transactions. It provides continuous availability and automatic data replication. The available security features are SSL/TLS encryption, client authentication, and authorisation. Some of the programming languages that are supported include Java, PHP, .Net, C++, and more.

Hive. Apache Hive [103] is a product built for the Hadoop framework. It is a data warehouse software that enables the management of large data sets. Hive uses an SQL-based language called Hive Query Language (HQL). HQL can be expanded with user-created instructions. Hive is not suited to OLTP. It is fault tolerant and loosely coupled to its input formats. Hive cannot be classified into one of the four NoSQL data stores. It imposes a schema on data when it is read. Data can be saved in various file formats such as row- or column-oriented formats. Hive provides only user authorisation and programmatic access in Java [104].

Impala. Apache Impala is being incubated, i.e. is under development, by the Apache Software Foundation. Impala is also software for Hadoop. It is similar to parallel RDBMSs with performance an order of magnitude faster than Hive. Impala uses HQL for data access; therefore, one can replace Hive with Impala. All the features of Hive apply to Impala due to their close integration. Impala is generally used for analytical queries and interactive exploration of data. Impala uses columnar storage to achieve the optimised performance of queries. Native Hadoop security is available along with Kerberos authentication. A Sentry module can be used to implement access control [105].

Redis. Redis [106] is an open source in-memory NoSQL data store and is owned by Redis Labs Inc. It can be used as a data store, a caching layer, or a message broker. When Redis is used as a data store, it operates as a key-value store. Redis provides high availability, partitioning, and fault tolerance. Scalability is provided through master-slave replication. The master-slave model of Redis prevents consistency guarantees, and writes may be lost forever. There are almost no security features, making Redis only usable within a trusted environment. Various programming languages are supported such as C, C#, Java, and PHP [107].

The data stores that were mentioned are summarised in Table 1. The features of each data store are given in the columns. The table allows for a quick comparison between the data stores. From the table, three data stores were identified for further evaluation, i.e. initial testing. The data stores that were chosen are MongoDB, Impala, and Couchbase.

Table 1: Comparison of data store system features

System name	Database type	Authentication	Authorisation	Connection security	Replication	Programming languages
Redis	key-value	single password	none	none	master-slave	C, C++, C#, PHP, and Java
Apache Impala	row-/column-oriented	Kerberos	none	none	master-slave	Java, Thrift and REST
Apache Hive	row-/column-oriented	user accounts	none	none	master-slave	Java, Thrift and REST
Apache Cassandra	row-/column-oriented	user accounts	user accounts and roles	TLS	approximately peer-to-peer	C++, C#, PHP, and Java
MongoDB Community Server	document	SCRAM-SHA-1 and X.509 certificates	user accounts and RBAC	TLS	master-slave	C, C++, .Net, PHP and Java
Apache HBase	column-oriented	SASL and Kerberos	access control lists	TLS (only for web access)	master-slave	Java (native) Thrift and REST (external)
Couchbase Server Community Edition	document	user accounts	none	none	master-slave	C, C++, .Net, PHP, and Java

2.6 Conclusion

Humanity has been storing data in the form of physical files since before computers. The introduction of computers in the late 1950s and early 1960s started digital data storage and processing. As technology evolved, it became possible to not only store and process data, but also to generate more data. The next evolution of data systems came in the form of database systems. With the introduction of database systems came an increase in the demand for information. This increased demand is mostly fuelled by the growth and internationalisation of organisations together with technological advancement. The first database system was the RDBMS.

As was discussed in Chapter 1, it is advantageous and profitable for organisations to collect and analyse data. Data management and analysis, however, require storage of the data on which it is performed. Herein lied the problem. The greatest part of the past 40 years has seen database systems dominating in terms of storing and managing data. In most cases, data was stored using an RDBMS. Data was forced into relational and strictly defined structures, which were not scalable for large data volumes, varieties, and velocities.

Relational databases store data as relations, which are two-dimensional tables. Each relation has rows, which are called records, and columns, which are called attributes. SQL was described as the language used to retrieve data from and write to relational databases. The final technicality discussed was the schema. Different schemas were discussed. However, simply stated, a schema describes the organisation and structure of data.

In the big data era, NoSQL data stores have emerged as a preferred solution for big data situations. These data stores are scalable and support unstructured and semi-structured data. The four types of NoSQL data stores were discussed, namely, key-value, document, column-oriented, and graph stores.

Key-value data stores persist data in the form of key and value pairs. Searches are performed on the keys, which must be unique. Document data stores are an extension of the key-value concept. A document can be thought of as a collection of key-value pairs. However, here values can contain complex data such as arrays of (sub)documents. Document stores also allow queries on values instead of just keys. Column-oriented data stores organise data with an emphasis on columns. They can support tables with many rows and columns, much more than relational databases. Graph stores use vertices and edges to create a data structure. Highly relational data is particularly suited to graph stores.

The guarantees that relational and NoSQL data stores provide were defined as ACID and BASE respectively. On a high level, relational databases place a greater emphasis on consistency. NoSQL data stores, however, tend to focus on availability.

Three big data system architectures were identified, namely, the Lambda architecture, polyglot persistence, and the data lake. The Lambda architecture was created to isolate complexity using three layers. These are

the batch, serving, and speed layers. Polyglot persistence advocated the use of different data stores for specific purposes. This contrasted with the one-size-fits-all mentality, which was usually followed with relational databases. The data lake concept focused on storing as much data in the rawest form possible. Having access to all data would allow for more detailed analysis and higher accuracies. Many systems using these architectures were investigated. This investigation yielded several possibilities for making system functionalities available (refer to Research Question 4).

The most popular method of making system functionalities available was found to be SOA. SOA makes use of software units with specific functions to construct a system. Most SOA systems used web services, either SOAP or REST, to expose the services. A further evolution of SOA was found in the form of microservices. Microservices take service-based architectures further by using small isolated applications with a single purpose.

Containers were found to possibly be the best way of implementing microservices. Similar to shipping containers, software containers provide isolation and portability to applications created with them. The container landscape is, however, dominated by Docker. Literature regarding the use of containers was studied, which indicated the necessity for container orchestration. From its name, container orchestration refers to the management and control, i.e. orchestration, of containers. Apache Mesos and Kubernetes, among others, were found to be possible orchestration software.

Overall, the systems found in the literature were not usable for this study's context. Some of the systems were used in another context, such as healthcare or sensor networks. Many of the systems used cloud services, which would be costly for small or medium organisations without big data experience. None of the systems used all three big data architectures, which are data lake, polyglot persistence, and the Lambda architecture. Few systems used SOA, microservices, and containers. Fewer of these systems addressed every aspect of big data, namely, storage, processing, and analysis. Consequently, the proposed system is novel in its use of SOA, microservices, and containers to achieve the various advantages that were discussed. It is unique in its use of all three big data architectures. Finally, the system addresses all aspects of big data.

The final section of this chapter investigated various NoSQL data stores. A table was created summarising the features and capabilities of each. Based on the summary, MongoDB, Apache Impala, and Couchbase were chosen for further evaluation. After the evaluation, a final choice of data store will be made.

CHAPTER 3 DATA MANAGEMENT SYSTEM DESIGN

3.1 Preamble

The literature of the previous section provided the knowledge base for the design of the proposed system. Section 3.2 describes the requirements analysis that was performed to obtain the requirements and specifications of the system. In Section 3.3, an initial test was performed on MongoDB, Impala, and Couchbase. The test results were used to make the final choice of data store. This test was done before design as the data store influenced the design of the system. Next, the system was created starting with the design activity, as discussed in Section 3.4. System design started from the abstract and moved into the physical. The result of this chapter was a functioning system to be implemented in a case study. Finally, this chapter is concluded in Section 3.5.

3.2 Requirements analysis

The requirements analysis process started by defining the user requirements for the system. The user requirements were defined as generally as possible. Design science, and software in general, requires an application context. The application context was the South African mining and industrial sectors, as described in Section 1.2. More specifically, the context was organisations that store data of companies in these sectors. For simplification, these types of organisation will be referred to as operational service companies.

The user requirements for a centralised data management system are as follows:

1. The system should be able to operate for extended periods of time with minimal downtime.
2. Data should be kept safe – even when the system experiences a fault.
3. Multiple systems should be able to integrate with the system. This integration allows the organisation to expand without being constrained by the proposed system.
4. The system should provide easy access to data for users. This requirement is similar to the previous requirement, except that this requirement focuses on the user.
5. A variety of data, including raw and processed data, should be stored in the system.
6. Additional computers should be added to the system to improve performance or support more users.
7. Different storage locations should be supported. Thereby, allowing the separation of unrelated or confidential data.
8. A system administrator must be able to control the access and operations of different users.
9. Communication with the system should be secure to protect the information that is sent and received.

From the user requirements, system specifications were created. The specifications that the system had to adhere to are as follows:

1. High availability should be the first focus of the system. Through high availability, data will be accessible even if some parts of the system fail.
2. The system should replicate data between multiple computers. Data replication will prevent data loss during system failures.
3. More than one external system should be able to integrate with the system through an API. The API will allow access to data within applications and other systems.
4. The system should provide a web service. This web service would allow any user with a web browser to connect to the system. Once connected, a user would be able to view stored data.
5. The system must support structured, unstructured, and semi-structured data. This support of all data types will increase the number of supported systems and allow for any data to be stored.
6. For the system to support more users, it must scale horizontally using commodity hardware.
7. The system should be able to create separate databases or repositories for different data. This feature would allow one to separate related and unrelated data. For example, one company's data will be stored in a location independent from another company's data.
8. User accounts or access control must be a native attribute of the system. This attribute would allow users to be limited to specific data and operations.
9. The system API should use encryption to protect the confidentiality of the information exchanged. Additionally, the integrity of information must be ensured along with user authentication.

3.3 Experimentation of database systems

MongoDB, Impala, Couchbase, and MySQL were tested on almost 3.7 million data points. MySQL was added for comparison between the NoSQL systems and an RDBMS. The data set consisted of 136 tags. A tag is a set of all measurement data for an entity. For example, a pump's power consumption is an entity, i.e. a tag. All the power consumption measurements for the pump is grouped as the tag's data. Each tag had at most 48 measurements or data points per day.

For each system, only the minimum number of settings was set to obtain correct operation. The systems were installed on the same VM with only one system being active at a time. Therefore, each system had the same resources when the test was performed. The first procedure that was completed was setting up the systems. This procedure was completed by following the steps below:

1. Install the software.
2. Load the test data into the system.
3. Verify that all test data was inserted and can be retrieved.

The final procedure consisted of multiple subtests. Each subtest was logically divided into four sets. Each set consisted of one or more queries, with the exact details given below.

- Set 1: Consisted of one query that computed a daily arithmetic mean for each tag over the entire data period.

- Set 2: Retrieved all attributes for eight different criteria. Each criterion was used in a separate query, which means that this set consisted of eight queries.
- Set 3: Retrieved only one attribute, namely, the measurement value, for eight different criteria.
- Set 4: Computed the arithmetic mean of the measurement value for 12 different criteria.

The steps to complete the final procedure were as follows:

1. Execute one query on the platform.
2. Record the completion time for the query in seconds.
3. Repeat Steps 1 and 2 for all the queries in each set.
4. Repeat Steps 1 to 3 for 100 iterations.
5. Repeat Steps 1 to 4 for each of the systems.

The final procedure was performed using Python scripts. Each system had a client driver in Python, whether created natively or by a third party. Therefore, each system was accessible through Python. Python was also used due to its high level of abstraction, which enabled the rapid creation of the test scripts.

This section will only show the results of four queries. All the query results are given in Appendix 1. The first query performed was the query of Set 1, and is shown in Figure 8. As stated previously, the query computed a daily arithmetic mean for each tag over all the measurement days. The figure shows the mean, worst- and best-case time for each system.

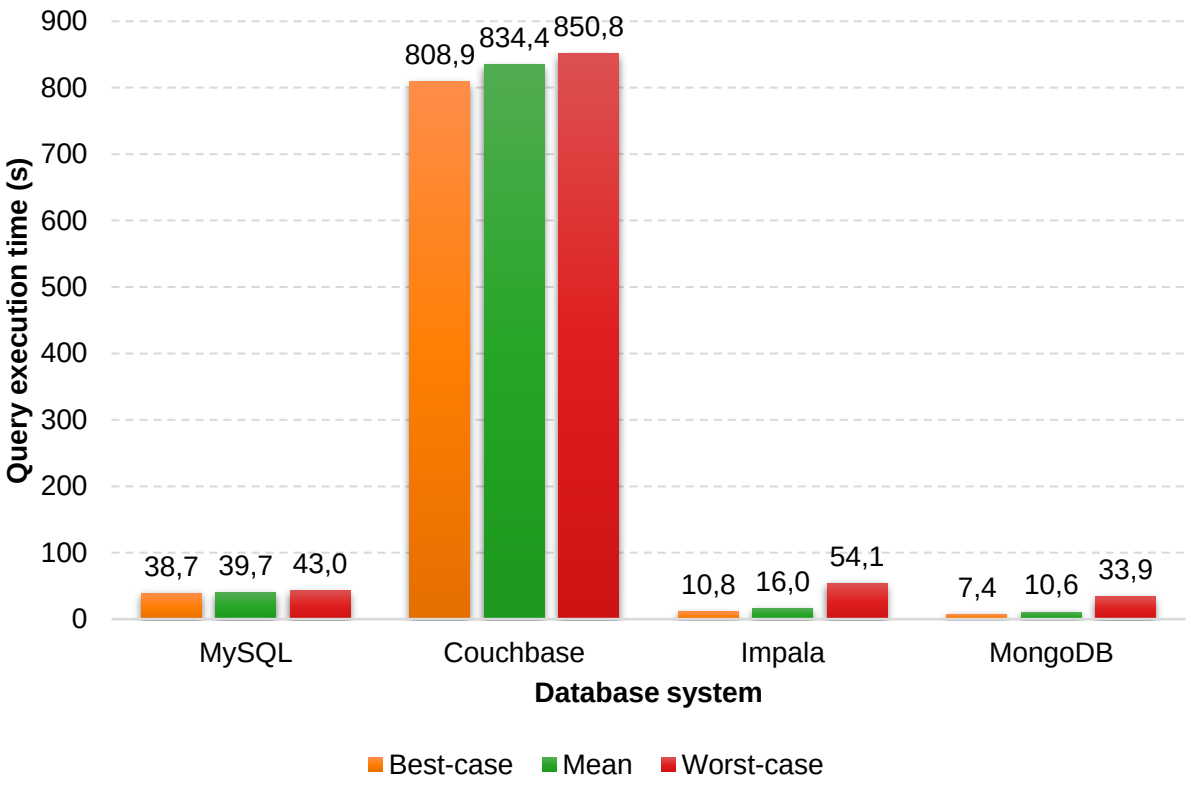


Figure 8: Comparison of the execution performances of the systems for Query 1

From Figure 8, it is evident that MongoDB performed slightly better than Impala. MongoDB was particularly better in the worst-case performance with a 37.3% improvement over Impala. Couchbase performed the worst of all the systems. MySQL was very consistent in its performance because of a small difference between its worst- and best-case performance. There were large variations in the performance of the NoSQL systems. On average, however, most of the NoSQL systems outperformed the RDBMS.

The second query performed was the second query of Set 2. This query retrieved all attributes for all data points. Simply stated, the query retrieved all the data. The mean, worst- and best-case performance of each system is illustrated in Figure 9.

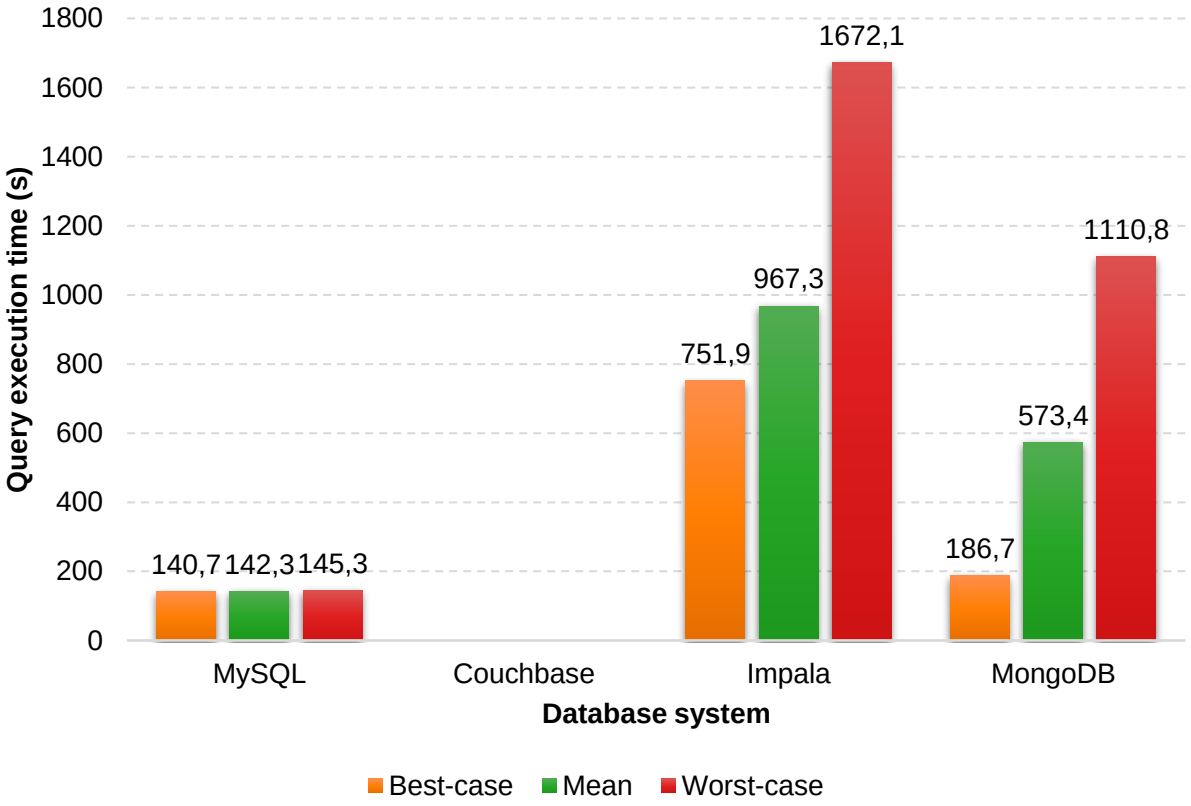


Figure 9: Comparison of the execution performances of the systems for Query 2

In Figure 9, there are no results for Couchbase because it was unable to perform the query. This test was the only time that Couchbase was unable to perform a query. Consequently, it might indicate a possible instability for certain types of query. MongoDB was again the best performer of the NoSQL systems. With this query, the maximum performance improvement of MongoDB over Impala was greater than the previous query. MongoDB was 75.1% faster than Impala during the best-case performance. MySQL had consistent performance, similar to the previous query. Here, however, the RDBMS outperformed the NoSQL systems.

The third query corresponds to the seventh query of Set 3. This query retrieved the measurement values for a specific tag on the 4th day of the 5th month in 2016. This day was chosen for the query because it had the most data points. The performance of each system is illustrated in Figure 10.

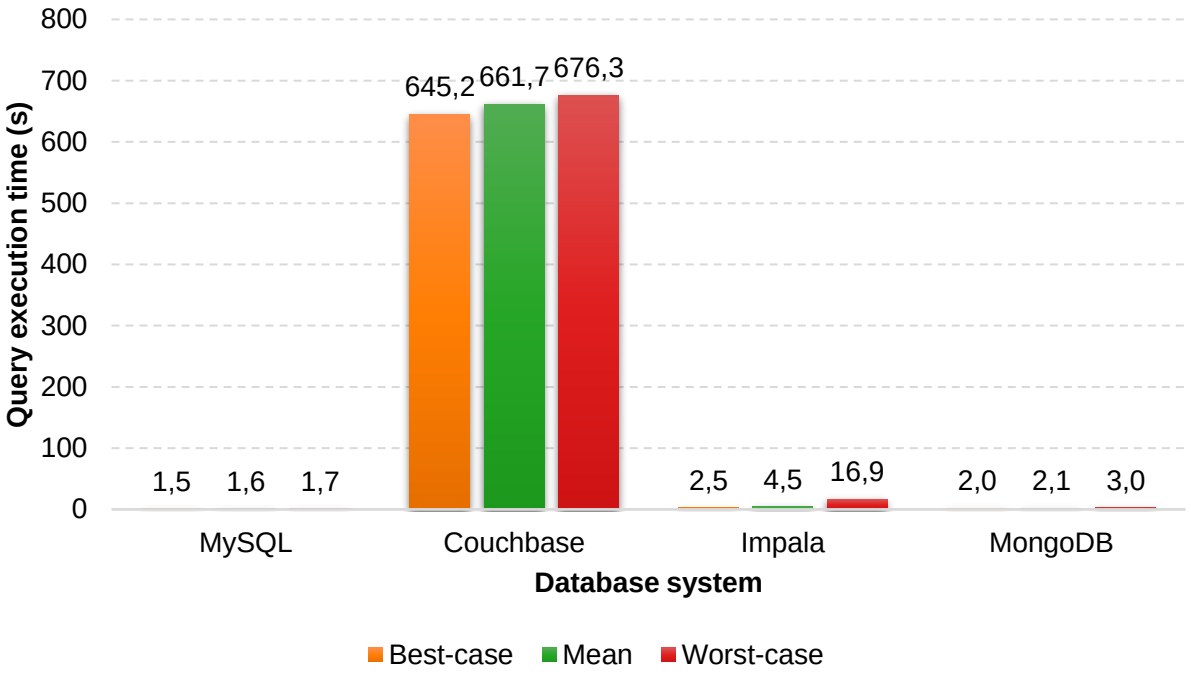


Figure 10: Comparison of the execution performances of the systems for Query 3

Figure 10 shows results similar to the previous two queries. MySQL had consistent performance and performed better than all the NoSQL systems. MongoDB performed the best of the NoSQL systems. For the worst-case performance, MongoDB was approximately 82.3% faster than Impala. Couchbase was many times slower than any of the other systems.

The final query was the second query of Set 4. The query calculated the combined arithmetic mean for each tag over the entire data period. This query differs from the first query in the following way: The first query performs many small calculations over short intervals within the data period. However, the second query of Set 4 performs a few large calculations over long intervals within the data period. The results of each system's performance are given in Figure 11.

For the final query, Figure 11 shows that Impala outperformed MongoDB. For the best-case performance, Impala had a time improvement of approximately 63% compared with MongoDB. MySQL again had consistent performance. As with the other queries, Couchbase was the worst performer.

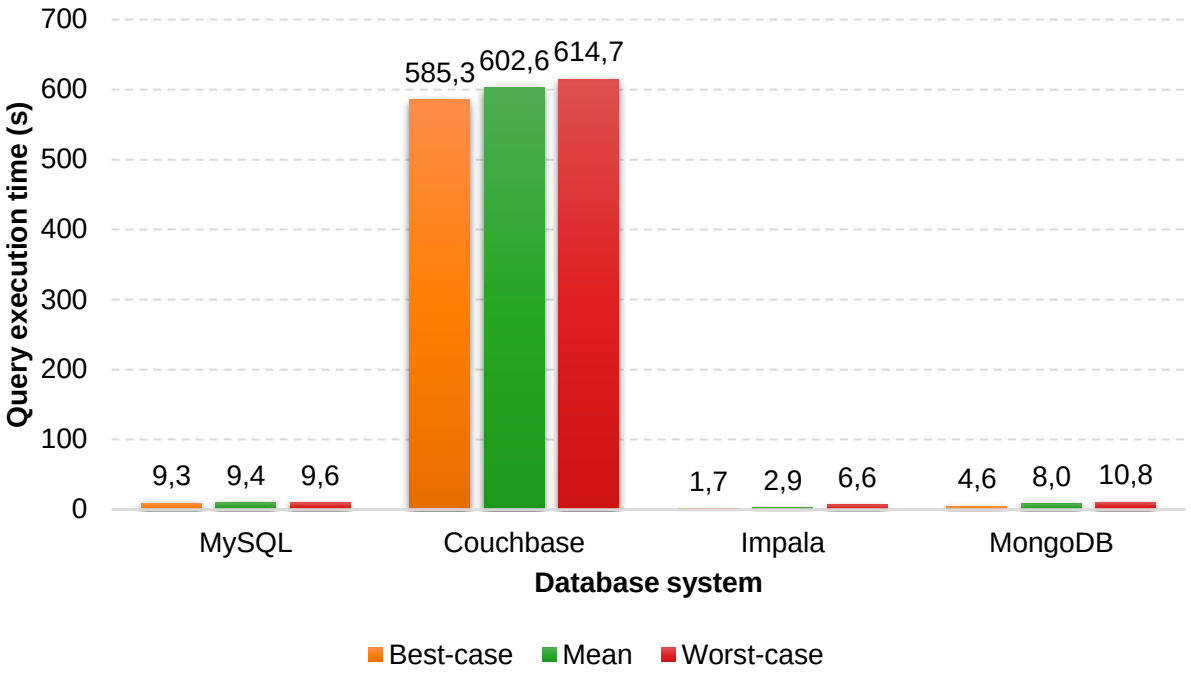


Figure 11: Comparison of the execution performances of the systems for Query 4

The four queries that were chosen consisted of a mixture of data retrieval and calculation tasks. Data retrieval was performed in Query 2 and Query 3. It showed results for two different scenarios. The first scenario is one where many data points were retrieved. The second scenario is one where searching had to be performed to retrieve a small set of data. The data calculations consisted of a query with many small calculations and one with a few large calculations. Therefore, the four queries represent a view of performance from four different scenarios.

For completeness, the poor performance of Couchbase, although not investigated, will be briefly discussed. The first possible explanation is that the version of Couchbase used could have been unstable. Another explanation could be that Couchbase requires greater or more thorough set-up. This greater set-up could require one to optimise the system for the data that would be stored. The final reason could be that too little system resources were available for Couchbase to function correctly. This would mean that Couchbase was too resource intensive for the test system. An attempt was not made to improve Couchbase's performance. The reason being that this experiment was created precisely for identifying instabilities or difficult configurations. The logic behind this is simple – in an organisation one would like to have a system that is stable and easy to maintain or change. Therefore, Couchbase would be inadequate.

To make a decision based on only the four queries shown here would be inadequate. The reason is that there may be dramatic shifts in performance for the remaining 25 queries not shown. Consequently, it was determined which system performed the best for each query. A system performed the best for a query if it had most of the lowest execution times. The number of times a system performed best were added together

and a percentage computed. These results are shown in Table 2. Note that MySQL is not included in the table because the comparison was done to determine the best NoSQL data store.

Table 2: Comparison of best performing NoSQL data store

Data store	Number of queries	Percentage of queries
MongoDB	24	82.8
Impala	5	17.2
Couchbase	0	0
Total	29	100

Taken together, the results of Figures 8–11 and Table 2 show that MongoDB was the best performing NoSQL system. Couchbase was the worst performing system. The evaluation of the systems also served as an evaluation of the Hadoop framework (Impala) compared with other frameworks. Therefore, the proposed system did not make extensive use of the Hadoop framework. In conclusion, the main data storage of the proposed system was provided for by MongoDB. An additional reason for choosing MongoDB was its selection as a leader in NoSQL data stores by Forrester Research [108].

The number of queries where MySQL was the best performer, although not shown in Table 2, were also calculated. In 21 queries, that is 72.4%, MySQL performed better than the best performing NoSQL system. This result does not, however, show the influence of increasing data volumes. It does create the expectation that MongoDB might not outperform an RDBMS in the final system. However, this study, as mentioned previously, was not an attempt at replacing an RDBMS with a NoSQL data store. Instead, the study focused on addressing the shortcomings of traditional data systems for the big data era.

3.4 System design

Multiple iterations of the design cycle were performed to produce the final system. The final design cycle, representing the completed system, will be discussed in this section. First, the system architecture will be defined. Following this, the components and software developed for the system will be discussed.

A hybrid architecture was chosen for the proposed system. Three big data system architectures were discussed in Section 2.3. From the three, polyglot persistence was chosen as the main architecture. Elements of the data lake and Lambda architectures were then combined with polyglot persistence to create the system architecture.

The top-level architecture of the system is a layered architecture. This architecture corresponds to the Lambda architecture, as discussed in Section 2.3.1. The Lambda architecture defined different layers for its system. By using layers, the Lambda architecture achieved complexity isolation. Therefore, the complexity of each layer was isolated from other layers.

The top-level architecture of the system consisted of three layers as is shown in Figure 12. The system's layered architecture approach is not completely the same as the Lambda architecture's approach. The Lambda architecture had the most complex layer at the top. Furthermore, the results of the most complex layer could be discarded in case of its failure. None of these characteristics apply to the proposed system's architecture. Therefore, the architecture of the proposed system was uniquely created during this study.

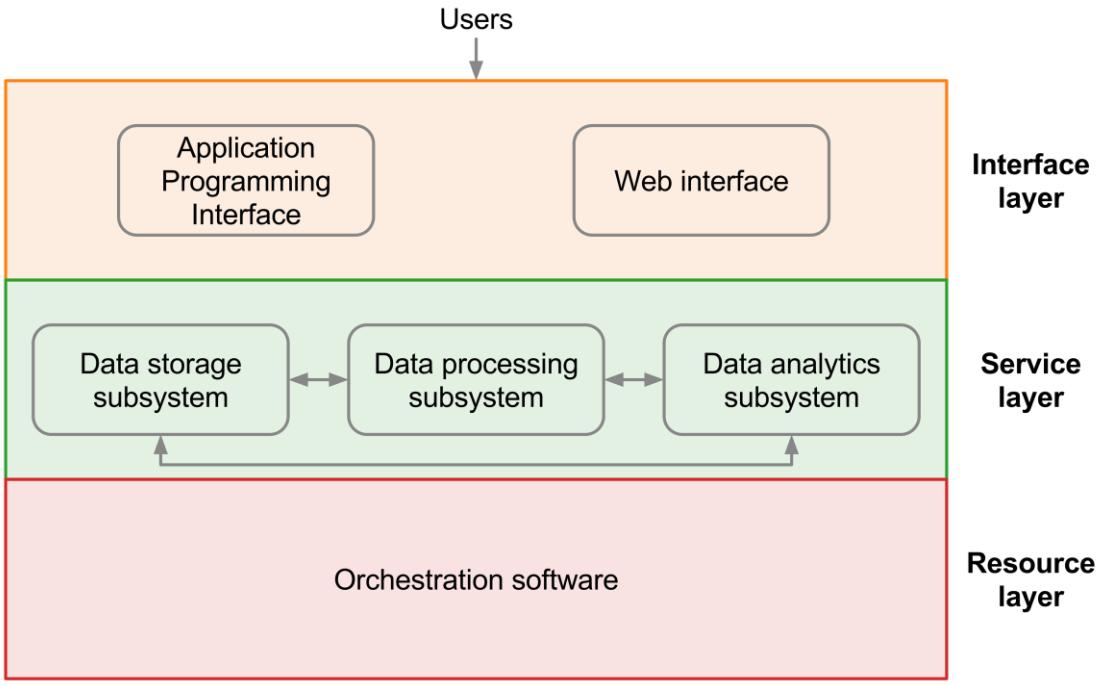


Figure 12: Top-level system architecture

Figure 12 shows that the proposed system is logically divided into an interface, service, and resource layer. Each layer is responsible for a specific function. The interface layer provides mechanisms for users to use the system. Two mechanisms are provided for system usage: an API and a web interface. In a typical scenario, applications and other systems will use the API. Human users would use the web interface. The interface layer serves as an abstraction of the service layer. Therefore, the interface layer hides many of the system's technicalities and internal functioning.

System functionalities, such as storage, processing and analysis of data, are provided by the service layer. In Figure 12, there are three subsystems that each fulfils a system function. The data storage subsystem stores all data. The data processing subsystem processes data, but only on a low level. For example, calculations and validation would be performed by the data processing subsystem. High-level processing is performed by the data analytics subsystem. Examples of such processing are data mining, sentiment analysis, and predictions.

The service layer requires hardware and high-level resources to function. These resources are provided by the resource layer. Orchestration software, which binds computers into a cluster, is the main component of the resource layer. The orchestration software is, therefore, an abstraction that hides hardware details.

Therefore, the orchestration software makes resources such as CPUs, memory, and storage available to the top layers.

Each of the architectural layers will be discussed subsequently in greater detail. A bottom-up approach will be followed, starting with the resource layer and finishing with the interface layer.

3.4.1 Resource layer design

The resource layer is the most important layer of the system. It is the base upon which the entire system is built. Any instabilities or errors in this layer would affect the entire system. Therefore, it was of the utmost importance to develop and use software that would be efficient, stable, and cost-effective. It was mentioned in Section 2.3 that system characteristics needed to be included in the design and architecture. Therefore, the resource layer had to support scalability, high availability, security, and extensibility[5]. Extensibility is important because one wants to have many choices for subsequent software and systems. That is, the choice of subsequent software and systems should be the least constrained.

As mentioned previously, the resource layer consists solely of orchestration software. The complexity and importance of orchestration software meant that it was not developed. Instead, open source projects with the required characteristics were sought. An evaluation of different orchestration software was beyond the scope of this study. The alternatives that were considered will, however, be mentioned.

Three orchestration software projects were identified, namely, Mesosphere DC/OS [109], Kubernetes [110], and Cloudify [111]. The three orchestration software packages all had similar features. These features include support for Docker images, horizontal scaling, monitoring, and self-healing. Here self-healing refers to the ability to restart applications or containers when they fail.

With such similar features, the choice of orchestration software was made based on long-term sustainability. It was felt that Mesosphere DC/OS would have long-term support and sustainability. The reason being that Mesosphere DC/OS has an enterprise version of the open source project. Furthermore, the DC/OS project is based on Apache Mesos and various other Apache projects. Apache is well known in the big data landscape. Thus, the sheer volume and size of all Apache projects would instil greater confidence in products based on them.

The end goal for the orchestration software was creating a computer cluster. Mesosphere DC/OS achieves this with the use of master and agent nodes. Master-agent operation is similar to the master-slave technique. Master nodes manage the cluster, which consists of agent nodes. Multiple master nodes can be created in a cluster; thereby, providing for high availability should a master node fail. Agent nodes perform tasks, such

[5] Extensibility considers the future growth of a system. In this context, it also refers to the ability to support a variety of systems and software.

as executing Docker containers, which are assigned by a master node. Mesosphere DC/OS further distinguishes between public and private agents. These details are not relevant in this study, but for completeness, all agents were private agents.

Mesosphere DC/OS supports various Linux-based operating systems. Ubuntu Server Edition was chosen as the computer operating system in this study. However, any operating system on which Mesosphere DC/OS can be installed would do. Furthermore, Mesosphere DC/OS can only be installed once on an operating system. Thus, a normal computer with one operating system can function either as a master or an agent node. Consequently, one would need many computers for multiple master and agent nodes. For small and medium organisations, this could be a problem. Ideally, one would like to minimise costs and maximise computer utilisation.

The single Mesosphere DC/OS problem was solved using VMs. In short, a VM allows for a virtual computer to be created. Many VMs can be created on one computer, each with its own operating system. The solution was, therefore, to create two VMs on each computer. Each VM used the Ubuntu Server operating system. One VM was used as a Mesosphere DC/OS master and the other as an agent. The computer that contains the VMs will henceforth be called the VM host.

It should be mentioned that the solution above would only be suitable for small clusters. The reason is that the Mesosphere DC/OS master node executes most of the DC/OS components [112]. As the cluster size increases, the master node/s will become busier. If the master node VMs have too little resources, cluster performance will decrease.

The risk with the solution is that if one computer fails, the cluster will lose both a master and an agent node. This risk can be minimised by adding more computers. By adding more computers, there will be more master and agent nodes to take over during failures. The advantage of the solution is fewer computers are required than the number of master and agent nodes combined. It should also be noted that a computer does not need to have a master and agent node VM. If the cluster has enough master nodes, the computer can be used exclusively as an agent node.

The cluster size for this study was three identical computers. The specifications of each computer are given in Table 3. These computers acted as the VM hosts. The VM specifications for the master and agent nodes are given in Table 4 and Table 5 respectively. The VMs were created identically on each computer.

Table 3: Specifications of each cluster computer

Name	Description
Operating system	Ubuntu server LTS 16.04
CPU	Intel Core i7-6700 3.40 GHz with four cores
Random access memory	16 GB DDR 3
Hard disk drive	2 TB, SATA 3, and 7200 rpm
Network interface	1 Gbps Ethernet

Table 4: Specifications of each Mesosphere DC/OS master VM

Name	Description
Operating system	Ubuntu server LTS 16.04
Processors	4
Memory	8 GB
Hard disk drive	100 GB

Table 5: Specifications of each Mesosphere DC/OS agent VM

Name	Description
Operating system	Ubuntu server LTS 16.04
Processors	4
Memory	6 GB
Hard disk drive	1.8 TB

A final component that was accepted implicitly is the existence of a computer network. An Internet Protocol version 4 (IPv4) computer network was created between the cluster computers. The network connects to each computer, thereby allowing communication and data transfer between them. The final system diagram for the resource layer is shown in Figure 13.

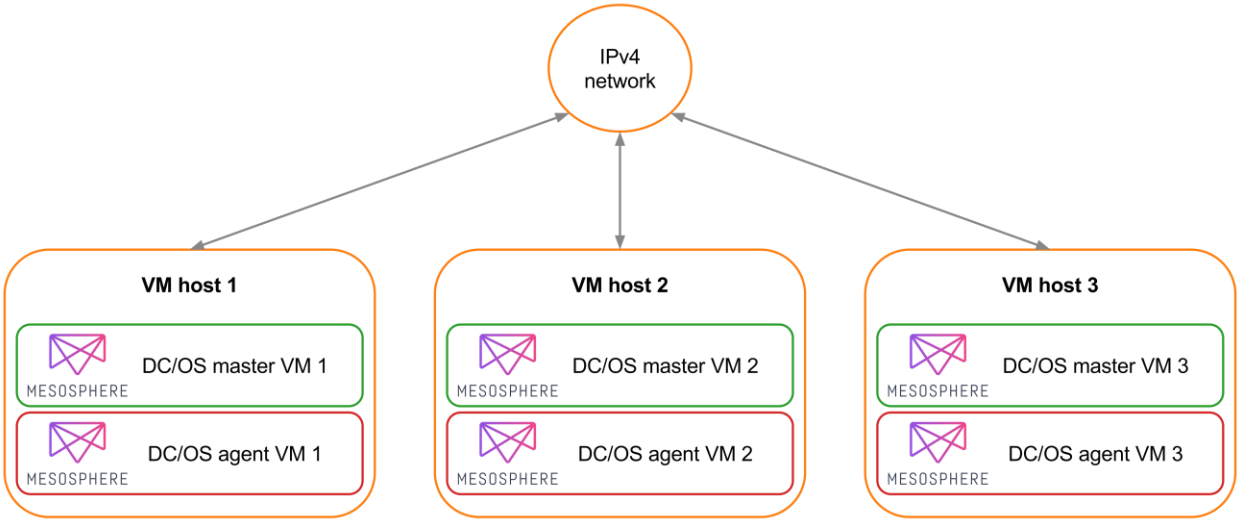


Figure 13: Resource layer final system diagram

Figure 13 shows the IPv4 network that interconnects each of the cluster computers, i.e. the VM hosts. Each VM host has two VMs, one for a Mesosphere DC/OS master and agent respectively.

A final aspect that will be mentioned in each of the layers is security. However, a detailed discussion of security is beyond the scope of the study. Mesosphere provides user authentication, which is enabled by default. The administrative user of the cluster is created during the installation of Mesosphere. After installation, the administrator is the only user able to manage the cluster. He/she is responsible for adding additional users, thereby allowing for strict access control to the cluster.

3.4.2 Service layer design

The service layer was designed based on a microservices architecture, as discussed in Section 2.4.2. Subsystems of this layer were, therefore, developed to provide specific services. Each service can be made available through the interface layer. Alternatively, external systems connecting to the service layer can make direct use of the services. The three subsystems of the service layer, as shown in Figure 12, provide services that are derived from their names. The data storage subsystem provides data storage services. The data processing subsystem provides (low-level) processing services. Finally, the data analytics subsystem provides high-level processing services.

Services in the service layer were created using containers, as discussed in Section 2.4.3. Docker was used to implement the containers. The use of containers is what necessitated orchestration software in the resource layer. By using Docker containers, many orchestration software could be used. The reason being that most orchestration software packages support Docker containers.

The fundamental benefit of using containers is a service layer that is loosely coupled to the resource layer. The containers or services of the service layer are still dependent on the resource layer. However, any

resource layer that can execute and orchestrate Docker containers can be used. Therefore, the resource layer is highly modular. If better orchestration software is created or discovered, it can be implemented in the resource layer. The resource layer can be changed without affecting the service layer. The service layer is, therefore, flexible and adaptable.

At this point, it is helpful to describe the difference between Docker images and containers. A Docker image can be seen as a blueprint for a house. The house that is built from the blueprint would be a Docker container. Thus, many houses can be built from the same blueprint. Similarly, many Docker containers can be created from the same Docker image. Therefore, the part of Docker that will execute on a computer will be the container.

The service layer architecture continued the hybrid architecture concept. Here, the architecture contained characteristics of the polyglot persistence and data lake architectures, as discussed in Section 2.3. From polyglot persistence, the polyglot characteristic was used. This characteristic meant that the architecture would make provision for various systems or services. Each service would have a specific purpose and consequently a specific software. The reason for choosing this characteristic was to obtain increased performance, as discussed in Section 2.3.2. This characteristic also avoids the one-size-fits-all pitfall.

Another characteristic used from polyglot persistence was that of avoiding impedance mismatch. Impedance mismatch was discussed in Section 2.3.2. One can only avoid impedance mismatch by correctly choosing or creating the software to be used.

The data lake characteristic that was used is the notion of storing data in a mostly raw and unprocessed format. This notion was discussed in Section 2.3.3. Its benefit is the ability to process and analyse different levels of processed data. For example, consider a sensor with measurements spaced one second from each other. This raw data can be processed into various forms. Some of which include per minute, per hour, or per day. It would, however, not be possible to obtain high-resolution data from low-resolution data. For example, hourly data cannot be used to obtain data for every second. A secondary indirect benefit is, therefore, that one would have access to high-resolution data.

Data storage subsystem

The data storage subsystem was the first to be designed and developed as it is the main focus of the system. Therefore, all other subsystems could be designed to be interoperable with the data storage subsystem. As already discussed in Section 3.3, MongoDB was chosen as the system's data store.

Another reason for choosing MongoDB was avoiding impedance mismatch as MongoDB uses BSON [113]. BSON, like JSON [114], is a data interchange format that is supported by all programming languages. This wide support is important as it does not constrain the choice of programming language. The fact that JSON is an Ecma standard means that new software and programming languages are likely to

support it. JSON and BSON can, therefore, be represented easily in a programming language. Therefore, the storage structure of data is no longer different from its structure in a programming language.

In-depth technical details of the MongoDB implementation will not be given as it is given in [115]. Unless stated otherwise, the details discussed can be found in [115]. MongoDB was implemented as a sharded cluster containing replica sets [116]. A replica set is a master-slave instantiation, which was discussed in Section 2.2.2. A sharded replica set replicates a data shard among the replica set members. One replica set member is a master and the rest are slaves. As there were only three cluster nodes, there could only be three MongoDB replica set members.

The MongoDB cluster contains three MongoDB components: the shard, config server, and router (known as mongos). The MongoDB cluster followed a production configuration. This configuration was recommended by MongoDB Inc. in [116]. Technically, the shard and config server components use the same application, called mongod. This application is only configured in a different way for each of the two components. The router component, however, uses a separate application, which is called mongos.

The shard component consisted of one sharded replica set. This sharded replica set will be referred to as the shard set. The shard set contained three replica set members. The MongoDB cluster is not fixed to one shard set. More shard sets can be added if required. The purpose of the shard set is to store the actual data.

The config server component consisted of one config server replica set. A config server replica set will be shortened to config server set. Three config server applications were deployed as a replica set. The purpose of the config servers is to store metadata to organise the entire MongoDB cluster. Unlike the shard set, more config server sets cannot be added. This restriction is stated by MongoDB Inc. in the production configuration.

The final component was the mongos router. Systems communicate via the mongos router application with the shard set. The mongos router application routes operations to the correct components in the shard set. Mongos, therefore, hides the complexities of the shard set from other systems. One mongos router application was deployed in the system. If the number of operations outgrows the mongos application, more can be deployed. The data storage subsystem as discussed thus far is shown in Figure 14.

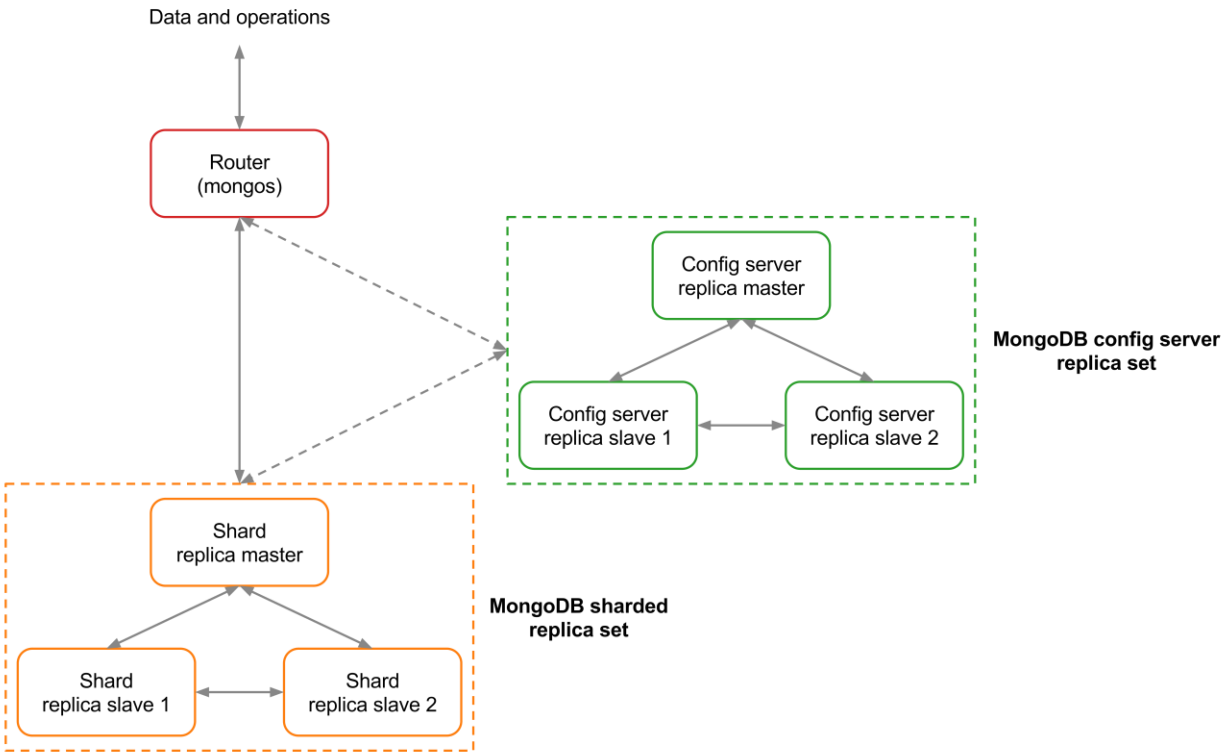


Figure 14: Data storage subsystem design with details on its usage

In Figure 14, the interface layer or directly connected systems send data and operations to the mongos router application. The mongos router implicitly communicates with the MongoDB config server replica set. Internally, the MongoDB config server replica set comprises a MongoDB master replica set member and two slave members. Data and operations would be sent from the mongos router to the MongoDB sharded replica set. Similar to the config server set, the MongoDB sharded replica set comprises one master and two slave replica set members. The shard set implicitly communicates with the config server set to ensure an organised cluster.

What is not shown in Figure 14 is the usage of Docker. In total, there are seven MongoDB components – each shown by a rounded rectangle in the figure. Each of the MongoDB components was implemented as a Docker container. Only one Docker image was needed to create the seven containers. The official MongoDB Docker image was used to create a custom optimised image for this study. The settings and configurations for each container were manually created. These settings and configurations enabled the different software to run within the containers.

Security was enabled among all the MongoDB components. X.509 certificates were used in conjunction with TLS/SSL version 1.2. These security settings provided confidentiality, integrity, and authentication for all communication. User authorisation was also configured. User authorisation limited the operations that each user could perform based on the user's identity. For example, except for the administrator, no user should be able to delete a MongoDB collection, which is similar to a database.

Data processing subsystem

The next subsystem to be designed was the data processing subsystem. Again, there were many possible software projects to use in this subsystem. The alternatives that were considered were Apache Spark [117], Apache Storm [118], Apache Hadoop MapReduce [119], and Apache Flink [120].

The Hadoop framework, which includes MapReduce, was a favourite in academic literature [33], [57], [61], [67], [121]. However, the tests performed in Section 3.3 showed that Hadoop is resource intensive. The reason is that Apache Hadoop is a framework that includes four modules. All four modules need to be active at the same time. Thus, if Hadoop had to be implemented alongside MongoDB, the resources needed would be too much.

Apache Spark was chosen to implement the data processing subsystem. Although it is part of Apache, it can be used independently from Hadoop. As an added benefit, it provided analytical features. Therefore, it could be used in the data analytics subsystem. Spark was also chosen because it had official integration with MongoDB. This integration was required for Spark to access data from MongoDB. The results of computations could then be stored in MongoDB.

Spark uses a master-slave architecture to form a Spark cluster. The Spark master node controls the Spark slave nodes. Tasks that need to be performed are first submitted to the master node. The master node would assign the task to one or more slave nodes. After a task completes, the master will obtain the results and return them. As mentioned previously, there were only three cluster nodes. Consequently, only three Spark slave nodes were created. One Spark master node was created. Thus, one computer had the Spark master and a slave node.

The choice of one Spark master node has the consequence of one point of failure. If the master node fails, no new tasks can be completed. While this is an undesirable situation, it can be prevented by the resource layer. As was discussed in Section 3.4.1, Mesosphere DC/OS can restart Docker containers. Thus, if the Spark master fails, Mesosphere DC/OS can restart it. This configuration reduces resource usage as the alternative would be to create multiple master nodes.

Apache did not provide an official Docker image for Spark. Therefore, a Docker image was created manually. The Spark image was created using the installation instructions given on the Apache Spark website [122]. Multiple versions of Docker images were created and tested until a working image was achieved. The final Docker image was used to create the master and slave nodes. The only difference between the nodes was in the configurations. The settings and configurations for the nodes were, therefore, created manually.

The complete data processing subsystem design is shown in Figure 15. The figure shows that system users communicate with the Spark master node. The Spark master node communicates with the Spark slave nodes

to perform tasks. In the figure, the presence of Docker is also shown. Each Spark node is an independent Docker container, which runs on the resource layer.

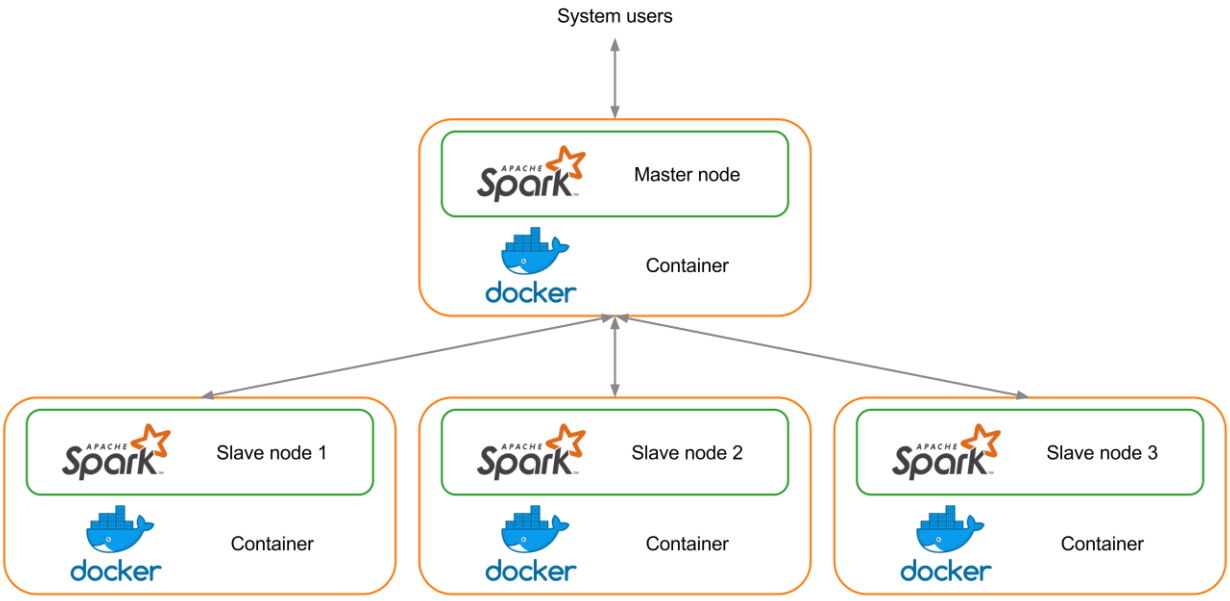


Figure 15: Data processing subsystem design

Data analytics subsystem

The final subsystem to be designed was the data analytics subsystem. The software components that were considered were those mentioned in literature [51], [57]. Two components were required in the data analytics subsystem. The first was a programming language. The programming language would provide analysis methods and the creation of analytics applications. The programming language that was chosen was Python [123]. Secondly, a software framework was required for large-scale data mining. The software framework chosen was Apache Spark. The advantage of using Python is the ability of using Spark. Users can write simple programs and access all of Spark's functionalities.

Instead of creating another Spark cluster, the cluster of the data processing subsystem was reused. A separate Docker container was not created for Python. For Python to be used with Spark, it must be installed alongside the Spark application. Consequently, Python was installed as part of the Spark Docker image. Python, however, still functions as a separate component in the data processing subsystem. One can use Python without using any Spark functions.

The complete service layer design is shown in Figure 16. In the figure, the various subsystems and their software components are shown. The interactions between the subsystems are also shown.

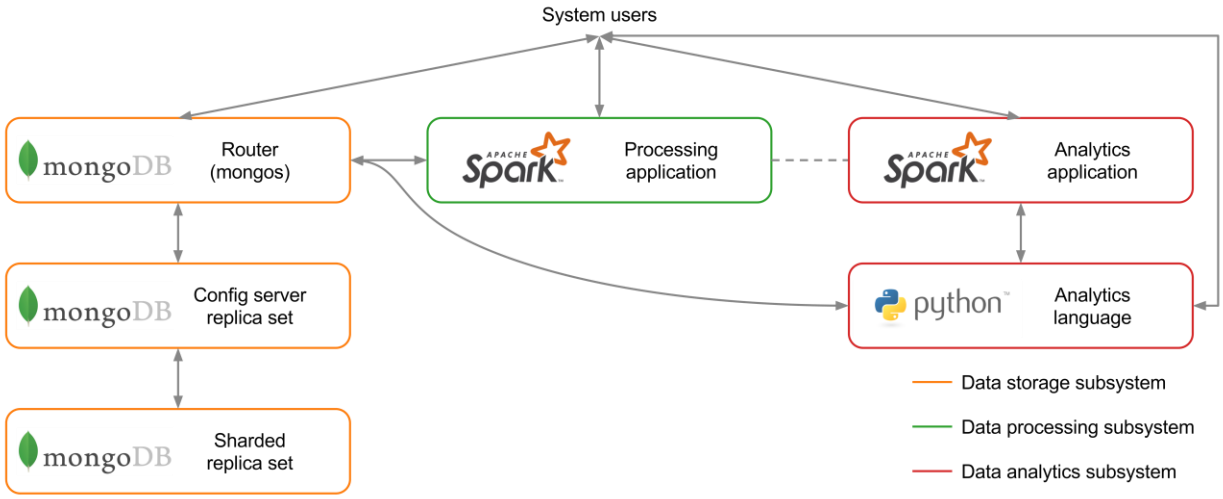


Figure 16: Service layer final system diagram

Figure 16 shows the data storage, processing, and analytics subsystems with their software. The dashed line between the two Spark applications indicates that they are one. Thus, only one Spark application would be executing in the system. Showing two Spark applications highlights their different uses in the respective subsystems. The data storage subsystem is only shown on a high level. Greater detail can be seen in Figure 14.

The final aspect shown in Figure 16 is system users that can connect to various software components. Python and Spark would be used for all data processing and analysis. Data storage can only be accessed via the MongoDB mongos router. The mongos router interacts internally with the different MongoDB components. The flexibility of Python is also worth noting. Python can interact directly with all the subsystems.

3.4.3 Interface layer design

The interface layer was designed based on two objectives. These two objectives relate to Requirement 3 and 4 in Section 3.2. The first objective was for systems to access data from the system. Thus, access was required to the data storage services in the service layer. MongoDB, as previously mentioned, provides APIs for many programming languages. Furthermore, organisations use a variety of programming languages. Thus, to ensure a generalised system, there needed to be a programming language independent way of accessing MongoDB. The solution, as discussed in Section 2.4.1, was provided by web services as part of the SOA approach.

The second objective of the interface layer was convenient user access. The same solution for the previous objective, i.e. web services, was used here. Such a web service would allow users to use the data processing and analytics services. This web service will be referred to as the web interface.

The two objectives meant that two software components were needed. The software component for the first objective, which will be called the API, was specifically created because of its dependence on the environment in which it would be used.

For the API to provide convenient and high-level data access, one needs to know two things: Firstly, who or what will access the data. Secondly, in what form data will be accessed and used. The details of this component are, therefore, less useful in a general sense. It is, nevertheless, logical for organisations that would use the system to want it modified for their needs. The API is such a component that would be modified based on organisational needs.

In brief, the API was implemented in the C# programming language. On a high level, the API is a Windows Communication Foundation (WCF) web service [124]. The protocol used by the web service is SOAP. The reasons for choosing SOAP were discussed in Section 2.4.1. C# and WCF were chosen because the test case organisation necessitated it. The organisation only had existing Windows hardware, and they preferred using Windows computers. Consequently, most of the test case organisation's systems were Windows-based. Therefore, the API had to be run on a Windows computer.

The API was not implemented as a Docker container. It is possible to create a Windows-based Docker image as discussed in [125]. However, the Windows Docker image has licensing, which would make it expensive to use. Consequently, the API resided on an external server. Even though the API was resident on an external server, it was still part of the system. As such, it is still shown in the interface layer as a component.

Open source software projects were found to implement the web interface. Some of the projects considered were Beaker [126], Apache Zeppelin [127], and Jupyter Notebook [128]. The requirement for the software was to support Spark and Python. All the listed software complied with these requirements, and therefore anyone would do. Apache Zeppelin was chosen due to its close ties with Apache projects such as Spark.

Apache Zeppelin allows users to analyse data interactively. Functionality is provided in the form of a notebook. The notebook is an environment where code can be written to perform calculations, access data, and draw graphs. Notebook results can be shared with people, thereby providing a real-time report. Notebooks can be shared among team members to provide for collaboration.

Unlike the API, the web interface was implemented as a Docker image. The installation instructions given on the Zeppelin website were followed to create the Zeppelin Docker image [129]. Additional settings and configurations were made manually to achieve the correct operation.

The interface layer is the least complex of the system's layers. It contains merely the two independent software components mentioned earlier. The final system diagram for the interface layer is shown in Figure 17. The figure shows that systems connect to the WCF web service to use it as an API. Users,

however, use the Zeppelin web service to access the system. Figure 17 shows the WCF web service explicitly as residing on an external server, as discussed earlier. The Zeppelin web service is, however, shown to reside within a Docker container.

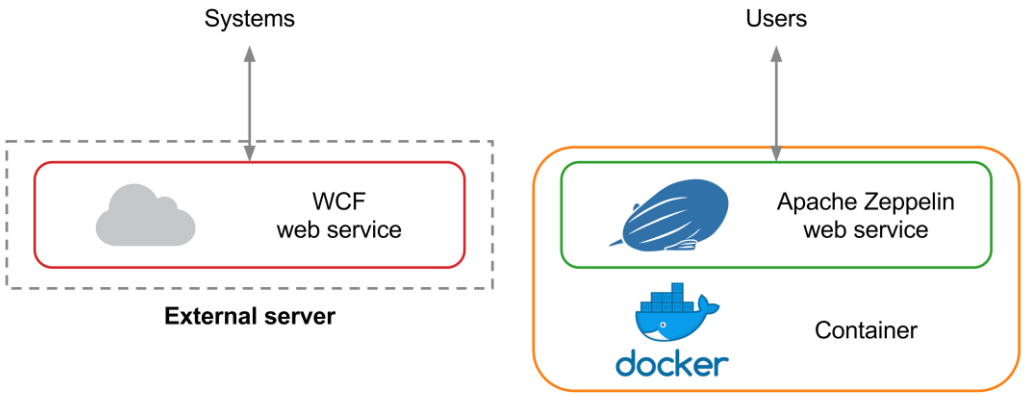


Figure 17: Interface layer final system diagram

3.4.4 MongoDB data design

The final design aspect was the modelling of data as it would be stored in MongoDB. Most of this activity was based on literature [45] and industry best practices [46]–[48]. Data modelling, as the name indicates, is concerned with data. Therefore, the solution described here was more dependent on the environment of the system. The solution described here is specifically applicable to the case study organisation. However, the concepts and process that were applied here are generally applicable.

The case study organisation uses tags to describe data. A tag is a collection of related data, as discussed in Section 3.3. This usage of tags was also prevalent at the industrial and mining clients of the case study organisation. Each tag would have many values, which are measurements for the tag. A tag also contained tag information, which described the characteristics of the tag.

The data modelling activity started by analysing the tag information that needed to be stored. Common information, such as tag name, measurement unit, and description, were identified easily. Additional information was added based on the limitations of the systems the organisation had. Addressing these limitations was an indirect advantage, which is to be expected from a new system. Therefore, the organisation could create the new system to improve their operation. The complete set of information that was identified for storage is given in Table 6. The table also describes how the information was stored in a MongoDB document.

Table 6: Details on the information stored in a tag document

Information	Storage in MongoDB document
TagNames:	Document array
AssignedTime	Date and time
TagName	String
SiteTagNames:	Document array
AssignedTime	Date and time
TagName	String
Interval:	Document
Year	Integer
Month	Integer
Day	Integer
Hour	Integer
Minute	Integer
Second	Integer
Millisecond	Integer
Unit	String
Source	String
Description	String
MeasurementType	String

The interpretation of the information in Table 6 is as follows: TagNames is an array of documents. Each document has AssignedTime and TagName attributes. Thus, each time a tag's name changes, a new document is added to the TagNames array. The new document contains the date on which the tag's name was changed to the new tag name. This structure allows tag names to be changed at will without any negative consequences. The SiteTagNames attribute is similar to the TagNames attribute. SiteTagNames, however, stores the tag name as used on a site, i.e. client premises.

Interval indicates the minimum amount of time that may be present between successive tag values. One might think that a date and time might suffice. MongoDB can, however, only store valid date and time values. For example, one could not indicate that there should be at least one millisecond between values. Similarly, one could not indicate that there should be at least one year between values. The solution was creating a document that could store integers representing time intervals as years, months, days, and even milliseconds.

Unit is the unit of measurement. This unit ensures conformity and therefore needed to be saved based on a predefined set of values. Source indicates the source from which the data came. Values for this attribute

would also be from a predefined set. Description is a self-explanatory attribute. When a tag is saved, a user can provide an optional description. Such a description could aid in understanding the tag's usage.

MeasurementType indicates what the tag's values are. For example, a total power tag would be the sum of multiple individual power usage tags. In other words, MeasurementType indicates if a calculation was performed on the tag's values. This attribute could be used to determine which further calculations could be applied to the tag.

As stated earlier, the actual measurements for a tag are called values or tag values. There are several ways of storing tag values. The first decision was to either embed or reference a tag value document [46]. Embedding means adding a tag value document within the tag document. Referencing means creating a value document separately. A link, i.e. reference, to this document is then created in the tag document. Industry best practices indicate that embedding should be used for one-to-few relationships. Referencing should be used for one-to-many and one-to-squillions[6] relationships.

The goal of the proposed system is providing high-resolution data. High-resolution data implies that tags would have many values. This scenario is covered under the one-to-squillions relationship. Consequently, referencing was chosen to link tag value documents to tags.

The one-to-squillions referencing technique is referred to as parent-referencing [46]. Thus, children documents, which are the tag value documents, would include an attribute that refers to their parent document, which is the tag document. The child attribute used for referencing always refers to the parent document ID. This requirement is necessary because the parent document ID is unique. When searching for a tag's values, one simply needs to know the ID of the tag document. One can then retrieve all the tag value documents whose reference attribute contain the given ID.

Choosing the referencing technique to link tag value documents implied that the tag value documents would be standalone documents. The tag value documents were saved in a separate collection from the tag document collection. Analysis of the required tag value information was then performed. Table 7 gives the information that tag value documents needed to contain.

Table 7: Details on the information stored in a tag value document

Information	Storage in MongoDB document
TagID	ObjectId
Timestamp	Date and time
Value	Dynamic

[6] Squillions is a very large indefinite number.

From Table 7, the TagID attribute is used to link a tag value document to a tag document. The TagID is a MongoDB ObjectId, which is used as a unique ID for documents. Timestamp contains the date and time when the measurement was made. The Value attribute stores the actual measurement. It is dynamic to enable the storage of any data type, such as strings, integers, and floating-point numbers.

More advanced data modelling concepts were also considered as described in [47] and [48]. Denormalization is a concept that was considered. Denormalization refers to the moving of attributes from the “one” document to the “squillions” document or vice versa. Zola suggested that one could even denormalize all information in the “one” document to the “squillions” document [47]. One would perform denormalization to speed up the completion of frequent queries. For example, if a tag name is frequently used to retrieve tag values, one could denormalize the TagNames attribute into the tag value documents. Although denormalization is discussed here regarding NoSQL data stores, it has already been used in relational data modelling [30, p. 379].

The two best candidates for denormalization were TagNames and SiteTagNames. The problem, however, is that frequently all the tag names and site tag names need to be retrieved. In such a situation, there is no need for the tag values. To retrieve all the unique tag and site tag names would require searching through all tag values. There could be millions of tag value documents, which would make the response of this type of query deteriorate as tag values increase. This reason was sufficient to motivate a separate tag document. It does, however, not address the denormalization for the rest of the tag attributes.

Consider the situation where one had a tag document with only the TagNames and SiteTagNames attributes. The rest of the tag information would be contained in a tag value document. If one needed to know information such as Interval or Unit given a tag name, one would perform two queries. The first query would be to find the tag document. The second query would search for any tag value document. If, however, the tag information was stored in the tag document, one would perform a single query. Updates to any of the tag information, when contained in the tag value document, would also be slow.

By not denormalizing TagNames and SiteTagNames, there is no sense in denormalizing the rest of the tag information. One might think this would be unnecessary if TagNames and SiteTagNames were not needed. SiteTagNames is needed because in most cases a client, i.e. industrial or mining company, merely provides their data. It would be unreasonable to expect them to change all their tag names to unique ObjectId values. Furthermore, both TagNames and SiteTagNames make it easier for humans to understand and work with tags. Finally, if one enforced descriptive tag names, one would derive some information regarding the tag from its name.

3.5 Conclusion

Many design cycles were completed in the design of the proposed system. This chapter, however, only detailed the final cycle of the completed system. The design process started with requirements analysis. User requirements for the system were obtained, and from them, system specifications were created. Table 8 summarises the system specifications that were created.

Table 8: Summary of system specifications

Specification number	Description
1	The system should focus on providing high availability.
2	Data stored by the system should be replicated between multiple computers.
3	An API should be available for external systems to integrate with the proposed system.
4	The system should provide a web interface for users to make use of its services.
5	Structured, unstructured, and semi-structured data should be supported by the system.
6	The system should scale horizontally using commodity hardware.
7	Different data within the system should be separated through separate repositories.
8	The system must control the access and privileges of users.
9	The system must use security measures to protect information.

Following the definition of specifications, the data store experiment was performed. This experiment was a continuation of Section 2.5. It tested the performance of MongoDB, Couchbase, Apache Impala, and MySQL. The purpose of the experiment was to identify the most appropriate NoSQL data store for this study's context. MySQL was used to compare the NoSQL data stores' performance to an RDBMS. Thus, it provided an expectation for the performance of the completed system.

In the experiment, four sets of tests were performed, which consisted of 29 tests or queries in total. MongoDB was the best performing data store in 24, which is 82.8% of the tests. Couchbase was the worst performing data store and did not perform the best in any of the tests. When comparing MySQL with MongoDB, MySQL performed better in 21 or 72.4% of the tests. The results of the experiment meant that MongoDB was chosen for the data storage of the system.

The next task was designing the system. The architecture of the system was created as a hybrid of the polyglot persistence, data lake, and Lambda architectures. At the top-level architecture, three layers, similar to the Lambda architecture, were defined. The three layers, from bottom to top, were the resource, service, and interface layers.

The resource layer was responsible for providing hardware and high-level resources to the top layers. Orchestration software, which was discussed in Section 2.4.3, was used in the resource layer to fulfil its

function. Three software projects were considered for use in the resource layer, namely, Mesosphere DC/OS, Kubernetes, and Cloudify. All three software supported Docker images, horizontal scaling, monitoring, and self-healing. Mesosphere DC/OS was chosen due to its use of Apache Mesos and its alignment with the Apache Software Foundation.

Mesosphere DC/OS allowed the creation of a cluster or private cloud service through master and agent nodes. Three computers were used to create the cluster. However, instead of three cluster nodes, six were created through the innovative use of VMs. Therefore, the cluster consisted of three master and agent nodes.

The service layer consisted of three subsystems, namely, the data storage, data processing, and data analytics subsystems. Each subsystem of this layer provided a service, which together implemented all functions of the system. Microservices architecture was used for the service layer, which meant that the services were created using Docker containers.

The data storage subsystem provided a data storage service by using MongoDB. A Docker image was created to run the MongoDB software. This image was used to implement a MongoDB cluster. The MongoDB cluster formed a production configuration sharded replica set. In such a configuration, data would be replicated to each of the three cluster computers.

The data processing subsystem provided low-level processing services such as calculations and validation. Four software projects were considered, namely, Apache Spark, Apache Storm, Apache Hadoop MapReduce, and Apache Flink. Apache Spark was chosen to implement this subsystem because it provided high-level data processing functions. There, Apache Spark could be used in the data analytics subsystem. A Spark cluster, consisting of one master and three slave nodes, was created to construct this subsystem. Again, a Docker image was created to use the Spark software.

High-level processing, for example, machine learning and data mining, was provided by the data analytics subsystem. This system allowed for applications to be created using the Python programming language. Apache Spark was also used here, which meant that the cluster of the data processing subsystem was reused. No Docker images were created, as Python was included in the previously created Spark image.

The interface layer made use of SOA, as discussed in Section 2.4.1. Thus, two web services were created – an API and a user interface. The API was a WCF web service on an external Windows server. It was external because the system used Linux computers and it was not possible to create a Windows Docker image. The API was created for applications and external systems to access data from the system in a business defined way.

For the user interface, Beaker, Apache Zeppelin, and Jupyter Notebook were considered as options. Apache Zeppelin was chosen due to its close ties with Apache projects such as Spark. Zeppelin allowed users to

use a web browser to work with Python, Spark, and MongoDB interactively. Thus, all system services could be accessed through the Zeppelin web interface.

The final design activity was to model the storage and usage of data. Within the context of the study, data was used as tags. A tag is the collection of measurements for an entity. For example, all power consumption values of a motor would be one tag. From the definition of a tag it follows that it has many values. Therefore, MongoDB was configured to save tag information in a tag document. The values of a tag would be saved in value documents. A tag document was then linked to its value documents via a one-to-many relationship.

After completing the system design, the system was implemented and deployed. The verification and validation followed, which are discussed in the next chapter.

CHAPTER 4 RESULTS OF SYSTEM USAGE

4.1 Preamble

The system designed in Chapter 3 was implemented in a case study for verification, validation, and evaluation. This chapter commences with a description of the case study on which the system was implemented, as detailed in Section 4.2. After this description, the details and results of the verification are given in Section 4.3. In Section 4.4, five validations are provided to show how this study's problem was addressed. Section 4.5 provides a discussion of this study's results, and this chapter concludes with Section 4.6. Therefore, this chapter preconcludes this study by providing evidence of its successful completion.

4.2 System implementation on a case study

The system was implemented and deployed at an engineering consultancy company. The company has clients in the South African industrial and mining sectors. Data from their clients are collected and processed to deliver various services. The data that is gathered is organised as tags, as described in Section 3.4.4. There are two data-related systems in the company, as shown in Figure 18.

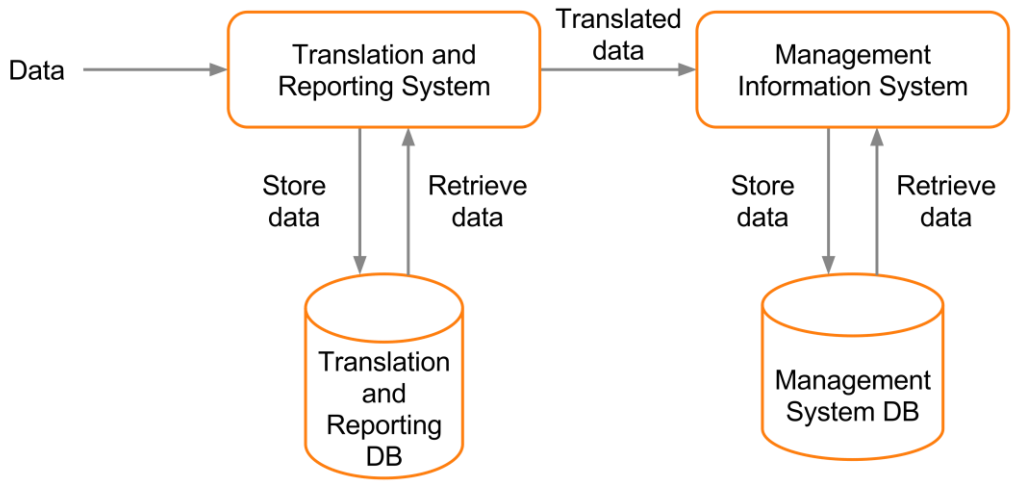


Figure 18: Case study organisation's systems overview

The first system was a translation and reporting system that translated and filtered data. Different clients use different data formats and, consequently, one needs to transform these formats into a standard format. The translation and reporting system performed this transformation of data into a standard format. Additionally, the data could also be processed, for example, aggregated. From this system, the data was either saved in its own data system or sent to the management information system.

The management information system was the second system of the organisation and also the biggest and most complex of the systems. High-level data processing was performed in this system. Projects were

tracked and monitored on its web interface. The system displayed data visualisations to show the current and historical data of a client. All the system's data was stored in a database system. It is important to note that both systems were located on separate computer systems. Communication was established between the systems via an IPv4 network.

The data system of the translation and reporting system was a custom created file-based storage system. Each client had a separate folder in which their data was stored. Data was organised into years and months, and stored in comma separated value (CSV) files. Thus, for one year there would be 12 data files. Each file contained data for its respective month and year.

The management information system used MySQL for its database system. This database system contained many databases that stored different clients' data. Each database had many relational tables of which further details will not be given due to their complexity. The MySQL system was not tested in its original state. Instead, a test MySQL system was created as discussed later. It is, however, important to note that the organisation's data was relational yet partially structured in its original form. The consequence of the data's relational nature meant that an RDBMS would have an advantage over the developed system.

The location where the centralised data management system was deployed is shown in Figure 19. The translation and reporting system would perform minimal processing and filtering on incoming data. This data would be stored in the centralised data management system. Subsequently, the translation and reporting system and the management information system would retrieve data from the centralised data management system. Any modification or addition of data would be done in a single location, namely, the centralised data management system.

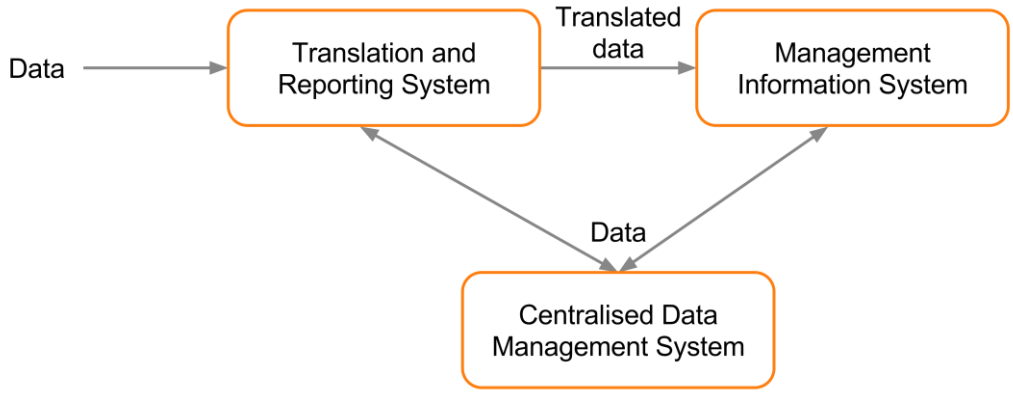


Figure 19: Location of the proposed system in the case study organisation

The validation, as discussed in Section 4.3, tested three systems, namely, the developed system, the MySQL test system, and the translation and reporting system's data system. No changes were made to the case study system; thus, it was tested in its original state. Additionally, all three systems were given the same computing resources.

The MySQL system was created with default settings and no indexes except for the required primary key index. Security settings were not enabled for the MySQL system, as this is how the management information system uses MySQL. The MySQL system was given the same resources as a single computer of the developed system. In fact, the MySQL system was deployed within the developed system.

The developed system was limited to the use of one computer to process commands. The remaining computers were still active and used in the system. All user commands were, however, directed to the single computer. This limiting of the developed system was done to allow for a fair comparison. Thus, the results are solely due to the artefact of this study and not external factors such as hardware.

The same data was used in all three systems and, where possible, structured in the same way. All three systems stored the same attributes of the data. Thus, the developed system did not store all the fields as discussed in Section 3.4.4, because this could have influenced its performance. Consequently, all these measures further prevented external factors from influencing the results.

4.3 Verification of system operation

In this study, verification was the process used to confirm whether the developed system met the specifications as given in Section 3.2. During this process, the operation of the system was ensured. Consequently, after verification, it was not only certain that the system functioned as designed, but also that it operated correctly. The verification process was performed during the design cycle of the research process. After verification, the system was deployed and subsequently validated.

Each of the specifications mentioned in Section 3.2 is in some way related to the components of the system. This relationship is shown in Table 9. In the subsections that follow, greater information will be provided on how each specification was verified. Instead of discussing the individual specifications, each component will be discussed.

Table 9: Specifications and the components related to them

Number	Specification	Related component
1	High availability	MongoDB and Mesosphere DC/OS
2	Data replication	MongoDB
3	External system integration	API and Mesosphere DC/OS
4	User accessibility	User interface, Python and Apache Spark
5	Multistructured data support	MongoDB
6	Horizontal scalability	MongoDB and Mesosphere DC/OS
7	Multiple data repositories	MongoDB
8	User access control	MongoDB and Mesosphere DC/OS
9	Data protection	MongoDB and API

4.3.1 Mesosphere DC/OS verification

Mesosphere DC/OS implicitly provides many of the specifications required by the system. However, all the Mesosphere functionalities that allowed adherence to the specifications are only applicable if Mesosphere functions correctly. Thus, the functioning of Mesosphere was verified as explained in the installation documentation [130]. Two methods of verification are described, namely, monitoring of the servers' status and viewing of components' health status.

The servers' status can be monitored by browsing to the Exhibitor website of the Mesosphere cluster. This website was visited after configuring the Mesosphere nodes. A screenshot of the website is shown in Figure 20. Overall, the figure shows that the status of each master node is green, i.e. the node is healthy and functional. Furthermore, the figure also shows that all the master nodes were detected. That is, the number of masters shown on the website corresponds to the number of masters that were configured.

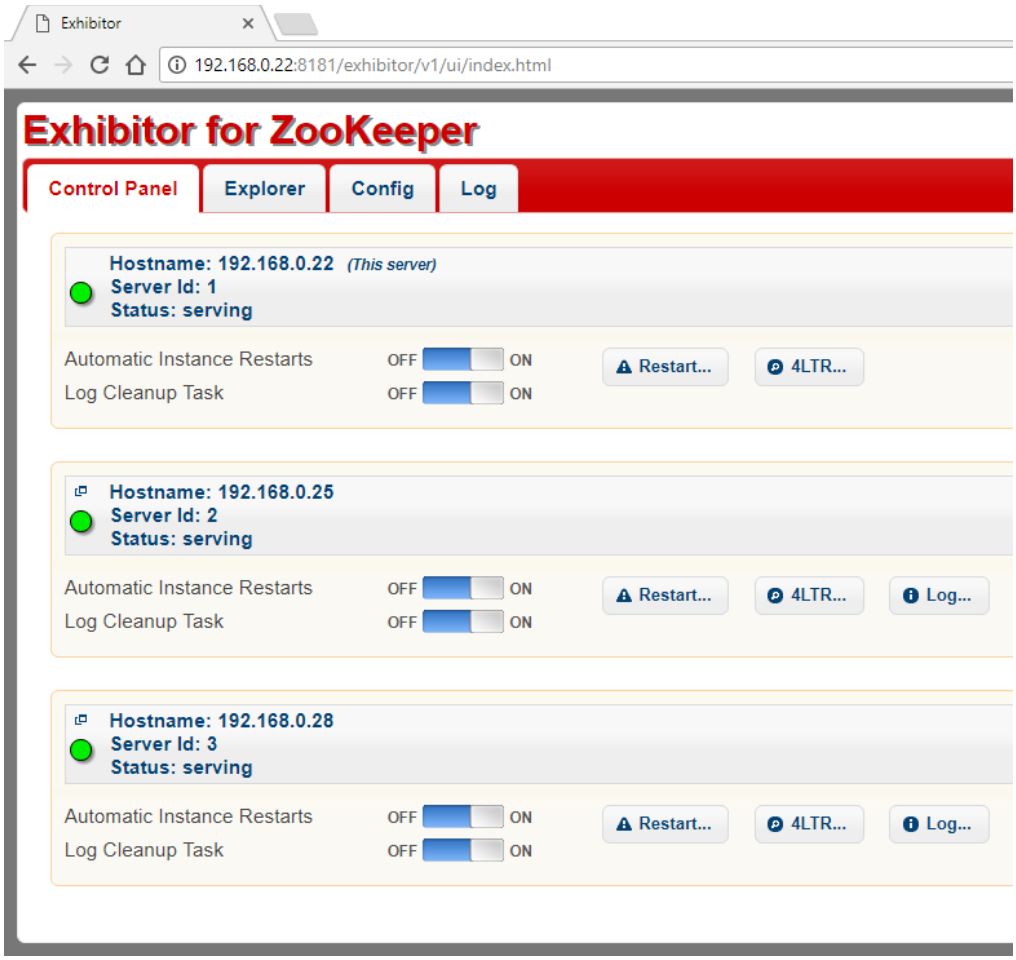


Figure 20: Screenshot of Exhibitor website showing each Mesosphere master node's status

Figure 20 does not show the status of the agent nodes. To view the status of the agent nodes, one had to proceed to the second verification method. The second verification method required one to log in to the Mesosphere web interface. This web interface is the main method for monitoring and controlling the cluster.

The actual screenshot of the web interface during verification is shown in Figure 21. One would see such a screen only upon logging in successfully.

The second method of verification was to view the health status of the Mesosphere components. These health statuses are already shown on the overview screen, as shown in the bottom right of Figure 21. This Component Health section, however, shows only a summary of the components' health. Greater information is obtained when clicking the View All components button. Completion of this action resulted in the Component page shown in Figure 22.

Figure 22 shows the healthy and unhealthy Mesosphere components. As can be seen in the figure, there were no unhealthy components. Therefore, all Mesosphere components functioned correctly. Although not shown in the figure, the health statuses are determined from all the nodes. Thus, the Component page indicates the entire Mesosphere cluster health.

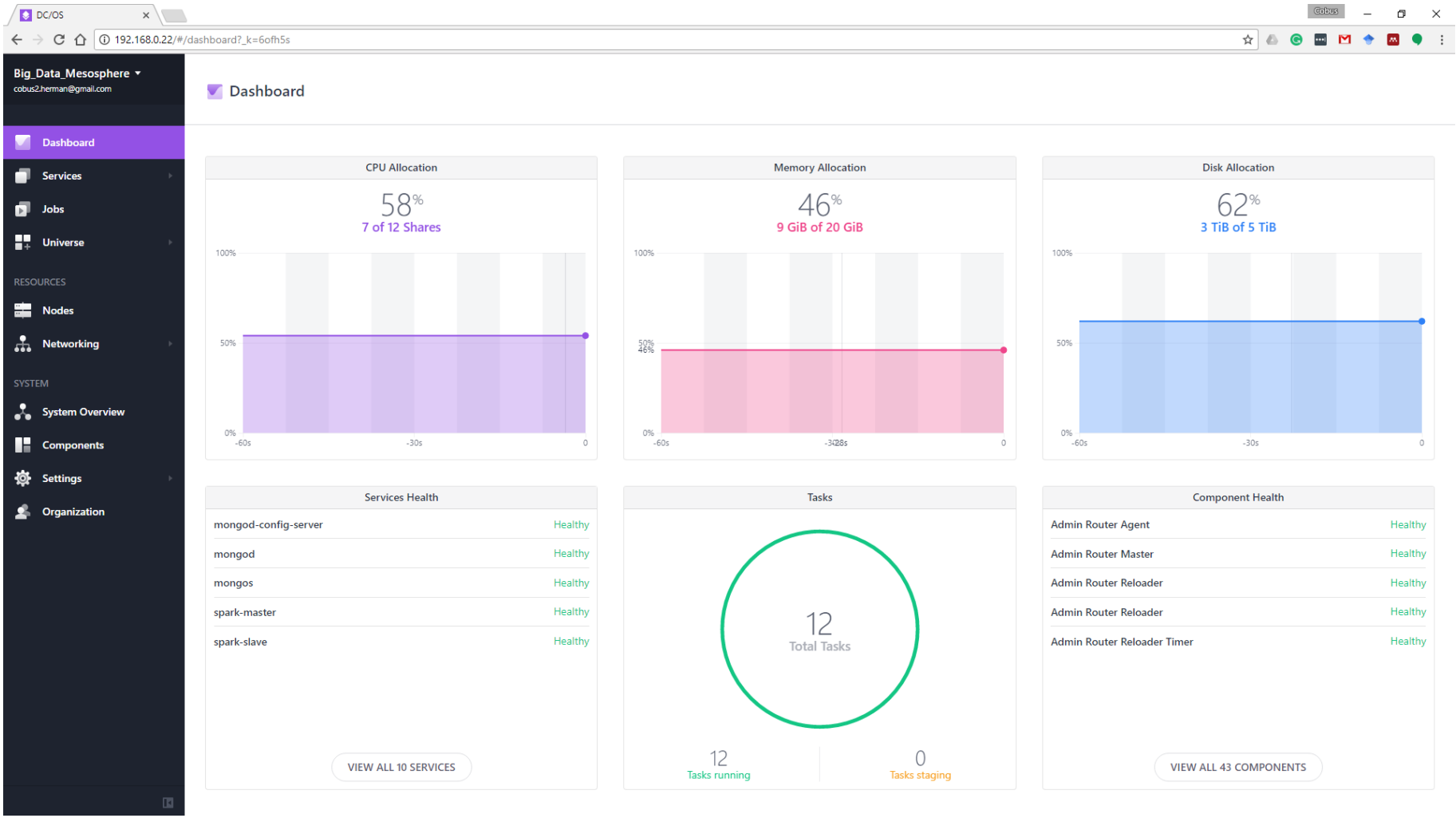


Figure 21: Mesosphere DC/OS web interface after a successful login

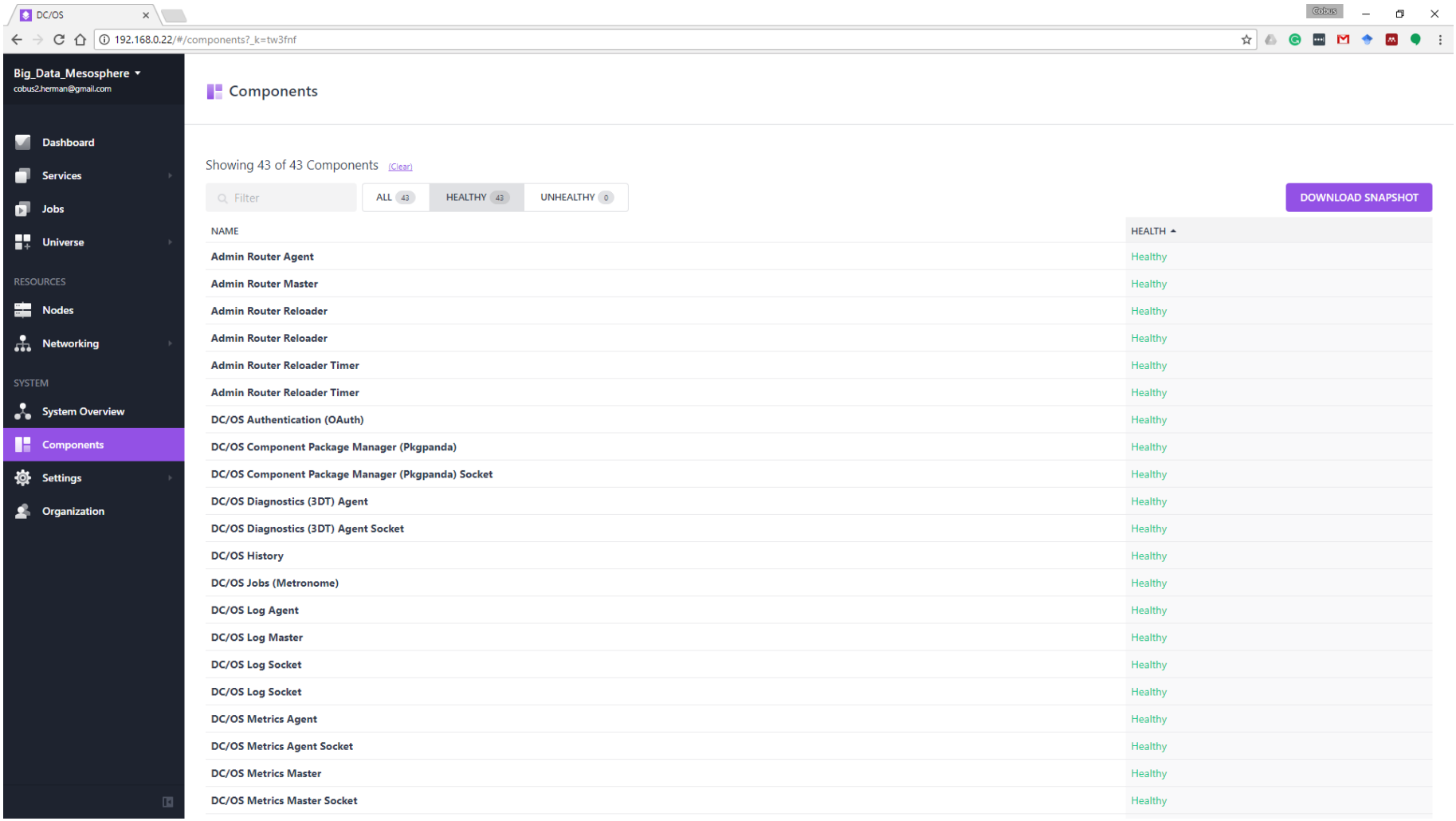


Figure 22: Component page showing the health of all Mesosphere DC/OS components

The correct operation of Mesosphere was established. Consequently, the rest of this subsection will discuss the way in which Mesosphere allowed some of the specifications to be met.

By default, Mesosphere provides for high availability with a master-agent configuration. For the proposed system, three master and three agent nodes were created. The master and agent nodes were configured on three commodity computers, with one master and agent on each computer. This configuration implicitly satisfied the high availability specification. If one of the master nodes failed, two more could take its place. In general, if an entire computer failed, the other two computers would still provide service.

It is elementary to show that using more computers increases the availability of a system. The proof is evident in the probability of independent events [131]. If, for example, the probability of a computer failing is 5%. That is:

$$P(Fail) = 0.05$$

Then the probability of two computers failing is:

$$P(Fail_1 \cap Fail_2) = P(Fail) \cdot P(Fail) = 0.0025 \quad \textbf{(1)}$$

The reason (1) is true is because the probability of one computer failing can be assumed to be independent of another computer failing. A logical consequence of (1) is that the more computers one adds to a system, the less the probability will be that all the computers will fail. The probability that all the computers fail is synonymous with the probability that the system is unavailable. Therefore, the greater the number of computers the system has, the higher the likelihood will be that the system will remain available.

Mesosphere provides self-healing as discussed in Section 3.4.1. This self-healing functionality also ensures high availability. High availability is ensured, because Mesosphere will automatically restart the software if there is a software fault or failure. Such a function is invaluable considering that most of the system is constructed from software. Most of the software components also function within Docker containers, which is yet more software. Therefore, the software could be highly available due to automatic restarting.

As stated in Section 3.4.1, Mesosphere provides horizontal scalability. This scalability is due to its distributed architecture and its use of master and agent nodes. Additional agent nodes can be added to the Mesosphere cluster at any time. A new computer should simply be set up with the Mesosphere software and then connected to the network. It is, however, not possible to add new master nodes. If more master nodes are required, the cluster should be recreated. The ability to add new agent nodes meant that the proposed system achieved the horizontal scalability specification.

The final specification achieved by using Mesosphere was user access control. Mesosphere, however, only controls access to the cluster. Thus, by implementing access control, users can be restricted from controlling the cluster. This control is important because the rest of the system's software can be controlled from the

Mesosphere web interface. Consequently, any incorrect actions taken on the cluster could result in a loss of service.

Mesosphere implemented user access control by only allowing authorised email addresses to log in. Therefore, when a user signs in to the web interface, their email address needs to be entered. If the email address is on the authorised list, the user can access the web interface. Otherwise, the user is blocked with an error message.

The web page that provides the user access controls is shown in Figure 23. The figure shows that only one email address was authorised, namely that of the author. Additional email addresses would appear on the page similar to the already authorised address.

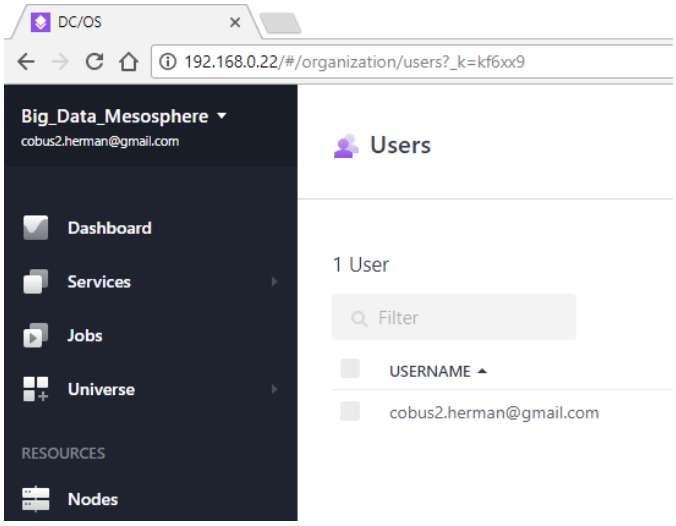


Figure 23: User access control page showing an authorised email address

4.3.2 MongoDB verification

The second and most important component verified was MongoDB. Naturally, all the specifications satisfied by Mesosphere DC/OS would also need to be satisfied by MongoDB. For example, the system could only be highly available if both MongoDB and Mesosphere DC/OS provided high availability. The reason for this is because of the layered architecture of the system. The top layer of the system can only satisfy the specifications if all the lower layers also satisfy the specifications. In addition to the Mesosphere DC/OS specifications, MongoDB had to satisfy other specifications as well.

All the specifications satisfied by MongoDB were implicitly part of the software. A white paper by MongoDB described the features that MongoDB contained [132]. The features that were part of the version of MongoDB used are:

1. High availability, which is enabled through master-slave replication and automated failover.
2. A flexible data model by supporting a dynamic schema. Thus, there is support for multistructured data.
3. Horizontal scalability through automatic sharding.

4. User access control with the use of authorisation and authentication.
5. Data protection through the encryption of data on the network.

Although not mentioned explicitly, MongoDB supports multiple repositories by default.

It is worth remembering that MongoDB was used in three different roles as mentioned in Section 3.4.2. The roles are shard, config server, and router. Therefore, three sets of verification were performed – one for each role. Verification focused on two parts. In the first part, the set-up of the software was inspected for correct functioning. The second part tested MongoDB operations from an external program. This part used a Python program, which served as verification of the Python component as well.

The first part started by verifying the shard role. In this part, the log file created by the master MongoDB member was inspected for warnings and errors. This log file was also inspected for the status of the replication and sharding processes. The reason for this is because, as mentioned in Section 3.4.2, the shard role was set up as a sharded replica set. Therefore, verification had to ensure correct sharding and replication behaviour. Only excerpts of the log file are shown in the text that follows.

The first excerpt of the log file created by MongoDB is shown in Figure 24. Viewing the whole file, including the excerpt, shows that there were no errors reported by the software. Additionally, there were only two warnings. These warnings were, however, only recommendations given by the software. This statement is evident in the use of the “We suggest …” phrase in the warnings. Therefore, the warnings have no ramifications for the operation of the software. In its entirety, the log file shows that the software was set up correctly.

```
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] MongoDB starting : pid=7 port=27018
dbpath=/mnt/mesos/sandbox/mongodb-storage/lib/mongodb 64-bit host=gondor-slave
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] db version v3.4.5
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] git version:
520b8f3092c48d934f0cd78ab5f40fe594f96863
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] OpenSSL version: OpenSSL 1.0.1t  3 May 2016
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] allocator: tcmalloc
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] modules: none
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] build environment:
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten]     distmod: debian81
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten]     distarch: x86_64
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten]     target_arch: x86_64
2017-08-10T06:28:38.476+0000 I CONTROL  [initandlisten] options: { config: "/etc/mongod.conf", net: { bindIp:
"0.0.0.0", port: 27018, ssl: { CAFile: "/etc/rootCA.pem", PEMKeyFile: "/etc/mongod-client.pem",
PEMKeyPassword: "<password>", allowInvalidHostnames: false, clusterFile: "/etc/mongod-cluster.pem",
clusterPassword: "<password>", disabledProtocols: "TLS1_0,TLS1_1", mode: "requireSSL" } }, replication: {
replSetName: "rs0" }, security: { authorization: "enabled", clusterAuthMode: "x509" }, setParameter: {
enableLocalhostAuthBypass: "false" }, sharding: { clusterRole: "shardsvr" }, storage: { dbPath:
"/mnt/mesos/sandbox/mongodb-storage/lib/mongodb", engine: "wiredTiger", journal: { enabled: true } },
systemLog: { destination: "file", logAppend: true, path: "/mnt/mesos/sandbox/mongodb-
storage/log/mongodb/mongod.log" } }
2017-08-10T06:28:38.479+0000 I STORAGE  [initandlisten] wiredtiger_open config:
create,cache_size=3479M,session_max=20000,eviction=(threads_min=4,threads_max=4),config_base=false,sta
tistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=1
00000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0),
2017-08-10T06:28:39.269+0000 I STORAGE  [initandlisten] Starting WiredTigerRecordStoreThread
local.oplog.rs
2017-08-10T06:28:39.269+0000 I STORAGE  [initandlisten] The size storer reports that the oplog contains
3763587 records totaling to 540289746 bytes
2017-08-10T06:28:39.269+0000 I STORAGE  [initandlisten] Sampling from the oplog between Jun 30 10:37:33:1
and Aug 10 05:18:18:1 to determine where to place markers for truncation
2017-08-10T06:28:39.269+0000 I STORAGE  [initandlisten] Taking 10 samples and assuming that each section
of oplog contains approximately 3739772 records totaling to 536870932 bytes
2017-08-10T06:28:39.305+0000 I STORAGE  [initandlisten] Placing a marker at optime Jul 27 09:37:50:353f
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten]
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten] ** WARNING:
/sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten]
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten] ** WARNING:
/sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2017-08-10T06:28:39.370+0000 I CONTROL  [initandlisten]
```

Figure 24: First MongoDB log file excerpt for the shard role showing the software activation

A second excerpt of the log file is shown in Figure 25. The excerpt shows that the sharding process started successfully. During the start of the process, the MongoDB host, from which the log file was taken, successfully connected to the config servers. As can be seen in the figure, three connections were made to config servers. Therefore, the three config servers were functioning correctly with communication established to each one. However, as the config servers were also starting, no primary member had been elected for the config server replica set. Thus, “No primary detected for set csReplSet” messages were displayed frequently. Later, the messages disappeared when the config server replica set elected a primary member.

```
2017-08-10T06:28:39.375+0000 I SHARDING [initandlisten] initializing sharding state with: { _id: "shardIdentity",
configsvrConnectionString: "csReplSet/192.168.0.21:27019,192.168.0.24:27019,192.168.0.27:27019",
shardName: "rs0", clusterId: ObjectId('59563ef59333513bb0e94b5d') }
2017-08-10T06:28:39.375+0000 I SHARDING [initandlisten] first cluster operation detected, adding sharding
hook to enable versioning and authentication to remote servers
2017-08-10T06:28:39.379+0000 I NETWORK  [initandlisten] Starting new replica set monitor for
csReplSet/192.168.0.21:27019,192.168.0.24:27019,192.168.0.27:27019
2017-08-10T06:28:39.380+0000 I SHARDING [thread1] creating distributed lock ping thread for process gondor-
slave:27018:1502346519:-6851901326932144878 (sleeping for 30000ms)
2017-08-10T06:28:39.400+0000 I NETWORK  [ReplicaSetMonitor-TaskExecutor-0] Successfully connected to
192.168.0.21:27019 (1 connections now open to 192.168.0.21:27019 with a 5 second timeout)
2017-08-10T06:28:39.412+0000 I NETWORK  [shard registry reload] Successfully connected to
192.168.0.24:27019 (1 connections now open to 192.168.0.24:27019 with a 5 second timeout)
2017-08-10T06:28:39.412+0000 I NETWORK  [replSetDistLockPinger] Successfully connected to
192.168.0.27:27019 (1 connections now open to 192.168.0.27:27019 with a 5 second timeout)
2017-08-10T06:28:39.413+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:39.414+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.27:27019
2017-08-10T06:28:39.414+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.24:27019
2017-08-10T06:28:39.466+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.27:27019, took 52ms (1 connections now open to 192.168.0.27:27019)
2017-08-10T06:28:39.475+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.24:27019, took 61ms (1 connections now open to 192.168.0.24:27019)
2017-08-10T06:28:39.917+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:40.421+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:40.924+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:41.428+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:41.932+0000 W NETWORK  [replSetDistLockPinger] No primary detected for set csReplSet
2017-08-10T06:28:42.437+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.21:27019
2017-08-10T06:28:42.458+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.21:27019, took 21ms (1 connections now open to 192.168.0.21:27019)
2017-08-10T06:28:47.093+0000 W SHARDING [replSetDistLockPinger] pinging failed for distributed lock pinger
:: caused by :: LockStateChangeFailed: findAndModify query predicate didn't match any lock document
```

Figure 25: Second MongoDB log file excerpt for the shard role showing the start of the sharding process

The functioning of the replication process is shown in the log file excerpt of Figure 26. The excerpt shows that the replication process failed at first. The reason being that no primary member had been elected for the replica set. The replication process successfully started at 2017-08-10T06:28:47.538 in the log file. After this time, various information messages were displayed regarding the replication process. The message “transition to SECONDARY” is of importance. The secondary state means that the MongoDB instance is eligible to be elected as a primary, i.e. master [133]. This state also means that the MongoDB instance is replicating the primary member’s data. The rest of the excerpt shows that communication with the remaining two replica set members was established. Both the replica set members were found in the secondary state.

The situation described in the previous paragraph may seem incorrect for it was stated that the MongoDB host, whose log file is being shown, should be the master of the replica set. The information shown in Figure 26 indicates that all replica set members are in the secondary state. This occurrence is, however, the natural state of progression for the replica set, as described in the MongoDB server documentation [133].

Another important message was displayed at 2017-08-10T06:28:47.458 in Figure 26. The message states that the host is listening for connections that are secured with SSL. This message verifies that SSL was set up correctly. The fact that connections were established with this and the other hosts, further confirms that SSL security was operational.

```
2017-08-10T06:28:47.095+0000 I NETWORK  [shard registry reload] Starting new replica set monitor for
rs0/192.168.0.21:27018,192.168.0.24:27018,192.168.0.27:27018
2017-08-10T06:28:47.095+0000 W NETWORK  [ReplicaSetMonitor-TaskExecutor-0] Failed to connect to
192.168.0.21:27018, in(checking socket for error after poll), reason: Connection refused
2017-08-10T06:28:47.096+0000 W NETWORK  [ReplicaSetMonitor-TaskExecutor-0] Failed to connect to
192.168.0.24:27018, in(checking socket for error after poll), reason: Connection refused
2017-08-10T06:28:47.097+0000 W NETWORK  [ReplicaSetMonitor-TaskExecutor-0] Failed to connect to
192.168.0.27:27018, in(checking socket for error after poll), reason: Connection refused
2017-08-10T06:28:47.097+0000 W NETWORK  [ReplicaSetMonitor-TaskExecutor-0] No primary detected for set
rs0
2017-08-10T06:28:47.097+0000 I NETWORK  [ReplicaSetMonitor-TaskExecutor-0] All nodes for set rs0 are
down. This has happened for 1 checks in a row.
2017-08-10T06:28:47.456+0000 I FTDC     [initandlisten] Initializing full-time diagnostic data capture with
directory '/mnt/mesos/sandbox/mongodb-storage/lib/mongodb/diagnostic.data'
2017-08-10T06:28:47.458+0000 I NETWORK  [thread2] waiting for connections on port 27018 ssl
2017-08-10T06:28:47.459+0000 W NETWORK  [replExecDBWorker-2] Failed to connect to 192.168.0.27:27018,
in(checking socket for error after poll), reason: Connection refused
2017-08-10T06:28:47.538+0000 I REPL     [replExecDBWorker-2] New replica set config in use: { _id: "rs0",
version: 1, protocolVersion: 1, members: [ { _id: 0, host: "192.168.0.21:27018", arbiterOnly: false, buildIndexes:
true, hidden: false, priority: 100.0, tags: {}, slaveDelay: 0, votes: 1 }, { _id: 1, host: "192.168.0.27:27018",
arbiterOnly: false, buildIndexes: true, hidden: false, priority: 1.0, tags: {}, slaveDelay: 0, votes: 1 }, { _id: 2, host:
"192.168.0.24:27018", arbiterOnly: false, buildIndexes: true, hidden: false, priority: 1.0, tags: {}, slaveDelay: 0,
votes: 1 } ], settings: { chainingAllowed: true, heartbeatIntervalMillis: 2000, heartbeatTimeoutSecs: 10,
electionTimeoutMillis: 10000, catchUpTimeoutMillis: 2000, getLastErrorModes: {}, getLastErrorDefaults: { w: 1,
wtimeout: 0 }, replicaSetId: ObjectId('595629ed33028def96d94d7a') } }
2017-08-10T06:28:47.538+0000 I REPL     [replExecDBWorker-2] This node is 192.168.0.21:27018 in the config
2017-08-10T06:28:47.538+0000 I REPL     [replExecDBWorker-2] transition to STARTUP2
2017-08-10T06:28:47.538+0000 I REPL     [replExecDBWorker-2] Starting replication storage threads
2017-08-10T06:28:47.541+0000 I REPL     [replExecDBWorker-2] Starting replication fetcher thread
2017-08-10T06:28:47.541+0000 I REPL     [replExecDBWorker-2] Starting replication applier thread
2017-08-10T06:28:47.541+0000 I REPL     [replExecDBWorker-2] Starting replication reporter thread
2017-08-10T06:28:47.542+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.27:27018
2017-08-10T06:28:47.542+0000 I REPL     [rsSync] transition to RECOVERING
2017-08-10T06:28:47.542+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.24:27018
2017-08-10T06:28:47.544+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Failed to connect to
192.168.0.27:27018 - HostUnreachable: Connection refused
2017-08-10T06:28:47.544+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Dropping all pooled
connections to 192.168.0.27:27018 due to failed operation on a connection
2017-08-10T06:28:47.545+0000 I REPL     [ReplicationExecutor] Error in heartbeat request to
192.168.0.27:27018; HostUnreachable: Connection refused
2017-08-10T06:28:47.545+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.27:27018
2017-08-10T06:28:47.545+0000 I REPL     [rsSync] transition to SECONDARY
2017-08-10T06:28:47.614+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.24:27018, took 72ms (1 connections now open to 192.168.0.24:27018)
2017-08-10T06:28:47.615+0000 I REPL     [ReplicationExecutor] Member 192.168.0.24:27018 is now in state
SECONDARY
2017-08-10T06:28:52.580+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.27:27018
2017-08-10T06:28:52.656+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.27:27018, took 77ms (1 connections now open to 192.168.0.27:27018)
2017-08-10T06:28:52.657+0000 I REPL     [ReplicationExecutor] Member 192.168.0.27:27018 is now in state
SECONDARY
```

Figure 26: Third MongoDB log file excerpt for the shard role showing the start of the replication process

Figure 27 shows a final excerpt of the MongoDB log file. The excerpt is at a later stage, and the intervening information is not shown. The first message in the excerpt shows that the election of the primary member was initiated. The election progressed without error and finished at 2017-08-10T06:28:57.781. After the election, a "transition to PRIMARY" message was displayed. This message indicates that the host of the log file was chosen as the primary member. Following this message, the host started the process of assuming the primary role. The transitioning to the primary role was completed at the last message of the excerpt. At that time, the replica set was fully functioning.

```
2017-08-10T06:28:57.689+0000 I REPL     [ReplicationExecutor] Starting an election, since we've seen no
PRIMARY in the past 10000ms
2017-08-10T06:28:57.689+0000 I REPL     [ReplicationExecutor] conducting a dry run election to see if we could
be elected
2017-08-10T06:28:57.691+0000 I REPL     [ReplicationExecutor] VoteRequester(term 13 dry run) received a yes
vote from 192.168.0.24:27018; response message: { term: 13, voteGranted: true, reason: "", ok: 1.0 }
2017-08-10T06:28:57.691+0000 I REPL     [ReplicationExecutor] dry election run succeeded, running for election
2017-08-10T06:28:57.781+0000 I REPL     [ReplicationExecutor] VoteRequester(term 14) received a yes vote
from 192.168.0.27:27018; response message: { term: 14, voteGranted: true, reason: "", ok: 1.0 }
2017-08-10T06:28:57.781+0000 I REPL     [ReplicationExecutor] election succeeded, assuming primary role in
term 14
2017-08-10T06:28:57.781+0000 I REPL     [ReplicationExecutor] transition to PRIMARY
2017-08-10T06:28:57.781+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.24:27018
2017-08-10T06:28:57.781+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.27:27018
2017-08-10T06:28:57.781+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.24:27018
2017-08-10T06:28:57.781+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Ending connection to host
192.168.0.24:27018 due to bad connection status; 2 connections to that host remain open
2017-08-10T06:28:57.856+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.27:27018, took 75ms (2 connections now open to 192.168.0.27:27018)
2017-08-10T06:28:57.869+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.24:27018, took 88ms (2 connections now open to 192.168.0.24:27018)
2017-08-10T06:28:57.870+0000 I REPL     [ReplicationExecutor] My optime is most up-to-date, skipping catch-
up and completing transition to primary.
2017-08-10T06:28:57.873+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.24:27018, took 92ms (2 connections now open to 192.168.0.24:27018)
2017-08-10T06:28:59.548+0000 I REPL     [rsSync] transition to primary complete; database writes are now
permitted
```

Figure 27: Final MongoDB log file excerpt for the shard role showing the completion of the replication process

The second verification set was that of the config server role. The MongoDB config servers were also configured as a replica set. This verification set used the log file created by the master of the replica set, which was a computer different from the master of the sharded replica set.

The first excerpt of the log file is shown in Figure 28. This excerpt shows the same warnings as were given to the sharded replica set. No errors were shown in the file, including the excerpt, which means the software was set up correctly. The same message verifying the SSL operation was shown at the end of the excerpt. Therefore, SSL was fully operational, which was verified again when successful communication was established in the next excerpt.

```
2017-08-10T06:28:29.629+0000 I CONTROL  [initandlisten] MongoDB starting : pid=7 port=27019
dbpath=/mnt/mesos/sandbox/mongodb-config-server-storage/lib/mongodb_configServer 64-bit host=rohan-slave
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] db version v3.4.5
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] git version:
520b8f3092c48d934f0cd78ab5f40fe594f96863
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] OpenSSL version: OpenSSL 1.0.1t  3 May 2016
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] allocator: tcmalloc
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] modules: none
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] build environment:
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten]     distmod: debian81
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten]     distarch: x86_64
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten]     target_arch: x86_64
2017-08-10T06:28:29.630+0000 I CONTROL  [initandlisten] options: { config: "/etc/mongod_configServer.conf",
net: { bindIp: "0.0.0.0", port: 27019, ssl: { CAFile: "/etc/rootCA.pem", PEMKeyFile: "/etc/mongod_configServer-
client.pem", PEMKeyPassword: "<password>", allowInvalidHostnames: false, clusterFile:
"/etc/mongod_configServer-cluster.pem", clusterPassword: "<password>", disabledProtocols: "TLS1_0,TLS1_1",
mode: "requireSSL" } }, replication: { replSetName: "csReplSet" }, security: { authorization: "enabled",
clusterAuthMode: "x509" }, setParameter: { enableLocalhostAuthBypass: "false" }, sharding: { clusterRole:
"configsvr" }, storage: { dbPath: "/mnt/mesos/sandbox/mongodb-config-server-
storage/lib/mongodb_configServer", engine: "wiredTiger", journal: { enabled: true } }, systemLog: { destination:
"file", logAppend: true, path: "/mnt/mesos/sandbox/mongodb-config-server-
storage/log/mongodb_configServer/mongod_configServer.log" } }
2017-08-10T06:28:29.659+0000 I STORAGE  [initandlisten] wiredtiger_open config:
create,cache_size=3479M,session_max=20000,eviction=(threads_min=4,threads_max=4),config_base=false,sta
tistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=1
00000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0),
2017-08-10T06:28:30.881+0000 I STORAGE  [initandlisten] Starting WiredTigerRecordStoreThread
local.oplog.rs
2017-08-10T06:28:30.881+0000 I STORAGE  [initandlisten] The size storer reports that the oplog contains
931063 records totaling to 158066525 bytes
2017-08-10T06:28:30.889+0000 I STORAGE  [initandlisten] Sampling from the oplog between Jun 30 12:06:59:1
and Aug 10 05:18:26:1 to determine where to place markers for truncation
2017-08-10T06:28:30.889+0000 I STORAGE  [initandlisten] Taking 2 samples and assuming that each section of
oplog contains approximately 3162344 records totaling to 536871003 bytes
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten]
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten] ** WARNING:
/sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten]
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten] ** WARNING:
/sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2017-08-10T06:28:31.089+0000 I CONTROL  [initandlisten]
2017-08-10T06:28:31.149+0000 I NETWORK  [thread2] waiting for connections on port 27019 ssl
```

Figure 28: First MongoDB log file excerpt for the config server role showing the software activation

The second log file excerpt, as shown in Figure 29, shows the successful start of the replication process. The host of the log file successfully transitioned to the secondary state at 2017-08-10T06:28:31.278. By the end of the excerpt, both remaining config servers were successfully detected, and communication was established with them. Both these config servers were in the correct state, i.e. the secondary state. Consequently, the replication process was started correctly; however, a primary or master still had to be elected. The successful establishment of communication between the replica set members further verifies the correct operation of SSL.

```
2017-08-10T06:28:31.272+0000 I REPL     [replExecDBWorker-0] This node is 192.168.0.27:27019 in the config
2017-08-10T06:28:31.272+0000 I REPL     [replExecDBWorker-0] transition to STARTUP2
2017-08-10T06:28:31.272+0000 I REPL     [replExecDBWorker-0] Starting replication snapshot thread
2017-08-10T06:28:31.273+0000 I REPL     [replExecDBWorker-0] Starting replication storage threads
2017-08-10T06:28:31.275+0000 I REPL     [replExecDBWorker-0] Starting replication fetcher thread
2017-08-10T06:28:31.275+0000 I REPL     [replExecDBWorker-0] Starting replication applier thread
2017-08-10T06:28:31.275+0000 I REPL     [replExecDBWorker-0] Starting replication reporter thread
2017-08-10T06:28:31.276+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.21:27019
2017-08-10T06:28:31.276+0000 I REPL     [rsSync] transition to RECOVERING
2017-08-10T06:28:31.276+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
192.168.0.24:27019
2017-08-10T06:28:31.278+0000 I REPL     [rsSync] transition to SECONDARY
2017-08-10T06:28:31.358+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.24:27019, took 82ms (1 connections now open to 192.168.0.24:27019)
2017-08-10T06:28:31.359+0000 I REPL     [ReplicationExecutor] Member 192.168.0.24:27019 is now in state
SECONDARY
2017-08-10T06:28:31.374+0000 I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully connected to
192.168.0.21:27019, took 98ms (1 connections now open to 192.168.0.21:27019)
2017-08-10T06:28:31.375+0000 I REPL     [ReplicationExecutor] Member 192.168.0.21:27019 is now in state
SECONDARY
```

Figure 29: Second MongoDB log file excerpt for the config server role showing the start of the replication process

The final excerpt of the log file is given in Figure 30. This excerpt shows that one of the other config servers elected itself as primary. This occurrence was detected in the first message of the excerpt. As a result, the host forced an election and was elected as the primary member at 2017-08-10T06:28:51.876. The host completed its transition to primary at the end of the excerpt. At that stage, the replica set was fully functioning with no errors detected.

```
2017-08-10T06:28:41.379+0000 I REPL     [ReplicationExecutor] Member 192.168.0.21:27019 is now in state
PRIMARY
2017-08-10T06:28:41.379+0000 I REPL     [ReplicationExecutor] Scheduling priority takeover at 2017-08-
10T06:28:51.772+0000
2017-08-10T06:28:51.772+0000 I REPL     [ReplicationExecutor] Canceling priority takeover callback
2017-08-10T06:28:51.772+0000 I REPL     [ReplicationExecutor] Starting an election for a priority takeover
2017-08-10T06:28:51.772+0000 I REPL     [ReplicationExecutor] conducting a dry run election to see if we could
be elected
2017-08-10T06:28:51.774+0000 I REPL     [ReplicationExecutor] VoteRequester(term 12 dry run) received a yes
vote from 192.168.0.21:27019; response message: { term: 12, voteGranted: true, reason: "", ok: 1.0 }
2017-08-10T06:28:51.774+0000 I REPL     [ReplicationExecutor] dry election run succeeded, running for election
2017-08-10T06:28:51.876+0000 I REPL     [ReplicationExecutor] VoteRequester(term 13) received a yes vote
from 192.168.0.24:27019; response message: { term: 13, voteGranted: true, reason: "", ok: 1.0 }
2017-08-10T06:28:51.876+0000 I REPL     [ReplicationExecutor] election succeeded, assuming primary role in
term 13
2017-08-10T06:28:51.876+0000 I REPL     [ReplicationExecutor] transition to PRIMARY
2017-08-10T06:28:53.360+0000 I REPL     [rsSync] transition to primary complete; database writes are now
permitted
```

Figure 30: Final MongoDB log file excerpt for the config server role showing the completion of the replication process

The MongoDB router role was the last to be verified and was also the simplest to verify. There were no special processes that needed to function correctly. Therefore, its verification would show the correct functioning of the software, and the detection of the config server replica set and master of the sharded replica set.

The first log file excerpt is shown in Figure 31. This excerpt and the file show that there were no errors or warnings. It should be clear that the mongos router program works differently than the mongod program, as used for the config server and sharded replica sets. Thus, the software was set up correctly.

```
2017-08-10T06:28:51.139+0000 I SHARDING [mongosMain] mongos version v3.4.5
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] git version:
520b8f3092c48d934f0cd78ab5f40fe594f96863
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] OpenSSL version: OpenSSL 1.0.1t  3 May 2016
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] allocator: tcmalloc
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] modules: none
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] build environment:
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     distmod: debian81
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     distarch: x86_64
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     target_arch: x86_64
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] db version v3.4.5
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] git version:
520b8f3092c48d934f0cd78ab5f40fe594f96863
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] OpenSSL version: OpenSSL 1.0.1t  3 May 2016
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] allocator: tcmalloc
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] modules: none
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] build environment:
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     distmod: debian81
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     distarch: x86_64
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain]     target_arch: x86_64
2017-08-10T06:28:51.139+0000 I CONTROL  [mongosMain] options: { config: "/etc/mongos.conf", net: { bindIp:
"0.0.0.0", port: 27017, ssl: { CAFile: "/etc/rootCA.pem", PEMKeyFile: "/etc/mongos-client.pem",
PEMKeyPassword: "<password>", allowInvalidHostnames: false, clusterFile: "/etc/mongos-cluster.pem",
clusterPassword: "<password>", disabledProtocols: "TLS1_0,TLS1_1", mode: "requireSSL" } }, security: {
clusterAuthMode: "x509" }, setParameter: { enableLocalhostAuthBypass: "false" }, sharding: { configDB:
"csReplSet/192.168.0.21:27019,192.168.0.27:27019,192.168.0.24:27019" }, systemLog: { destination: "file",
logAppend: true, path: "/mnt/mesos/sandbox/mongos-storage/log/mongos/mongos.log" } }
2017-08-10T06:28:51.145+0000 I NETWORK  [mongosMain] Starting new replica set monitor for
csReplSet/192.168.0.21:27019,192.168.0.24:27019,192.168.0.27:27019
2017-08-10T06:28:51.146+0000 I SHARDING [thread1] creating distributed lock ping thread for process gondor-
slave:27017:1502346531:-4198709130731404243 (sleeping for 30000ms)
```

Figure 31: First MongoDB log file excerpt for the router role showing the software activation

The final excerpt, as given in Figure 32, shows that the host connected successfully to each of the config servers. It also shows that communication was successfully established with the master of the sharded replica set. The message showing the correct activation of SSL is given at 2017-08-10T06:28:51.271 in the excerpt. The final message in the excerpt shows that communication with the entire config server replica set was successful. Therefore, it can be concluded that the MongoDB router functioned correctly with SSL enabled.

```
2017-08-10T06:28:51.208+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.24:27019
2017-08-10T06:28:51.250+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.24:27019, took 49ms (1 connections now open to 192.168.0.24:27019)
2017-08-10T06:28:51.271+0000 I NETWORK  [thread2] waiting for connections on port 27017 ssl
2017-08-10T06:40:28.492+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.21:27018
2017-08-10T06:40:28.511+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.21:27018, took 19ms (1 connections now open to 192.168.0.21:27018)
2017-08-10T06:40:52.937+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.21:27019
2017-08-10T06:40:52.938+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Connecting to
192.168.0.27:27019
2017-08-10T06:40:52.989+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.21:27019, took 52ms (2 connections now open to 192.168.0.21:27019)
2017-08-10T06:40:53.004+0000 I ASIO     [NetworkInterfaceASIO-ShardRegistry-0] Successfully connected to
192.168.0.27:27019, took 66ms (2 connections now open to 192.168.0.27:27019)
2017-08-10T06:41:13.944+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-0-0] Connecting to
192.168.0.27:27019
2017-08-10T06:41:13.996+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-0-0] Successfully
connected to 192.168.0.27:27019, took 53ms (1 connections now open to 192.168.0.27:27019)
2017-08-10T06:41:14.001+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-1-0] Connecting to
192.168.0.27:27019
2017-08-10T06:41:14.040+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-1-0] Successfully
connected to 192.168.0.27:27019, took 39ms (1 connections now open to 192.168.0.27:27019)
2017-08-10T06:41:14.046+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-2-0] Connecting to
192.168.0.27:27019
2017-08-10T06:41:14.094+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-2-0] Successfully
connected to 192.168.0.27:27019, took 48ms (1 connections now open to 192.168.0.27:27019)
2017-08-10T06:41:14.128+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-3-0] Connecting to
192.168.0.27:27019
2017-08-10T06:41:14.206+0000 I ASIO     [NetworkInterfaceASIO-TaskExecutorPool-3-0] Successfully
connected to 192.168.0.27:27019, took 78ms (1 connections now open to 192.168.0.27:27019)
2017-08-10T06:41:14.289+0000 I NETWORK  [conn16] Successfully connected to
csReplSet/192.168.0.21:27019,192.168.0.24:27019,192.168.0.27:27019 (1 connections now open to
csReplSet/192.168.0.21:27019,192.168.0.24:27019,192.168.0.27:27019 with a 0 second timeout)
```

Figure 32: Final MongoDB log file excerpt for the router role showing the operation of the software

The excerpts shown above proved that there were no faults with the set-up of each MongoDB role. All the MongoDB software functioned correctly. Replication, sharding, and SSL were shown to function correctly as required for the different roles.

The final part of the verification was to test if data could be written to, read from, and aggregated by the complete MongoDB subsystem. For this purpose, a Python program was written; its source code is given in Appendix 2. The program started by inserting one tag into its own collection. The tag was immediately retrieved via its ID field to ensure it was stored and retrieved correctly. After this, the program wrote three sets of value data to a separate value collection. In total, 33 values were stored in the value collection with each one linked to the single tag inserted previously. After writing the value data, the program verified that MongoDB had returned 33 ID numbers. The significance of this is that an ID number is only returned for a successful insertion.

Table 10 shows an example of one value data set that was used. The first column shows the date and time for which the value was created. The second column shows the value, i.e. measurement, of the tag. The

third column shows the interval over which each aggregation operation was performed. The remaining columns show the respective expected minimum, maximum, and summation results for the corresponding interval.

Table 10: Example value data set with expected results for three aggregation operations

Date and time	Value	Interval time	Minimum	Maximum	Sum
2016-01-01 00:10	1	00:00	1	5	6
2016-01-01 00:20	5				
2016-01-01 00:30	3	00:30	3	9	16
2016-01-01 00:40	4				
2016-01-01 00:50	9				
2016-01-01 01:00	17	01:00	0	17	22
2016-01-01 01:10	0				
2016-01-01 01:20	5				
2016-01-01 01:30	0	01:30	0	4	5
2016-01-01 01:40	1				
2016-01-01 01:50	4				
2016-01-01 02:00	7	02:00	2	9	18
2016-01-01 02:10	9				
2016-01-01 02:20	2				
2016-01-01 02:30	1	02:30	1	1	1

The final part of the program retrieved and aggregated each set of value data in multiple ways. A minimum, maximum, and summation operation were performed for each set. Upon completion, the program verified that the computed result was equal to the expected result. The computed results of the example data set are shown in Table 11. The table shows the expected results again for effortless comparison.

Table 11: Expected and computed aggregation results of the example value data set

Expected minimum	Computed minimum	Expected maximum	Computed maximum	Expected sum	Computed sum
1	1	5	5	6	6
3	3	9	9	16	16
0	0	17	17	22	22
0	0	4	4	5	5
2	2	9	9	18	18
1	1	1	1	1	1

From Table 10 and Table 11, it is clear that the MongoDB system aggregated the example value data correctly. The minimum, maximum, and summation operations yielded the expected results. Therefore, the data was stored, retrieved, and processed correctly. Consequently, the MongoDB system in its entirety satisfied all its specifications and functioned correctly.

4.3.3 Apache Spark verification

The role of Apache Spark, as stated in Section 3.4.2, was to provide data processing and analysis functionality. It fulfilled this role by providing integration with MongoDB and Python. By allowing access to Spark through Python, many people could make use of its features. This statement is true because Python is a programming language that is easy to learn and to use, and it is one of the most popular languages. The potential user pool would, therefore, be large. In contrast, there would be a smaller user pool if a more specialised programming language was used.

The verification of Spark focused on confirming the correct operation thereof. This verification was performed in two parts. The first part focused on the set-up of Spark. The second part focused on the successful execution of two Python programs.

The first verification part used the Spark master web interface to confirm the cluster set-up. As was mentioned in Section 3.4.2, Spark was deployed with one master and three slave nodes. The web interface showed the operational status of each Spark slave node. Upon visiting the web interface, the screenshot shown in Figure 33 was displayed.

The fact that the Spark master web interface was visited meant that the master node was already operating correctly. Figure 33 shows that all slave nodes were “ALIVE”, which means they were fully operational. The figure further shows that Spark detected 12 CPU cores and 20.4 GB memory. These resources detected by Spark are the total resources for all slave nodes. Busy and completed applications would also be shown on the web interface. However, because Spark was started just before Figure 33 was created, there were no busy or completed applications. Therefore, the corresponding applications sections shown in Figure 33 were empty.

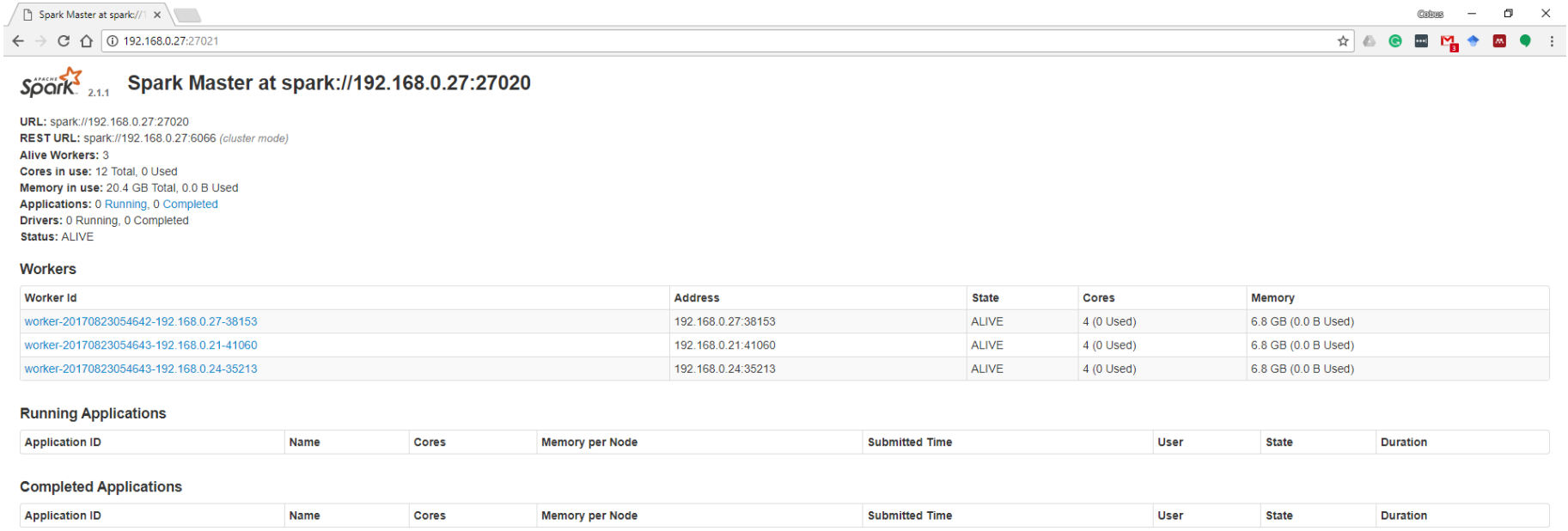


Figure 33: Apache Spark master web interface showing cluster information

For the second verification part, a pi computing Python program was executed first. The pi program, also called PythonPi, was created by the Spark creators. It is provided as an example to test Spark's execution of Python programs. The PythonPi program was submitted to Spark for execution. An excerpt of the output generated is given in Figure 34. The complete output is given in Appendix 3.

```
17/08/23 06:06:40 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
17/08/23 06:06:41 INFO DAGScheduler: ResultStage 0 (reduce at
/usr/local/spark/examples/src/main/python/pi.py:43) finished in 8.580 s
17/08/23 06:06:41 INFO DAGScheduler: Job 0 finished: reduce at
/usr/local/spark/examples/src/main/python/pi.py:43, took 11.616557 s
Pi is roughly 3.140856
17/08/23 06:06:43 INFO SparkUI: Stopped Spark web UI at http://192.168.0.27:4040
17/08/23 06:06:43 INFO StandaloneSchedulerBackend: Shutting down all executors
17/08/23 06:06:43 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Asking each executor to shut down
17/08/23 06:06:43 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
17/08/23 06:06:44 INFO MemoryStore: MemoryStore cleared
17/08/23 06:06:44 INFO BlockManager: BlockManager stopped
17/08/23 06:06:44 INFO BlockManagerMaster: BlockManagerMaster stopped
17/08/23 06:06:44 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
17/08/23 06:06:44 INFO SparkContext: Successfully stopped SparkContext
17/08/23 06:06:54 INFO ShutdownHookManager: Shutdown hook called
17/08/23 06:06:54 INFO ShutdownHookManager: Deleting directory /mnt/mesos/sandbox/spark-
storage/local/spark-b0296a7d-84fd-4415-9908-0fecb71a419b/pyspark-982ac810-b8bb-4b1f-89d4-c397de4662ea
17/08/23 06:06:54 INFO ShutdownHookManager: Deleting directory /mnt/mesos/sandbox/spark-
storage/local/spark-b0296a7d-84fd-4415-9908-0fecb71a419b
```

Figure 34: Excerpt of Spark PythonPi program output

Figure 34 shows mostly technical details, most of which are information messages. The sixth message in the figure shows the final output of the PythonPi program. "Pi is roughly …", shows that the program completed successfully by providing an approximate value for pi. The excerpt, as well as the complete output given in Appendix 3, showed no error or warning messages. Thus, Spark executed the program successfully. For additional verification, the master web interface was visited again. The output displayed is illustrated in the screenshot of Figure 35.

One entry is shown in the "Completed Applications" section of Figure 35. The entry was created for the PythonPi program that had been executed previously. The state of the program was given as "FINISHED", which means that the program did indeed complete successfully. Therefore, Figure 35 serves as another method of verifying Spark's correct operation.

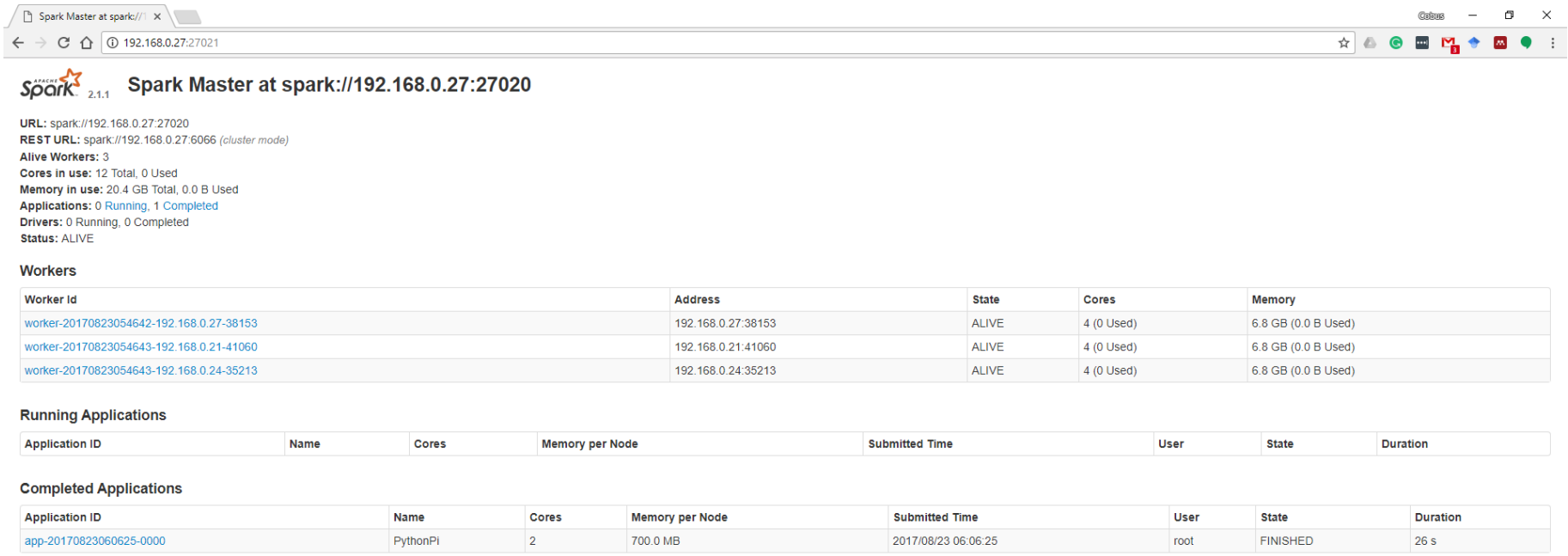


Figure 35: Apache Spark master web interface showing completed Python program

The second Python program submitted to Spark was the one created for the MongoDB verification. This program was described in Section 4.3.2. By submitting this program to Spark, it was established whether Spark could access MongoDB. The Python program was submitted to Spark and completed successfully. An excerpt of the output displayed by the program is shown in Figure 36.

```
Main class:
org.apache.spark.deploy.PythonRunner
Arguments:
file:/mnt/mesos/sandbox/spark-storage/mongo_test_Spark.py
null
System properties:
spark.executor.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.executor.memory -> 700M
spark.driver.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.cores.max -> 2
SPARK_SUBMIT -> true
spark.files -> file:/mnt/mesos/sandbox/spark-storage/mongo_test_Spark.py
spark.app.name -> mongo_test_Spark.py
spark.jars -> file:/usr/local/spark/jars/mongo-spark-connector.jar,file:/usr/local/spark/jars/mongo-java-driver.jar
spark.submit.deployMode -> client
spark.master -> spark://192.168.0.27:27020
spark.driver.extraClassPath -> /usr/local/spark/jars/mongo-spark-connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Classpath elements:
/usr/local/spark/jars/mongo-spark-connector.jar
/usr/local/spark/jars/mongo-java-driver.jar


Starting tests...
Finished tests.
```

Figure 36: Excerpt of MongoDB Python program output

Figure 36 shows the messages that were displayed by Spark upon executing the MongoDB Python program. The two information messages displayed by the program are shown at the end of the excerpt. By displaying these two messages, the program successfully tested MongoDB as described in Section 4.3.2.

The first verification completed for Spark showed that Spark was set up correctly. It also showed that the Spark cluster functioned correctly. The second verification part showed that Spark could execute Python programs. Secondly, it confirmed that Spark could read from and write to MongoDB. Therefore, Spark functioned correctly in its entirety, and it achieved its specifications.

4.3.4 API verification

The API had to achieve two specifications. The first specification was external system integration, which was a more implicit specification. This specification was satisfied by designing the API as a web service based on SOAP, as discussed in Section 3.4.3. Part of the specification was, however, verified with the verification of the second specification. The second specification was data protection. This specification was physically verifiable and was tested with a C# program.

The C# program used for verification was created during the development of the API web service software. This program, in software engineering terms, was an integration test for the complete API. What this means is that the program tested all the functions of the API externally. Thus, the program used the API in the same way another application or system would use it. Unit tests, which test all functions internally, were also created. These unit tests were used during implementation to correct any faults.

The execution of the C# program was not enough to verify if data was protected. Therefore, third-party software was used to inspect the communication between the C# program and the API web service during the execution of the program. The third-party software used was Wireshark [134]. An excerpt of the communication between the software is shown in Figure 37.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Client	Server	TCP	66	50114 → 8011 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.000991	Server	Client	TCP	66	8011 → 50114 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.001071	Client	Server	TCP	54	50114 → 8011 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4	0.001610	Client	Server	TLSv1.2	201	Client Hello
5	0.019195	Server	Client	TCP	1514	8011 → 50114 [ACK] Seq=1 Ack=148 Win=131328 Len=1460 [TCP segment of a reassembled PDU]
6	0.019195	Server	Client	TLSv1.2	1171	Server Hello, Certificate, Server Key Exchange, Server Hello Done
7	0.019280	Client	Server	TCP	54	50114 → 8011 [ACK] Seq=148 Ack=2578 Win=65536 Len=0
8	0.021670	Client	Server	TLSv1.2	268	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
9	0.025649	Server	Client	TLSv1.2	161	Change Cipher Spec, Encrypted Handshake Message
10	0.028227	Client	Server	TLSv1.2	347	Application Data
11	0.029623	Server	Client	TLSv1.2	155	Application Data
12	0.029787	Client	Server	TCP	1514	50114 → 8011 [ACK] Seq=655 Ack=2786 Win=65280 Len=1460 [TCP segment of a reassembled PDU]
13	0.029799	Client	Server	TCP	1514	50114 → 8011 [ACK] Seq=2115 Ack=2786 Win=65280 Len=1460 [TCP segment of a reassembled PDU]
14	0.029806	Client	Server	TCP	1514	50114 → 8011 [ACK] Seq=3575 Ack=2786 Win=65280 Len=1460 [TCP segment of a reassembled PDU]
15	0.029813	Client	Server	TCP	1514	50114 → 8011 [ACK] Seq=5035 Ack=2786 Win=65280 Len=1460 [TCP segment of a reassembled PDU]
16	0.029820	Client	Server	TLSv1.2	155	Application Data
17	0.031280	Server	Client	TCP	60	8011 → 50114 [ACK] Seq=2786 Ack=3575 Win=131328 Len=0
18	0.031280	Server	Client	TCP	60	8011 → 50114 [ACK] Seq=2786 Ack=6495 Win=131328 Len=0
19	0.031281	Server	Client	TLSv1.2	139	Encrypted Handshake Message
20	0.031755	Client	Server	TLSv1.2	283	Encrypted Handshake Message
21	0.049717	Server	Client	TCP	1514	8011 → 50114 [ACK] Seq=2871 Ack=6825 Win=130816 Len=1460 [TCP segment of a reassembled PDU]
22	0.049723	Server	Client	TCP	1514	8011 → 50114 [ACK] Seq=4331 Ack=6825 Win=130816 Len=1460 [TCP segment of a reassembled PDU]
23	0.049724	Server	Client	TLSv1.2	1363	Encrypted Handshake Message
24	0.049945	Client	Server	TCP	54	50114 → 8011 [ACK] Seq=6825 Ack=7100 Win=65536 Len=0

Figure 37: Wireshark capture of TLS messages exchanged between client and server

The sequence of events illustrated in Figure 37 requires some background to be understood. One needs to be aware of the TLS 1.2 protocol and its handshake protocol [135]. This protocol will not be discussed in detail, but a brief explanation follows.

For data to be encrypted via TLS 1.2, the handshake protocol must complete successfully. Upon its completion, all network traffic between two network entities will be encrypted. In this case, the client entity was the C# program, and the server was the API. The sequence of messages required to complete the handshake protocol is given in Table 12.

Table 12: Messages sent between client and server for the TLS 1.2 handshake protocol

Client sends	Server sends
1. Client hello	
	2. Server hello
	3. Optional: Certificate or Server key exchange or Certificate request
	4. Server hello done
5. Optional: Certificate	
6. Client key exchange	
7. Optional: Certificate verify	
8. Change cipher spec	
Finished	
	9. Change cipher spec
	Finished

When comparing the required messages of Table 12 with the messages sent in Figure 37, it is evident that the TLS 1.2 handshake protocol completed successfully. The C# program sent a Client hello message (No. 4 in Figure 37). The API service responded with the Server hello, Certificate, Server key exchange, and Server hello done messages (No. 6 in Figure 37). After this message, the C# program sent the Client key exchange and Change cipher spec messages before finishing (No. 8 in Figure 37). Finally, the API service sent its Change cipher spec message and finished (No. 9 in Figure 37).

Upon finishing the handshake protocol, the C# program and API service exchanged data securely. This situation is evident in message No. 10 and onward in Figure 37. It is also worth noting that Wireshark reported TLS 1.2 as the communication protocol, as shown in the protocol column of the figure. Therefore, TLS 1.2 was used successfully by the API.

4.3.5 User interface verification

The user interface service was the final component verified. Its only specification was to provide user access to the services of the system. To comply with this specification, it had to provide a functioning web interface connected to Python, MongoDB, and Apache Spark. It was stated in Section 3.4.3 that Apache Zeppelin was used for the user interface service. Apache Zeppelin already provided a web interface, and therefore all that remained was to verify its operation.

The web interface of Zeppelin was visited with a screenshot thereof shown in Figure 38. The successful display of the welcome screen allowed for a notebook to be created. After creating a notebook, the Python program, as mentioned in Sections 4.3.2 and 4.3.3, was used to verify Zeppelin's integration with Python, Spark, and MongoDB. The Python program was slightly modified for execution in Zeppelin. Zeppelin did

not execute the Python program itself. Instead, it submitted the program to Spark, which performed the execution. The output displayed upon execution of the program is shown in Figure 39.

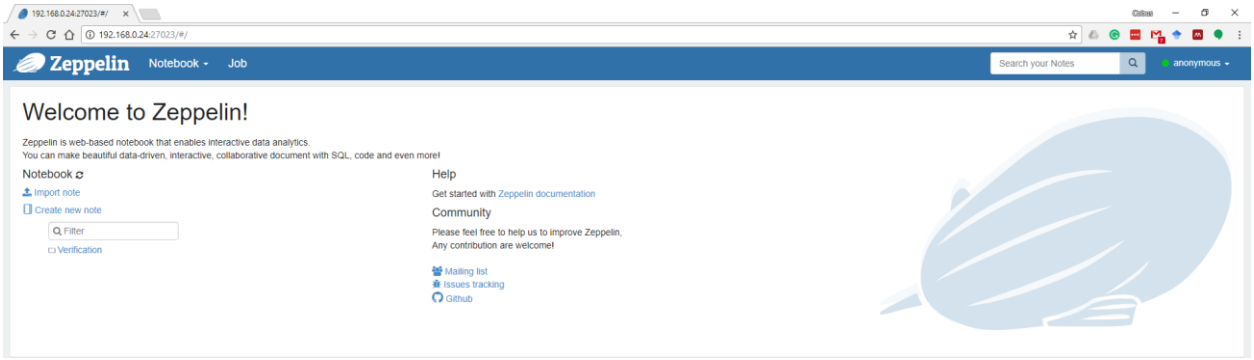


Figure 38: Apache Zeppelin web interface welcome screen

Figure 39 shows the Zeppelin notebook with the information messages displayed by the Python program. These information messages were also seen during the verification of Spark, as discussed in Section 4.3.3. Therefore, the Python program once again executed correctly. Furthermore, Python, Spark, and MongoDB were again verified for correct operation. Consequently, Zeppelin was successfully verified and achieved the specification set for it.

Before commencing with validation, it must be stated that a section for Python was not given. The reason is that by verifying the other components, Python was also verified. Thus, Python was implicitly verified because no verification could have used Python if Python did not function correctly. Furthermore, Python was not configured or modified such as the other software. Thus, the functioning of Python was outside the control of the author.

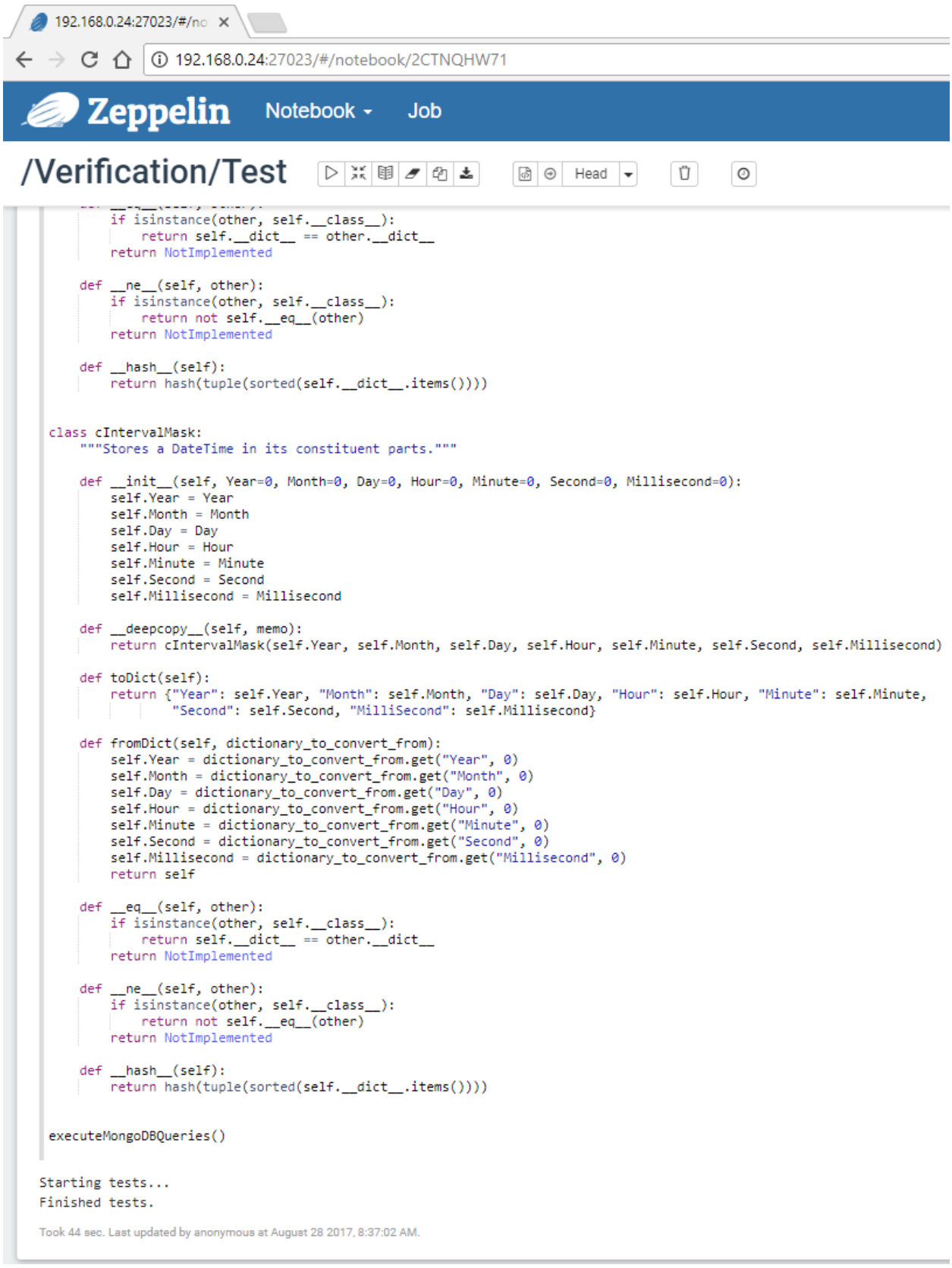


Figure 39: Apache Zeppelin notebook with Python test program output

4.4 System operation validation

In this section, the validation of the system is described. Five validation methods were completed. The first validation determined the cost efficiency of the system. Secondly, the general time-based performance of the system was evaluated. In the third validation, performance was measured with increasing data volumes. The fourth validation determined performance in the presence of increasing numbers of users. Finally, validation was performed regarding the impact of using the system for sustainable energy consumption. This final validation was, however, detailed in a journal article that was submitted for publishing.

4.4.1 System cost efficiency

Although cloud services were not considered in this study, the developed system should still be financially feasible. When compared with cloud services, the developed system should at least be the same cost. Otherwise, there would be no motivation to create and use the system as opposed to using cloud services. For this reason, the developed system's cost was compared with the cost of three cloud providers. A comparison to the most affordable cloud provider was not made as there are too many providers to consider. The three biggest cloud providers were used in the comparison, namely, AWS, Microsoft Azure, and the Google Cloud Platform [136].

Table 13 shows the cost of the developed system and the offering from the three cloud providers. For each of these options, only storage and processing resources were considered. In all cases, the same number of resources was obtained. The costs for all the options were calculated on the same day (10 October 2017). The exchange rate used to convert the cloud services' cost to South African rand was R13.75 per United States dollar.

Table 13: Cost comparison between developed system and three cloud services

Detail	Cost for option (R/year)			
	Developed system	AWS	Google Cloud	Microsoft Azure
Processing cost	35 332.35	52 539.30	61 920.93	115 148.88
Storage cost		53 730.60	48 660.43	40 550.40
Total	**35 332.35**	**106 269.90**	**110 581.35**	**155 699.28**

Although the developed system cost is shown as a yearly cost, it is not a yearly cost. The fact is that the hardware of the system could last for many years. However, the hardware would last at least one year due to the one-year warranty of the system. Thus, it could be considered as a yearly cost, assuming that the hardware will last only for the warranty period.

Table 13 shows that the cost of the developed system is far lower than the cloud services. This result was expected because cloud providers take the full responsibility of supplying computing resources. They manage, purchase, and maintain hardware to keep supplying their services. For this reason, there are more costs involved for the cloud providers, which result in a higher system cost.

Table 13 does not, however, show the costs of additional equipment and human resources. For example, the network infrastructure for the developed system is not included. Similarly, cloud services have additional charges for network traffic to their data centres. For the developed system, human resources, who would manage and set up the additional equipment, were also not considered. These costs may be significant but are complex to calculate. The reasons for excluding such costs are: 1) the scope of this study is concerned with the system, and 2) the case study organisation, like many organisations, had existing infrastructure and human resources.

Based on the previous statements, cloud services are more feasible with insufficient infrastructure and limited human resources. Alternatively, if an organisation already makes exclusive use of cloud services, then creating a cloud system would be preferable. However, in this study, the developed system was more cost-efficient and, therefore, financially feasible. This feasibility makes the business case for the developed system stronger, as the initial cost of ownership is lower than for cloud services.

4.4.2 System general performance

In this subsection and those that follow, results are given to validate the developed system regarding performance. These results were obtained by executing many tests on the case study organisation's original system, a MySQL test system, and the developed system. A total of 400 tags were created for the test. Each tag had half hourly data for approximately four years, which equated to more than 28.6 million data points.

The case study organisation's (original) system was used as it was at the point in time of the tests. Thus, no performance enhancements were made that could alter the test results. As was mentioned in Section 4.2, the case study system was the data system of the translation and reporting system. The MySQL test system was used to emulate the data system of the management information system. A test MySQL system was used because the management information system used a different structure and organisation for data. This difference meant that its performance could have been influenced.

Three C# programs were created to perform the tests; one for each system that was tested. Each test executed a query for 50 iterations while saving the time taken for each iteration. The times taken for all iterations were analysed to obtain the results given in this and the following subsections. In each subsection, the results are given first, followed by a complete discussion at the end.

Test: single-day single-tag

The first set of tests comprised eight tests. The first test retrieved one day's data for the 400th tag. Thus, this test had to search for the 400th tag and return its data for the first day. The results of this test are illustrated in Figure 40.

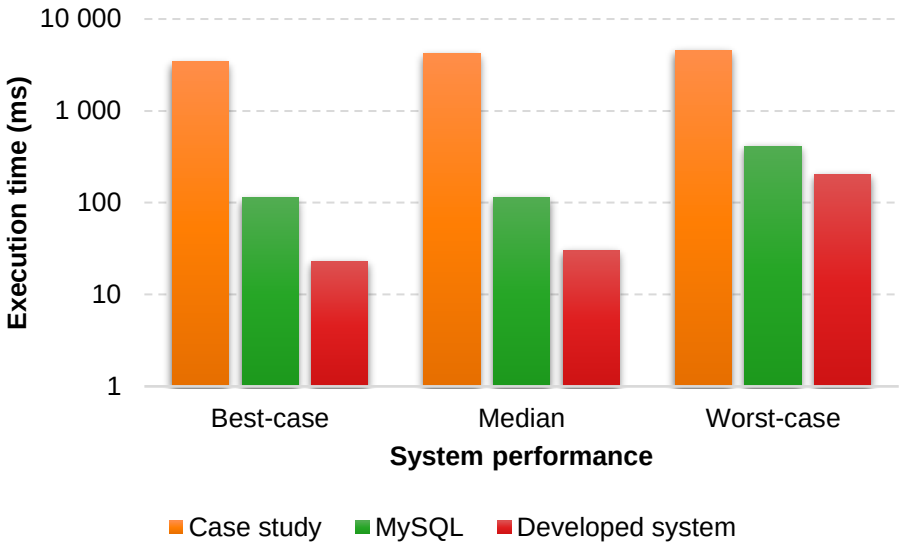


Figure 40: Execution time of systems on a logarithmic axis for the single-day single-tag test

In Figure 40, the best-case, median, and worst-case execution time for each system is given. The figure shows that the developed system had the fastest execution time in all three cases. The MySQL test system was the second fastest, while the case study system was orders of magnitude slower. Half of the time, the developed system achieved an approximate improvement of 99.28% compared with the case study and 73.82% compared with the MySQL system.

Test: single-day all-tags

The second test retrieved one day's data for all tags, i.e. all data for one day. The results of this test are displayed in Figure 41.

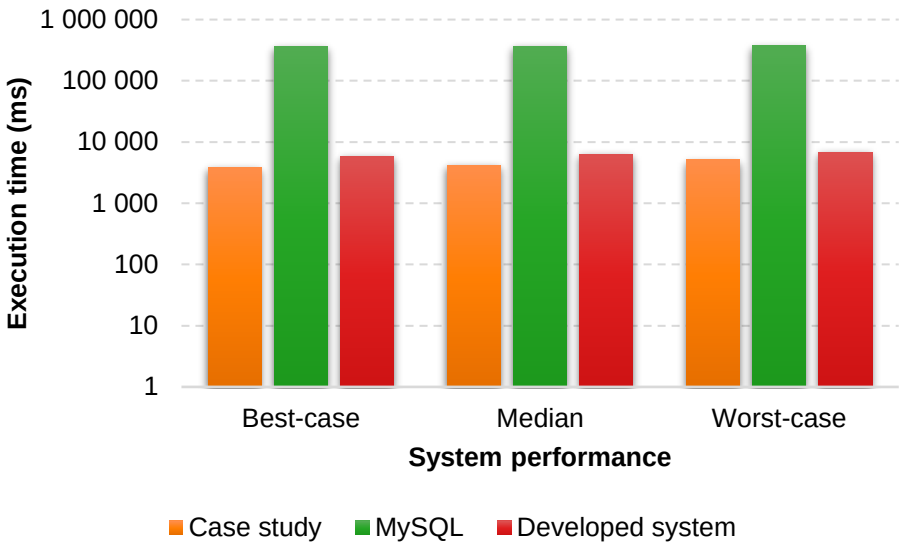


Figure 41: Execution time of systems on a logarithmic axis for the single-day all-tags test

From Figure 41, the case study system was the quickest of the systems. The developed system was the second fastest, and the MySQL system was orders of magnitude slower. For the median-case, the developed system increased the time taken by approximately 50.46% compared with the case study system. However, the developed system realised an improvement of approximately 98.29% compared with the MySQL system.

Test: all-data single-tag

The third test retrieved all data for the 400th tag. The results of this test are shown in Figure 42.

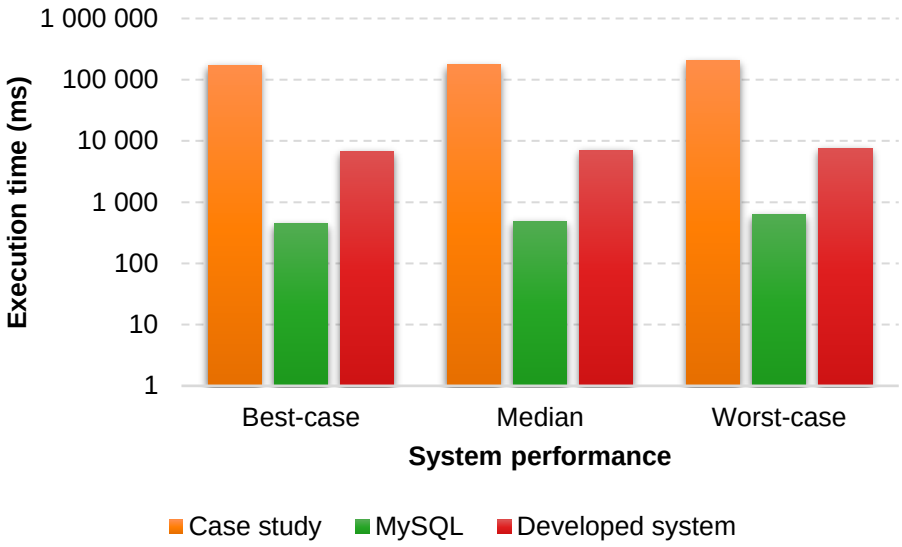


Figure 42: Execution time of systems on a logarithmic axis for the all-data single-tag test

Figure 42 shows that the MySQL system was many times faster than the developed and case study systems. The developed and case study systems were the second fastest and slowest performers respectively. For the median-case, the developed system achieved an improvement of approximately 96.02% compared with the case study system. Compared with the MySQL system, the developed system's time was approximately 1355.41% greater.

Test: all-data all-tags

The fourth test performed a query to retrieve the data of all the tags. Therefore, the query retrieved the entire data set. The resulting execution times of each system are shown in Figure 43.

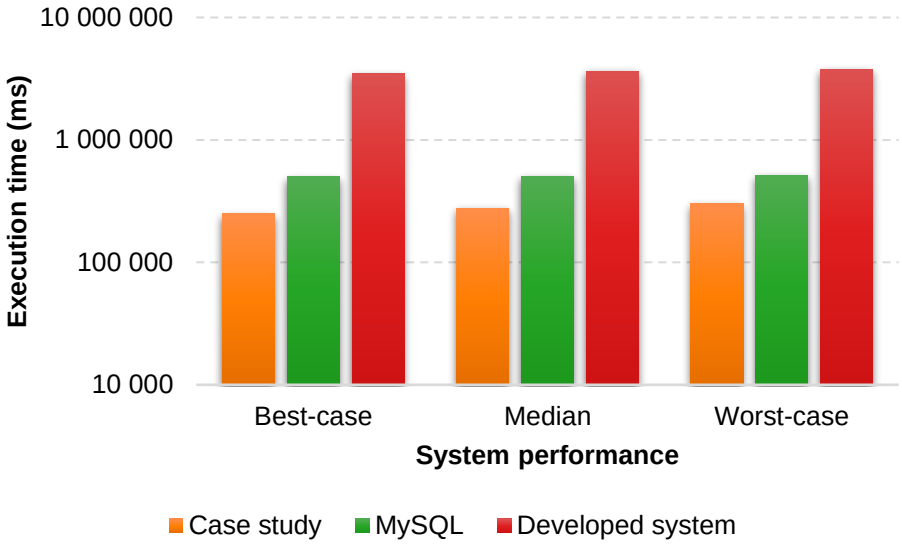


Figure 43: Execution time of systems on a logarithmic axis for the all-data all-tags test

In Figure 43, the developed system was the worst performer. The case study system was the best performer, followed by the MySQL system. The developed system was for the median-case 1210.05% slower than the case study system and 621.28% slower than the MySQL system.

Test: day-average single-tag

The fifth test computed one day's average for the 400th tag. These results are illustrated in Figure 44.

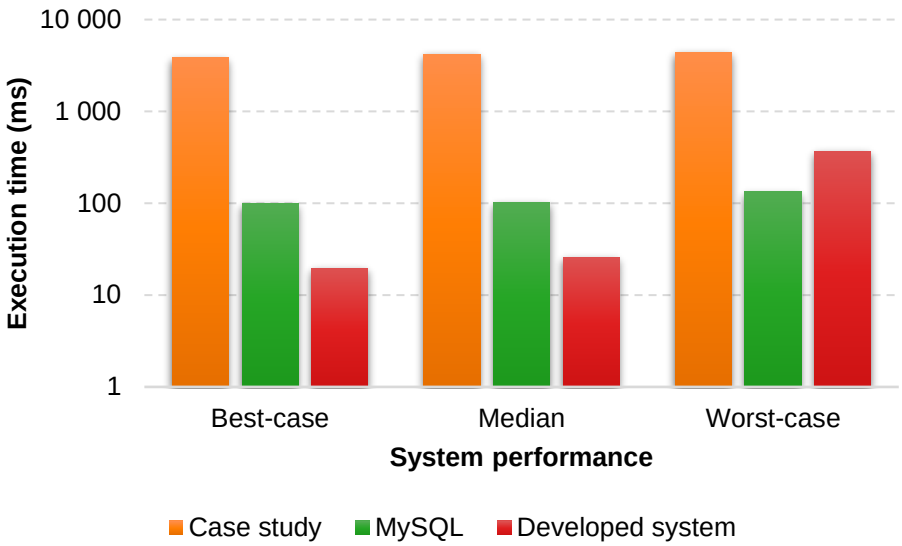


Figure 44: Execution time of systems on a logarithmic axis for the day-average single-tag test

Figure 44 shows that the developed system was the best performer in all but the worst-case. However, the median performance was so close to the best-case performance that the developed system could be the best performer most of the time. The third quartile was calculated and resulted in 102.2 ms for the MySQL system and 26.9 ms for the developed system. Therefore, the developed system was the best performer for most of the test. For the median-case, the developed system obtained an improvement of 99.39% compared with the case study system. Similarly, an improvement of 74.35% was achieved compared with the MySQL system.

Test: day-average all-tags

For the sixth test, one day's average was computed for all the tags. The results of this test are illustrated in Figure 45.

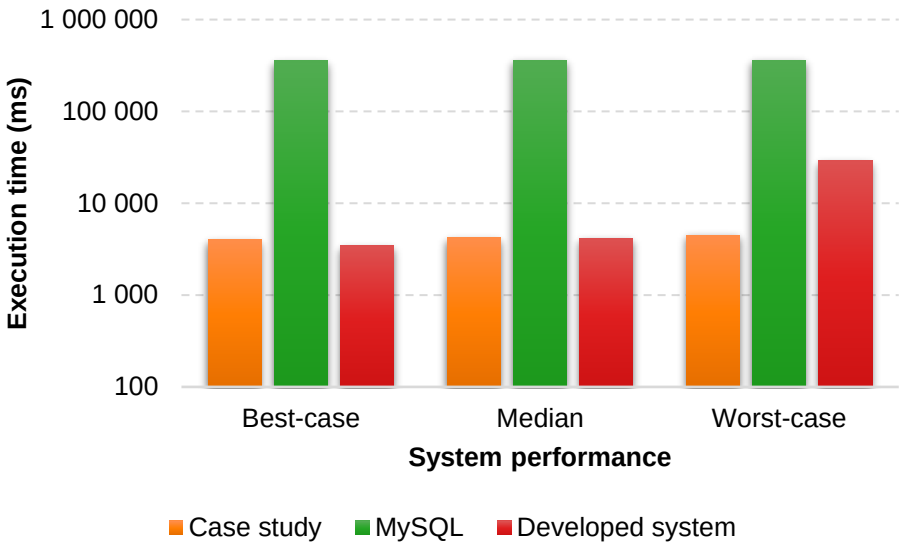


Figure 45: Execution time of systems on a logarithmic axis for the day-average all-tags test

The results of Figure 45 are similar to those of Figure 44, except that the MySQL system was the worst performer in this test. From Figure 45, the developed system was narrowly the best performer in the best- and median-case. In the worst-case, the case study system was the best performer. Calculating the third quartile for the worst-case performance again showed that the developed system was the best performer. The third quartiles were 4265.7 ms for the case study and 4169.6 ms for the developed system. For the median-case, the developed system realised an improvement of approximately 2.52% compared with the case study, and 98.86% compared with the MySQL system.

Test: all-days-average single-tag

The seventh test computed a daily average of all the days for the 400th tag. In total, the test computed 1492 averages for the tag. The results of the test are given in Figure 46.

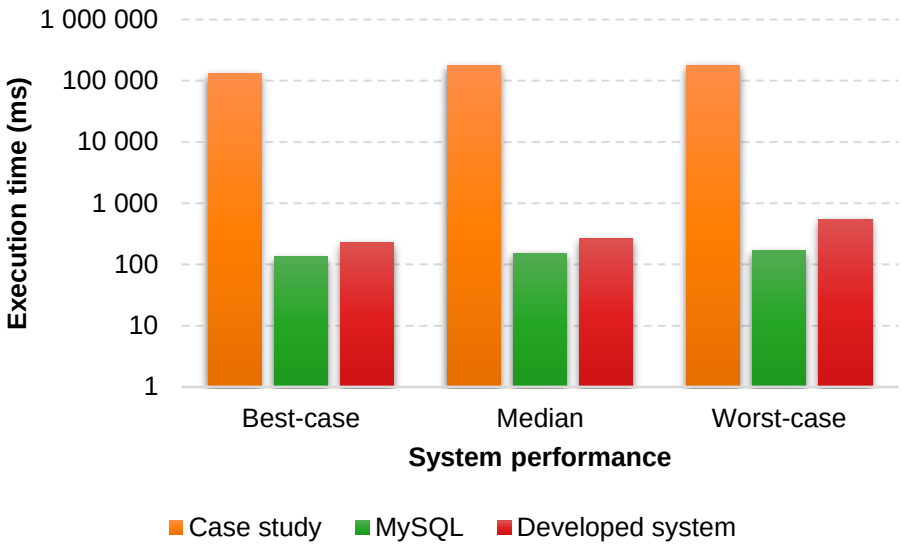


Figure 46: Execution time of systems on a logarithmic axis for the all-days-average single-tag test

From Figure 46, the case study system was the slowest followed by the developed system. Here, the MySQL system was the best performer. For the median-case, the developed system obtained an improvement of 99.85% over the case study system. Compared with the MySQL system, the developed system caused an increase of 72.10%.

Test: all-days-average all-tags

The last general test computed a daily average of all the days for all the tags. This computation resulted in 596 800 average values. The test results are given in Figure 47.

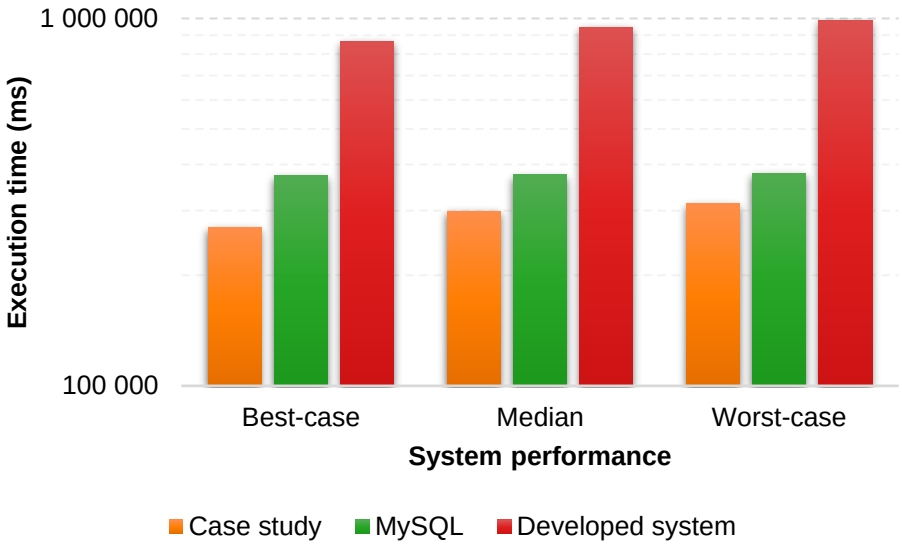


Figure 47: Execution time of systems on a logarithmic axis for the all-days-average all-tags test

Figure 47 shows that the developed system was the worst performer. The case study system was the best performer, followed by the MySQL system. When considering the median performance, the developed system caused a 216.25% increase compared with the case study system. When compared with the MySQL system, the developed system caused an increase of 151.67%.

Discussion

The eight tests showed that the developed system had improved the performance in most of the situations. In five of the tests, the developed system outperformed the case study system. In four of the tests, the developed system outperformed the MySQL system.

The expectation was for the developed system to be faster. However, some tests showed remarkably better performance. Tests showing this remarkably better performance are the single-day single-tag test, and the day-average single-tag test. In contrast, the all-data all-tags test showed remarkably worse performance for the developed system.

There are two possible explanations for the developed system's poor performance in some of the tests. The first explanation is the increased security of the system. In fact, the system has double security: the API and the data storage subsystem are both secured with TLS 1.2. It is widely known that implementing security increases system strain [137]–[139]. This increased strain is due to the extra processing involved in the encryption, key exchange, authentication, authorisation, and integrity-checking processes.

While the increased security decreased performance, it is nevertheless necessary due to the confidential data of the case study. Security is, however, a benefit, which can make the organisation more efficient. Consider, for example, if security was merely added to the system. This separate security would not only result in additional costs, but it would also decrease performance more than native security. Lack of security could also cause compliance and legal issues, which would affect an organisation's operations or services. Therefore, the security of the system increases the protection and privacy efficiency of an organisation.

The second possible cause is the distributed nature of the system. The test computer had to communicate with the API web service, which resided on another networked computer. Thus, the API web service had to communicate with the data storage subsystem to reach the data. The data storage subsystem was again on a different networked computer. Therefore, all communication needed to pass through two computers and had to be transmitted over two network connections. Naturally, the more computers and network connections there are, the longer data takes to reach its destination.

The fact that the developed and MySQL systems were accessible over a network is clearly visible in the large data tests. This network effect can be seen in the tests that required the transmission of large amounts of data. These large data tests are the single-day all-tags, all-data all-tags, (to a lesser degree) day-average all-tags, and all-days-average all-tags tests. In all these tests, the case study system was faster than both the

MySQL and the developed systems. The exception is the day-average all-tags test, where only the developed system was faster than the case study system. For these four tests, the common denominator between the MySQL and developed systems was a network.

The converse of the previous paragraph is also true. That is, an increase in performance was possible because the case study system was present on the same computer. The case study system had to be present on the test computer because it had no network capabilities. Consequently, there were no additional influences on its performance.

The results of this subsection can be taken further. Consider, for example, a day in which users use a system. During this day, various queries would be performed that consist of those reported in the eight tests. The total number of each query performed could be as indicated in Table 14. The numbers given in the table assume that users would be less likely to query all the data. Similarly, it would be more useful to query raw data instead of averages, as one can do any calculation on the raw data. A higher number was given to the single-tag queries to account for the fact that users could query a subset of the tags.

Table 14: Theoretical distribution of queries for a day

Type of query	Number of queries	Probability of query
Single-day single-tag	1 000	0.663
Single-day all-tags	50	0.033
All-data single-tag	100	0.066
All-data all-tags	5	0.003
Day-average single-tag	300	0.199
Day-average all-tags	2	0.001
All-days-average single-tag	50	0.033
All-days-average all-tags	2	0.001
Total	**1 509**	**1**

Using the numbers in Table 14, the median time to complete the set of queries was calculated. The time to complete the set of queries was calculated for each system based on the results of this subsection. These completion times are given in Figure 48.

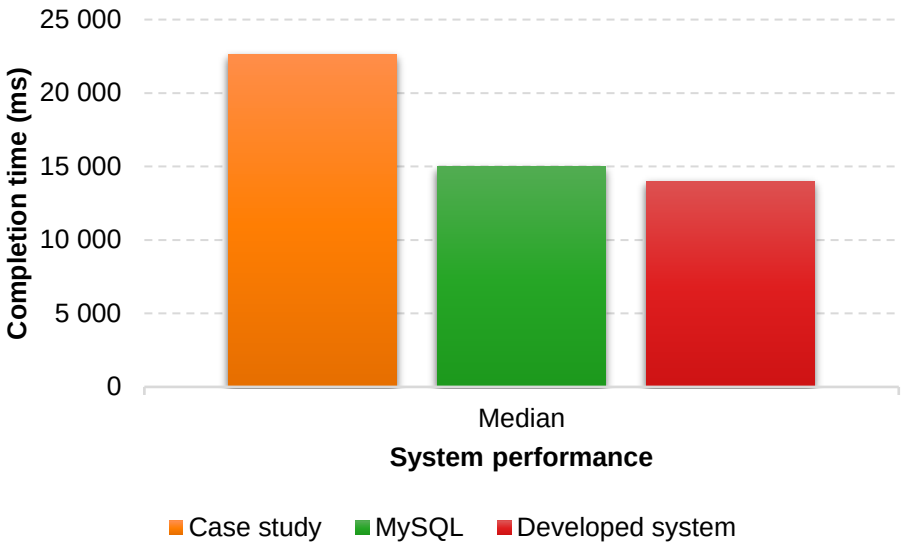


Figure 48: Completion time of systems for a theoretical set of queries

Figure 48 illustrates that for the theoretical set of queries, the developed system would complete it the fastest. The MySQL system would complete the queries the second fastest, and the case study system would complete it the slowest. Calculating the time saved by the developed systems yields 8632.7 ms compared with the case study system and 979.5 ms compared with the MySQL system. Thus for 365 days, i.e. a normal year, the developed system would save approximately 52.5 minutes compared with the case study system, and 6.0 minutes compared with the MySQL system.

It is clear that the developed system's performance is comparative to MySQL. This result is important because, as mentioned in Section 4.2, the case study data benefitted MySQL. Despite MySQL's advantage, the developed system still achieved a faster general performance.

The next subsection further investigates the systems' performance as the amount of data retrieved increases.

4.4.3 System increasing data volume performance

This subsection continues from the previous subsection by investigating each system's performance with an increasing data volume. The daily data retrieval and daily average tests of the previous subsection were repeated here. However, the number of days used in the queries were varied. Each test started with 10 days' data and increased with 10 until reaching 300 days. For each number of days, 50 iterations were performed. After performing all iterations, the median execution time was calculated, which is reported here. As with the previous subsection, a detailed discussion is given at the end of this subsection.

Test: varying-days single-tag

The first test queried an increasing number of days' worth of data for the 400th tag. By increasing the number of days, the amount of data increased linearly. Therefore, this test, as well as the subsequent tests,

determined the sensitivity of the systems to increasing data. The results of the first test are illustrated in Figure 49.

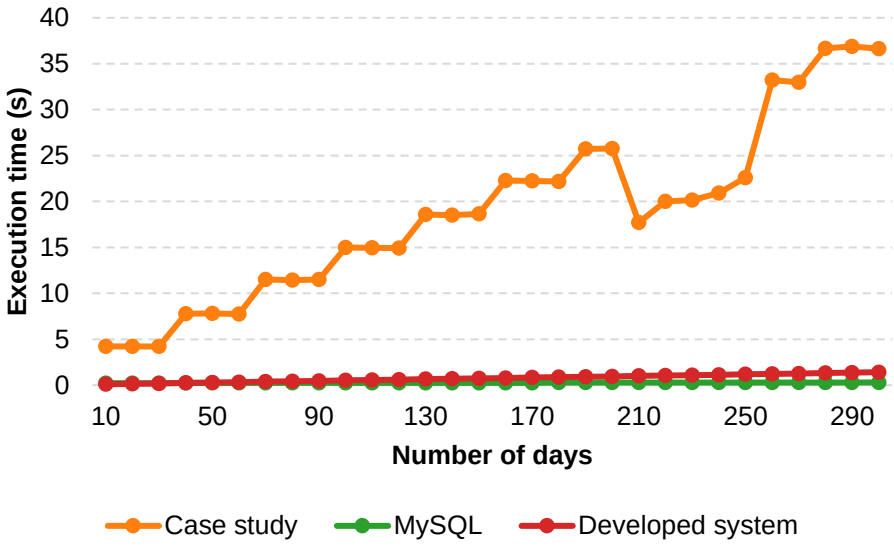


Figure 49: Median execution time of systems for the varying-days single-tag test

From Figure 49, the case study system had the worst response to increasing data. Therefore, its rate of execution time increase was the greatest. The developed system started with the fastest execution time but ended slower than the MySQL system. Its rate of execution time increase was the second largest. Consequently, the MySQL system had the best response with an increasing amount of data. The figure implies that the MySQL and developed systems can support greater data volumes than the case study system.

The anomaly in the line graph of the case study system warrants explanation. From Figure 49, it is evident that the case study system followed a steplike pattern. This pattern was, however, broken at days 210 to 250. There are two explanations for this anomaly. The first explanation is that there could have been a flaw or error in the case study system. This flaw could have caused the test to terminate prematurely, resulting in a lower execution time. The second explanation is a network communication problem, as the system communicates internally with other systems. Any communication problem would cause a premature stop in the test.

Despite the anomaly as mentioned above, the pattern or relationship for the case study system remains clear. The anomaly does not alter the overall outcome of the test.

Test: varying-days all-tags

The second test retrieved an increasing number of days' data for all tags. Recording the execution times of this test resulted in the graph shown in Figure 50.

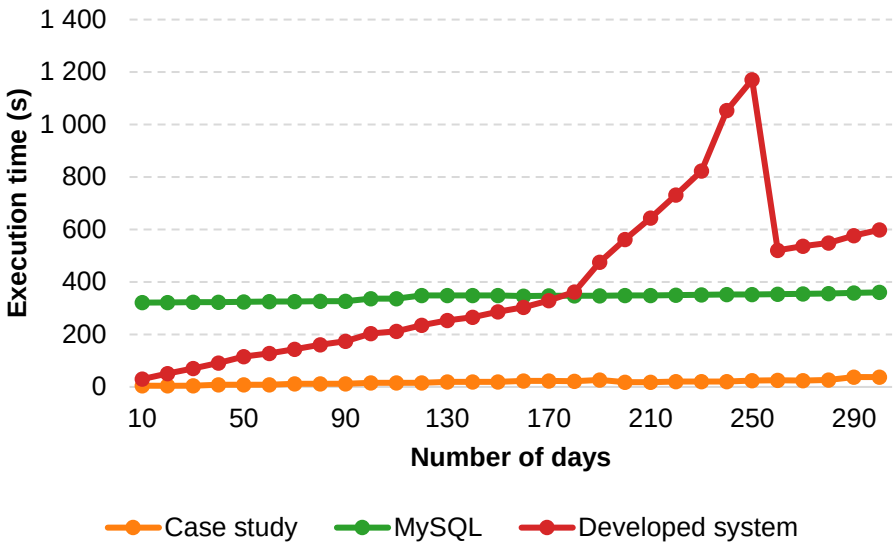


Figure 50: Median execution time of systems for the varying-days all-tags test

Figure 50 shows that the case study and MySQL systems had a small increase in execution time as the number of days increased. The case study system was the fastest of the systems, followed by the MySQL system. The developed system was initially the second-fastest system, but at 180 days it became the slowest. The rate of increase for the developed system was the highest, which means it is more sensitive to this type of query.

There was another anomaly in this test; this time for the developed system. From 190 to 250 days, the developed system's execution times were significantly higher. A possible explanation for this anomaly is a network-related problem. This problem is possible because the systems were tested at the case study organisation. Although the tests were performed after hours and over weekends, the likelihood of network-related user activity remained. Significant network usage could cause decreased network bandwidth and increase network transmission time.

It is clear, however, that the system did not fail as that would have resulted in either very small or very large execution times. A very large execution time refers to a time greater than 3600 s, as this was the time limit for each test iteration. Irrespective of the anomalous execution times, the relationship of the developed system's execution times resembles a straight line. Furthermore, the results of the test were not influenced by the anomaly.

Test: varying-days day-average single-tag

For the third test, an increasing number of daily averages were computed for the 400th tag. These results are displayed in Figure 51.

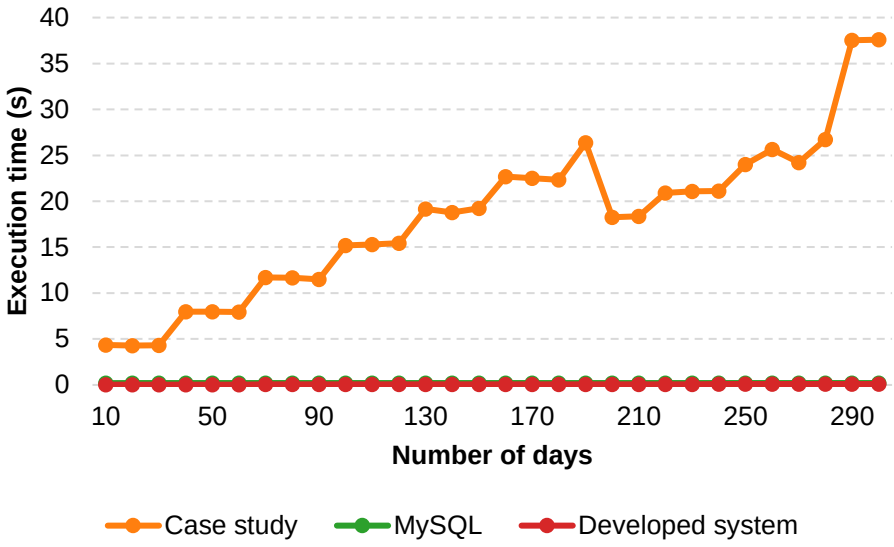


Figure 51: Median execution time of systems for the varying-days day-average single-tag test

In Figure 51, the case study system was the slowest system with the largest rate of increase. The execution times of the case study system followed a steplike pattern, similar to that in Figure 49. For the MySQL and developed systems, the figure is less clear. Consequently, the section containing the MySQL and developed systems were enlarged and is given in Figure 52.

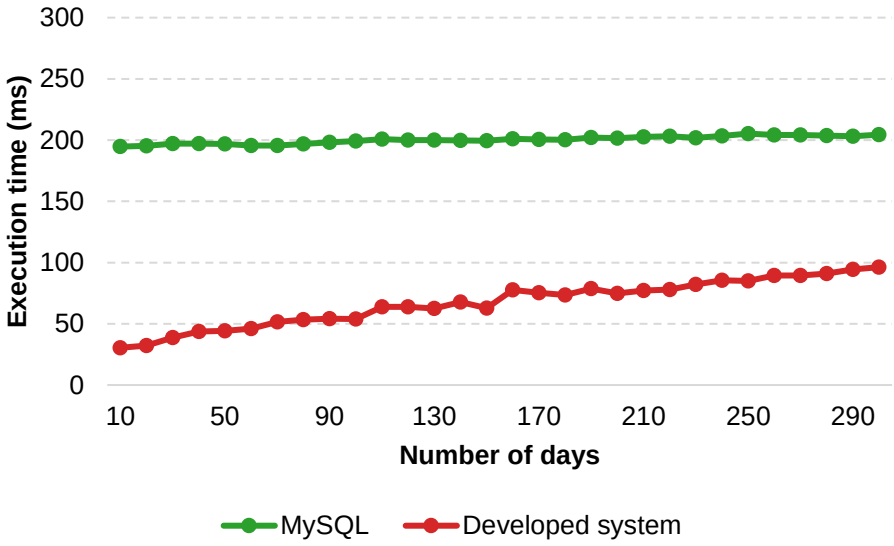


Figure 52: Enlarged region of Figure 51 for the MySQL and developed systems

Figure 52 shows that the MySQL system had the smallest rate of increase, but was still the second-fastest system. The developed system was the fastest, but it had the second-highest rate of increase. Both the MySQL and the developed system had followed an approximate straight-line pattern. It is clear, however, that the developed system would eventually surpass the MySQL system, but only if their illustrated relationships continued.

This test again had an anomaly from days 200 to 280 for the case study system. This anomaly is similar to that discussed in the varying-days single-tag test. Despite the anomaly, the results of the test remained unaffected.

Test: varying-days day-average all-tags

The final test computed daily averages for each of the tags on an increasing number of days. This test yielded the results illustrated in Figure 53.

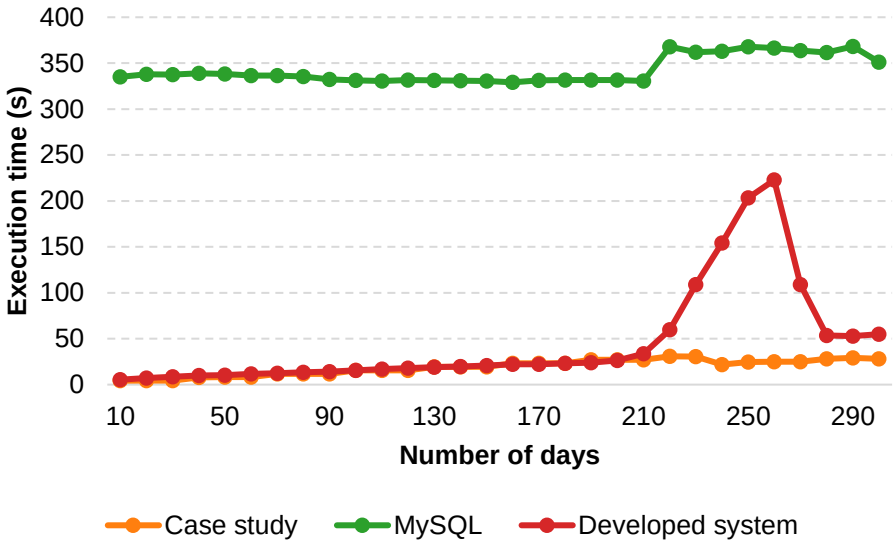


Figure 53: Median execution time of systems for the varying-days day-average all-tags test

It is evident from Figure 53 that the MySQL system was the slowest but had the least rate of increase. The case study and developed systems had almost the same performance, except for closer to the end of the day range. Near the end, the case study system achieved quicker performance with an overall second-largest rate of change. The developed system was, therefore, the second-slowest system with the largest rate of increase.

For this test, there were anomalies for both the MySQL and developed systems. The MySQL system showed greater variation in the execution time from 220 to 300 days. The developed system exhibited suspicious execution times from 220 to 280 days. As both these systems are dependent on network communication, this was the likely cause for their increased execution times. Despite the two anomalies, a pattern is still visible for the MySQL and developed systems. Thus, the result outcomes remain the same.

Discussion

The results of this subsection showed that the performance of the tested systems depends on the quantity of data. As more data is used in the computations and queries, the execution time increases. When looking at each system individually, the MySQL system was the least influenced by increasing data volumes. Its

performance was more stable than the performance of the other systems. However, the MySQL system had a large offset, i.e. base execution time, even for small data volumes. Examples of the large offsets are visible in Figures 50, 52, and 53.

The developed system was equal to the case study system regarding its relationship to increasing data volumes. In two tests, the developed system had a lower rate of increasing execution time. In the varying-days all-tags test, the developed system had a greater rate of increasing execution time than the case study system. The varying-days day-average all-tags test showed that the developed system narrowly exceeded the rate of increase for the case study system.

Overall, the four tests showed that the developed system is not as performant. The reason for this is the same as was discussed in Section 4.4.2. That is, the security and additional network communication required increase the time taken by the developed system. The tests also showed that a conclusion regarding the impact of data volume is difficult to make. System performance varies according to the type of query and the data volume.

The next subsection investigates a different aspect regarding performance, which is the number of users supported.

4.4.4 System simultaneous user performance

The final set of tests that validated the system was done by increasing the number of simultaneous users. Two tests, i.e. two queries, were executed. For each test, the number of users connected was varied from 10 to 50 in intervals of 10. Users were simulated by creating connections to a system with each user executing the same query. The median time it took for all queries to be completed was then recorded. As with the previous tests, each test for a given number of users was performed 50 times. A discussion of the two tests follows at the end of this subsection.

The limit for the number of users was chosen because it represents half of the case study organisation's workforce. It was reasoned that not all the workforce would perform data retrieval at the same time. Furthermore, not all employees work with tag data, for example, janitors, administration personnel, and chief officer personnel.

Test: varying-users single-day single-tag

The first test is the single-day single-tag test, which is described in Section 4.4.2. The results of this test are shown in Figure 54.

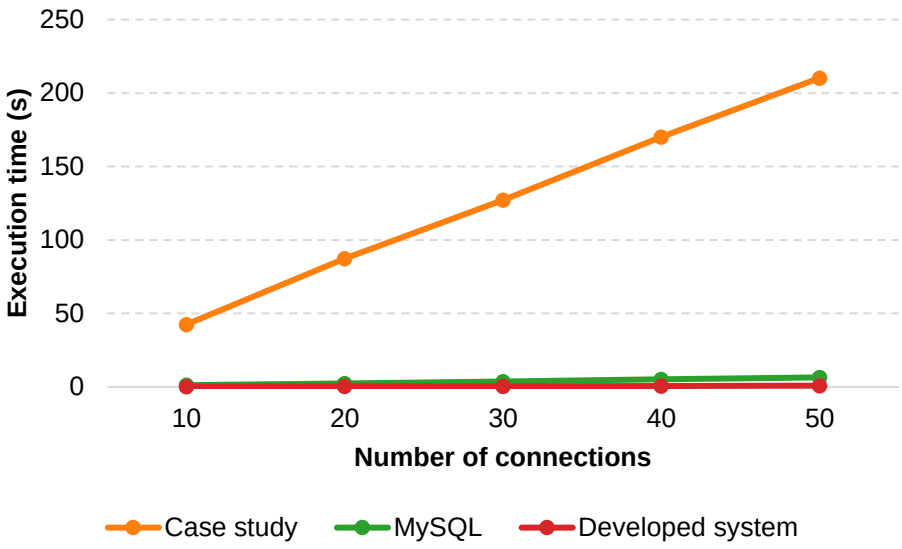


Figure 54: Median execution time of systems for the varying-users single-day single-tag test

Figure 54 shows that the case study was the worst performer for any number of simultaneous user connections. The rate of increase was the largest for the case study system. To better view the remaining systems' performance, an enlarged illustration of the figure is shown in Figure 55.

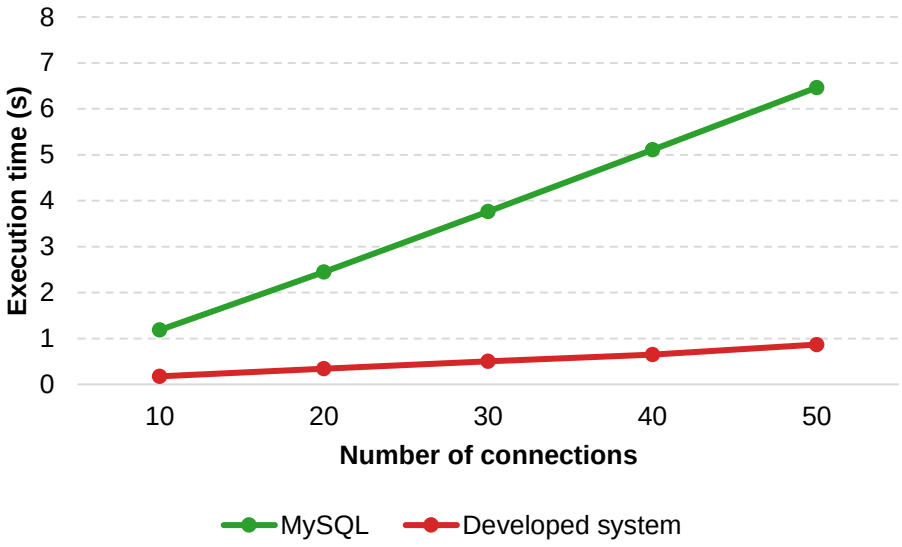


Figure 55: Enlarged region of Figure 54 for the MySQL and developed systems

From Figure 55, the developed system was indeed the best performer with the lowest rate of increase. The MySQL was the second-best performer, yet far better than the case study system. The MySQL system had the second-lowest rate of increase. Overall, all the systems' execution time increased linearly with an increase in user connections. For the maximum connections, the developed system achieved an approximate 99.59% performance increase compared with the case study system. Compared with the MySQL system, the developed system achieved an 86.58% increase.

Test: varying-users single-day all-tags

For the final test, the single-day all-tags test was performed, as discussed in Section 4.4.2. The results obtained for this test are shown in Figure 56.

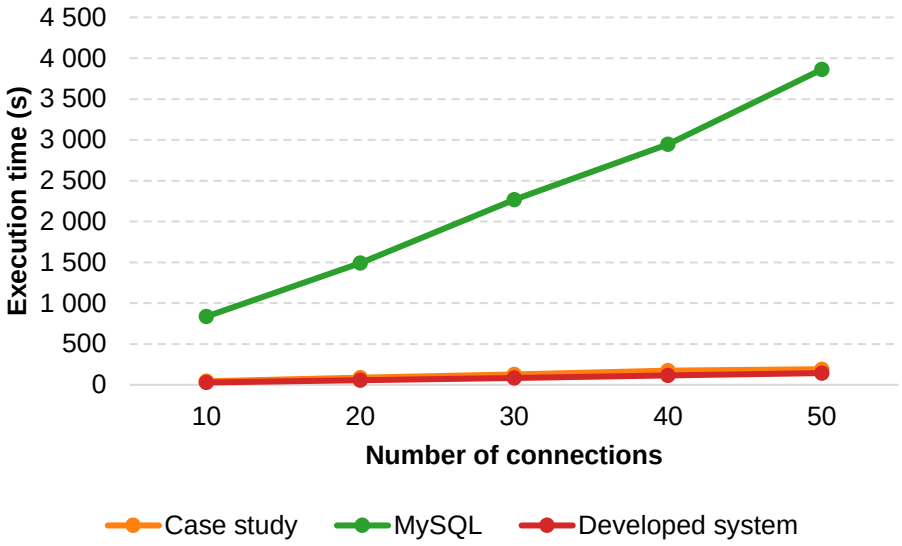


Figure 56: Median execution time of systems for the varying-users single-day all-tags test

In Figure 56, the MySQL system was the slowest with the highest rate of increase. Again, the case study and developed systems' performance is more clearly shown in Figure 57.

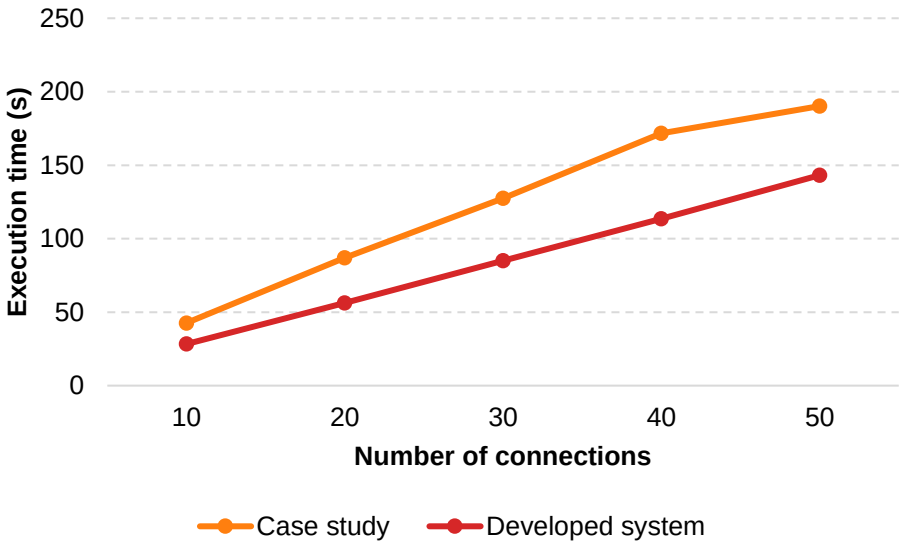


Figure 57: Enlarged region of Figure 56 for the case study and developed systems

Figure 57 shows that the developed system was the best performer with the lowest rate of increase. Consequently, the case study system was the second-slowest performer. All three systems had an approximate linear increasing relationship with the number of connected users. For the maximum

connections, the developed system obtained a 24.72% increase compared with the performance of the developed system. Furthermore, the developed system realised a 96.29% performance increase compared with the MySQL system.

Discussion

This subsection's results showed the impact that multiple simultaneous connections had on the systems' performance. All three systems showed an increase in execution time as the simultaneous connections increased. In both the tests, the developed system was the best performer with the lowest increase in execution time for a given number of connections. The performance of the case study and MySQL systems were equal, with each being the worst performer in a different test.

The developed system obtained a performance increase of at least 24.72% compared with the case study system. Compared with the MySQL system, the developed system achieved an increase of at least 86.58%. Therefore, the developed system increased the case study organisation's user capacity. This increase means that for the same number of simultaneous users, the organisation is more efficient. The organisation is more efficient because more users can perform queries than with the previous system. Alternatively, the same number of users can perform queries faster than with the previous system.

The benefits of the developed system extend to the employees. If more employees can do their work at the same time, fewer employees would have to wait. Alternatively, if the same number of employees can perform their queries faster, they would have more time for other work. In both situations, employees become more efficient.

The results were, however, dependent on the type of query performed. Due to this conclusion, it was decided only to perform the two tests. If it were known which queries a user might perform, more tests could have been performed. However, it is only plausible to do a small number of tests, as it becomes difficult to test all queries that could be performed. The previous tests of Sections 4.4.2 and 4.4.3 were also dependent on the type of query. Therefore, it was necessary to limit the number of tests performed due to the limited time for this study.

4.4.5 System usage impact

The final validation method was to determine the impact of using the system. To this end, a journal article was submitted for publication in the Special Volume of the Journal of Cleaner Production: Sustainable Consumption and Big Data. The purpose of the article was to show insights into sustainable energy consumption in the industrial and mining sectors. Although the article is not discussed in detail, it is given in Appendix 4.

The article provides validation of the system regarding its operational influence. In the article, multiple case studies are given to illustrate energy-saving projects that were identified. By assisting with the identification of energy-saving projects, the organisation could do more business. Thus, the organisation became more profitable and delivered more services. Consequently, the efficiency of the organisation increased.

Although the previous four validations provide quantitative benefits of the system, there are implicit benefits of the system. The replication of MongoDB in the service layer, as discussed in Section 3.4.2, provides automatic data redundancy or backups. Having multiple copies of all data prevents data loss during faults and failures.

Furthermore, the data set replicas can allow for even greater user concurrency. This greater concurrency is possible because MongoDB allows users to read data from slave replica members. There is, however, the risk of outdated data because of eventual consistency. However, data is read more than it is updated in this study, thus making it less likely of reading outdated data. The validation tests did not exploit the system's increased concurrency. The reason is it would have benefitted the system unfairly, making comparisons with the other systems meaningless.

A second implicit benefit is that the system provides a platform for future big data projects. This characteristic was, however, already discussed in Chapter 3. Still, the system is modular and extensible to make use of future big data software and technologies. Furthermore, the system is deployable to cloud services, which is important for its sustainability.

The final implicit benefit, as discussed in Chapter 3, is the addition of computers to increase performance. This characteristic is possible because of horizontal scalability. If the system reaches a point where the data volume is too great, more computers can be added as nodes in the cluster. Data sharding can also be extended to maintain acceptable performance.

4.5 Research results

This final part of the chapter will discuss the most important results and the outcome of this research. Thus, it will be described how the research objectives were achieved. Finally, the research questions will also be answered.

This study's objective was to design, develop, and implement a centralised data management system to improve organisational efficiency using big data. The value of big data was to be illustrated to organisations toward the adoption of big data. This research had to focus on mining and industrial sectors to provide an example for the practical implementation of a big data system. The data silos problem had to be addressed by the proposed system.

Was the main objective attained and the problem solved? Yes. Chapters 3 and 4 provide evidence of the following:

- The system supports and makes use of big data technologies.
- Future big data technologies and projects can be used in the system through its extensible and modular design.
- Security was considered at the design level, which makes data safer and ensures its privacy.
- The developed system saves cost as it is more affordable than cloud services.
- Compared with a case study's previous system, the developed system took less time to perform queries. This faster performance increases the speed of an organisation, i.e. time efficiency.
- The developed system supports a greater number of simultaneous connections allowing more employees to use the system. This support of more users makes them and their organisation more efficient.
- Using the system improves operational efficiency as detailed in a journal article of this study.

Therefore, this study detailed the design, development, and implementation of a centralised data management system. This system provides an example and framework for the deployment of a big data system thereby aiding in the adoption of big data in organisations. More importantly, the system's implementation on a case study in the mining and industrial sectors illustrates the benefits and value of big data to such organisations. The data silos problem was solved through the system's dynamic nature and support for multiple data types.

The research questions were answered in the study as described in Table 15.

The specific objectives of this study, as given in Section 1.3, were also met. Each of the literature objectives was addressed as follows:

1. The characteristics of big data systems were compared with traditional data systems and thoroughly investigated in Sections 2.2 and 2.3.
2. Big data system architectures were identified and investigated in Section 2.3.
3. The methods and best practices used to make the functionalities of big data systems available were discussed in Section 2.4.
4. Big data software for storage, processing, and management were studied throughout Section 2.5 as well as Chapter 3.

Table 15: Summary of research questions as answered in this study

Research question	Section or chapter of detail	Summarised answer
1	2.2 and 3.4.4	• A document data store provides flexibility for any data type. • For MongoDB, parent-referencing allowed optimal structuring of tag and value storage.
2	2.3 and 3.4	• The Lambda architecture, polyglot persistence, and data lake were identified and investigated. • A combination of the three architectures was used in the system's design.
3	2.2, 2.5 and 3.3	• Four types of NoSQL data store were discovered and investigated. • A summary of possible data store software was made. • MongoDB, Couchbase, Impala, and MySQL were evaluated experimentally.
4	2.4 and 3.4	• SOA and microservices were identified as the preferred approach. • Containers were investigated as the recommended implementation of microservices. • Microservices with Docker containers were used to create all system components.
5	4	The system: • Improved an organisation's security. • Reduced time taken. • Increased user capacity. • Decreased costs compared with cloud services.

The empirical objectives were addressed as discussed below:

1. The developed system was found to be financially feasible as discussed in Section 4.4.1.
2. Compared with the case study's previous system, the developed system was quicker as discussed in Section 4.4.2.
3. The developed system was less performant for increasing data volumes, which indicates room for improvement as discussed in Section 4.4.3.
4. Increased support for simultaneous users was realised by the developed system, which was discussed in Section 4.4.4.
5. The impact of security on the performance of the developed system was illustrated throughout the validation of the system.

In summary, this section stated that this study addressed the research problem, objectives, and questions.

4.6 Conclusion

This chapter started by providing details regarding the case study on which the system was implemented. The engineering consultancy company deployed the developed system alongside a translation and reporting system together with a management information system. Data from the case study organisation's clients was stored in the developed system. Traditional data systems were still used by the case study organisation in the form of MySQL. However, the traditional data systems only stored data that was meant for a specific system. Thus, the developed system was used to centralise the data of the case study organisation.

The developed system was verified by confirming adherence to the specifications of Section 3.2. This verification process further ensured the correct operation of each system component as described in Section 3.4. The components that were explicitly verified were Mesosphere DC/OS, MongoDB, Apache Spark, the API web service, and the user interface (Apache Zeppelin). Python was implicitly verified through the verification of MongoDB, Apache Spark, and the user interface. All system components were functioning correctly, and each adhered to its specifications.

Following verification, the developed system was validated. The first validation compared the cost of the developed system with the costs of cloud services from the three largest cloud providers, which were AWS, Google Cloud Platform, and Microsoft Azure. The hardware cost of the developed system was much more affordable than the least expensive cloud service. Being more affordable means that implementing the developed system was financially viable and would save costs compared with cloud services.

The second validation investigated the general performance of the case study system, a default MySQL system, and the developed system. Eight tests were completed that consisted of computational and data retrieval queries. The performance of each system varied based on the query or test completed. However, the developed system was the best performer overall in five of the tests when compared with the case study system. When compared with the MySQL system, the developed system was faster in four of the tests.

The third validation was to determine the influence of the amount of data on the performance of the three test systems. For the four tests performed, the MySQL system showed the lowest increase in time taken for increased data volumes. The case study and the developed system performed approximately similarly, with each being better in half of the tests. This part of the validation meant that the developed system was not as performant as in the previous validation. Thus, this test showed an area of improvement.

For the fourth validation, two tests were performed for increasing simultaneous connections. This part of the validation determined the user capacity of each system. In both tests, the developed system was the best performer. The developed system was quicker for the same or a greater number of simultaneous connections. Thus, this validation showed that the developed system increased employee efficiency.

Employee efficiency is increased by allowing more employees to do work at the same time or fewer employees to do their work quicker.

The final validation was done to show the system's operational impact. This validation was detailed in a journal article written by the author, which was submitted for publication. Thus, only a brief description was given regarding the improvement of an organisation's ability to deliver services. The full journal article is given in Appendix 4 and should be considered as a continuation of this study.

The reason the developed system was not the best in each of the performance tests was due to its security and distributed architecture. It is widely known that the more thorough security is, the more performance is impacted. The distributed architecture's effect can be seen in network connectivity. For example, if the network of the developed system is congested due to user traffic, performance will suffer. A congested network and any other network-related problem are, however, detrimental to any networked system and not just the developed system.

This chapter has shown that the system was verified with all specifications being met. The system was proven to function correctly. Finally, the system was validated by a case study in terms of cost, performance, data volume, user capacity, and operational efficiency. Thus, the developed system succeeded in solving the problem identified by this study. This problem was to improve the efficiency of organisations through designing and implementing a big data system. The research questions and objectives were, therefore, addressed by this study.

CHAPTER 5 CONCLUSION

5.1 Summary

Big data is continuing its disruption of the modern data landscape. By embracing big data, organisations can become and are becoming more competitive, innovative, and efficient. Big data requires organisations to rethink their usage of traditional relational database systems. Traditional relational database systems, referred to as SQL databases, have dominated the database environment for more than 40 years. However, to make use of big data, organisations need to adopt NoSQL data stores and related big data technologies.

The study found that organisations, specifically in the South African mining and industrial sectors, were slow to adopt big data. Organisations were uncertain of the value and impact that big data could yield. This uncertainty was compounded by the incorrect use of traditional relational database systems. Data silos were found as a threat to the successful utilisation of big data.

This study proposed the development of a centralised data management system to solve these previous problems and realise big data's benefits. The proposed system would provide a practical framework for the creation of big data systems. The proposed system had the objectives of improving organisational efficiency while enabling the adoption of big data. Being futureproof was an important aspect of the system as the big data landscape is ever evolving with new and improved technologies.

This study was structured according to the design science research methodology. Consequently, the stages that enabled the completion of the study were problem identification, investigation, design, development, verification, and evaluation. Each of these stages was addressed in specific chapters. Chapter 1 addressed the problem identification stage. Chapter 2 covered the investigation stage. Chapter 3 detailed the design and development stages. Chapter 4 addressed the verification and evaluation stages.

The investigation stage comprised a literature study on data systems, their design, and implementations. Certain questions, as given in Section 1.3, needed to be answered for a successful system design. Firstly, data systems were introduced with their progression from physical file systems to database systems. Relational databases, as managed by an RDBMS, were the first data systems. RDBMSs are, however, unable to support big data due to their relational model and characteristics.

NoSQL data stores can be seen as the next evolution of database systems. Four types of NoSQL data store were investigated, namely, the key-value store, document store, graph store, and columnar store. Each of the data store types has a different approach to data storage and structuring. This uniqueness means that there is no one-size-fits-all solution.

Data storage software is, however, just one part of a big data system. Thus, the architectures of big data systems needed to be investigated. Three architectures were identified to answer this question. The Lambda

architecture was found as an architecture that isolates complexity using different layers. Polyglot persistence is a concept that prevents the one-database-fits-all-data pitfall. Essentially, polyglot persistence makes use of different data stores for their specific intended purpose. Finally, the data lake architecture advocates the storage of data in its raw form and the collection of the most data possible.

A method was needed to make the system's functionality available to users and other systems. This question was answered in Section 2.4. Literature prescribed SOA as the preferred architecture or way to manage the complexity of a system's different features. Further investigation revealed that the microservices pattern is the latest trend in the software industry. Microservices is related to SOA and could be considered a specialisation thereof. By using microservices, a system is divided into small applications; each with a single responsibility.

Discovering microservices raised the question of how it is implemented. Literature and industry experience suggested using containers. Containers allow the isolation of software into small units, hence its applicability to microservices. A few containers technologies were found, but Docker was consistently recommended.

Using containers resulted in other requirements regarding their management and deployment to create a system. Hence, container orchestration was studied as a means of implementing containers in a distributed system. Again, many orchestration software was found.

The final question was one regarding which big data platforms to use in the development of the system. As already stated, there are four types of NoSQL data store; each of which has many software projects. This question was answered in two parts. The first part, completed during the literature study, was the evaluation of different data store software. This evaluation summarised the characteristics of a few data stores. From there, MongoDB, Impala, and Couchbase were chosen for a performance evaluation. The performance evaluation constituted the second part of the answer to the question above.

The literature study provided a knowledge base for the design and development stages. Here, the first task was to determine the requirements and specifications for the proposed system. The requirements analysis was followed by the database system performance evaluation. In the performance evaluation, MySQL was tested in addition to the three chosen data stores. This evaluation concluded that MongoDB would be the most appropriate data store for this study's context and system. Additionally, MySQL was discovered to have performance better or equal to MongoDB in general. The reason being that relational data was used which favoured MySQL. Such data was, however, required as it was similar to the case study data.

The system was designed by making use of characteristics of the Lambda architecture, polyglot persistence, and the data lake. Three layers were created for the system's architecture, namely the resource, service, and interface layers.

The resource layer consisted of Mesosphere DC/OS orchestration software. This layer provided the resources for the next two layers. Therefore, the resource layer was the closest to hardware. The resource layer was used to create a cluster of computers on which containers would be deployed. By designing a resource layer, the system can be deployed to cloud services. Thus, in a private context, the resource layer will be implemented on an organisation's infrastructure. However, if necessary, cloud services can become the resource layer, therefore providing the system with portability and extensibility.

The service layer, as the name indicates, consisted of subsystems that provided the main services of the system. Three subsystems were placed in this layer to provide data storage, processing, and analysis. Each subsystem or service was created as a microservice using Docker containers. Thus, Docker is the only requirement of the service layer, which allows for any resource layer software now or in the future. MongoDB provided the data storage of this layer. Furthermore, Apache Spark and the Python programming language enabled data processing and analysis.

The interface layer is the layer that provides an interface for business processes and users. Logically, the business logic would reside in this layer. An API and user interface were created to implement this layer. The API, through which systems and applications use the system, was implemented as a WCF SOAP web service. Apache Zeppelin was implemented, again as a container, to provide a web browser environment for users.

The final design task was modelling how data was stored in the system. This task addressed the research question regarding the data model of the proposed system. Data modelling was closely tied to the type of data store and characteristics of the software. Hence, literature regarding MongoDB was studied and used to create the structure of documents, which contain the stored data.

Upon the successful design of the system, it was implemented on a case study for the verification and evaluation stages. The case study chosen was an engineering services company with clients in the South African mining and industrial sectors. In the case study organisation, the system integrated with existing systems that translated and used data.

Verification, which followed the system implementation, confirmed that the system and its components adhered to the specifications. Furthermore, verification ensured that the system and its services were functioning correctly. Mesosphere DC/OS, MongoDB, Apache Spark, the API, and the user interface were explicitly verified. Python was implicitly verified during the other components' verification.

Validation was performed during the last stage, which is evaluation. It was determined that the system was financially feasible and more affordable compared with the cloud services of the three largest cloud providers. This affordability makes an organisation cost-effective.

The general performance of the system was determined alongside the case study's original system and a default MySQL system. Based on a general mixture of queries, the developed system achieved an approximate yearly time saving of 52.5 minutes compared with the case study system, and 6.0 minutes compared with the MySQL system. Although these savings are small, the advanced security and dynamic nature of the system outweigh these savings.

Testing the performance of the three systems with increasing data volumes showed that the performance of the developed system was equal to that of the case study system. The MySQL system was the better performer, which indicates a possible area of improvement for the developed system.

The fourth validation method determined the performance of the three systems with multiple simultaneous connections, which are equivalent to users. For this validation method, the developed system was the best performer. The developed system achieved a performance increase of at least 24.72% compared with the case study system and 86.58% compared with the MySQL system. What this validation meant was that employees of the case study organisation are more efficient, and so too is the organisation.

In addition to this study, a journal article on the developed system was written by the author and submitted for publishing. The journal article provides the final validation that shows the operational impact of the system. The article shows how the system provided insights to increase sustainable energy consumption by assisting with energy-saving project identification. Appendix 4 provides the article, which is to be considered as a continuation of this dissertation.

Possible causes for the decreased performance of the system are due to its networked architecture and thorough security measures. However, a system cannot claim to support big data without employing horizontal scalability. Horizontal scalability means using distributed computers as if they were one to achieve greater performance. Therefore, horizontal scalability will always make use of computer networks, including their limitations. The final cause, security, is known to reduce performance. However, proper security makes an organisation safer and ensures privacy. Considering the case study's confidential data means that the system's security further improves its efficiency.

Finally, the system provides implicit benefits giving it a greater organisational impact. Firstly, data is implicitly backed up through redundant copies; meaning additional backups are not needed. Secondly, the system provides a platform for future big data projects and technologies. Lastly, performance can be furthered increased, or more data supported, by adding more computers, i.e. cluster nodes.

5.2 Study closure

When considering the system design and the choices made, the following was experienced:

- The choice of a hybrid architecture, which used characteristics from the Lambda architecture, polyglot persistence, and data lake, was the correct one. Using a hybrid architecture simplified the system into discrete units, which were easier to maintain and develop.
- Using microservices and Docker simplified the construction and deployment of the system. Again, the small software units, as ensured by microservices, were simple to create and rectify.
- Evaluating and testing NoSQL data stores were invaluable as it allowed the best software to be chosen for this study's context. The fact that so many NoSQL software packages were available made it difficult to make an informed decision. Therefore, some way of evaluating the software was needed.
- The software chosen, namely, Mesosphere DC/OS, MongoDB, Apache Spark, Apache Zeppelin, and Python, was experienced as very stable and reliable. No faults or errors were discovered on the part of the software. Human error caused any errors or faults.

Improvements were also identified, which are as follows:

- Using one computer for both a Mesosphere master and agent should be reconsidered. Although this idea resulted in greater computer utilisation, resources for software are reduced immediately. Limited resources were not a problem in this study. However, expansion of the system will be halted sooner rather than later. The best strategy remains to use one computer for either a Mesosphere master or agent.
- The API SOAP web service, although more secure, proved to be less compatible with programming languages other than C# .Net. A web service is still the best solution, but support for many programming languages should be a priority. It is not clear which alternative would be better, but a REST web service would be recommended.
- It would have been advantageous to determine the most used queries in the case study organisation. These queries could have been used in each of the performance validations for greater relevance.

Despite the limitations of the study, all research objectives were met. Each of the five research questions and specific objectives was answered and addressed in detail. The problem of the study was addressed and solved by the artefact. Finally, research contributions were made to the knowledge base as necessitated by design science research. This study and the publishing of a journal article make these contributions.

5.3 Recommendations for future work

Future studies can improve on this study by expanding to key-value, graph, and columnar data stores. This study focused on document data stores; however, the developed system provides a basis for polyglot persistence. Thus, any number of software packages can be deployed in the developed system.

This study can be applied to other contexts. The context of this study was the South African mining and industrial sectors. However, there are many contexts with data different from that described in this study. These other data types tie in with the previously mentioned expansion to other data stores. Thus, while the data of this study was relational with a partial structure, other contexts would require different solutions.

Cloud services were not extensively discussed in this study. However, the industry is moving to greater cloud service adoption. Therefore, an in-depth study can be performed to provide greater clarity regarding the decision for or against cloud services. Such a study could use the developed system and deploy it to the cloud. The cost comparison of this study did, however, consider cloud services. A greater investigation of all costs could still be done to compare the TCO of the developed system with that of cloud services.

One aspect that the study did not investigate was the perception of users and their usage of the system. Therefore, future research can study user adoption and usage of the system. User adoption is important for if users do not use a system to its full potential, time would have been wasted, and the full benefits would not be realised.

It was mentioned that increased security results in decreased performance. Therefore, there is a need to investigate the trade-off regarding performance and security. Such a study could consider different contexts to guide organisations in terms of best practices. Consequently, the quantification of the performance and security trade-off would be required.

Finally, this study was limited to the software and technology available at the time of writing. Future developments could provide better options for consideration. Thus, this study can be revisited with new technologies and software to improve upon the developed system.

BIBLIOGRAPHY

[1] S. Mohanty, M. Jagadeesh, and H. Srivatsa, *Big Data Imperatives: Enterprise Big Data Warehouse, BI Implementations and Analytics*, 1st ed. Berkeley, CA: Apress, 2013.

[2] D. Feinleib, *Big Data Bootcamp: What Managers Need to Know to Profit from the Big Data Revolution*, 1st ed. Berkeley, CA: Apress, 2014.

[3] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *Int. J. Inf. Manage.*, vol. 35, no. 2, pp. 137–144, Apr. 2015.

[4] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," McKinsey & Company, New York, NY, May 2011.

[5] P. B. Brandtzæg, "Big data: For better or worse," SINTEF ICT, Oslo, Norway, May 2013.

[6] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton, "The digital universe of opportunities: Rich data and increasing value of the internet of things," IDC, Framingham, MA, Apr. 2014.

[7] Y. Li, Z. Liu, and H. Zhu, "Enterprise search in the big data era," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1717–1718, Aug. 2014.

[8] N. Piedra, E. Tovar, R. Colomo-Palacios, J. Lopez-Vargas, and J. A. Chicaiza, "Consuming and producing linked open data: The case of OpenCourseWare," *Program*, vol. 48, no. 1, pp. 16–40, Jan. 2014.

[9] L. Wiese, *Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases*. Berlin, Germany: De Gruyter, 2015.

[10] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Greenwich, CT: Manning Publications, 2015.

[11] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Hoboken, NJ: Pearson Education, 2012.

[12] R. van der Meulen, "Gartner survey reveals investment in big data is up but fewer organizations plan to invest," *Gartner Newsroom*, 2016. [Online]. Available: http://www.gartner.com/newsroom/id/3466117. [Accessed: 26-Feb-2017].

[13] Knowledgent, "2015 Big data survey: Current implementation challenges," New York, NY, Apr. 2015.

[14] International Institute for Analytics, "Advanced analytics & big data adoption report," IIA and Dell Digital Business Services, Portland, OR, 2016.

[15] G. van Zyl, "SA's finance sector banking more on big data," *Fin24 Tech*, 2014. [Online]. Available: http://www.fin24.com/Tech/Companies/SAs-finance-sector-banking-more-on-big-data-20141207. [Accessed: 02-Mar-2017].

[16] W. Thompson, "Big data and data analytics to bring efficiency," *MoneyWeb*, 2017. [Online]. Available: https://www.moneyweb.co.za/mineweb/environment-and-sustainability/listen-big-data-and-data-analytics-is-going-to-bring-a-step-change-in-efficiency/. [Accessed: 02-Mar-2017].

[17] X. Jin, B. W. Wah, X. Cheng, and Y. Wang, "Significance and challenges of big data research," *Big Data Res.*, vol. 2, no. 2, pp. 59–64, Jun. 2015.

[18] A. Abbasi, S. Sarker, and R. H. L. Chiang, "Big data research in information systems: Toward an inclusive research agenda," *J. Assoc. Inf. Syst.*, vol. 17, no. 2, pp. 1–32, 2016.

[19] S. Zillner, T. Becker, R. Munné, K. Hussain, S. Rusitschka, H. Lippell, E. Curry, and A. Ojo, "Big data-driven innovation in industrial sectors," in *New Horizons for a Data-Driven Economy*, Cham, Switzerland: Springer International Publishing, 2016, pp. 169–178.

[20] W. A. Günther, M. H. Rezazade Mehrizi, M. Huysman, and F. Feldberg, “Debating big data: A literature review on realizing value from big data,” *J. Strateg. Inf. Syst.*, vol. 26, no. 3, pp. 191–209, Sep. 2017.

[21] S. F. Wamba, S. Akter, A. Edwards, G. Chopin, and D. Gnanzou, “How ‘big data’ can make big impact: Findings from a systematic review and a longitudinal case study,” *Int. J. Prod. Econ.*, vol. 165, pp. 234–246, Jul. 2015.

[22] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. S. Netto, and R. Buyya, “Big data computing and clouds: Trends and future directions,” *J. Parallel Distrib. Comput.*, vol. 79–80, pp. 3–15, May 2015.

[23] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, “The rise of ‘big data’ on cloud computing: Review and open research issues,” *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015.

[24] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, Dec. 2007.

[25] M. P. Uysal, “Towards a software engineering research framework: Extending design science research,” *Int. Res. J. Eng. Technol.*, vol. 3, no. 2, pp. 22–26, 2016.

[26] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2008.

[27] A. R. Hevner, “A three cycle view of design science research,” *Scand. J. Inf. Syst.*, vol. 19, no. 2, pp. 87–92, 2007.

[28] A. Hevner and S. Chatterjee, “Design science research in information systems,” in *Design Research in Information Systems: Theory and Practice*, Boston, MA: Springer US, 2010, pp. 9–22.

[29] J. van Aken, A. Chandrasekaran, and J. Halman, “Conducting and publishing design science research,” *J. Oper. Manag.*, vol. 47–48, pp. 1–8, Nov. 2016.

[30] P. Ponniah, *Database Design and Development: An Essential Guide for IT Professionals*. Hoboken, NJ: John Wiley & Sons, 2003.

[31] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 9th ed. Essex, England: Pearson Education, 2013.

[32] E. C. Foster and S. V. Godbole, *Database Systems: A Pragmatic Approach*, 2nd ed. Berkeley, CA: Apress, 2016.

[33] M. Chen, S. Mao, and Y. Liu, “Big data: A survey,” *Mob. Networks Appl.*, vol. 19, no. 2, pp. 171–209, Apr. 2014.

[34] A. B. M. Moniruzzaman and S. A. Hossain, “NoSQL database: New era of databases for big data analytics: Classification, characteristics and comparison,” *Int. J. Database Theory Appl.*, vol. 6, no. 4, pp. 1–14, 2013.

[35] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*, 2nd ed. Sebastopol, CA: O’Reilly Media, 2015.

[36] E. Brewer, “CAP twelve years later: How the ‘rules’ have changed,” *IEEE Comput.*, vol. 45, no. 2, pp. 23–29, Feb. 2012.

[37] A. Fox and E. A. Brewer, “Harvest, yield, and scalable tolerant systems,” in *Proc. Seventh Workshop Hot Topics Operating Systems*, 1999, pp. 174–178.

[38] E. A. Brewer, “Towards robust distributed systems,” in *Proc. Nineteenth Annu. ACM Symp. Principles Distributed Computing – PODC ’00*, 2000, p. 7.

[39] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, “Cluster-based scalable network services,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 78–91, Dec. 1997.

[40] D. Lee, J. Choi, and H. Shin, “A scalable and flexible repository for big sensor data,” *IEEE Sens. J.*, vol. 15, no. 12, pp. 7284–7294, Dec. 2015.

[41] E. Zeydan, E. Bastug, M. Bennis, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah, “Big data caching for networking: Moving from cloud to edge,” *IEEE Commun. Mag.*, vol. 54, no. 9, pp. 36–42, Sep. 2016.

[42] N. Sadashiv and S. M. D. Kumar, “Cluster, grid and cloud computing: A detailed comparison,” in *2011 Sixth Int. Conf. Comput. Sci. & Edu. (ICCSE)*, 2011, pp. 477–482.

[43] A. Li, X. Yang, S. Kandula, and M. Zhang, “Comparing public-cloud providers,” *IEEE Internet Comput.*, vol. 15, no. 2, pp. 50–53, Mar. 2011.

[44] H. Jiang, Q. Yu, and K. Huang, “Design and implementation of an improved cloud storage system,” in *2016 Twelfth Int. Conf. Natural Computation, Fuzzy Syst. and Knowledge Discovery (ICNC-FSKD)*, 2016, pp. 1816–1823.

[45] Y.-S. Kang, I.-H. Park, J. Rhee, and Y.-H. Lee, “MongoDB-based repository design for IoT-generated RFID/sensor big data,” *IEEE Sens. J.*, vol. 16, no. 2, pp. 485–497, Jan. 2016.

[46] W. Zola, “6 Rules of thumb for MongoDB schema design: Part 1,” *MongoDB Blog*, 2014. [Online]. Available: http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1. [Accessed: 14-Nov-2016].

[47] W. Zola, “6 Rules of thumb for MongoDB schema design: Part 2,” *MongoDB Blog*, 2014. [Online]. Available: http://blog.mongodb.org/post/87892923503/6-rules-of-thumb-for-mongodb-schema-design-part-2. [Accessed: 14-Nov-2016].

[48] W. Zola, “6 Rules of thumb for MongoDB schema design: Part 3,” *MongoDB Blog*, 2014. [Online]. Available: http://blog.mongodb.org/post/88473035333/6-rules-of-thumb-for-mongodb-schema-design-part-3. [Accessed: 14-Nov-2016].

[49] S. Banerjee, R. Shaw, A. Sarkar, and N. C. Debnath, “Towards logical level design of big data,” in *2015 IEEE Thirteenth Int. Conf. Ind. Informatics (INDIN)*, 2015, pp. 1665–1671.

[50] A. Sarkar, “Conceptual level design of semi-structured database system: Graph-semantic based approach,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 2, no. 10, pp. 112–121, 2011.

[51] E. Begoli and J. Horey, “Design principles for effective knowledge discovery from big data,” in *2012 Joint Working IEEE/IFIP Conf. Software Architecture and European Conf. Software Architecture*, 2012, pp. 215–218.

[52] H. M. Chen, R. Kazman, and S. Haziyev, “Agile big data analytics for web-based systems: An architecture-centric approach,” *IEEE Trans. Big Data*, vol. 2, no. 3, pp. 234–248, Sep. 2016.

[53] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “DevOps,” *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, May 2016.

[54] M. Callanan and A. Spillane, “DevOps: Making it easy to do the right thing,” *IEEE Softw.*, vol. 33, no. 3, pp. 53–59, May 2016.

[55] L. Zhu, L. Bass, and G. Champlin-Scharff, “DevOps and its practices,” *IEEE Softw.*, vol. 33, no. 3, pp. 32–34, May 2016.

[56] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, “Lambda architecture for cost-effective batch and speed big data processing,” in *2015 IEEE Int. Conf. Big Data (Big Data)*, 2015, pp. 2785–2792.

[57] R. Ranjan, “Streaming big data processing in datacenter clouds,” *IEEE Cloud Comput.*, vol. 1, no. 1, pp. 78–83, May 2014.

[58] M. Fowler, “PolyglotPersistence,” *Martin Fowler*, 2011. [Online]. Available: https://martinfowler.com/bliki/PolyglotPersistence.html. [Accessed: 10-Apr-2017].

[59] K. Kaur and R. Rani, “Managing data in healthcare information systems: Many models, one solution,” *IEEE Comput.*, vol. 48, no. 3, pp. 52–59, Mar. 2015.

[60] K. Kaur and R. Rani, “A smart polyglot solution for big data in healthcare,” *IT Prof.*, vol. 17, no. 6, pp. 48–55, Nov. 2015.

[61] S. Prasad and S. B. Avinash, “Application of polyglot persistence to enhance performance of the energy data management systems,” in *2014 Int. Conf. Advances Electron., Comput. and Commun.*, 2014, pp. 1–6.

[62] S. Prasad and M. S. N. Sha, “NextGen data persistence pattern in healthcare: Polyglot persistence,” in *2013 Fourth Int. Conf. Computing, Commun. and Networking Technol. (ICCCNT)*, 2013, pp. 1–8.

[63] N. Miloslavskaya and A. Tolstoy, “Application of big data, fast data, and data lake concepts to information security issues,” in *2016 IEEE Fourth Int. Conf. Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 148–153.

[64] J. Dixon, “Pentaho, Hadoop, and data lakes,” *James Dixon’s Blog*, 2010. [Online]. Available: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. [Accessed: 21-Feb-2017].

[65] J. Dixon, “Data lakes revisited,” *James Dixon’s Blog*, 2014. [Online]. Available: https://jamesdixon.wordpress.com/2014/09/25/data-lakes-revisited/. [Accessed: 22-Feb-2017].

[66] D. E. O’Leary, “Embedding AI and crowdsourcing in the big data lake,” *IEEE Intell. Syst.*, vol. 29, no. 5, pp. 70–73, Sep. 2014.

[67] H. J. Watson, “Data lakes, data labs, and sandboxes,” *Bus. Intell. J.*, vol. 20, no. 1, pp. 4–7, 2015.

[68] D. Clegg, “Evolving data warehouse and BI architectures: The big data challenge,” *Bus. Intell. J.*, vol. 20, no. 1, pp. 19–24, 2015.

[69] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, “The evolution of distributed systems towards microservices architecture,” in *2016 Eleventh Int. Conf. Internet Technol. and Secured Trans. (ICITST)*, 2016, pp. 318–325.

[70] N. Serrano, J. Hernantes, and G. Gallardo, “Service-oriented architecture and legacy systems,” *IEEE Softw.*, vol. 31, no. 5, pp. 15–19, Sep. 2014.

[71] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, “SOAP version 1.2 Part 1: Messaging framework,” 2007.

[72] J. Thönes, “Microservices,” *IEEE Softw.*, vol. 32, no. 1, pp. 116–116, Jan. 2015.

[73] C. Szyperski, M. Petitclerc, and R. Barga, “Three experts on big data engineering,” *IEEE Softw.*, vol. 33, no. 2, pp. 68–72, Mar. 2016.

[74] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “Microservices in practice, Part 1: Reality check and service design,” *IEEE Softw.*, vol. 34, no. 1, pp. 91–98, Jan. 2017.

[75] M. Richards, *Microservices vs. Service-oriented Architecture*. Sebastopol, CA: O’Reilly Media, 2015.

[76] A. Singleton, “The economics of microservices,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 16–20, Sep. 2016.

[77] R. Stephens, *Beginning Software Engineering*, 1st ed. Indianapolis, IN: John Wiley & Sons, 2015.

[78] A. Sill, “The design and architecture of microservices,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 76–80, Sep. 2016.

[79] D. S. Linthicum, “Practical use of microservices in moving workloads to the cloud,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 6–9, Sep. 2016.

[80] C. Pahl, “Containerization and the PaaS cloud,” *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, May 2015.

[81] Docker Inc., “What is a container,” *Docker*, 2016. [Online]. Available: https://www.docker.com/what-container. [Accessed: 06-Apr-2017].

[82] D. Bernstein, “Containers and cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[83] The Linux Foundation, “Open Container Initiative: About,” *Open Container Initiative*, 2015. [Online]. Available: https://www.opencontainers.org/about. [Accessed: 06-Apr-2017].

[84] CoreOS Inc., “rkt vs other projects,” *CoreOS*, 2017. [Online]. Available: https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html. [Accessed: 06-Apr-2017].

[85] CoreOS Inc., “App Container basics,” *CoreOS*, 2017. [Online]. Available: https://coreos.com/rkt/docs/latest/app-container.html. [Accessed: 06-Apr-2017].

[86] N. Kratzke, “A lightweight virtualization cluster reference architecture derived from open source PaaS platforms,” *Open J. Mob. Comput. Cloud Comput.*, vol. 1, no. 2, pp. 17–30, 2014.

[87] J. Wood, “Container orchestration: Moving from fleet to Kubernetes,” *CoreOS Blog*, 2017. [Online]. Available: https://coreos.com/blog/migrating-from-fleet-to-kubernetes.html. [Accessed: 01-Jun-2017].

[88] J. Crandall and P. Lipton, “OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC,” *OASIS*, 2017. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. [Accessed: 01-Jun-2017].

[89] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, “Portable cloud services using TOSCA,” *IEEE Internet Comput.*, vol. 16, no. 3, pp. 80–85, May 2012.

[90] K. Liu, K. Aida, S. Yokoyama, and Y. Masatani, “Flexible container-based computing platform on cloud for scientific workflows,” in *2016 Int. Conf. Cloud Computing Res. and Innovations (ICCCRI)*, 2016, pp. 56–63.

[91] M. Stillwell and J. G. F. Coutinho, “A DevOps approach to integration of software components in an EU research project,” in *Proc. First Int. Workshop Quality-Aware DevOps – QUDOS 2015*, 2015, pp. 1–6.

[92] D. Cao, P. Liu, W. Cui, Y. Zhong, and B. An, “Cluster as a service: A resource sharing approach for private cloud,” *Tsinghua Sci. Technol.*, vol. 21, no. 6, pp. 610–619, Dec. 2016.

[93] M. Fazio, A. Celesti, A. Puliafito, and M. Villari, “Big data storage in the cloud for smart environment monitoring,” *Procedia Comput. Sci.*, vol. 52, pp. 500–506, 2015.

[94] V. D. Le, M. M. Neff, R. V. Stewart, R. Kelley, E. Fritzinger, S. M. Dascalu, and F. C. Harris, “Microservice-based architecture for the NRDC,” in *2015 IEEE Thirteenth Int. Conf. Ind. Informatics (INDIN)*, 2015, pp. 1659–1664.

[95] D. Feinberg, M. Adrian, N. Heudecker, A. M. Ronthal, and T. Palanca, “Magic quadrant for operational database management systems,” Gartner, Stamford, CT, Nov. 2017.

[96] Couchbase Inc., “Couchbase Server,” *Couchbase*, 2017. [Online]. Available: https://www.couchbase.com/nosql-databases/couchbase-server. [Accessed: 06-May-2017].

[97] Couchbase Inc., “Couchbase Server Editions,” *Couchbase Server Developer*, 2017. [Online]. Available: https://developer.couchbase.com/documentation/server/4.6/introduction/editions.html. [Accessed: 06-May-2017].

[98] Apache Software Foundation, “Apache HBase,” *Apache HBase*, 2017. [Online]. Available: https://hbase.apache.org/index.html. [Accessed: 06-May-2017].

[99] Apache Software Foundation, “Apache HBase reference guide,” *Apache HBase*, 2017. [Online]. Available: http://hbase.apache.org/book.html. [Accessed: 06-May-2017].

[100] MongoDB Inc., “What is MongoDB?,” *MongoDB*, 2017. [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Accessed: 06-May-2017].

[101] Apache Software Foundation, “What is Cassandra?,” *Apache Cassandra*, 2017. [Online]. Available: http://cassandra.apache.org/. [Accessed: 06-May-2017].

[102] Apache Software Foundation, “Apache Cassandra,” *Apache Cassandra Git Repository*, 2017. [Online]. Available: https://github.com/apache/cassandra. [Accessed: 06-May-2017].

[103] Apache Software Foundation, “Apache Hive,” *Apache Hive*, 2017. [Online]. Available: https://hive.apache.org/. [Accessed: 06-May-2017].

[104] Apache Software Foundation, “Apache Hive Home,” *Apache Hive Documentation*, 2017. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/Home. [Accessed: 06-May-2017].

[105] Cloudera Inc., “Apache Impala (incubating): Interactive SQL,” *Cloudera*, 2017. [Online]. Available: http://www.cloudera.com/documentation/enterprise/latest/topics/impala.html. [Accessed: 06-May-2017].

[106] Redis Labs Inc., “Introduction to Redis,” *Redis*, 2017. [Online]. Available: https://redis.io/topics/introduction. [Accessed: 06-May-2017].

[107] Redis Labs Inc., “Redis Documentation,” *Redis*, 2017. [Online]. Available: https://redis.io/documentation. [Accessed: 06-May-2017].

[108] N. Yuhanna, L. Gene, and C. Austin, “The Forrester Wave™: Big data NoSQL, Q3 2016,” Forrester Research Inc., Cambridge, MA, Aug. 2016.

[109] Mesosphere Inc., “DC/OS,” *Mesosphere DC/OS*, 2017. [Online]. Available: https://dcos.io/. [Accessed: 19-May-2017].

[110] The Kubernetes Authors, “Kubernetes,” *Kubernetes*, 2017. [Online]. Available: https://kubernetes.io/. [Accessed: 19-May-2017].

[111] GigaSpaces Technologies Inc., “Cloudify,” *Cloudify*, 2017. [Online]. Available: http://cloudify.co/. [Accessed: 19-May-2017].

[112] Mesosphere Inc., “DC/OS system requirements,” *Mesosphere DC/OS*, 2017. [Online]. Available: https://dcos.io/docs/1.9/installing/custom/system-requirements/#master-nodes. [Accessed: 20-May-2017].

[113] MongoDB Inc., “BSON,” *BSON specification*, 2017. [Online]. Available: http://bsonspec.org/. [Accessed: 22-May-2017].

[114] Ecma International, “The JSON data interchange format,” Standard ECMA-404, Geneva, Switzerland, 2013.

[115] MongoDB Inc., “The MongoDB manual,” *MongoDB Server Documentation*, 2017. [Online]. Available: https://docs.mongodb.com/manual/. [Accessed: 22-May-2017].

[116] MongoDB Inc., “Sharded cluster components,” *MongoDB Server Documentation*, 2017. [Online]. Available: https://docs.mongodb.com/manual/core/sharded-cluster-components/. [Accessed: 22-May-2017].

[117] Apache Software Foundation, “Apache Spark,” *Apache Spark*, 2017. [Online]. Available: https://spark.apache.org/. [Accessed: 25-May-2017].

[118] Apache Software Foundation, “Apache Storm,” *Apache Storm*, 2017. [Online]. Available: http://storm.apache.org/. [Accessed: 25-May-2017].

[119] Apache Software Foundation, “Apache Hadoop,” *Apache Hadoop*, 2017. [Online]. Available: http://hadoop.apache.org/. [Accessed: 25-May-2017].

[120] Apache Software Foundation, “Apache Flink,” *Apache Flink*, 2017. [Online]. Available: https://flink.apache.org/. [Accessed: 25-May-2017].

[121] H. Fang, “Managing data lakes in big data era: What’s a data lake and why has it became popular in data management ecosystem,” in *2015 IEEE Int. Conf. Cyber Technol. Automation, Control, and Intell. Syst. (CYBER)*, 2015, pp. 820–824.

[122] Apache Software Foundation, “Spark standalone mode,” *Apache Spark*, 2017. [Online]. Available: https://spark.apache.org/docs/latest/spark-standalone.html. [Accessed: 27-May-2017].

[123] Python Software Foundation, “Python,” *Python*, 2017. [Online]. Available: https://www.python.org/. [Accessed: 25-May-2017].

[124] Microsoft Corporation, “Windows Communication Foundation,” *Microsoft Developer Network*, 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx. [Accessed: 26-May-2017].

[125] Microsoft Corporation, “Microsoft Windows Server 2016 Server Core,” *Docker Hub Microsoft Repository*, 2017. [Online]. Available: https://hub.docker.com/r/microsoft/windowsservercore/. [Accessed: 29-May-2017].

[126] Two Sigma Open Source, “Beaker,” *Beaker*, 2017. [Online]. Available: http://beakernotebook.com/. [Accessed: 26-May-2017].

[127] Apache Software Foundation, “Apache Zeppelin,” *Apache Zeppelin*, 2017. [Online]. Available: https://zeppelin.apache.org/. [Accessed: 26-May-2017].

[128] Project Jupyter, “Jupyter Notebook,” *Jupyter*, 2017. [Online]. Available: http://jupyter.org/. [Accessed: 26-May-2017].

[129] Apache Software Foundation, “Apache Zeppelin quick start,” *Apache Zeppelin*, 2017. [Online]. Available: https://zeppelin.apache.org/docs/latest/install/install.html. [Accessed: 29-May-2017].

[130] Mesosphere Inc., “Advanced DC/OS installation guide,” *Mesosphere DC/OS*, 2017. [Online]. Available: https://dcos.io/docs/1.9/installing/custom/advanced/. [Accessed: 26-Jul-2017].

[131] D. D. Wackerly, W. Mendenhall, and R. L. Scheaffer, “Conditional probability and the independence of events,” in *Mathematical Statistics with Applications*, 7th ed., Belmont, CA: Brooks/Cole Cengage Learning, 2008, pp. 51–56.

[132] MongoDB Inc., “MongoDB architecture guide,” New York, NY, Jun. 2016.

[133] MongoDB Inc., “Replica set member states,” *MongoDB Server Documentation*, 2017. [Online]. Available: https://docs.mongodb.com/manual/reference/replica-states/#replica-set-member-states. [Accessed: 09-Aug-2017].

[134] G. Combs, “Wireshark,” *Wireshark*, 2017. [Online]. Available: https://www.wireshark.org/. [Accessed: 27-Aug-2017].

[135] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, IETF, Fremont, CA, 2008.

[136] L. Leong, R. Bala, C. Lowery, and D. Smith, “Magic quadrant for cloud infrastructure as a service, worldwide,” Gartner, Stamford, CT, Jun. 2017.

[137] C. Shen, E. Nahum, H. Schulzrinne, and C. P. Wright, “The impact of TLS on SIP server performance: Measurement and modeling,” *IEEE/ACM Trans. Netw.*, vol. 20, no. 4, pp. 1217–1230, Aug. 2012.

[138] S. Muller, D. Bermbach, S. Tai, and F. Pallas, “Benchmarking the performance impact of transport layer security in cloud database systems,” in *2014 IEEE Int. Conf. Cloud Eng.*, 2014, pp. 27–36.

[139] Synopsys Inc., “Data-driven software security insights: TLS performance overhead for MongoDB,” San Francisco, CA, 2017.

APPENDIX 1 DATABASE SYSTEM EVALUATION RESULTS

The complete database system evaluation results, as discussed in Section 3.3, are given in Tables 16–18. An explanation of what each query entailed is provided below.

- Set 1 Query 1: Retrieved the year, month, day, tag name, and average value grouped by year, month, day, and tag name. That is, this query retrieved daily averages for each tag over the entire period.
- Set 2 Query 1: Retrieved all data for a specific tag by searching for its tag name.
- Set 2 Query 2: Retrieved all data for all tags.
- Set 2 Query 3: Retrieved all data in the year 2016 in the 5th month for a specific tag.
- Set 2 Query 4: Retrieved all data in the year 2016 in the 5th month for all tags.
- Set 2 Query 5: Retrieved all data in the year 2016 in the 20th week for a specific tag.
- Set 2 Query 6: Retrieved all data in the year 2016 in the 20th week for all tags.
- Set 2 Query 7: Retrieved all data in the year 2016 in the 5th month on the 4th day for a specific tag.
- Set 2 Query 8: Retrieved all data in the year 2016 in the 5th month on the 4th day for all tags.
- Set 3 Query 1: Retrieved the value attribute or column of all data for a specific tag.
- Set 3 Query 2: Retrieved the value attribute or column of all data for all tags.
- Set 3 Query 3: Retrieved the value attribute or column of all data in the year 2016 in the 5th month of a specific tag.
- Set 3 Query 4: Retrieved the value attribute or column of all data in the year 2016 in the 5th month for all tags.
- Set 3 Query 5: Retrieved the value attribute or column of all data in the year 2016 in the 20th week of a specific tag.
- Set 3 Query 6: Retrieved the value attribute or column of all data in the year 2016 in the 20th week for all tags.
- Set 3 Query 7: Retrieved the value attribute or column of all data in the year 2016 in the 5th month on the 4th day of a specific tag.
- Set 3 Query 8: Retrieved the value attribute or column of all data in the year 2016 in the 5th month on the 4th day for all tags.
- Set 4 Query 1: Retrieved the average of the value attribute or column for all data of a specific tag. That is, the query calculated the global arithmetic mean over all the values of a specific tag.
- Set 4 Query 2: Retrieved the average of the value attribute or column grouped by tag name for all data of all tags. That is, the query calculated the global arithmetic mean over all the values for each tag individually.
- Set 4 Query 3: Retrieved the average of the value attribute or column for all data in the year 2016 in the 5th month of a specific tag.

- Set 4 Query 4: Retrieved the average of the value attribute or column grouped by tag name for all data in the year 2016 in the 5th month of all tags.
- Set 4 Query 5: Retrieved the average of the value attribute or column for all data in the year 2016 in the 20th week of a specific tag.
- Set 4 Query 6: Retrieved the average of the value attribute or column grouped by tag name for all data in the year 2016 in the 20th week of all tags.
- Set 4 Query 7: Retrieved the average of the value attribute or column for all data in the year 2016 in the 5th month on the 4th day of a specific tag.
- Set 4 Query 8: Retrieved the average of the value attribute or column grouped by tag name for all data in the year 2016 in the 5th month on the 4th day of all tags.
- Set 4 Query 9: Retrieved the average of the value attribute or column for all data. That is, the query calculated an arithmetic mean over all the values of all tags.
- Set 4 Query 10: Retrieved the average of the value attribute or column for all data in the year 2016 in the 5th month.
- Set 4 Query 11: Retrieved the average of the value attribute or column for all data in the year 2016 in the 20th week.
- Set 4 Query 12: Retrieved the average of the value attribute or column for all data in the year 2016 in the 5th month on the 4th day.

Table 16: Database system execution times in seconds for part of the tests

Test	MongoDB			Impala			Couchbase			MySQL		
	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case
Set 1 Query 1	7.42	10.55	33.86	10.84	16.02	54.07	808.94	834.38	850.82	38.71	39.65	43.00
Set 2 Query 1	2.53	2.57	2.77	8.28	11.11	17.39	513.93	525.17	542.95	2.75	2.84	3.08
Set 2 Query 2	186.70	573.37	1 110.81	751.85	967.28	1 672.14	NA	NA	NA	140.67	142.27	145.32
Set 2 Query 3	13.59	22.02	115.11	6.77	13.86	224.40	599.67	615.05	623.59	2.15	2.26	2.85
Set 2 Query 4	3.21	3.45	5.51	41.91	57.35	96.61	618.62	641.03	659.95	9.11	9.26	9.43
Set 2 Query 5	2.03	2.07	2.47	3.90	6.43	10.82	700.32	718.67	731.15	1.71	1.77	1.85
Set 2 Query 6	2.27	2.32	2.64	12.45	16.50	27.63	674.45	693.75	703.53	3.54	3.59	3.65
Set 2 Query 7	1.99	2.02	2.09	3.73	6.02	10.59	641.08	661.78	673.48	1.69	1.74	1.82
Set 2 Query 8	2.02	2.05	2.10	4.57	7.26	11.47	611.40	628.13	639.21	2.01	2.07	2.15
Set 3 Query 1	2.53	2.63	3.15	1.25	2.31	4.64	504.01	520.35	532.47	1.78	1.83	1.90
Set 3 Query 2	10.92	11.05	11.85	99.00	131.38	231.54	1 174.92	1 237.07	1 305.20	58.41	59.15	59.97
Set 3 Query 3	2.17	2.25	3.44	2.88	4.95	30.50	708.70	814.07	910.90	1.67	1.72	1.80

Table 17: Database system execution times in seconds for second part of the tests

Test	MongoDB			Impala			Couchbase			MySQL		
	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case
Set 3 Query 4	2.61	2.65	2.77	7.32	10.24	17.27	619.20	648.74	666.05	4.39	4.48	4.56
Set 3 Query 5	2.08	2.11	2.17	2.75	4.46	12.26	703.96	721.35	745.55	1.54	1.59	1.66
Set 3 Query 6	2.17	2.22	3.95	3.29	5.26	19.83	668.16	688.87	702.04	2.11	2.16	2.23
Set 3 Query 7	2.03	2.08	2.97	2.49	4.52	16.88	645.21	661.69	676.33	1.53	1.59	1.68
Set 3 Query 8	2.04	2.08	2.41	2.35	4.47	11.88	607.29	626.48	639.10	1.47	1.52	1.59
Set 4 Query 1	3.14	6.22	12.17	0.94	2.25	3.81	502.34	514.73	525.59	1.32	1.37	1.42
Set 4 Query 2	4.60	7.99	10.80	1.66	2.91	6.58	585.30	602.63	614.73	9.28	9.43	9.59
Set 4 Query 3	2.14	2.18	2.26	2.55	4.39	7.30	594.58	613.03	628.67	1.52	1.58	1.67
Set 4 Query 4	2.24	2.29	2.47	2.87	4.28	8.03	575.13	587.88	599.21	2.00	2.06	2.13
Set 4 Query 5	2.04	2.07	2.13	2.55	4.29	7.45	643.89	664.47	679.24	1.52	1.58	1.66
Set 4 Query 6	2.07	2.10	2.17	2.81	4.42	7.44	663.82	680.06	692.17	1.79	1.85	1.91
Set 4 Query 7	2.00	2.04	2.76	2.37	4.31	8.75	644.38	660.30	671.67	1.54	1.59	1.71

Table 18: Database system execution times in seconds for final part of the tests

Test	MongoDB			Impala			Couchbase			MySQL		
	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case	Best-case	Mean	Worst-case
Set 4 Query 8	2.00	2.04	2.11	2.69	4.31	7.58	613.83	625.42	639.42	1.69	1.76	1.85
Set 4 Query 9	2.76	3.12	5.62	1.15	2.56	4.81	529.95	541.83	553.19	1.07	1.12	1.16
Set 4 Query 10	2.13	2.18	2.56	2.01	3.94	6.73	566.63	583.09	598.62	1.36	1.42	1.50
Set 4 Query 11	2.03	2.06	2.11	2.52	4.21	7.51	567.38	580.01	593.79	1.40	1.45	1.51
Set 4 Query 12	1.99	2.02	2.08	2.13	3.96	7.07	609.04	624.31	634.95	1.37	1.43	1.48

APPENDIX 2 PYTHON VERIFICATION PROGRAM CODE

The source code of the Python program used in Sections 4.3.2, 4.3.3, and 4.3.5 is given below.

```
from pymongo import MongoClient
import ssl
import datetime
import copy
import numpy as np
from bson.son import SON


def executeMongoDBQueries():
    """Performs all of the verification queries for MongoDB.

    All the verification queries are performed with the execution of this
function.
    If a certain test fails an error will be printed and the script will
exit.
    """
    # Print a message stating that the tests will start.
    print("Starting tests...")

    # Establish a connection to the MongoDB service.
    ssl_directory = r'C:\Users\Jacobus
Herman\Documents\Masters_degree_code\Verification\Mongo_SSL_certs'
    client = MongoClient(host='192.168.0.76',
                         port=27017,
                         ssl=True,
                         ssl_certfile=ssl_directory + r'\mongo-client.pem',
                         ssl_ca_certs=ssl_directory + r'\rootCA.pem',
                         ssl_pem_passphrase='Hv@cint006',
                         ssl_cert_reqs=ssl.CERT_NONE)

    if not client["Test"].authenticate('Gandalf', r'Hv@cint006',
source='admin'):
        raise RuntimeError("Could not authenticate with the provided
credentials.")

    database = client["Test"]
    database.drop_collection("Tag")
    database.drop_collection("Value")
    tag_collection = database.Tag
    value_collection = database.Value

    # Perform the insertion test.
    tag_names = [cTagName("TagName 0", datetime.datetime(2014, 1, 1, 0, 0,
0))]
    site_tag_names = [cTagName("SiteTagName 0", datetime.datetime(2014, 1, 1,
0, 0, 0))]
    interval_mask = cIntervalMask(0, 0, 0, 0, 1, 0, 0)
    tag = cTagDocument(tag_names, site_tag_names, interval_mask, "GWh",
"PDI", "Test description 0", "Number")
    tag_to_insert = tag.toDict()
    tag_id = tag_collection.insert_one(tag_to_insert).inserted_id
    tag_from_db = tag_collection.find_one({"_id": tag_id})
    tag_read = cTagDocument().fromDict(tag_from_db)
    assert tag_read == tag

    values_to_insert = list()
```

```
    date_to_use = datetime.datetime(2016, 1, 1, 0, 10, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 0, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 0, 30, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
3000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 0, 40, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 0, 50, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 0, 55, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 2, 15, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
3000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 2, 45, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 3, 31, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1000.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 1, 3, 59, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2000.0).toDict())

    date_to_use = datetime.datetime(2016, 1, 2, 0, 10, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 0, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
5.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 0, 30, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
3.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 0, 40, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
4.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 0, 50, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
9.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 0, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
17.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 10, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
0.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
5.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 30, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
0.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 40, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 1, 50, 0)
```

```
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
4.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 2, 0, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
7.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 2, 10, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
9.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 2, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
2.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 2, 2, 30, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
1.0).toDict())

    date_to_use = datetime.datetime(2016, 1, 3, 0, 0, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
5.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 0, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
17.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 0, 40, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
9.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 0, 55, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
3.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 1, 5, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
5.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 1, 20, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
8.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 1, 40, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
10.0).toDict())
    date_to_use = datetime.datetime(2016, 1, 3, 1, 50, 0)
    values_to_insert.append(cValueDocument(date_to_use, tag_id,
6.0).toDict())

    result = value_collection.insert_many(values_to_insert)
    assert len(result.inserted_ids) == 33

    # Perform the sum test.
    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 1, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 8000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 3, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 1, 0, 0, 0),
                                        datetime.datetime(2016, 1, 1, 4, 0,
0), [0, 0, 0, 0, 30], "sum")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 2, 0, 0, 0)
```

```
    expected_values.append(cValueDocument(date_to_use, tag_id, 6.0))
    date_to_use = datetime.datetime(2016, 1, 2, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 16.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 22.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 5.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 18.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 2, 0, 0, 0),
                                        datetime.datetime(2016, 1, 2, 2, 30,
0), [0, 0, 0, 0, 30], "sum")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 3, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 22.0))
    date_to_use = datetime.datetime(2016, 1, 3, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 12.0))
    date_to_use = datetime.datetime(2016, 1, 3, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 13.0))
    date_to_use = datetime.datetime(2016, 1, 3, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 16.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 3, 0, 0, 0),
                                        datetime.datetime(2016, 1, 3, 1, 50,
0), [0, 0, 0, 0, 30], "sum")
    assert np.array_equal(actual_values, expected_values)

    # Perform the min test.
    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 1, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 3, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1000.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 1, 0, 0, 0),
                                        datetime.datetime(2016, 1, 1, 4, 0,
0), [0, 0, 0, 0, 30], "min")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 2, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1.0))
    date_to_use = datetime.datetime(2016, 1, 2, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 0.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 0.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1.0))
```

```
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 2, 0, 0, 0),
                                        datetime.datetime(2016, 1, 2, 2, 30,
0), [0, 0, 0, 0, 30], "min")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 3, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 5.0))
    date_to_use = datetime.datetime(2016, 1, 3, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3.0))
    date_to_use = datetime.datetime(2016, 1, 3, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 5.0))
    date_to_use = datetime.datetime(2016, 1, 3, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 6.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 3, 0, 0, 0),
                                        datetime.datetime(2016, 1, 3, 1, 50,
0), [0, 0, 0, 0, 30], "min")
    assert np.array_equal(actual_values, expected_values)

    # Perform the max test.
    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 1, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 3000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2000.0))
    date_to_use = datetime.datetime(2016, 1, 1, 3, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 2000.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 1, 0, 0, 0),
                                        datetime.datetime(2016, 1, 1, 4, 0,
0), [0, 0, 0, 0, 30], "max")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 2, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 5.0))
    date_to_use = datetime.datetime(2016, 1, 2, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 9.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 17.0))
    date_to_use = datetime.datetime(2016, 1, 2, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 4.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 9.0))
    date_to_use = datetime.datetime(2016, 1, 2, 2, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 1.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 2, 0, 0, 0),
                                        datetime.datetime(2016, 1, 2, 2, 30,
0), [0, 0, 0, 0, 30], "max")
    assert np.array_equal(actual_values, expected_values)

    expected_values = list()
    date_to_use = datetime.datetime(2016, 1, 3, 0, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 17.0))
    date_to_use = datetime.datetime(2016, 1, 3, 0, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 9.0))
```

```
    date_to_use = datetime.datetime(2016, 1, 3, 1, 0, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 8.0))
    date_to_use = datetime.datetime(2016, 1, 3, 1, 30, 0)
    expected_values.append(cValueDocument(date_to_use, tag_id, 10.0))
    actual_values = getAggregatedValues(client, "Test", tag_id,
datetime.datetime(2016, 1, 3, 0, 0, 0),
                                        datetime.datetime(2016, 1, 3, 1, 50,
0), [0, 0, 0, 0, 30], "max")
    assert np.array_equal(actual_values, expected_values)

    # Close the connection to the Mongo database.
    client.close()

    # Print a message stating that the tests are finished.
    print("Finished tests.")


def getAggregatedValues(mongo_client, database_name, tag_id, begin_date,
end_date, aggregation_period,
                        calculation_command):
    collection_to_use = mongo_client[database_name]["Value"]
    first_match_stage = {"$match": {"TagID": tag_id}}
    first_time_match = {"$match": {"Timestamp": {"$gte": begin_date}}}
    second_time_match = {"$match": {"Timestamp": {"$lte": end_date}}}

    # Create the sort stage.
    sort_stage = {"$sort": {}}
    preceding_field = "_id"

    for index, element in enumerate(aggregation_period):
        if element <= 0:
            continue
        if index == 0:
            sort_stage["$sort"][preceding_field + ".YearGrouped"] = 1
            continue
        if index == 1:
            sort_stage["$sort"][preceding_field + ".MonthGrouped"] = 1
            continue
        if index >= 2:
            sort_stage["$sort"][preceding_field + ".GroupedMillisecond"] = 1
            break

    # Create the group stage.
    group_stage = {"$group": {"_id": {}}}

    for index, element in enumerate(aggregation_period):
        if element <= 0:
            continue
        if index == 0:

createDateTimeAggregationField(group_stage["$group"][preceding_field],
element, "YearGrouped", "year", "Timestamp")
            continue
        if index == 1:
            number_to_subtract = 0
            if element > 1:
                number_to_subtract = element

createDateTimeAggregationField(group_stage["$group"][preceding_field],
element, "MonthGrouped", "month", "Timestamp",
                                           number_to_subtract)
            continue
        if index >= 2:
```

```
            period_length = computeMillisecondPeriod(aggregation_period)

createMillisecondAggregationField(group_stage["$group"][preceding_field],
period_length, "GroupedMillisecond", "Timestamp",
                                                  begin_date)
            break

    group_stage["$group"]["AverageValue"] = {"$" + calculation_command:
"$Value"}

    pipeline = [first_match_stage, first_time_match, second_time_match,
group_stage, sort_stage]
    aggregated_values = list(collection_to_use.aggregate(pipeline))
    return_documents = list()

    for element in aggregated_values:
        return_documents.append(
            cValueDocument(determineDateTimeFromRawDocument(begin_date,
aggregation_period, preceding_field, element), tag_id,
                           element["AverageValue"]))

    return return_documents


def createDateTimeAggregationField(input_dict: dict, period_length: int,
new_field_name: str, date_command: str,
                                   date_field_in_db: str,
number_to_subtract:int=0):
    if period_length == 1:
        input_dict[new_field_name] = {"$" + date_command: "$" +
date_field_in_db}
        return input_dict

    input_dict[new_field_name] = {
        "$let": {
            "vars": {
                "newDateField": {
                    "$subtract": [{"$" + date_command: "$" +
date_field_in_db}, number_to_subtract]
                }
            },
            "in": {
                "$subtract": ["$$newDateField", {"$mod": ["$$newDateField",
period_length]}]
            }
        }
    }
    return input_dict


def createMillisecondAggregationField(input_dict: dict, period_length: int,
new_field_name: str, date_field_in_db: str,
                                      date_to_subtract: datetime.datetime):
    input_dict[new_field_name] = {
        "$let": {
            "vars": {
                "newMillisecondField": {
                    "$subtract": ["$" + date_field_in_db, date_to_subtract]
                }
            },
            "in": {
                "$subtract": ["$$newMillisecondField", {"$mod":
["$$newMillisecondField", period_length]}]
```

```
                }
            }
        }
        return input_dict


def computeMillisecondPeriod(aggregation_period: list):
    return_millisecond = 0

    for index, element in enumerate(aggregation_period[2:]):
        if element <= 1:
            continue
        if index == 0:
            return_millisecond += element * 24 * 3600 * 1000
            continue
        if index == 1:
            return_millisecond += element * 3600 * 1000
            continue
        if index == 2:
            return_millisecond += element * 60 * 1000
            continue
        if index == 3:
            return_millisecond += element * 1000
            continue
        return_millisecond += element

    return return_millisecond


def determineDateTimeFromRawDocument(begin_date: datetime.datetime,
aggregation_period: list, preceding_db_field: str,
                                     input_document_dict: dict):
    if next((i for i in aggregation_period[2:] if i > 0), -1) != -1:
        return begin_date + datetime.timedelta(

milliseconds=input_document_dict[preceding_db_field]["GroupedMillisecond"])

    year = 0
    month = 0

    if aggregation_period[0] > 1:
        year += begin_date.year

    if aggregation_period[0] > 0:
        year += input_document_dict[preceding_db_field]["YearGrouped"]
    else:
        year = begin_date.year

    if aggregation_period[1] > 1:
        month += begin_date.month

    if aggregation_period[1] > 0:
        month += input_document_dict[preceding_db_field]["MonthGrouped"]
    else:
        month = begin_date.month

    return datetime.datetime(year, month, 0, 0, 0, 0)


class cValueDocument:
    """Stores the data of a document in the Tag Collection."""
```

```
    def __init__(self, Timestamp=None, TagID=None, Value=None,
FileReferenceID=None, _id=None):
        self.Timestamp = Timestamp
        self.TagID = TagID
        self.Value = Value
        self.FileReferenceID = FileReferenceID
        self._id = _id

    def __deepcopy__(self, memo):
        return cValueDocument(self.Timestamp, self.TagID, self.Value,
self.FileReferenceID, self._id)

    def toDict(self):
        return_dict = {}
        return_dict["Timestamp"] = self.Timestamp
        return_dict["TagID"] = self.TagID
        return_dict["Value"] = self.Value
        return_dict["FileReferenceID"] = self.FileReferenceID

        if not self._id is None:
            return_dict["_id"] = self._id

        return return_dict

    def fromDict(self, dictionary_to_convert_from):
        if "Timestamp" in dictionary_to_convert_from:
            self.Timestamp = dictionary_to_convert_from["Timestamp"]

        if "TagID" in dictionary_to_convert_from:
            self.TagID = dictionary_to_convert_from["TagID"]

        if "Value" in dictionary_to_convert_from:
            self.Value = dictionary_to_convert_from["Value"]

        if "FileReferenceID" in dictionary_to_convert_from:
            self.FileReferenceID =
dictionary_to_convert_from["FileReferenceID"]

        if "_id" in dictionary_to_convert_from:
            self._id = dictionary_to_convert_from["_id"]
        else:
            self._id = None

        return self

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return (self.Timestamp == other.Timestamp and
                    self.TagID == other.TagID and
                    self.Value == other.Value and
                    self.FileReferenceID == other.FileReferenceID)
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, self.__class__):
            return not self.__eq__(other)
        return NotImplemented

    def __hash__(self):
        return hash(tuple(sorted(self.__dict__.items())))


class cTagDocument:
```

```
    """Stores the data of a document in the Tag Collection."""

    def __init__(self, TagNames=None, SiteTagNames=None, IntervalMask=None,
Unit=None, Source=None, Description=None,
                 MeasurementType=None, _id=None):
        self.TagNames = TagNames
        self.SiteTagNames = SiteTagNames
        self.IntervalMask = IntervalMask
        self.Unit = Unit
        self.Source = Source
        self.Description = Description
        self.MeasurementType = MeasurementType
        self._id = _id

    def __deepcopy__(self, memo):
        return cTagDocument(copy.deepcopy(self.TagNames),
copy.deepcopy(self.SiteTagNames),
                            copy.deepcopy(self.IntervalMask), self.Unit,
self.Source, self.Description,
                            self.MeasurementType, self._id)

    def toDict(self):
        return_dict = {}
        return_dict["TagNames"] = []

        for element in self.TagNames:
            return_dict["TagNames"].append(element.toDict())

        return_dict["SiteTagNames"] = []

        for element in self.SiteTagNames:
            return_dict["SiteTagNames"].append(element.toDict())

        return_dict["IntervalMask"] = self.IntervalMask.toDict()
        return_dict["Unit"] = self.Unit
        return_dict["Source"] = self.Source
        return_dict["Description"] = self.Description
        return_dict["MeasurementType"] = self.MeasurementType

        if not self._id is None:
            return_dict["_id"] = self._id

        return return_dict

    def fromDict(self, dictionary_to_convert_from):
        if "TagNames" in dictionary_to_convert_from:
            self.TagNames = list()
            for element in dictionary_to_convert_from["TagNames"]:
                self.TagNames.append(cTagName().fromDict(element))

        if "SiteTagNames" in dictionary_to_convert_from:
            self.SiteTagNames = list()
            for element in dictionary_to_convert_from["SiteTagNames"]:
                self.SiteTagNames.append(cTagName().fromDict(element))

        if "IntervalMask" in dictionary_to_convert_from:
            self.IntervalMask =
cIntervalMask().fromDict(dictionary_to_convert_from["IntervalMask"])

        if "Unit" in dictionary_to_convert_from:
            self.Unit = dictionary_to_convert_from["Unit"]

        if "Source" in dictionary_to_convert_from:
```

```
            self.Source = dictionary_to_convert_from["Source"]

        if "Description" in dictionary_to_convert_from:
            self.Description = dictionary_to_convert_from["Description"]

        if "MeasurementType" in dictionary_to_convert_from:
            self.MeasurementType =
dictionary_to_convert_from["MeasurementType"]

        if "_id" in dictionary_to_convert_from:
            self._id = dictionary_to_convert_from["_id"]
        else:
            self._id = None

        return self

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return (np.array_equal(self.TagNames, other.TagNames) and
                    np.array_equal(self.SiteTagNames, other.SiteTagNames) and
                    self.IntervalMask == other.IntervalMask and
                    self.Unit == other.Unit and
                    self.Source == other.Source and
                    self.Description == other.Description and
                    self.MeasurementType == other.MeasurementType)
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, self.__class__):
            return not self.__eq__(other)
        return NotImplemented

    def __hash__(self):
        return hash(tuple(sorted(self.__dict__.items())))


class cTagName:
    """Stores the values associated with a TagDocument's TagNames and
SiteTagNames members."""

    def __init__(self, TagName="", AssignedTime=datetime.datetime.utcnow()):
        self.TagName = TagName
        self.AssignedTime = AssignedTime

    def __deepcopy__(self, memo):
        return cTagName(self.TagName, self.AssignedTime)

    def toDict(self):
        return {"TagName": self.TagName, "AssignedTime": self.AssignedTime}

    def fromDict(self, dictionary_to_convert_from):
        self.TagName = dictionary_to_convert_from.get("TagName", "")
        self.AssignedTime = dictionary_to_convert_from.get("AssignedTime",
datetime.datetime.utcnow())
        return self

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.__dict__ == other.__dict__
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, self.__class__):
```

```
            return not self.__eq__(other)
        return NotImplemented

    def __hash__(self):
        return hash(tuple(sorted(self.__dict__.items())))


class cIntervalMask:
    """Stores a DateTime in its constituent parts."""

    def __init__(self, Year=0, Month=0, Day=0, Hour=0, Minute=0, Second=0,
Millisecond=0):
        self.Year = Year
        self.Month = Month
        self.Day = Day
        self.Hour = Hour
        self.Minute = Minute
        self.Second = Second
        self.Millisecond = Millisecond

    def __deepcopy__(self, memo):
        return cIntervalMask(self.Year, self.Month, self.Day, self.Hour,
self.Minute, self.Second, self.Millisecond)

    def toDict(self):
        return {"Year": self.Year, "Month": self.Month, "Day": self.Day,
"Hour": self.Hour, "Minute": self.Minute,
                "Second": self.Second, "MilliSecond": self.Millisecond}

    def fromDict(self, dictionary_to_convert_from):
        self.Year = dictionary_to_convert_from.get("Year", 0)
        self.Month = dictionary_to_convert_from.get("Month", 0)
        self.Day = dictionary_to_convert_from.get("Day", 0)
        self.Hour = dictionary_to_convert_from.get("Hour", 0)
        self.Minute = dictionary_to_convert_from.get("Minute", 0)
        self.Second = dictionary_to_convert_from.get("Second", 0)
        self.Millisecond = dictionary_to_convert_from.get("Millisecond", 0)
        return self

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.__dict__ == other.__dict__
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, self.__class__):
            return not self.__eq__(other)
        return NotImplemented

    def __hash__(self):
        return hash(tuple(sorted(self.__dict__.items())))


if __name__ == "__main__":
    executeMongoDBQueries()
```

APPENDIX 3 APACHE SPARK FULL VERIFICATION LOGS

The log output that is given in this section relates to the verification of Apache Spark, as discussed in Section 4.3.3. The full log obtained during the execution of the PythonPi program is given below.

```
docker exec aa9b9ede210f bash -c "spark-submit --verbose --master
spark://192.168.0.27:27020 --executor-memory 700M --total-executor-cores 2 --
num-executors 2 /usr/local/spark/examples/src/main/python/pi.py 10"
Using properties file: /usr/local/spark/conf/spark-defaults.conf
Adding default property:
spark.driver.extraLibraryPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Adding default property: spark.jars=/usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
Adding default property: spark.master=spark://spark-master-
spark.marathon.slave.mesos:27020
Adding default property:
spark.executor.extraLibraryPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Adding default property:
spark.driver.extraClassPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Parsed arguments:
  master                  spark://192.168.0.27:27020
  deployMode              null
  executorMemory          700M
  executorCores           null
  totalExecutorCores      2
  propertiesFile          /usr/local/spark/conf/spark-defaults.conf
  driverMemory            null
  driverCores             null
  driverExtraClassPath    /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  driverExtraLibraryPath  /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  driverExtraJavaOptions  null
  supervise               false
  queue                   null
  numExecutors            2
  files                   null
  pyFiles                 null
  archives                null
  mainClass               null
  primaryResource
file:/usr/local/spark/examples/src/main/python/pi.py
  name                    pi.py
  childArgs               [10]
  jars                    /usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
  packages                null
  packagesExclusions      null
  repositories            null
  verbose                 true

Spark properties used, including those specified through
 --conf and those from the properties file /usr/local/spark/conf/spark-
defaults.conf:
  spark.executor.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
```

```
  spark.driver.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  spark.jars -> /usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
  spark.master -> spark://spark-master-spark.marathon.slave.mesos:27020
  spark.driver.extraClassPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar


Main class:
org.apache.spark.deploy.PythonRunner
Arguments:
file:/usr/local/spark/examples/src/main/python/pi.py
null
10
System properties:
spark.executor.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.executor.memory -> 700M
spark.driver.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.cores.max -> 2
SPARK_SUBMIT -> true
spark.files -> file:/usr/local/spark/examples/src/main/python/pi.py
spark.app.name -> pi.py
spark.jars -> file:/usr/local/spark/jars/mongo-spark-
connector.jar,file:/usr/local/spark/jars/mongo-java-driver.jar
spark.submit.deployMode -> client
spark.master -> spark://192.168.0.27:27020
spark.driver.extraClassPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Classpath elements:
/usr/local/spark/jars/mongo-spark-connector.jar
/usr/local/spark/jars/mongo-java-driver.jar


Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
17/08/23 06:06:07 INFO SparkContext: Running Spark version 2.1.1
17/08/23 06:06:08 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
17/08/23 06:06:08 INFO SecurityManager: Changing view acls to: root
17/08/23 06:06:08 INFO SecurityManager: Changing modify acls to: root
17/08/23 06:06:08 INFO SecurityManager: Changing view acls groups to:
17/08/23 06:06:08 INFO SecurityManager: Changing modify acls groups to:
17/08/23 06:06:08 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users  with view permissions: Set(root); groups
with view permissions: Set(); users  with modify permissions: Set(root);
groups with modify permissions: Set()
17/08/23 06:06:09 INFO Utils: Successfully started service 'sparkDriver' on
port 36341.
17/08/23 06:06:09 INFO SparkEnv: Registering MapOutputTracker
17/08/23 06:06:09 INFO SparkEnv: Registering BlockManagerMaster
17/08/23 06:06:09 INFO BlockManagerMasterEndpoint: Using
org.apache.spark.storage.DefaultTopologyMapper for getting topology
information
17/08/23 06:06:09 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint
up
17/08/23 06:06:09 INFO DiskBlockManager: Created local directory at
/mnt/mesos/sandbox/spark-storage/local/blockmgr-0d116d28-115e-4b4a-8076-
f08ab1f65004
17/08/23 06:06:09 INFO MemoryStore: MemoryStore started with capacity 366.3
MB
```

```
17/08/23 06:06:09 INFO SparkEnv: Registering OutputCommitCoordinator
17/08/23 06:06:10 INFO Utils: Successfully started service 'SparkUI' on port
4040.
17/08/23 06:06:10 INFO SparkUI: Bound SparkUI to 192.168.0.27, and started at
http://192.168.0.27:4040
17/08/23 06:06:10 INFO SparkContext: Added JAR
file:/usr/local/spark/jars/mongo-spark-connector.jar at
spark://192.168.0.27:36341/jars/mongo-spark-connector.jar with timestamp
1503468370225
17/08/23 06:06:10 INFO SparkContext: Added JAR
file:/usr/local/spark/jars/mongo-java-driver.jar at
spark://192.168.0.27:36341/jars/mongo-java-driver.jar with timestamp
1503468370226
17/08/23 06:06:10 INFO SparkContext: Added file
file:/usr/local/spark/examples/src/main/python/pi.py at
spark://192.168.0.27:36341/files/pi.py with timestamp 1503468370729
17/08/23 06:06:10 INFO Utils: Copying
/usr/local/spark/examples/src/main/python/pi.py to /mnt/mesos/sandbox/spark-
storage/local/spark-b0296a7d-84fd-4415-9908-0fecb71a419b/userFiles-0d69f730-
0421-40bc-81df-ebceff5b49f3/pi.py
17/08/23 06:06:11 INFO StandaloneAppClient$ClientEndpoint: Connecting to
master spark://192.168.0.27:27020...
17/08/23 06:06:11 INFO TransportClientFactory: Successfully created
connection to /192.168.0.27:27020 after 50 ms (0 ms spent in bootstraps)
17/08/23 06:06:26 INFO StandaloneSchedulerBackend: Connected to Spark cluster
with app ID app-20170823060625-0000
17/08/23 06:06:26 INFO Utils: Successfully started service
'org.apache.spark.network.netty.NettyBlockTransferService' on port 36838.
17/08/23 06:06:26 INFO NettyBlockTransferService: Server created on
192.168.0.27:36838
17/08/23 06:06:27 INFO BlockManager: Using
org.apache.spark.storage.RandomBlockReplicationPolicy for block replication
policy
17/08/23 06:06:27 INFO BlockManagerMaster: Registering BlockManager
BlockManagerId(driver, 192.168.0.27, 36838, None)
17/08/23 06:06:27 INFO BlockManagerMasterEndpoint: Registering block manager
192.168.0.27:36838 with 366.3 MB RAM, BlockManagerId(driver, 192.168.0.27,
36838, None)
17/08/23 06:06:27 INFO BlockManagerMaster: Registered BlockManager
BlockManagerId(driver, 192.168.0.27, 36838, None)
17/08/23 06:06:27 INFO BlockManager: Initialized BlockManager:
BlockManagerId(driver, 192.168.0.27, 36838, None)
17/08/23 06:06:27 INFO StandaloneSchedulerBackend: SchedulerBackend is ready
for scheduling beginning after reached minRegisteredResourcesRatio: 0.0
17/08/23 06:06:27 INFO SharedState: Warehouse path is
'file:/usr/local/spark/spark-warehouse'.
17/08/23 06:06:29 INFO SparkContext: Starting job: reduce at
/usr/local/spark/examples/src/main/python/pi.py:43
17/08/23 06:06:30 INFO DAGScheduler: Got job 0 (reduce at
/usr/local/spark/examples/src/main/python/pi.py:43) with 10 output partitions
17/08/23 06:06:30 INFO DAGScheduler: Final stage: ResultStage 0 (reduce at
/usr/local/spark/examples/src/main/python/pi.py:43)
17/08/23 06:06:30 INFO DAGScheduler: Parents of final stage: List()
17/08/23 06:06:30 INFO DAGScheduler: Missing parents: List()
17/08/23 06:06:30 INFO DAGScheduler: Submitting ResultStage 0 (PythonRDD[1]
at reduce at /usr/local/spark/examples/src/main/python/pi.py:43), which has
no missing parents
17/08/23 06:06:31 INFO MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 4.9 KB, free 366.3 MB)
17/08/23 06:06:32 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes
in memory (estimated size 3.4 KB, free 366.3 MB)
17/08/23 06:06:32 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on 192.168.0.27:36838 (size: 3.4 KB, free: 366.3 MB)
```

```
17/08/23 06:06:32 INFO SparkContext: Created broadcast 0 from broadcast at
DAGScheduler.scala:996
17/08/23 06:06:32 INFO DAGScheduler: Submitting 10 missing tasks from
ResultStage 0 (PythonRDD[1] at reduce at
/usr/local/spark/examples/src/main/python/pi.py:43)
17/08/23 06:06:32 INFO TaskSchedulerImpl: Adding task set 0.0 with 10 tasks
17/08/23 06:06:33 INFO StandaloneAppClient$ClientEndpoint: Executor added:
app-20170823060625-0000/0 on worker-20170823054643-192.168.0.21-41060
(192.168.0.21:41060) with 1 cores
17/08/23 06:06:33 INFO StandaloneSchedulerBackend: Granted executor ID app-
20170823060625-0000/0 on hostPort 192.168.0.21:41060 with 1 cores, 700.0 MB
RAM
17/08/23 06:06:33 INFO StandaloneAppClient$ClientEndpoint: Executor added:
app-20170823060625-0000/1 on worker-20170823054642-192.168.0.27-38153
(192.168.0.27:38153) with 1 cores
17/08/23 06:06:33 INFO StandaloneSchedulerBackend: Granted executor ID app-
20170823060625-0000/1 on hostPort 192.168.0.27:38153 with 1 cores, 700.0 MB
RAM
17/08/23 06:06:35 INFO CoarseGrainedSchedulerBackend$DriverEndpoint:
Registered executor NettyRpcEndpointRef(null) (192.168.0.21:41286) with ID 0
17/08/23 06:06:36 INFO StandaloneAppClient$ClientEndpoint: Executor updated:
app-20170823060625-0000/1 is now RUNNING
17/08/23 06:06:36 INFO StandaloneAppClient$ClientEndpoint: Executor updated:
app-20170823060625-0000/0 is now RUNNING
17/08/23 06:06:36 INFO BlockManagerMasterEndpoint: Registering block manager
192.168.0.21:35283 with 193.5 MB RAM, BlockManagerId(0, 192.168.0.21, 35283,
None)
17/08/23 06:06:36 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
192.168.0.21, executor 0, partition 0, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:38 INFO CoarseGrainedSchedulerBackend$DriverEndpoint:
Registered executor NettyRpcEndpointRef(null) (192.168.0.27:58966) with ID 1
17/08/23 06:06:38 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
192.168.0.27, executor 1, partition 1, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:38 INFO BlockManagerMasterEndpoint: Registering block manager
192.168.0.27:34504 with 193.5 MB RAM, BlockManagerId(1, 192.168.0.27, 34504,
None)
17/08/23 06:06:39 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on 192.168.0.21:35283 (size: 3.4 KB, free: 193.5 MB)
17/08/23 06:06:39 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on 192.168.0.27:34504 (size: 3.4 KB, free: 193.5 MB)
17/08/23 06:06:39 INFO TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2,
192.168.0.21, executor 0, partition 2, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 3.0 in stage 0.0 (TID 3,
192.168.0.21, executor 0, partition 3, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 4018 ms on 192.168.0.21 (executor 0) (1/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 4.0 in stage 0.0 (TID 4,
192.168.0.21, executor 0, partition 4, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2)
in 349 ms on 192.168.0.21 (executor 0) (2/10)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3)
in 247 ms on 192.168.0.21 (executor 0) (3/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 5.0 in stage 0.0 (TID 5,
192.168.0.21, executor 0, partition 5, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 4.0 in stage 0.0 (TID 4)
in 81 ms on 192.168.0.21 (executor 0) (4/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 6.0 in stage 0.0 (TID 6,
192.168.0.21, executor 0, partition 6, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 5.0 in stage 0.0 (TID 5)
in 93 ms on 192.168.0.21 (executor 0) (5/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 7.0 in stage 0.0 (TID 7,
192.168.0.21, executor 0, partition 7, PROCESS_LOCAL, 6202 bytes)
```

```
17/08/23 06:06:40 INFO TaskSetManager: Finished task 6.0 in stage 0.0 (TID 6)
in 148 ms on 192.168.0.21 (executor 0) (6/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 8.0 in stage 0.0 (TID 8,
192.168.0.27, executor 1, partition 8, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 2285 ms on 192.168.0.27 (executor 1) (7/10)
17/08/23 06:06:40 INFO TaskSetManager: Starting task 9.0 in stage 0.0 (TID 9,
192.168.0.21, executor 0, partition 9, PROCESS_LOCAL, 6202 bytes)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7)
in 106 ms on 192.168.0.21 (executor 0) (8/10)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8)
in 116 ms on 192.168.0.27 (executor 1) (9/10)
17/08/23 06:06:40 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9)
in 197 ms on 192.168.0.21 (executor 0) (10/10)
17/08/23 06:06:40 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
17/08/23 06:06:41 INFO DAGScheduler: ResultStage 0 (reduce at
/usr/local/spark/examples/src/main/python/pi.py:43) finished in 8.580 s
17/08/23 06:06:41 INFO DAGScheduler: Job 0 finished: reduce at
/usr/local/spark/examples/src/main/python/pi.py:43, took 11.616557 s
Pi is roughly 3.140856
17/08/23 06:06:43 INFO SparkUI: Stopped Spark web UI at
http://192.168.0.27:4040
17/08/23 06:06:43 INFO StandaloneSchedulerBackend: Shutting down all
executors
17/08/23 06:06:43 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Asking
each executor to shut down
17/08/23 06:06:43 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
17/08/23 06:06:44 INFO MemoryStore: MemoryStore cleared
17/08/23 06:06:44 INFO BlockManager: BlockManager stopped
17/08/23 06:06:44 INFO BlockManagerMaster: BlockManagerMaster stopped
17/08/23 06:06:44 INFO
OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
17/08/23 06:06:44 INFO SparkContext: Successfully stopped SparkContext
17/08/23 06:06:54 INFO ShutdownHookManager: Shutdown hook called
17/08/23 06:06:54 INFO ShutdownHookManager: Deleting directory
/mnt/mesos/sandbox/spark-storage/local/spark-b0296a7d-84fd-4415-9908-
0fecb71a419b/pyspark-982ac810-b8bb-4b1f-89d4-c397de4662ea
17/08/23 06:06:54 INFO ShutdownHookManager: Deleting directory
/mnt/mesos/sandbox/spark-storage/local/spark-b0296a7d-84fd-4415-9908-
0fecb71a419b
```

The full log obtained during the execution of the MongoDB Python verification program is given below.

```
docker exec aa9b9ede210f bash -c "spark-submit --verbose --master
spark://192.168.0.27:27020 --executor-memory 700M --total-executor-cores 2 --
num-executors 1 /mnt/mesos/sandbox/spark-storage/mongo_test_Spark.py"
Using properties file: /usr/local/spark/conf/spark-defaults.conf
Adding default property:
spark.driver.extraLibraryPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Adding default property: spark.jars=/usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
Adding default property: spark.master=spark://spark-master-
spark.marathon.slave.mesos:27020
Adding default property:
spark.executor.extraLibraryPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
```

```
Adding default property:
spark.driver.extraClassPath=/usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Parsed arguments:
  master                  spark://192.168.0.27:27020
  deployMode              null
  executorMemory          700M
  executorCores           null
  totalExecutorCores      2
  propertiesFile          /usr/local/spark/conf/spark-defaults.conf
  driverMemory            null
  driverCores             null
  driverExtraClassPath    /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  driverExtraLibraryPath  /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  driverExtraJavaOptions  null
  supervise               false
  queue                   null
  numExecutors            1
  files                   null
  pyFiles                 null
  archives                null
  mainClass               null
  primaryResource         file:/mnt/mesos/sandbox/spark-
storage/mongo_test_Spark.py
  name                    mongo_test_Spark.py
  childArgs               []
  jars                    /usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
  packages                null
  packagesExclusions      null
  repositories            null
  verbose                 true

Spark properties used, including those specified through
 --conf and those from the properties file /usr/local/spark/conf/spark-
defaults.conf:
  spark.executor.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  spark.driver.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
  spark.jars -> /usr/local/spark/jars/mongo-spark-
connector.jar,/usr/local/spark/jars/mongo-java-driver.jar
  spark.master -> spark://spark-master-spark.marathon.slave.mesos:27020
  spark.driver.extraClassPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar


Main class:
org.apache.spark.deploy.PythonRunner
Arguments:
file:/mnt/mesos/sandbox/spark-storage/mongo_test_Spark.py
null
System properties:
spark.executor.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.executor.memory -> 700M
spark.driver.extraLibraryPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
spark.cores.max -> 2
SPARK_SUBMIT -> true
spark.files -> file:/mnt/mesos/sandbox/spark-storage/mongo_test_Spark.py
```

```
spark.app.name -> mongo_test_Spark.py
spark.jars -> file:/usr/local/spark/jars/mongo-spark-
connector.jar,file:/usr/local/spark/jars/mongo-java-driver.jar
spark.submit.deployMode -> client
spark.master -> spark://192.168.0.27:27020
spark.driver.extraClassPath -> /usr/local/spark/jars/mongo-spark-
connector.jar:/usr/local/spark/jars/mongo-java-driver.jar
Classpath elements:
/usr/local/spark/jars/mongo-spark-connector.jar
/usr/local/spark/jars/mongo-java-driver.jar


Starting tests...
Finished tests.
```

APPENDIX 4 JOURNAL ARTICLE OF STUDY

An article was submitted for publication in the Special Volume of the Journal of Cleaner Production: Sustainable Consumption and Big Data. The article was submitted on 28 September 2017 and starts on the next page to ensure its original layout. At the time of writing, the article was accepted by the editor of the journal and entered the peer review stage. During the peer review stage, at least one expert reviewer will assess the article.

Word count: 8954

Using Big Data for Insights into Sustainable Energy Consumption in the Industrial and Mining Sectors

Jacobus Herman [a, *], Hendrik Herman [a], Marc John Mathews [a], Jan Corné Vosloo [a]

[a] Centre for Research and Continued Engineering Development, North-West University, P.O. Box 11207, Silver Lakes, 0054, South Africa

Abstract

Human activities related to the production and consumption of energy are negatively impacting the environment and need to be addressed. This can be done by focusing on cleaner production. This article investigates sustainable energy consumption in the industrial and mining sectors. To this end, big data was applied after finding that its adoption was still limited. Hence, a big data system was designed which makes novel use of microservices and containers for modularity and extensibility. A hybrid architecture, consisting of characteristics from polyglot persistence, data lakes and the Lambda architecture, was used. MongoDB served as the main data store. The system was implemented in an engineering services company, which yielded seven case studies. These case studies show the impact big data has on the various levels of management within an organisation, delivering valuable insights to personnel on different levels of an organisational hierarchy. By creating these insights for the appropriate personnel, improved decisions can be made and actions taken to achieve sustainable energy consumption reductions. The total maximum impact resulting from the case studies was a saving of 75 622 MWh, a reduction of 74 111 tonne carbon dioxide, and a saving of 108 141 m^3 water. Continued reductions in energy consumption lead to cleaner production and eventually aids in addressing the various energy challenges that exist in our world.

Keywords: big data, sustainable energy consumption, microservices, containers, organisational hierarchy insights, hybrid architecture

Energy is an interconnected commodity that is essential for our modern world to thrive and continuously advance. The importance of energy is best understood through the water-energy-food nexus (Bazilian et al., 2011; Bizikova et al., 2013) and the more complex water-energy-land-food nexus (Ringler et al., 2013). The aforementioned networks emphasise that the use and misuse of energy cause ripples that can adversely affect our entire world, such as climate change (Hansen et al., 2013).

The energy sector's unsustainable dependence on fossil fuels is responsible for at least two-thirds of global greenhouse gas emissions (International Energy Agency, 2016a). More specifically, the

* Corresponding author
Email address: j.herman.za@ieee.org

industrial and mining sectors are two of the largest energy consumers. Due to these high levels of energy consumption, they have a large potential for energy efficiency (International Energy Agency, 2016a; Rahman et al., 2016). Over time, various commitments, such as the Paris Agreement (United Nations, 2016), have been made to reduce greenhouse gas emissions and energy consumption. However, the underlying mechanisms of reducing emissions can only come from sustainable energy utilisation by the consumer (International Energy Agency, 2016b).

The need to lower energy consumption or at least improve its sustainability in industry is evident. While there are many possibilities for contributing to sustainable energy consumption (Abdelaziz et al., 2011; Chu et al., 2016), only some have applied big data to this end (Song et al., 2017; Zhou et al., 2016). Background on big data will not be discussed as it has already been done by others (Alharthi et al., 2017; Feinleib, 2014; Gandomi and Haider, 2015; Mohanty et al., 2013). What is important, however, is that big data adoption in the industry is still in its infancy.

Research shows that companies are experimenting with big data projects. In a Knowledgent white paper, it was found that 25 of 100 respondents had implemented a big data solution (Knowledgent, 2015). Furthermore, a report by the International Institute for Analytics (International Institute for Analytics, 2016) showed that 29% of 194 companies in the United States and internationally had operational big data systems. However, of the 194 companies, 31% were still in the process of implementing a big data system. Finally, Gartner shows that big data adoption is increasing with 48% of 199 companies investigated having invested in big data during 2016 (van der Meulen, 2016). However, only 15% of the companies deployed their big data project to production. Thus, there is a need for incorporating sustainable energy consumption into big data solutions.

In the energy production sector, big data has been applied to smart grids to reduce electricity consumption using Dynamic Demand Response (Simmhan et al., 2013). Simmhan et al. used a cloud computing platform to perform big data analytics on renewable energy, and Dynamic Demand Response to reduce electricity consumption and decrease supply fluctuations. The reduction in energy consumption was only performed during high demand cycles and was not sustained. Thus, the approach of Simmhan et al. makes progress toward sustainable energy consumption. However, as suggested by the International Energy Agency (International Energy Agency, 2016a), it is critical to focus on energy efficiency in the short term.

A study by Dessureault found that the mining sector has large repositories of data which are mostly unexploited (Dessureault, 2016). He noted that most data are relational, followed by analogue and control data, which are referred to as process data. Furthermore, he found that the minority of data is unstructured, although this is changing with the adoption of next-generation equipment and Internet of

Things (IoT) systems. Dessureault concluded that existing mining data holds immense potential. This data could be exploited to make sustainable energy consumption decisions.

The findings from a study by Erkayaoğlu identified the same data types in the mining industry as Dessureault (Erkayaoğlu, 2017). However, Erkayaoğlu went into greater detail regarding data warehousing and the applications of big data in the mining sector. He identified the shift needed in management perspective and mining culture to make use of data for improved decision-making. However, no practical implementation was given.

IoT is an area of research which could aid in the modernisation of the industrial sector (Laskier, 2016). However, Laskier found that the industrial and mining sectors have made little progress toward IoT adoption. Applying IoT technologies to the mining and industrial industries would create even more data. This would likely require the use of big data processing, storage, and analytics.

Perrons and McAuley gave a possible explanation for the hesitation of the mining and industrial sectors to make use of big data (Perrons and McAuley, 2015). They investigated the notion of collecting all data for use in predictive analytics. Frequently, in literature and media, an "N=all" approach to data gathering is proclaimed, where "N=all" refers to the collection of all data. Perrons and McAuley found that in mining it was more feasible to use an "n « all" approach for big data and predictive analytics. Here "n « all" refers to the collection of a lot less than the entire population of data.

The reason Perrons and McAuley proposed their approach was due to the high cost of data storage and capturing that results from mines' technical and geographical challenges. The authors did concede that as technology advances and costs decrease, the "n « all" approach would become less relevant. Their work does, however, show that big data systems are not easy to implement on mines, which could explain the low rate of big data adoption.

It is evident from the literature that there is a need for sustainable energy consumption, specifically in industry. The possible actions for achieving this goal are a progression toward clean and renewable energy sources (Jacobson et al., 2017) as well as an increase in energy efficiency (Lukman et al., 2016). The focus of this paper is on the use of big data systems for sustainable energy consumption in the industrial sector with the following contributions:

1. A practical application of big data in the mining and industrial sectors through the implementation of a big data system. The system is novel in its use of containers and microservices to provide modularity, and it addresses the need for energy big data systems.
2. Case studies relating to the use of big data which specifically focus on the consumption of energy and its impact on sustainability.

Word count: 8954

As stated in the introduction, there is a clear need for energy-focused big data systems. The system proposed in this study is unique as it was developed for the storage of energy-related data within both the industrial and mining sectors. Containers and microservices (as discussed later) further add to the novelty of the system.

The proposed system was designed to be deployable on cloud infrastructure or on-premises with commodity hardware. It is true that most cloud providers such as Amazon, Google, IBM, and Microsoft already offer services that encompass all the requirements for deploying a big data system. However, organisations with existing infrastructure, or the requirement for complete ownership, would need a private cluster-based system. To this end, the proposed system details the creation of a private cluster and the possibility of deploying to cloud services.

The system used a hybrid architecture which combined elements from three big data concepts or architectures, namely data lake (Clegg, 2015; Fang, 2015; O'Leary, 2014; Watson, 2015), polyglot persistence (Fowler, 2011; Sadalage and Fowler, 2012; Wiese, 2015) and the Lambda architecture (Marz and Warren, 2015). However, the primary focus was on the polyglot persistence concept.

Different layers characterised the system organisation, as shown in Figure 58. These layers were chosen to maintain consistency with the Lambda architecture and to achieve complexity isolation. As the name indicates, complexity isolation allowed the complexity of each layer to be isolated from that of other layers and thus yielded a more simplified system.

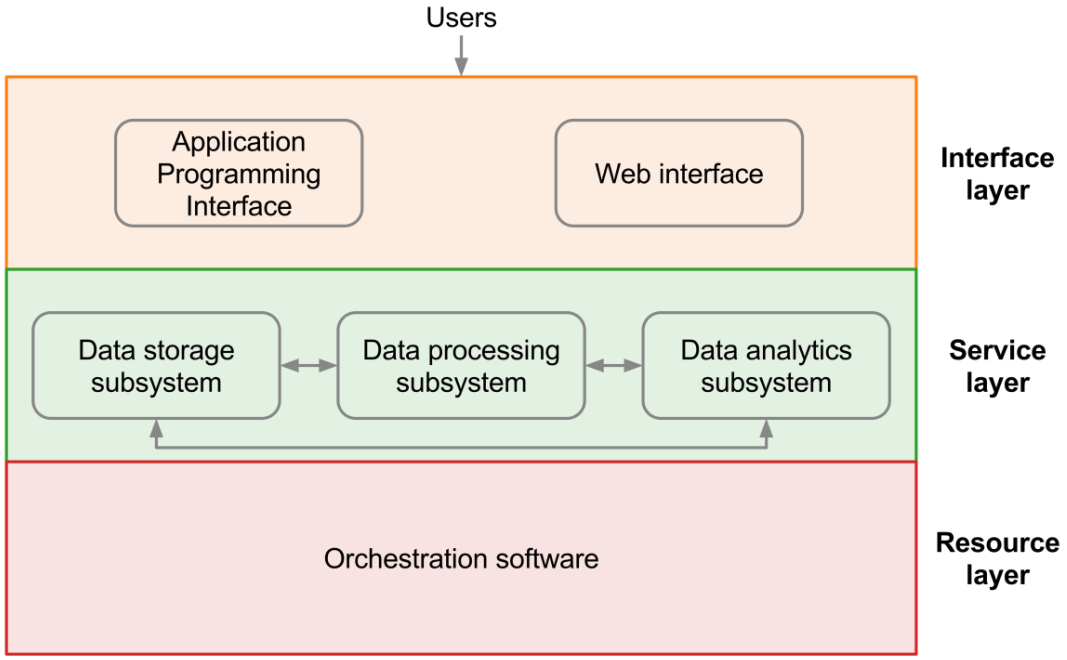


Figure 58: Top-level layers of the system architecture

Figure 58 shows the three layers of the system, namely the interface, service and resource layers. Users access the system through the interface layer, which contains an application programming interface (API) as well as a web interface. The API would be used by other systems and applications to obtain service from the system. The web interface is provided for human users, as most people are familiar with web browsers and websites.

The interface layer can only access lower-level functionality through the service layer. The service layer provides functionalities such as storage, processing and analytics. Figure 58 shows the aforementioned three functional subsystems contained within the service layer. The first is the data storage subsystem. This stores both the raw and processed data. Next is the data processing subsystem which provides for low-level data processing. High-level processing, such as data mining and predictions, is made available by the data analytics subsystem.

For the service layer to function, hardware and high-level computer system resources are required. These resources are supplied through the resource layer. The main component of the resource layer, as shown in Figure 58, is orchestration software. Orchestration software is used to create a cluster from multiple computers (Kratzke, 2014; Pahl, 2015). This software pools and coordinates the resources on the cluster and provides additional features such as monitoring, service discovery, and automatic recovery of failed software. Thus, orchestration software provides hardware abstraction and supplies resources such as CPUs, memory and storage, to the higher layers.

Each layer will be discussed briefly in subsections that follow.

2.1. Resource layer design

In a private cloud, the resource layer would be the most important layer as it is the base on which the entire system runs. Instabilities or errors in this layer would impact the upward layers. Therefore, the software for this layer needed to be efficient, stable and cost-effective, while also supporting scalability, high availability, security and extensibility. Here, extensibility takes the future growth of a system into account by supporting a variety of systems and software. Extensibility was important to prevent constraints on the choices of subsequent software.

The orchestration software identified for use were Mesosphere DC/OS (Mesosphere Inc., 2017), Kubernetes (The Kubernetes Authors, 2017) and Cloudify (GigaSpaces Technologies Inc., 2017). Each of the software projects had similar features, such as support for Docker images, horizontal scaling, monitoring and self-healing. Note that self-healing is a feature that automatically restarts applications or containers if they fail.

With the orchestration software projects having similar features, the final decision rested on long-term sustainability. Mesosphere DC/OS was deemed as having the best long-term support and sustainability due to its open-source nature and enterprise version. Further, Mesosphere DC/OS is based on various Apache Software Foundation projects which are prominent in the big data landscape.

Mesosphere constructs a cluster with the use of a master-agent configuration. The master node(s) manage the cluster and run most of the DC/OS software components. Agent nodes can either be public or private, which refers to the scope of their accessibility. The agent nodes perform work by executing the tasks which are deployed to the cluster. As alluded to earlier, multiple master nodes can be present in a cluster to ensure high availability. The proposed system consisted of three master and three private agent nodes.

It is important to note that an operating system can only support one Mesosphere installation, which could either be a master or an agent node. For small clusters, this situation would require superfluous single operating system hosts. This paper, therefore, proposes the use of computers with two virtual machines (VMs) per computer. One VM would be a master node and the other an agent node. Such a configuration is advantageous as, for small clusters, a single operating system host would have underutilised resources which could be better used elsewhere. Therefore, collocating master and agent nodes on the same computer reduces initial equipment costs. The disadvantage occurs when the cluster increases in size, necessitating more powerful master nodes. In this situation, the master and agent nodes would need to be recreated. Another disadvantage is that if one computer fails the cluster will lose both a master and an agent.

The resource layer comprised three identical computers with the operating system of the VMs and host being Ubuntu server LTS 16.04. Table 1 lists the resources of the master and agent node VMs. The Kernel-based Virtual Machine (KVM) software was used to create the aforementioned VMs. The computers were connected via an Ethernet network to facilitate communication and data transfer. The system diagram of the resource layer is shown in Figure 59.

Table 19: Resources for the two Mesosphere DC/OS node types

Resource type	Amount	
	Master node	**Agent node**
Processors	4	4
Memory	8 GB	6 GB
Hard disk drive	100 GB	1.8 TB

Word count: 8954

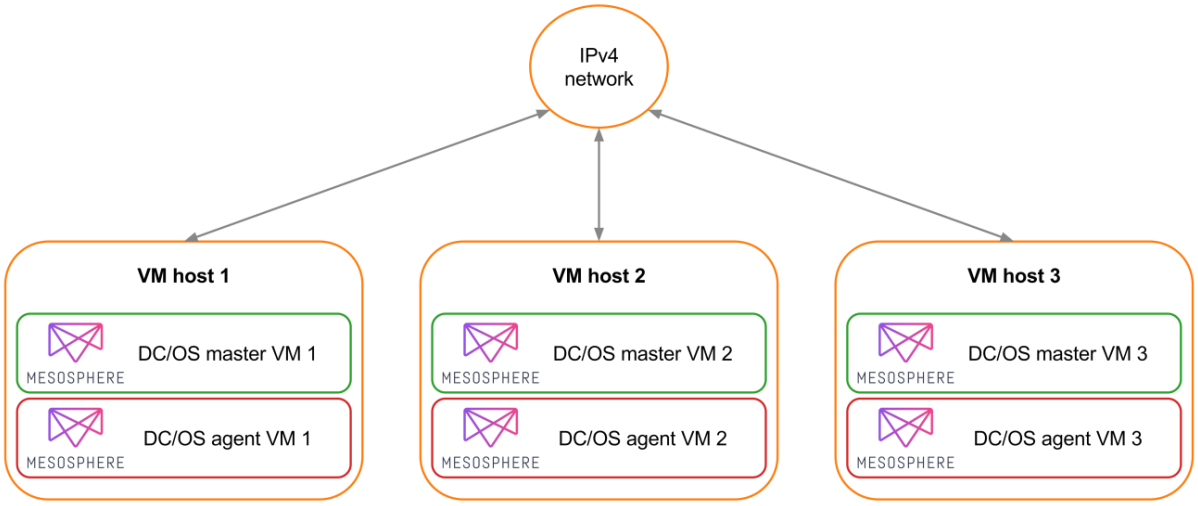


Figure 59: Resource layer implementation diagram

2.2. Service layer design

The service layer design was based on a microservices architecture (Linthicum, 2016; Sill, 2016; Singleton, 2016; Thones, 2015). While microservices are beyond the scope of this paper, it essentially entails the creation of small components which provide a specific service. Microservices provide benefits such as modularity, loose coupling between components and increased scalability. The service layer, therefore, consisted of various components which were logically grouped into subsystems. As previously shown in Figure 58, three subsystems comprised the service layer, namely the data storage, data processing and data analytics subsystems.

The microservices of the service layer were created with the use of (software) containers, which are thoroughly discussed in (Docker Inc., 2016; Linthicum, 2016; Pahl, 2015). Software containers are similar to shipping containers in that they provide isolation and portability. Microservices are ideal for containers, but can be implemented without containers as discussed in (Singleton, 2016) and (Sill, 2016). Containers were found to be the most popular, or even best, way to implement microservices (Linthicum, 2016; Pahl, 2015; Sill, 2016). The Docker software platform was used to create the images and containers of the system. The motivation for using Docker was due to its dominance of container software and its universal support among orchestration software.

As mentioned in the previous paragraph, containers provide portability. This portability allows the system to be deployed to any cloud provider that either supports Docker containers or permits the installation of the Docker software. In such a situation, the resource layer of the system would become the cloud provider, but the rest of the layers would remain the same. In another scenario, imagine improved orchestration software becomes available. In this situation, the orchestration software only needs to support Docker containers or allow installation of the Docker software to become a new resource layer for the system.

The subsections that follow briefly explain each of the subsystems in the service layer.

2.2.1. Data storage subsystem design

The data storage subsystem was the main component of the service layer as it stored all the data. MongoDB was selected as the most appropriate data store for the system and context. However, any number of data stores could have been implemented within this subsystem.

MongoDB is a document store which provides a dynamic schema and subsequently allows for multi-structured data storage. Its use of the Binary JavaScript Object Notation (BSON) (MongoDB Inc., 2017a), which is a binary form of the JavaScript Object Notation (JSON) (Ecma International, 2013), was instrumental in its choice. The reason being that JSON and BSON are supported in most programming languages, resulting in interoperable data interchange. Furthermore, using a document store prevented impedance mismatch, which means the structure in which data is stored does not differ from its structure in a programming language.

MongoDB was implemented in a production configuration (MongoDB Inc., 2017b) consisting of one mongos router, a sharded replica set and a config server replica set. The sharded replica set and config server replica set consisted of three MongoDB hosts each, where one host was the master and the rest slaves. For clarity, a replica set is the mechanism through which MongoDB provides for fault tolerance through master-slave replication. Sharding enables horizontal scalability by partitioning data.

Figure 60 shows the layout of the data storage subsystem. Data and operations are sent to the MongoDB router which communicates with the config server replica set regarding the location to which the data and operations must be sent. Once the MongoDB router determines the correct location, it sends the data and operations to the sharded replica set. The output from the sharded replica set is then sent to the MongoDB router which sends it to the client. Internally, the config server replica set communicates with the sharded replica set to store the state and organization of the data and cluster components.

Word count: 8954

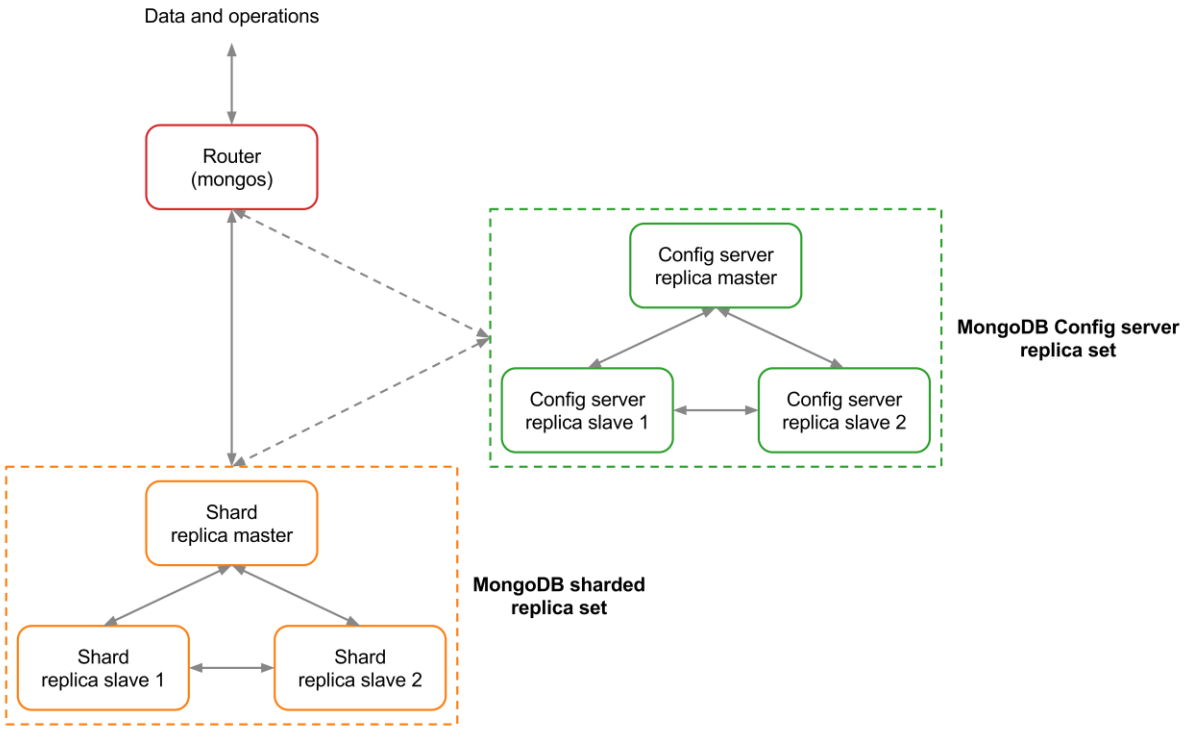


Figure 60: Implementation diagram of the data storage subsystem

Although this investigation made use of only one sharded replica set, given sufficient data, one would ideally deploy multiple sharded replica sets. Each sharded replica set would then contain a set of the total data. In such a situation, the utility of the config servers become apparent for, without the config servers, the client would have to know the location of which sharded replica set contains which data before the execution of queries. The config servers prevent this situation and allow the client to send queries regardless of the actual data's location.

The data storage subsystem is modular in that, as the need arises, more hosts can be added to the sharded and config server replica sets. More mongos applications can also be deployed to support increased query volumes and rates. Adding more hosts would, however, not increase the performance of the subsystem as only more sharded replica sets may increase performance. This increase in performance depends on the way in which sharding was configured.

MongoDB version 3.4.5 was used, and implementation was based on the official MongoDB Docker image. The official Docker image could have been used as is, but additional features were required for improved operation. Additionally, the MongoDB configuration included transport layer security (TLS) version 1.2 and user authentication. Using TLS 1.2 required X.509 certificates for each instance of MongoDB, which totals seven. TLS provided secure communication from within and outside the data storage subsystem. User authentication provided access control for when clients connected to the data storage subsystem.

2.2.2. Data processing subsystem

Following the data storage subsystem was the design of the data processing subsystem. As with the data storage subsystem, there were multiple software projects to use in this subsystem. The software projects considered were Apache Spark (Apache Software Foundation, 2017a), Apache Storm (Apache Software Foundation, 2017b), Apache Hadoop MapReduce (Apache Software Foundation, 2017c) and Apache Flink (Apache Software Foundation, 2017d).

The Hadoop framework, which includes MapReduce, was a favourite in academic literature (Chen et al., 2014; Fang, 2015; Prasad and Avinash, 2014; Ranjan, 2014; Watson, 2015). However, Hadoop was found to be too resource intensive under normal operation. Therefore, Hadoop was not considered and instead Apache Spark was selected. Apache Spark is often used alongside Hadoop, however, for the proposed system it was implemented in a standalone configuration. The advantage of Spark was its native analytical features, which meant it could also be used in the data analytics subsystem. Another reason for choosing Spark was its official integration with MongoDB, which allows reading from and writing to the data storage subsystem.

Spark was deployed as a cluster with the use of its master-slave architecture. When in operation, tasks are submitted to the Spark master node. From there, the task is assigned to one or more slave nodes. On completion of the task, the master obtains and returns the results. For the Spark cluster, three slave nodes and one master node were created. This meant the system that hosted the master node also had a slave node to perform work.

Deploying one Spark master node results in a single point of failure. However, this holds true for any master-slave architecture with one master. Although this is an undesirable situation, its severity is reduced by the resource layer and the usage of Mesosphere DC/OS. The reasoning, as previously stated, is that Mesosphere DC/OS can restart Docker containers automatically. Nevertheless, there exists a trade-off between the number of masters and the resources available: the more masters there are, the fewer resources are available for slaves to perform work.

An official Docker image was not available for Spark and thus one was subsequently created. The Spark Docker image was created by applying the installation instructions given on the Apache Spark website (Apache Software Foundation, 2017e) with the Bitnami Minideb Docker image serving as the base. Both master and slave nodes used the same Docker image with only a difference in the configuration.

The implementation diagram of the data processing subsystem is shown in Figure 61. System users would communicate only with the Spark master which assigns tasks to one or more slaves. On completion of the work, the master would communicate data back to the users. The figure also illustrates how the Spark software is encapsulated within Docker containers.

Word count: 8954

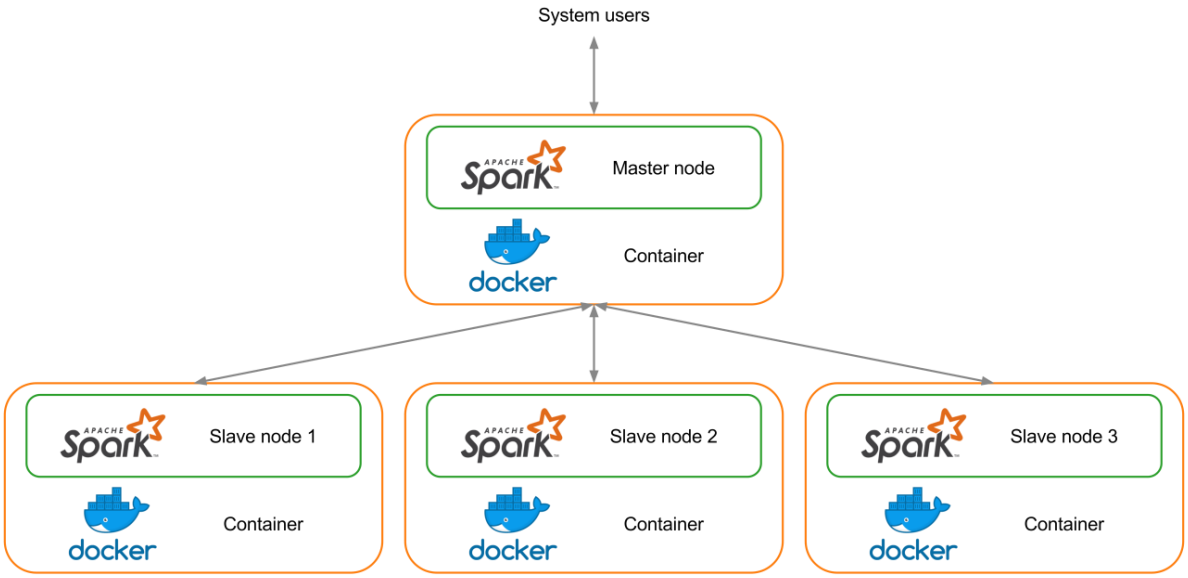


Figure 61: Implementation diagram of the data processing subsystem

2.2.3. Data analytics subsystem

The last part of the service layer was the design of the data analytics subsystem. This subsystem consisted of two components as described in literature (Begoli and Horey, 2012; Ranjan, 2014). The first component was a programming language. The programming language was needed for analysis methods and to enable the creation of analytics applications. Python (Python Software Foundation, 2017) was chosen as the programming language.

The second component was a framework for large-scale data processing and mining. Apache Spark was chosen as the processing and mining framework. The reasons for choosing Python was due to its popularity and general-purpose nature, which was important as normal users would use it. Thus, providing a tool that people are more likely to know or would be willing to learn, is the most important criterion. As mentioned in the previous section, Spark was a natural choice. Furthermore, Spark provides an API that is accessible through Python.

A new Spark cluster was not created for the data analytics subsystem. Instead, the cluster of the data processing subsystem was reused. Python was also not contained within a separate Docker image because it could only be used with Spark when installed alongside it. Python can, however, still be used independently from Spark despite being installed with it. Miniconda, from Continuum Analytics, was used to install Python 3.6 as part of the base Spark Docker image.

The complete service layer is shown in Figure 62. The figure shows two Spark components merely to highlight its usage in two subsystems. The dashed line symbolises that the two Spark components are one, i.e. only one Spark cluster would be present. Less detail is shown for the data storage subsystem

as greater detail can be seen in Figure 60. Each subsystem is shown in a different colour, orange for the data storage subsystem, green for the data processing subsystem and red for the data analytics subsystem.

Figure 62 shows where system users would connect to the software components. The figure shows the flexibility of Python, which is connected to all the subsystems. This is so because Python has libraries for communicating with all of the system software.

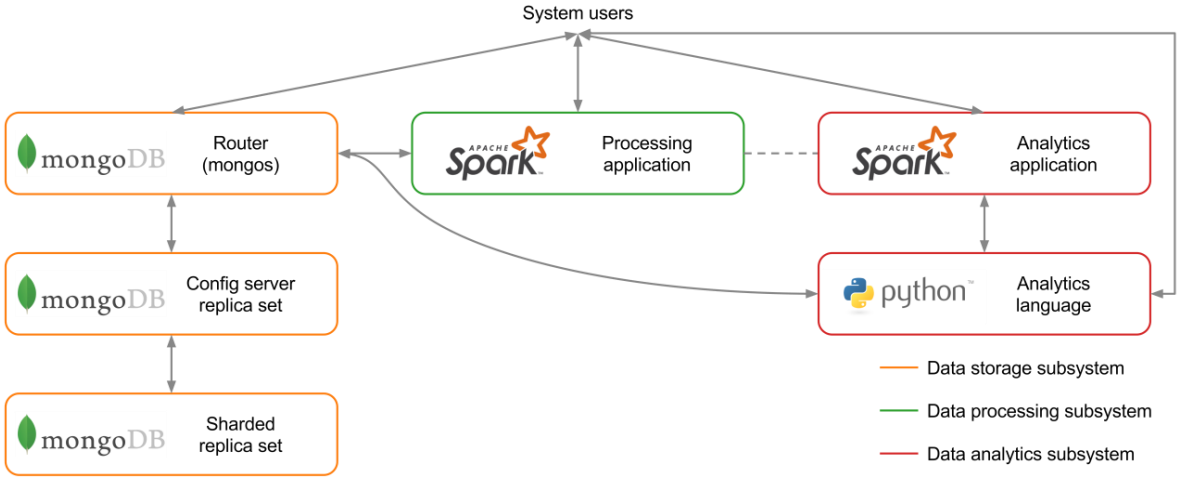


Figure 62: Service layer implementation diagram

2.3. Interface layer design

The design of the interface layer focused on two objectives, as was shown in Figure 58. These two objectives were the provisioning of an API and a web interface. The API was required to encapsulate business logic and functions into an interoperable component. Organisations could make use of multiple programming languages for their systems or applications. Therefore, a means of accessing MongoDB independent of programming language was necessary. The web interface was needed to provide users with easy access to data processing and analysis. Thus, the web interface focused on Python and Spark accessibility.

The API was created as a web service based on the services-oriented architecture (SOA) approach (Serrano et al., 2014). The reason for a web service is that most programming languages have functions for web connectivity. Furthermore, most SOA approaches use a web service. Up to this point, the system was generalised. However, the API is the component that would be tailored to the context in which the system is deployed. Different organisations would require different features. Therefore, the API (to be discussed further) was specific to the case study organisation.

The API was implemented as a Windows Communication Foundation (WCF) web service in the C# .Net programming language (Microsoft Corporation, 2017a). This web service made use of the

simple object access protocol because of the additional security features required (Serrano et al., 2014). In particular, the web service was secured with TLS 1.2 and message security, which provides confidentiality and integrity. The X.509 certificates were also purposefully verified to provide authorisation, which prevented unauthorised clients from accessing the web service. The web service was not implemented within a Docker image because Windows would be required as the base image. A Windows-based Docker image can, however, be created as explained in (Microsoft Corporation, 2017b), but the licensing of the base image would make it expensive to use. Hence, the web service was deployed to a separate Windows server.

The web interface, which is technically also a web service, was implemented with an open source software project. Some of the software considered were Beaker (Two Sigma Open Source, 2017), Apache Zeppelin (Apache Software Foundation, 2017f) and Jupyter Notebook (Project Jupyter, 2017). All of these software projects complied with the requirement to support Spark and Python, which means any one of them could have been used. Apache Zeppelin was finally chosen for it had close ties to Apache projects, such as Spark.

The benefit of the web interface is that it allows users to use Python and Spark through a logical notebook interactively. The notebook is an environment, provided in a web browser, in which code can be written to perform, for example, calculations, data access and to draw graphs. No processing is performed on the user's computer and no installation of software is required.

A Docker image was used to encapsulate the web interface. The installation instructions given on the Zeppelin website (Apache Software Foundation, 2017g) were followed to create the Docker image. Zeppelin also needed to reside with Python and Spark. Therefore, the Spark Docker image was reused as a base for the Zeppelin image.

Figure 63 shows the final system diagram of the interface layer. As seen in the figure, the interface layer is the least complex of the system layers. It consists of only the API web service and the web interface. The WCF service is shown as residing on an external server and the Apache Zeppelin software, for the web interface, is deployed as a Docker container.

Word count: 8954

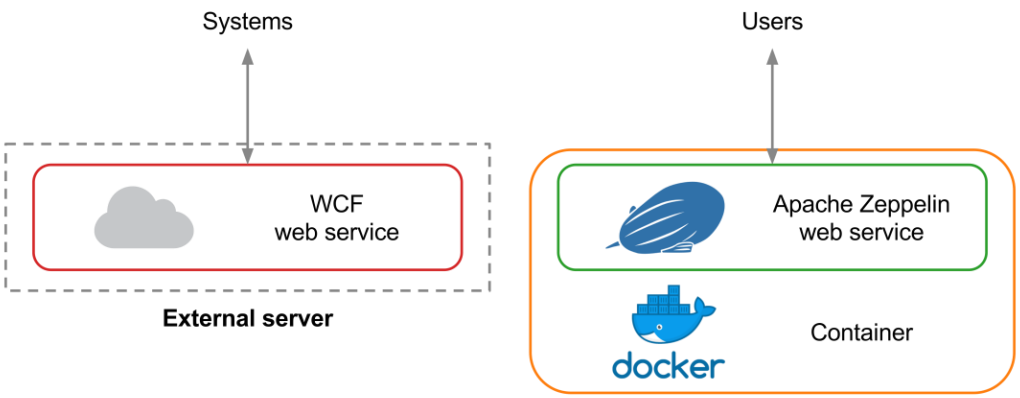


Figure 63: Interface layer implementation diagram

The system, as described in the previous section, was deployed in an engineering consultancy company working with industrial and mining companies' data. The engineering consultancy company and their clients used the system to improve energy efficiency. Data for each of the consultancy company's clients were stored, processed and analysed by the deployed system. The general nature of the API allowed it to be used for all their clients' data. Personnel of the company used the web interface to view the analysed and processed data.

Overall the system contained the data of ten mining and industrial companies which equates to an excess of 50 million data points per year. A variety of case studies were taken from the company to show the insights gained from big data and how it assisted with sustainable energy consumption. Each case study focused on a different part of the management structure, which is shown in Figure 64. The case studies were also distributed with regard to industry, and comprised mining, cement production and steel production. In each case study, the underlying big data system was used to collect, aggregate and analyse available data.

Figure 64 shows a typical organisational hierarchy, starting with top-level management at the top and operational personnel at the bottom. The figure implies that the higher up in the hierarchy, the more power and responsibility personnel have. Therefore, higher-level personnel would only be interested in high impact, company-wide information to make management decisions on energy usage. In contrast, lower-level personnel are only responsible for specific sections or tasks. These personnel are more interested in information specific to their work when making the operational decisions that reduce energy consumption. The reasoning just described was used by the developed system to bring the correct information to the attention of the correct personnel. The case studies are described by showing what information was available to the personnel and what decisions could be made therewith.

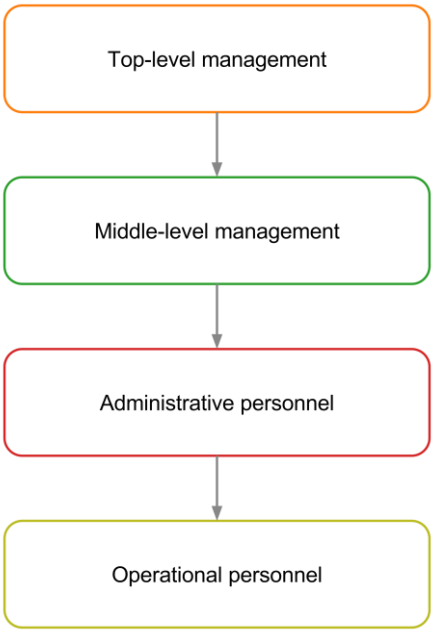


Figure 64: Organisational management hierarchy

3.1. Case study 1: Compressed air wastage

A case study was done on a gold mine's compressed air distribution network. As with any equipment, usage thereof results in damage, i.e. wear and tear. Wear and tear manifests as leaks in pipes, which cause a loss of compressed air and wastage of the energy that was used to compress the air. Using the data from various measurement points in a mining shaft, the big data system computed compressed air losses. This information was placed into a report to create awareness. Figure 65 is an example of the compressed air losses for one day.

Word count: 8954

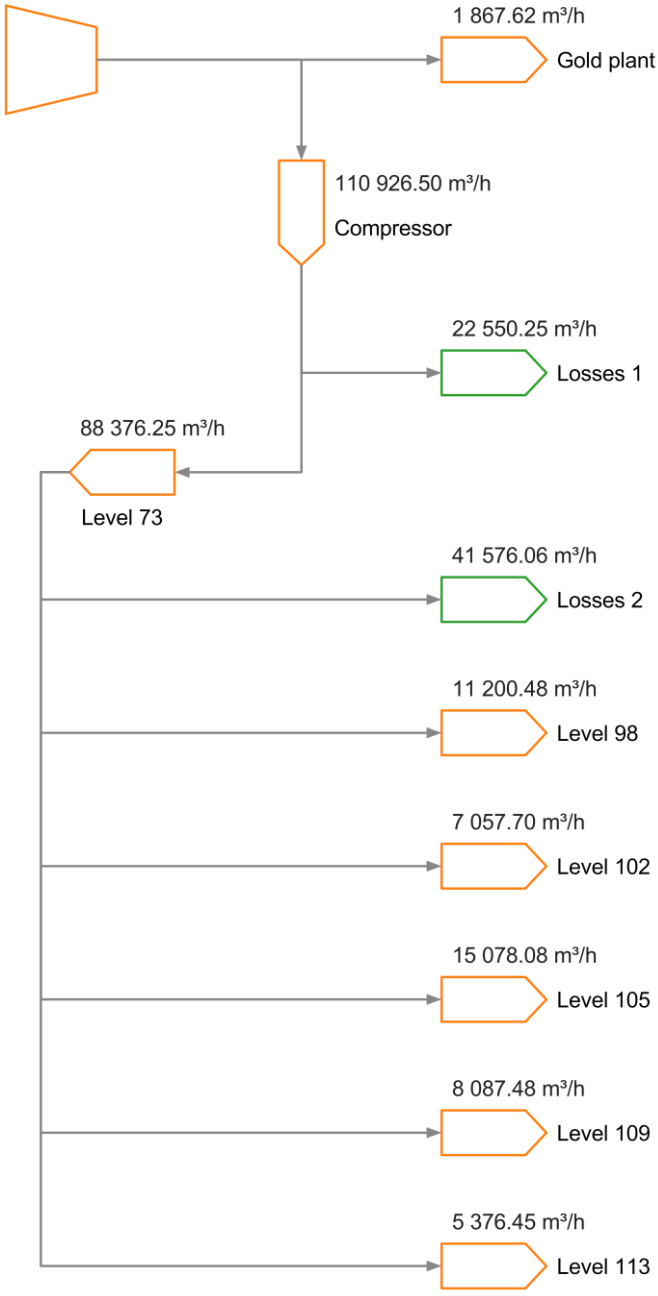


Figure 65: Compressed air distribution diagram for one day

Figure 65 shows the high-level, aggregated volumetric air flow rates to various users in a compressed air network. The volumetric flow rates of the actual data are shown at the orange pentagons. Losses are calculated from the data and are shown at the green pentagons. This figure would form part of a report sent to mining personnel. Middle-level management such as the shaft engineer and energy or production managers of the mine would then use this information to make operational decisions to fix pipes and leaks to curb energy losses. For example, in this case study a reduction of air wastage results in a potential compressed air saving of 57.8%.

3.2. Case study 2: Furnace gas optimisation

For this case study, gas optimisation was investigated at a steel plant. The steel plant consists of five mills, which are used to roll steel into specific profiles. Each mill uses a reheating furnace to heat the steel before rolling. The reheating furnaces can operate with a mixture of Coke Oven Gas (COG) and natural gas. The big data system uses data analytics to determine the efficiency characteristics of each reheating furnace. This determines how efficient the mill is with different gas mixtures. Furthermore, COG is a by-product gas which is available "for free", whereas natural gas needs to be purchased. Therefore, the big data system uses mill data to determine if the steel plant optimally operates their mills. These optimisations need to occur in near real time to be useful. Figure 66 shows the gas mixture at a particular time, with the suggested optimised gas mixture shown in Figure 67.

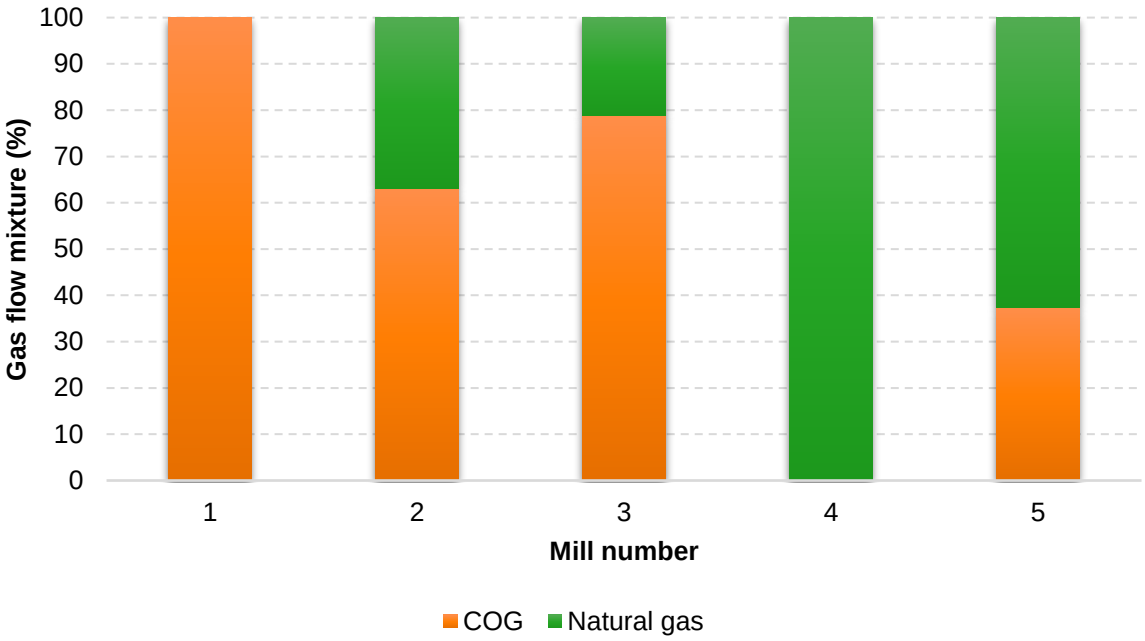


Figure 66: Actual mill furnace gas mixture

Word count: 8954

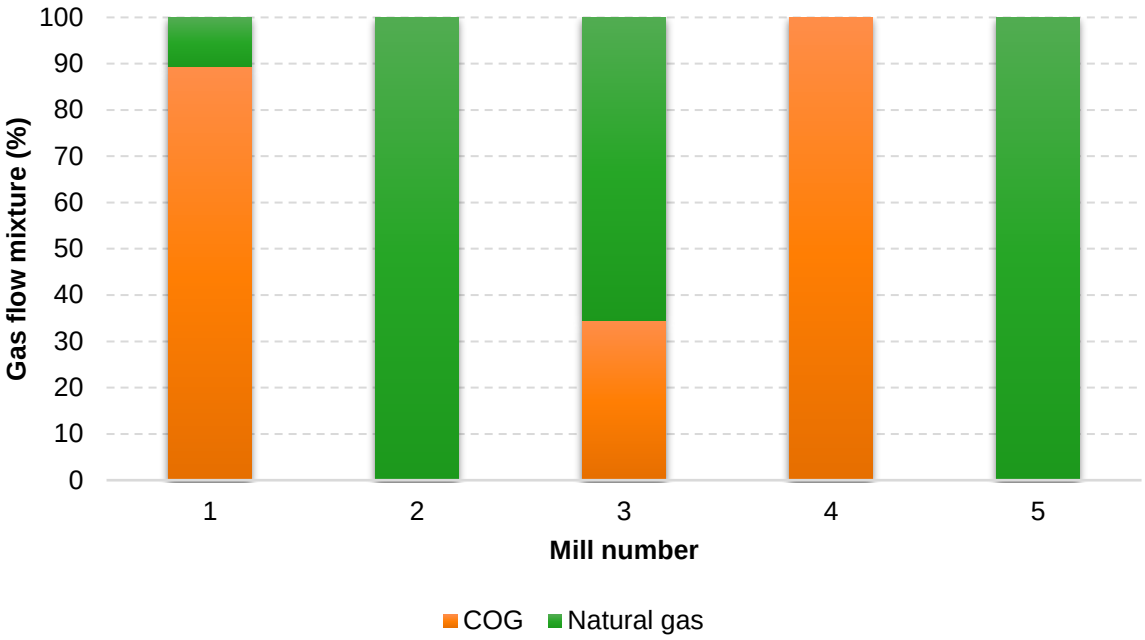


Figure 67: Optimised mill furnace gas mixture

The big data system monitors the data in real time, analyses it and provides a suggestion in the form of Figure 67. Control room and equipment operators (operational personnel) use Figure 66 to see the current gas mixture to each mill's furnace. Using the information provided by the big data system, the operators can adjust the gas mixture to optimise the furnace operation. The aim, therefore, is to optimise the gas mixture of each mill for the available COG. The impact of this information is an increase in mill efficiency and a subsequent decrease in (gas) energy usage. Secondly, full utilisation of the COG would decrease the usage of natural gas, thereby decreasing gas production and reducing emissions. Acting on the information illustrated in the figures, 2190 m^3/h of natural gas was saved.

3.3. Case study 3: Gas flaring identification

This case study continues with the steel plant of section 3.2. In section 3.2, COG was described as a by-product gas which can be reused by steel plant mills. Another by-product gas that can be used by a steel plant to generate electricity is Blast Furnace Gas (BFG). However, if BFG is not optimally used it needs to be flared, i.e. burned. The reason is that only a finite amount of gas can be stored by the steel plant in a gas holder. Once the gas holder is saturated, any subsequent gas needs to be flared. Therefore, unnecessary flaring of BFG wastes energy, increases emissions and requires the production of additional grid electricity.

Figure 68 provides information for the monitoring of BFG flaring. The line graphs show the level of the two gas holders during different times of a day. The orange and green lines show the BFG and COG holder levels respectively. The bars or spikes indicate the flow, on the secondary axis, through a flare stack, which is responsible for flaring. This data is presented by the big data system on a real-time basis

to a control room operator (operational personnel) and production manager (middle-level management). They use the figures to identify periods of energy wastage. These identifications lead to the optimisation of BFG usage. By maximising BFG usage, the steel plant decreases its need for grid electricity, reducing grid demand and subsequently reducing emissions. For the day in question, 55.13 MWh could have been generated by using BFG instead of flaring it.

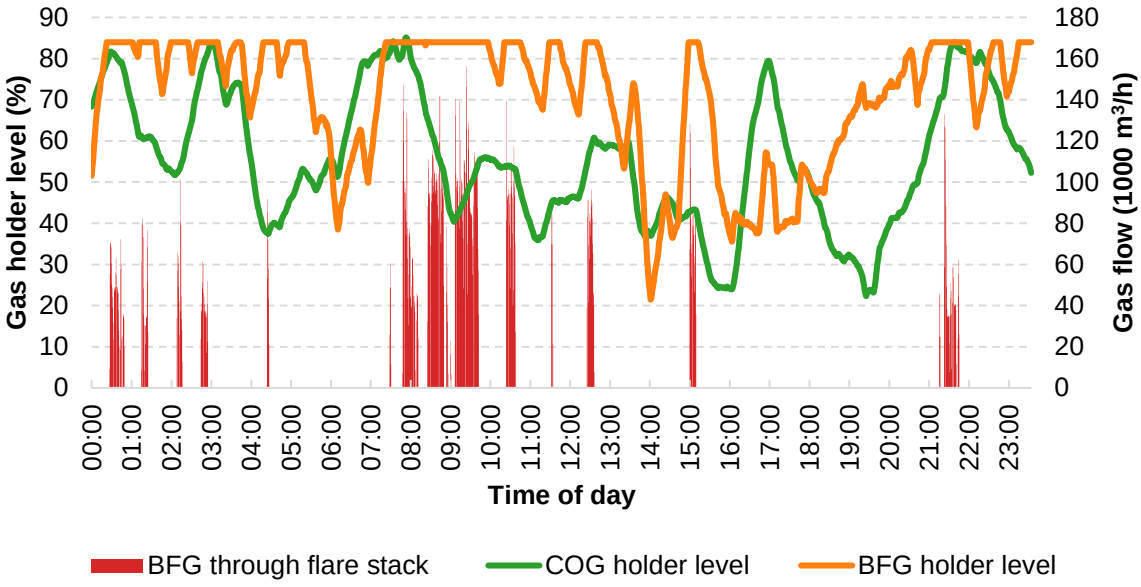


Figure 68: Gas holder levels and flaring during a day

3.4. Case study 4: Energy wastage identification

The next case study was done on a cement plant, specifically the finishing milling section. This plant section is the final step in the process to create cement before it is bagged. The big data system was used to capture all energy information across the entire plant, and data analysis was conducted thereon. The results of the data analysis are displayed on a monthly basis to operations and energy managers (middle-level management) to identify areas of improvement and wastage, thus possibly leading to energy-saving projects. The finishing milling section consisted of a few major energy consumers, whose consumption are depicted in Figure 69 for a one-month period.

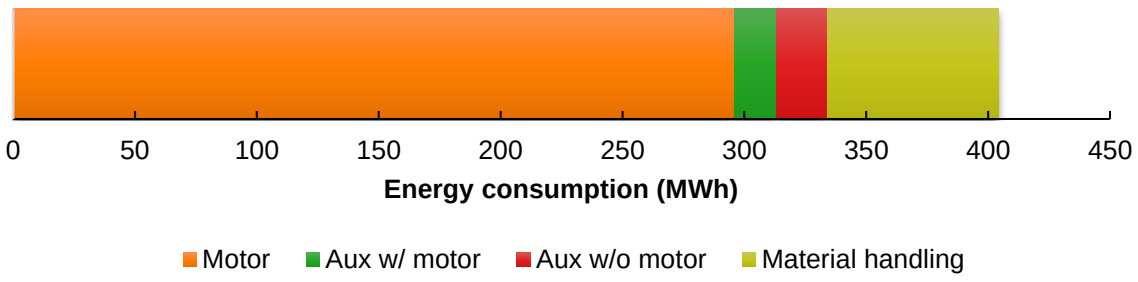


Figure 69: Energy consumption contributions for a month

Figure 69 shows there were three energy consumers namely a motor, auxiliaries and material handling. The graph looks correct; however, the auxiliaries must only be used when the motor is used. Therefore, the red section of the graph shows a situation where the auxiliaries were consuming energy even though the motor was turned off. Consequently, energy was wasted, which had gone unnoticed sans the insights from the data analysis. Acting on this information by middle management ensured that the same problems did not occur again. Wastage of approximately 20.89 MWh was identified for the month illustrated.

3.5. Case study 5: Financial impact

The big data system was implemented across an entire mining group. These mining groups typically have many operations which could be shafts, plants or buildings that consume energy. As with all organisations, a mining company creates budgets to plan company expenditure and track expenses, which includes energy consumption. Administrative personnel, typically in the financial department and top-level management, usually want to track energy usage and expenses across all operations. By using the energy consumption of all users of each operation, the big data system can compute actual total energy usage and costs. Furthermore, data analysis of historical data is used to determine energy budgets to benchmark performance. Some of this information is summarised in the table in Figure 70, which was taken from a report that was generated by the big data system.

Operations	Budget (kWh)	Actual (kWh)	Difference
Operation 1 ☺	1 705 532	1 546 393	-9.3%
Operation 2 ☺	1 330 317	1 263 623	-5.0%
Operation 3 ☺	345 768	268 424	-22.4%
Operation 4 ☺	472 651	406 907	-13.9%
Operation 5 ☺	1 215 383	1 131 937	-6.9%
Operation 6 ☹	573 781	602 213	5.0%
Operation 7 ☹	1 654 715	1 902 776	15.0%
Operation 8 ☹	259 179	301 807	16.4%
Total kWh ☺	**7 557 327**	**7 424 080**	**-1.8%**

Figure 70: Energy consumption of various mining operations

Figure 70 shows the performance of all operations of a mining company for one day. For each operation, the budgeted energy consumption and actual consumption is shown. The percentage of over or under usage is shown in the difference column. In the bottom row, the total budgeted consumption, actual consumption and percentage difference are shown in the respective columns. Management can thus see the cost saving for the mine and which operation was the most energy efficient. Continuous energy awareness and involvement by management aided in facilitating the 133 MWh savings achieved.

Word count: 8954

3.6. Case study 6: Total energy consumption

For the final two case studies, we again return to a mining group, different from the one mentioned in section 3.5. The top-level management of the company needed to see the bottom-line impact of various operational-level energy-saving initiatives. By using the big data system to analyse and process consumption data for the entire group, total energy consumption information was obtained. This total energy consumption information is illustrated in Figure 71.

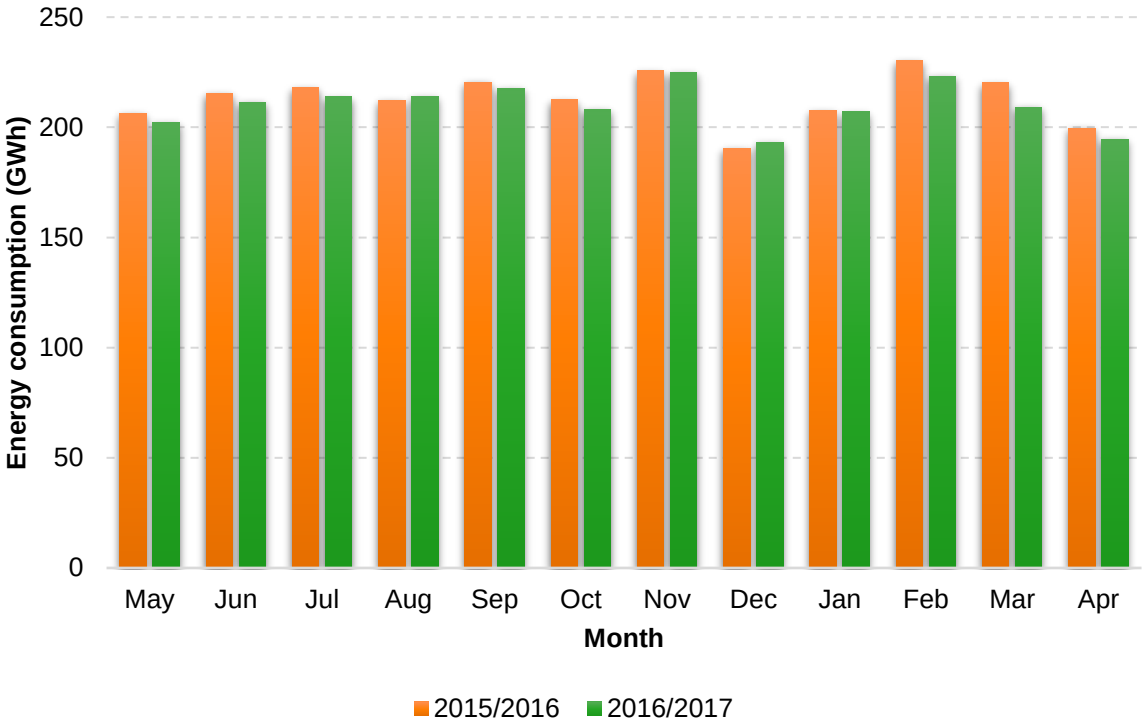


Figure 71: Comparison of monthly energy consumption over two financial years

Figure 71 shows a comparison of two years' energy consumption for specific months of the year. This graph contains the least detailed information, which makes it suitable for top management to use to make management decisions. The big data system delivers this information in the form of annual reports which management can use to compare the total energy consumption year-on-year. With this information, various actions can be taken such as the implementation of new projects, improvement of existing projects and the allocation of capital. By showing low detail information with high value, the big data system enables personnel with the most authority to make informed decisions, hopefully for the benefit of the environment. In this case study, continuous energy consumption improvement caused improved year-on-year energy intensity (0.15 MWh/tonne vs 0.13 MWh/tonne).

Word count: 8954

3.7. Case study 7: Carbon dioxide intensity

This final case study ties in with the previous case study found in section 3.6. The end goal of obtaining energy-related data is to be able to see the effect of production on the environment. The big data system was used to monitor all carbon dioxide (CO2) emission data, from all sources, for the mining company. Using the CO2 emissions coupled with production data results in CO2 intensity information. This information, as used on the web interface of the big data system, is illustrated in Figure 72.

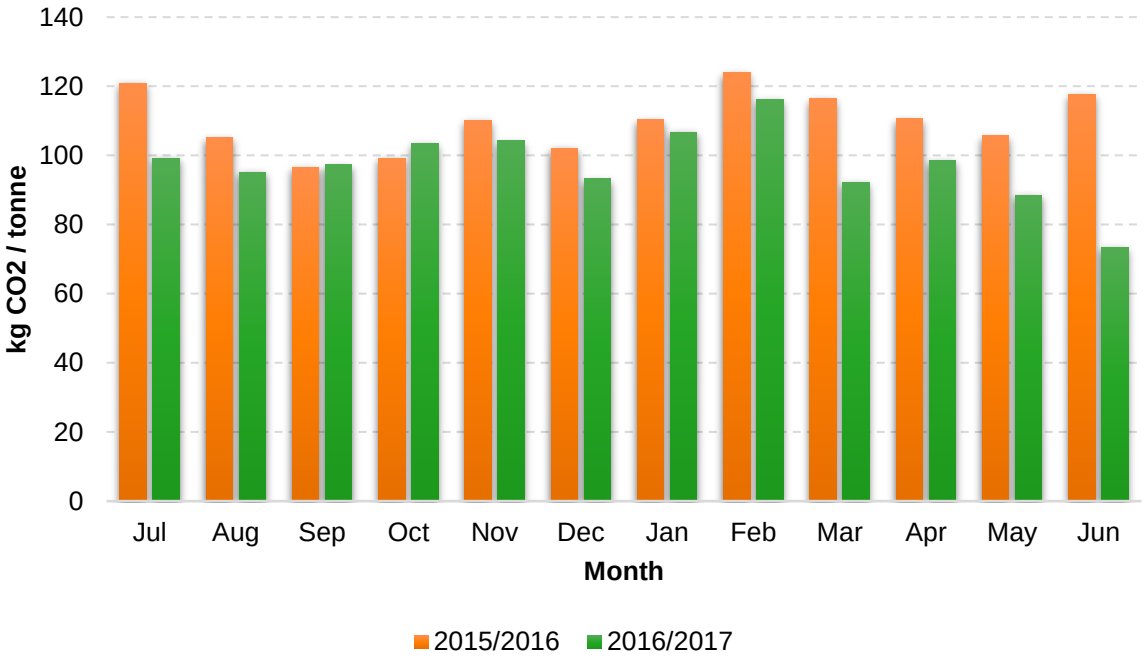


Figure 72: Carbon dioxide intensity comparison for two financial years

The CO2 intensity of Figure 72 illustrates how carbon intensive each tonne of production was for each month of the year. The graph shows how clean production was for the previous two financial years. It is clear that production became less carbon-intensive, with an average intensity of 97.35 kg CO2/tonne compared to the 109.91 kg CO2/tonne from the previous year. This is due to the focus on energy efficiency by the mining company in the latest financial year, which included the use of the big data system. Top-level management can use this information to report to stakeholders, as well as budget and plan for new energy savings and emission reducing initiatives. The biggest impact the graph has, however, is to see the impact (payoff) of energy and emission reduction measures. Top management can use this to help motivate future projects that would further improve sustainable consumption and cleaner production.

Word count: 8954

3.8. Discussion

The impact on sustainable consumption and cleaner production is further illustrated in Table 20. For each case study, the average monthly energy saved was calculated. Subsequently, the CO2 emissions prevented and water saved were computed. The factors that were used to derive the emissions prevented and water saved were taken from (Eskom, 2017). As the table shows, every MWh saved resulted in an approximate one tonne of CO2 prevented and 1.5 m^3 of water saved.

Table 20: Environmental impact associated with the energy savings for each case study

Case study	Average energy saved (MWh per month)	CO2 emissions prevented (tonne per month)	Water saved (m^3 per month)
1	≤ 4 222	≤ 4 138	≤ 6 038
2	≤ 4 587	≤ 4 496	≤ 6 560
3	≤ 965	≤ 946.07	≤ 1 380
4	21	20	30
5	133	131	191
6 & 7	65 694	64 380	93 942
Total	**≤ 75 622**	**≤ 74 111**	**≤ 108 141**

This study provides a perspective of energy consumption from only the mining and industrial sectors. Other sectors such as the residential and transportation sectors could also be candidates for big data adoption. Due to the design of the big data system, it is possible to use the designed system with customised APIs for a variety of sectors and situations.

Advancement of technology and the development of software will impact software choices made during the system design. However, the modularity and extensibility of the system ensure a simplified substitution of improved components. Therefore, as improved software is created, it can replace existing components in the system.

The case studies presented were limited in their application of big data due to the volume and variety of available data. Thus, as the industry adopts IoT and improved sensor technology, more data would be available. This would allow for more advanced and in-depth analytics, which could include predictions.

Sustainable use of energy is important for various reasons. It is important for large industries such as the industrial and mining sectors to strive towards cleaner production by actively managing their energy efficiency. In this regard, big data can help. This study found a lack of big data usage in the mining and industrial sectors, and when adopted the usage of big data was in its infancy.

A novel big data system for energy usage in the industrial and mining sectors was designed. This system is modular and customisable and was implemented in an engineering services company. The system is unique in its use of microservices and containers, which enables deployment in the cloud or on-premise. Its use of web services, in the form of an API and web interface, allow it to be utilised for a variety of situations.

The study provided seven case studies for the practical usage of big data. Combined, the case studies focused on providing insights throughout the four levels of an organisational hierarchy: top-level management, middle-level management, administrative personnel, and operational personnel. The case studies varied in their type of organisation, which included mines, a steel plant, and a cement plant. The case studies focused on using the big data system to either gain insight or actively manage energy. Taken together, the case studies showed the insights provided through big data could aid in increasing energy efficiency to the benefit of the environment. By focusing on the entire organisational hierarchy, the impact of big data was far-reaching in providing personnel with the power to make improved decisions. The total maximum impact resulting from the case studies was a saving of 75 622 MWh, a reduction of 74 111 tonne CO2, and a saving of 108 141 m^3 water.

Future developments in big data software could broaden the implementation of the proposed big data system. Furthermore, the modularity and extensibility of the system ensure adoption of future software and technology. This study was also limited to the use of data that was available, which by all accounts will increase and diversify, especially in the industrial and mining sectors. Finally, there remains an opportunity for advanced analytics, more so than was illustrated in this study.

Acknowledgements

TEMM International (Pty) Ltd funded this study and ETA Operations (Pty) Ltd deployed the system and allowed for the use of their clients' data. We also acknowledge the following people for their contributions to this paper: Kate Lowes (proofreading), Sybrand van Niekerk (case study 2), Alexander Ludick (case study 3), Wiehan Pelser (case study 4), Philip Maré (case study 5) and Johan Bredenkamp (case study 6).

References

Abdelaziz, E.A., Saidur, R., Mekhilef, S., 2011. A review on energy saving strategies in industrial sector. Renew. Sustain. Energy Rev. 15, 150–168. doi:10.1016/j.rser.2010.09.003

Alharthi, A., Krotov, V., Bowman, M., 2017. Addressing barriers to big data. Bus. Horiz. 60, 285–292. doi:10.1016/j.bushor.2017.01.002

Apache Software Foundation, 2017a. Apache Spark [WWW Document]. Apache Spark website. URL https://spark.apache.org/ (accessed 5.25.17).

Apache Software Foundation, 2017b. Apache Storm [WWW Document]. Apache Storm website. URL http://storm.apache.org/ (accessed 5.25.17).

Apache Software Foundation, 2017c. Apache Hadoop [WWW Document]. Apache Hadoop website. URL http://hadoop.apache.org/ (accessed 5.25.17).

Apache Software Foundation, 2017d. Apache Flink [WWW Document]. Apache Flink website. URL https://flink.apache.org/ (accessed 5.25.17).

Apache Software Foundation, 2017e. Spark Standalone Mode [WWW Document]. Apache Spark website. URL https://spark.apache.org/docs/latest/spark-standalone.html (accessed 5.27.17).

Apache Software Foundation, 2017f. Apache Zeppelin [WWW Document]. Apache Zeppelin website. URL https://zeppelin.apache.org/ (accessed 5.26.17).

Apache Software Foundation, 2017g. Apache Zeppelin Quick Start [WWW Document]. Apache Zeppelin website. URL https://zeppelin.apache.org/docs/latest/install/install.html (accessed 5.29.17).

Bazilian, M., Rogner, H., Howells, M., Hermann, S., Arent, D., Gielen, D., Steduto, P., Mueller, A., Komor, P., Tol, R.S.J., Yumkella, K.K., 2011. Considering the energy, water and food nexus: Towards an integrated modelling approach. Energy Policy 39, 7896–7906. doi:10.1016/j.enpol.2011.09.039

Begoli, E., Horey, J., 2012. Design Principles for Effective Knowledge Discovery from Big Data, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. IEEE, pp. 215–218. doi:10.1109/WICSA-ECSA.212.32

Bizikova, L., Roy, D., Swanson, D., Venema, H.D., McCandless, M., 2013. The water-energy-food security nexus: Towards a practical planning and decision-support framework for landscape investment and risk management. International Institute for Sustainable Development, Winnipeg, Manitoba, Canada.

Chen, M., Mao, S., Liu, Y., 2014. Big Data: A Survey. Mob. Networks Appl. 19, 171–209. doi:10.1007/s11036-013-0489-0

Chu, S., Cui, Y., Liu, N., 2016. The path towards sustainable energy. Nat. Mater. 16, 16–22. doi:10.1038/nmat4834

Clegg, D., 2015. Evolving Data Warehouse and BI Architectures: The Big Data Challenge. Bus. Intell. J. 20, 19–24.

Dessureault, S., 2016. Understanding big data. CIM Mag. 11.

Docker Inc., 2016. What is a Container [WWW Document]. Docker website. URL https://www.docker.com/what-container (accessed 4.6.17).

Ecma International, 2013. The JSON Data Interchange Format. Stand. ECMA-404 1–5.

Erkayaoğlu, M., 2017. Data warehousing and big data applications in mining industry, in: IMCET 2017: New Trends in Mining - Proceedings of 25th International Mining Congress of Turkey. pp. 242–249.

Eskom, 2017. Eskom Integrated Report 2017.

Fang, H., 2015. Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem, in: 2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER). IEEE, pp. 820–824. doi:10.1109/CYBER.2015.7288049

Feinleib, D., 2014. Big Data Bootcamp: What Managers Need to Know to Profit from the Big Data Revolution, 1st ed. Apress, Berkely, CA, USA.

Fowler, M., 2011. PolyglotPersistence [WWW Document]. Martin Fowler Pers. website. URL https://martinfowler.com/bliki/PolyglotPersistence.html (accessed 4.10.17).

Gandomi, A., Haider, M., 2015. Beyond the hype: Big data concepts, methods, and analytics. Int. J. Inf. Manage. 35, 137–144. doi:10.1016/j.ijinfomgt.2014.10.007

GigaSpaces Technologies Inc., 2017. Cloudify [WWW Document]. Cloudify website. URL http://cloudify.co/ (accessed 5.19.17).

Hansen, J., Kharecha, P., Sato, M., Masson-Delmotte, V., Ackerman, F., Beerling, D.J., Hearty, P.J., Hoegh-Guldberg, O., Hsu, S.-L., Parmesan, C., Rockstrom, J., Rohling, E.J., Sachs, J., Smith, P., Steffen, K., Van Susteren, L., von Schuckmann, K., Zachos, J.C., 2013. Assessing "Dangerous Climate Change": Required Reduction of Carbon Emissions to Protect Young People, Future Generations and Nature. PLoS One 8, 1–26. doi:10.1371/journal.pone.0081648

International Energy Agency, 2016a. World Energy Outlook 2016 (Executive Summary), World Energy Outlook. Paris, France.

International Energy Agency, 2016b. Energy Efficiency Market Report 2016. Paris, France.

International Institute for Analytics, 2016. Advanced Analytics & Big Data Adoption Report. IIA and Dell Digital Business Services Inc., Portland, Oregon, U.S.A.

Jacobson, M.Z., Delucchi, M.A., Bauer, Z.A.F., Goodman, S.C., Chapman, W.E., Cameron, M.A., Bozonnat, C., Chobadi, L., Clonts, H.A., Enevoldsen, P., Erwin, J.R., Fobi, S.N., Goldstrom, O.K., Hennessy, E.M., Liu, J., Lo, J., Meyer, C.B., Morris, S.B., Moy, K.R., O'Neill, P.L., Petkov, I., Redfern, S., Schucker, R., Sontag, M.A., Wang, J., Weiner, E., Yachanin, A.S., 2017. 100% Clean and Renewable Wind, Water, and Sunlight All-Sector Energy Roadmaps for 139 Countries of the World. Joule 1, 108–121. doi:10.1016/j.joule.2017.07.005

Knowledgent, 2015. 2015 Big Data Survey - Current Implementation Challenges. New York, NY, U.S.A.

Kratzke, N., 2014. A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms. Open J. Mob. Comput. Cloud Comput. 1, 17–30.

Laskier, R., 2016. Modernizing the Mining Industry With the Internet Of Things, in: Internet of Things and Data Analytics Handbook. John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 521–543. doi:10.1002/9781119173601.ch32

Linthicum, D.S., 2016. Practical Use of Microservices in Moving Workloads to the Cloud. IEEE Cloud

Comput. 3, 6–9. doi:10.1109/MCC.2016.114

Lukman, R.K., Glavič, P., Carpenter, A., Virtič, P., 2016. Sustainable consumption and production – Research, experience, and development – The Europe we want. J. Clean. Prod. 138, 139–147. doi:10.1016/j.jclepro.2016.08.049

Marz, N., Warren, J., 2015. Big Data: Principles and Best Practices of Scalable Realtime Data Systems, 1st ed. Manning Publications Co., Greenwich, CT, USA.

Mesosphere Inc., 2017. DC/OS [WWW Document]. Mesos. DC/OS website. URL https://dcos.io/ (accessed 5.19.17).

Microsoft Corporation, 2017a. Windows Communication Foundation [WWW Document]. Microsoft Dev. Netw. website. URL https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx (accessed 5.26.17).

Microsoft Corporation, 2017b. Microsoft Windows Server 2016 Server Core [WWW Document]. Docker Hub Microsoft Repos. URL https://hub.docker.com/r/microsoft/windowsservercore/ (accessed 5.29.17).

Mohanty, S., Jagadeesh, M., Srivatsa, H., 2013. Big Data Imperatives: Enterprise Big Data Warehouse, BI Implementations and Analytics, 1st ed. Apress, Berkely, CA, USA.

MongoDB Inc., 2017a. BSON [WWW Document]. BSON Specif. URL http://bsonspec.org/ (accessed 5.22.17).

MongoDB Inc., 2017b. Sharded Cluster Components [WWW Document]. MongoDB Serv. Doc. URL https://docs.mongodb.com/manual/core/sharded-cluster-components/ (accessed 5.22.17).

O'Leary, D.E., 2014. Embedding AI and Crowdsourcing in the Big Data Lake. IEEE Intell. Syst. 29, 70–73. doi:10.1109/MIS.2014.82

Pahl, C., 2015. Containerization and the PaaS Cloud. IEEE Cloud Comput. 2, 24–31. doi:10.1109/MCC.2015.51

Perrons, R.K., McAuley, D., 2015. The case for "n«all": Why the Big Data revolution will probably happen differently in the mining sector. Resour. Policy 46, 234–238. doi:10.1016/j.resourpol.2015.10.007

Prasad, S., Avinash, S.B., 2014. Application of polyglot persistence to enhance performance of the energy data management systems, in: 2014 International Conference on Advances in Electronics Computers and Communications. IEEE, pp. 1–6. doi:10.1109/ICAECC.2014.7002444

Project Jupyter, 2017. Jupyter Notebook [WWW Document]. Jupyter website. URL http://jupyter.org/ (accessed 5.26.17).

Python Software Foundation, 2017. Python [WWW Document]. Python website. URL https://www.python.org/ (accessed 5.25.17).

Rahman, M.S., Noman, A.H.M., Shahari, F., Aslam, M., Gee, C.S., Isa, C.R., Pervin, S., 2016. Efficient energy consumption in industrial sectors and its effect on environment: A comparative analysis between G8 and Southeast Asian emerging economies. Energy 97, 82–89.

doi:10.1016/j.energy.2015.12.109

Ranjan, R., 2014. Streaming Big Data Processing in Datacenter Clouds. IEEE Cloud Comput. 1, 78–83. doi:10.1109/MCC.2014.22

Ringler, C., Bhaduri, A., Lawford, R., 2013. The nexus across water, energy, land and food (WELF): potential for improved resource use efficiency? Curr. Opin. Environ. Sustain. 5, 617–624. doi:10.1016/j.cosust.2013.11.002

Sadalage, P.J., Fowler, M., 2012. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Pearson Education.

Serrano, N., Hernantes, J., Gallardo, G., 2014. Service-Oriented Architecture and Legacy Systems. IEEE Softw. 31, 15–19. doi:10.1109/MS.2014.125

Sill, A., 2016. The Design and Architecture of Microservices. IEEE Cloud Comput. 3, 76–80. doi:10.1109/MCC.2016.111

Simmhan, Y., Aman, S., Kumbhare, A., Liu, R., Stevens, S., Zhou, Q., Prasanna, V., 2013. Cloud-Based Software Platform for Big Data Analytics in Smart Grids. Comput. Sci. Eng. 15, 38–47. doi:10.1109/MCSE.2013.39

Singleton, A., 2016. The Economics of Microservices. IEEE Cloud Comput. 3, 16–20. doi:10.1109/MCC.2016.109

Song, M., Cen, L., Zheng, Z., Fisher, R., Liang, X., Wang, Y., Huisingh, D., 2017. How would big data support societal development and environmental sustainability? Insights and practices. J. Clean. Prod. 142, 489–500. doi:10.1016/j.jclepro.2016.10.091

The Kubernetes Authors, 2017. Kubernetes [WWW Document]. Kubernetes website. URL https://kubernetes.io/ (accessed 5.19.17).

Thones, J., 2015. Microservices. IEEE Softw. 32, 116–116. doi:10.1109/MS.2015.11

Two Sigma Open Source, 2017. Beaker [WWW Document]. Beaker website. URL http://beakernotebook.com/ (accessed 5.26.17).

United Nations, 2016. Paris Agreement. Treaty No. 54113, United Nations Office of Legal Affairs, Treaty Section, Paris, France.

van der Meulen, R., 2016. Gartner Survey Reveals Investment in Big Data Is Up but Fewer Organizations Plan to Invest [WWW Document]. Gart. Newsroom. URL http://www.gartner.com/newsroom/id/3466117 (accessed 2.26.17).

Watson, H.J., 2015. Data Lakes, Data Labs, and Sandboxes. Bus. Intell. J. 20, 4–7.

Wiese, L., 2015. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases, De Gruyter Textbook. De Gruyter.

Zhou, K., Fu, C., Yang, S., 2016. Big data driven smart energy management: From big data to big insights. Renew. Sustain. Energy Rev. 56, 215–225. doi:10.1016/j.rser.2015.11.050