EXPERIMENTING WITH A BIG DATA FRAMEWORK FOR SCALING A DATA QUALITY QUERY SYSTEM

A thesis submitted to the University of Manchester for the degree of Master of Philosophy in the Faculty of Science and Engineering

2016

By
Sonia Cisneros Cabrera
School of Computer Science
Contents

Abstract 13
Declaration 14
Copyright 15
Dedication 16
Acknowledgements 17

1 Introduction 18
  1.1 Motivation ................................................................. 18
  1.2 Research Goals and Contributions ................................. 20
  1.3 Research Method ......................................................... 21
  1.4 Thesis Overview ......................................................... 25

2 Background and Related Work 26
  2.1 Data Quality: Wrangling and the Big Data Life Cycle ............ 26
  2.2 Big Data and Parallelism .............................................. 31
  2.3 Big Data Frameworks: Tools and Projects ...................... 33
    2.3.1 MapReduce, Hadoop and Spark ................................. 35
  2.4 Scalability of Data Management Operators .................... 37

3 Designing a Scalable \text{DQ}^2S 42
  3.1 The Data Quality Query System (\text{DQ}^2S) .................... 42
    3.1.1 The Framework .................................................... 43
    3.1.2 The Query .......................................................... 44
    3.1.3 The Datasets ....................................................... 46
    3.1.4 The Algorithms Included in the \text{DQ}^2S Database Algebra Implementation ....................................................... 48
3.2 The Artifact

3.2.1 The Java DQ^2S

3.2.2 The Python DQ^2S Implementation

3.2.3 The Optimised Python DQ^2S Implementation

3.2.4 The PySpark DQ^2S Implementation

3.3 Summary

4 Experiment design

4.1 Objectives and Scope of the Experiment

4.2 Data and Tools

4.3 Setup: Spark Cluster and Application Settings

4.3.1 The Apache Spark Local Mode

4.3.2 The Apache Spark Standalone Mode

4.4 Experiment Procedure

4.5 Algorithm Comparison

4.6 Experiment Evaluation

4.6.1 Performance Metrics

4.6.2 Validity Evaluation

4.7 Summary

5 Evaluation of Experimental Results

5.1 Experiments Results

5.2 Algorithm Comparison

5.3 Summary

6 Conclusions and Future Work

6.1 Research Summary and Key Results

6.2 Research Contributions and Impact

6.3 Future Work

Bibliography

A DQ^2S Source code

A.1 The Java DQ^2S

A.2 The Python DQ^2S
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3</td>
<td>The Optimised Python DQ²S</td>
<td>186</td>
</tr>
<tr>
<td>A.4</td>
<td>The PySpark DQ²S</td>
<td>191</td>
</tr>
<tr>
<td>B</td>
<td>Apache Spark set up</td>
<td>198</td>
</tr>
<tr>
<td>B.1</td>
<td>Configuration of Spark’s Basic Elements</td>
<td>198</td>
</tr>
<tr>
<td>B.2</td>
<td>Single Machine Deployment</td>
<td>199</td>
</tr>
<tr>
<td>B.3</td>
<td>PySpark Submission to the DPSF Cluster</td>
<td>201</td>
</tr>
<tr>
<td>C</td>
<td>Testing Results</td>
<td>205</td>
</tr>
<tr>
<td>C.1</td>
<td>Raw Results</td>
<td>205</td>
</tr>
<tr>
<td>C.2</td>
<td>Performance Metrics Results</td>
<td>224</td>
</tr>
<tr>
<td>D</td>
<td>Apache Spark User Level Proposal</td>
<td>227</td>
</tr>
</tbody>
</table>

Word Count: 37242
List of Tables

1.1 The Design Science Research guidelines ................................. 23
2.1 Big data life cycle processes definitions ............................... 30
2.2 Big data frameworks available for data quality tasks ............... 34
2.3 Summary of the most relevant state-of-the-art related studies .... 41
3.1 Operators comprising the DQ^2S Timeliness query ................. 50
3.2 Java DQ^2S technical description ..................................... 58
3.3 Python DQ^2S technical description ................................... 60
3.4 Optimised Python DQ^2S technical description ...................... 62
3.5 PySpark DQ^2S technical description ................................ 70
4.1 Research Question’s solution plan ..................................... 73
4.2 Relation of number of rows and size per each input dataset utilised in the experimentation ........................................... 74
4.3 Hardware specification of the desktop machine (PC) and the cluster (DPSF) utilised in the experimentation ...................... 75
4.4 Tools used in the empirical evaluation of the research ............. 76
4.5 Spark configuration locations ...................................... 79
4.6 General overview of the experiment’s aim ............................ 79
4.7 Number of runtimes and averages collected by experiment ........ 80
4.8 Conclusion validity threats applicable to the research presented. Details and addressing approach ................................. 96
4.9 Internal validity threats applicable to the research presented. Details and addressing approach ................................. 97
4.10 Construct validity threats applicable to the research presented. Details and addressing approach ................................. 98
4.11 External validity threats applicable to the research presented. Details and addressing approach ................................. 99
5.1 Relation of the dataset sizes and results when executing Optimised Python DQ²S in a single machine for Experiment 4.

5.2 Size in bytes for the first dataset that caused a failure (12 000 000 rows), and the maximum number of rows processed by the Optimised Python DQ²S (11 500 000).

5.3 Relation of seconds decreased from different configurations using 1 worker (W) and 1 to 4 cores (C) with the Standalone mode in a single machine.

5.4 Times Slower: Relation between the runtime results of the PySpark instance with 1 (sequential) and 8 threads (best time). Where local[1]<local[8] indicates the PySpark DQ²S in local mode with 8 threads is slower than the same implementation with 8 threads as many times as indicated.

5.5 Size in GB for the datasets sizes processed on the cluster.

5.6 Relation of the dataset sizes and results when executing the Optimised Python DQ²S in the cluster for Experiment 9.

5.7 Times Slower: Relation between Original Java and Python runtime results. Where Python <Java indicates the Python DQ²S is slower than the Java implementation as many times as indicated.

5.8 Times Slower: Relation between Python and Optimised Python runtime results. Where Python <Optimised Python indicates the Python DQ²S is slower than the Optimised Python implementation as many times as indicated.

5.9 Times Slower: Relation between Original Java and Optimised Python runtime results. Where Java <Optimised Python indicates the Java DQ²S is slower than the Optimised Python implementation as many times as indicated.

5.10 Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 1 thread. Where local[1]<Optimised Python indicates the PySpark DQ²S in local mode with 1 thread is slower than the Optimised implementation as many times as indicated.

5.11 Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 8 threads. Where local[8]<Optimised Python indicates the PySpark DQ²S in local mode with 8 threads is slower than the Optimised implementation as many times as indicated.
5.12 Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 1 and 8 threads for 11 500 000 rows. Where the notation implementation \textless implementation indicates the left implementation is slower than the implementation on the right as many times as indicated.

B.1 Apache Spark processes’ default values for Standalone mode, where NA stands for “Not Applicable”.

B.2 Apache Spark processes’ general configuration parameters and location, where NA stands for “Not Applicable”.

C.1 Original Java DQ$^2$S execution times.

C.2 Python DQ$^2$S execution times.

C.3 Optimised Python DQ$^2$S execution times.

C.4 Optimised Python DQ$^2$S execution times in nanoseconds and seconds for 11 500 000 rows.

C.5 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 1 core.

C.6 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 2 cores each one.

C.7 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 3 cores each one.

C.8 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 4 cores each one.

C.9 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 2 workers and 1 core each one.

C.10 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 2 workers and 2 cores each one.

C.11 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 3 workers and 1 core each one.

C.12 Execution times obtained from PySpark DQ$^2$S Standalone mode in a single machine over 4 workers and 1 core each one.

C.13 Execution times obtained from PySpark DQ$^2$S Local mode using 1 core in a single machine.

C.14 Execution times obtained from PySpark DQ$^2$S Local mode using 2 cores in a single machine.
C.15 Execution times obtained from PySpark DQ²S Local mode using 3 cores in a single machine. ........................................... 216
C.16 Execution times obtained from PySpark DQ²S Local mode using 4 cores in a single machine. ........................................... 217
C.17 Execution times obtained from PySpark DQ²S Local mode using 5 cores in a single machine. ........................................... 218
C.18 Execution times obtained from PySpark DQ²S Local mode using 6 cores in a single machine. ........................................... 218
C.19 Execution times obtained from PySpark DQ²S Local mode using 7 cores in a single machine. ........................................... 219
C.20 Execution times obtained from PySpark DQ²S Local mode using 8 cores in a single machine. ........................................... 219
C.21 Execution times obtained from PySpark DQ²S Local mode using 1 to 6 cores in a single machine for 11 500 000 rows. ....... 220
C.22 Execution times obtained from PySpark DQ²S Local mode using 7 to 10 cores in a single machine for 11 500 000 rows. ....... 220
C.23 Execution times obtained from PySpark DQ²S Local mode using 1 to 6 cores in a single machine for 35 000 000 rows. ....... 221
C.24 Execution times obtained from PySpark DQ²S Local mode using 7 to 10, 12, and 16 cores in a single machine for 35 000 000 rows. ....... 221
C.25 Execution times obtained from PySpark DQ²S Standalone mode processing 35 000 000 rows with 1 to 4 worker nodes on a cluster. ....... 222
C.26 Execution times obtained from PySpark DQ²S Standalone mode processing 70 000 000 rows with 1 to 4 worker nodes on a cluster. ....... 222
C.27 Execution times obtained from PySpark DQ²S Standalone mode processing 105 000 000 rows with 1 to 4 worker nodes on a cluster. ....... 223
C.28 Optimised Python DQ²S execution times in nanoseconds and seconds for 35 000 000 rows. ........................................... 223
C.29 Performance Metrics for Parallel Algorithms: Values calculated for 11 500 000 rows processing from PySpark Local mode in single machine execution. ........................................... 225
C.30 Performance Metrics for Parallel Algorithms: Values calculated from PySpark Standalone mode in single machine execution. ........................................... 225
C.31 Performance Metrics for Parallel Algorithms: Values calculated from PySpark Local mode in single machine execution. ........................................... 226
List of Figures

1.1 The Design Science General Process ........................................ 22

2.1 The big data life cycle ................................................................. 27

3.1 Timeliness query expressed with DQ2L [26] .................................. 46
3.2 Timeliness query expressed with SQL [26] .................................... 46
3.3 OrderT dataset’s first 9 rows ......................................................... 46
3.4 StatusTimelinessQR dataset’s first 9 rows .................................... 46
3.5 General sequence diagram for the Timeliness query, build from the Java DQ2S instance ......................................................... 51
3.6 Class diagram for the Java DQ2S engine .......................................... 52

4.1 Flow structure of Chapter 4 .......................................................... 71

5.1 Runtime registered in seconds for the Original Java DQ2S processing 50 000, 100 000 and 1 000 000 rows .................................................. 101
5.2 Runtime registered in seconds for the Python DQ2S processing 50 000, 100 000 and 1 000 000 rows .................................................. 102
5.3 Runtime registered in seconds for the Optimised Python DQ2S processing 50 000, 100 000 and 1 000 000 rows .................................................. 103
5.4 Runtime registered in seconds for the PySpark DQ2S processing 50 000, 100 000 and 1 000 000 rows in Standalone mode implemented in a single machine, where \( W \) refers to workers and \( C \) to cores in the Apache Spark definition. The marked line indicates the lower runtime obtained per dataset size .................................................. 107
5.5 Runtime registered in seconds for the PySpark DQ2S processing 50 000, 100 000 and 1 000 000 rows in Local mode implemented in a single machine. The marked line indicates the lower runtime obtained per dataset size .................................................. 109
5.6 Runtime registered in seconds for the PySpark DQ²S processing 50 000 rows in Local mode implemented in a single machine.

5.7 Runtime registered in seconds for the PySpark DQ²S processing 100 000 rows in Local mode implemented in a single machine.

5.8 Runtime registered in seconds for the PySpark DQ²S processing 1 000 000 rows in Local mode implemented in a single machine.

5.9 Runtime registered in seconds for the PySpark DQ²S processing 11 500 000 and 35 000 000 rows in Local mode implemented in a single machine. The marked line indicates the lower runtime obtained per dataset size.

5.10 Runtime registered in seconds for the PySpark DQ²S processing 11 500 000 rows in Local mode implemented in a single machine.

5.11 Runtime registered in seconds for the PySpark DQ²S processing 35 000 000 rows in Local mode implemented in a single machine.

5.12 Runtime registered in seconds for the processing of 35 000 000, 70 000 000, and 105 000 000 rows with PySpark DQ²S in Standalone mode used in a cluster, utilising from 1 to 4 nodes (worker nodes). Each node was configured to contain 1 worker process with 8 executors, each executor with 2 cores and 32GB of available memory.

5.13 Comparison of the runtime in seconds obtained when processing 50 000 rows with the non-big data instances.

5.14 Comparison of the runtime in seconds obtained when processing 100 000 rows with the non-big data instances.

5.15 Comparison of the runtime in seconds obtained when processing 1 000 000 rows with the non-big data instances.

5.16 Comparison of the runtime in seconds obtained when processing 50 000, 100 000 and 1 000 000 rows with the best non-big data instance (sequential) and its implementation with a big data instance using 1 and 8 threads.

5.17 Comparison of the runtime in seconds obtained when processing 11 500 000 rows with the best non-big data instance and the PySpark one with 1 and 8 threads.

5.18 Ideal Speedup compared against the speedup results obtained from the PySpark DQ²S Local mode in a single machine.

5.19 Ideal Efficiency compared against the speedup results obtained from the PySpark DQ²S Local mode in a single machine.
5.20 Performance Metrics for Parallel Algorithms: Values calculated for 11,500,000 rows processing for the PySpark Local mode DQ^2S instance. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations.

5.21 Runtime in seconds for the PySpark DQ^2S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 50,000 rows.

5.22 Runtime in seconds for the PySpark DQ^2S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 100,000 rows.

5.23 Runtime in seconds for the PySpark DQ^2S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 1,000,000 rows.

5.24 Difference in seconds for 50,000, 100,000 and 1,000,000 rows when executing the PySpark DQ^2S with Standalone and Local mode, where Local mode was the quickest for all of the three dataset sizes.

5.25 Speedup and Efficiency individual heat map for the Standalone and Local mode PySpark DQ^2S. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations within each set of tables.

5.26 Speedup and Efficiency global heat map for the Standalone and Local mode PySpark DQ^2S. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations of all of the tables.

5.27 Runtimes obtained from the execution of the Timeliness query with 35,000,000 rows datasets, for PySpark in a single machine with Local mode and a Standalone mode in a cluster (DPSF).

5.28 Runtime in seconds for 35,000,000 rows processed in the DPSF cluster with the Optimised Python DQ^2S and the PySpark instance in Standalone mode.
| D.1 Proposed big data frameworks user classification with regards to experience and expertise. | 227 |
Abstract

EXPERIMENTING WITH A BIG DATA FRAMEWORK FOR SCALING A DATA QUALITY QUERY SYSTEM
Sonia Cisneros Cabrera
A thesis submitted to The University of Manchester
for the degree of Master of Philosophy, 2016

The work presented in this thesis comprises the design, implementation and evaluation of extensions made to the Data Quality Query System (DQ²S), a state-of-the-art data quality-aware query processing framework and query language, towards testing and improving its scalability when working with increasing amounts of data. The purpose of the evaluation is to assess to what extent a big data framework, such as Apache Spark, can offer significant gains in performance, including runtime, required amount of memory, processing capacity, and resource utilisation, when running over different environments. DQ²S enables assessing and improving data quality within information management by facilitating profiling of the data in use, and leading to the support of data cleansing tasks, which represent an important step in the big data life-cycle. Despite this, DQ²S, as the majority of data quality management systems, is not designed to process very large amounts of data. This research describes the journey of how data quality extensions from an earlier implementation that processed two datasets with 50 000 rows each one in 397 seconds, were designed, implemented and tested to achieve a big data solution capable of processing 105 000 000 rows in 145 seconds.

The research described in this thesis provides a detailed account of the experimental journey followed to extend DQ²S towards exploring the capabilities of a popular big data framework (Apache Spark), including the experiments used to measure the scalability and usefulness of the approach. The study also provides a roadmap for researchers interested in re-purposing and porting existing information management systems and tools to explore the capabilities provided by big data frameworks, particularly useful given that re-purposing and re-writing existing software to work with big data frameworks is a less costly and risky approach when compared to greenfield engineering of information management systems and tools.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in The University’s policy on presentation of Theses.
Dedication

To my husband, my parents, and my four grandparents.
Acknowledgements

“If I have seen further it is by standing on the shoulders of Giants.”
— Isaac Newton

I would like to thank my supervisors Sandra Sampaio, and Pedro Sampaio, whose support and guidance allowed me to pursue my own path, with the confidence of having wide and kind open doors whenever I faced doubts and confusion. I am deeply grateful to them for encouraging me to keep going, and to keep growing.

I am thankful to the Research Infrastructure Team within Research IT at The University of Manchester, for their tireless technical support towards the Data Processing Shared Facility cluster being able to work with Apache Spark, special thanks to George Leaver, Penny Richardson, and Anja Le Blanc.

I met wonderful people during my studies at the School of Computer Science, staff, fellow students, professors and graduates, who made me feel incredibly fortunate to coincide with them in this time and place since the first time I arrived to the University of Manchester. I am particularly grateful to Michele Filannino, and Oscar Flores-Vargas for showing me the passion and kindness of providing academic support and share their expertise to others, including me.

I also would like to thank Jessica Cueva, Atenea Manzano, and Estrella Armenta, for their support given during the process of my enrolment to the University of Manchester, a very important and crucial phase to make this work happen. Thanks for inspiring me to dream big.

I owe a lifetime debt of gratitude to Jesus Carmona-Sanchez, Delphine Dourdou, and Lily Carmona, for being the family Manchester gave me.

And last, but not least, I want to thank my sponsor, the National Council on Science and Technology of Mexico (CONACyT for its acronym in Spanish), for giving me the opportunity and funding to study a Master’s degree at the University of Manchester.
Chapter 1

Introduction

The work presented in this thesis aims to design, implement and evaluate extensions to the Data Quality Query System (DQ^2S) [26], a quality-aware query processing framework and query language, towards leveraging the Apache Spark big data framework. This study sets out to investigate the usefulness of it towards scalability for processing data quality techniques by converting a current successful algorithm into a big data capable one. The purpose of the evaluation is to assess to what extent a big data framework offers significant gains in performance (runtime, memory required, processing capacity, resource utilisation), maintaining the same levels of data quality that an algorithm executed without using the big data framework.

1.1 Motivation

Data appears to be one of the most valuable assets of organisations in the digital era [59, 91, 120], probably due to the power of the information provided by it, as Luciano Floridi and Phyllis Illiari mention, “The most developed post-industrial societies live by information, and Information and Communication Technologies (ICTs) keep them oxygenated” [42].

The importance of data could also be weighted by looking to the efforts done globally to protect data, to improve its management or to get a better understanding of it. An example of this can be found in the International Organization for Standardization (ISO) norm for Information Security Management (ISO/IEC 27001), which gives the guidelines and certifiable best practices regarding information, and therefore data, considering it as a valuable asset within organisations. Another example could be found in the recognised best practices in information technology management services, such as
1.1. MOTIVATION

The Information Technology Infrastructure Library (ITIL), which grew based on the claim that information is a pillar of business and not a separate part of it \[13\].

Nowadays, data is not only created by companies, but by individuals. For example, while walking on a street that has a sensor, sending out a text message, writing an email or even just turning on an internet connected television and watching a show; all of those actions generate data in one or another way. It is said that in 2013, more than 90% of the worldwide amount of data had appeared just in the biennium before \[22, 36, 60\] and it had kept on growing, making an estimate of “40 Zettabytes of data created by 2020” \[58\].

It is so easy to generate data, that storage and processing technologies are running out of capacity to cope with demand. The concept “big data” refers not only to large amounts of data in terms of storage size (Volume), but to its inherent characteristic of being created very fast (Velocity), its uncontrollable nature of heterogeneity (Variety) \[58\], its measurable degree of accuracy (Veracity), and its potential to unveil relevant insights (Value) \[58, 82\]. Thus, new technologies capable of handling big data are essential to the future of information technology.

One of the most important characteristics of big data is the value that could be retrieved from it, even into a level of obtaining information that was not known about its existence beforehand. For instance, the data gathered from an engine of commercial airplanes, with around 5 000 factors being monitored, can generate predictions on failures, or even make prognosis about turbulence based on forecast provided by the same data. In this sense, by analysing the information produced by sensors it is possible to generate savings of more than 30 million USD for airlines with this technology, by helping them to reduce costs and extend usage of their planes as a result of the insight obtained by the large amount of data that sensors provided\[9\], in this case, the collection of the big data from the airplanes’ engines leads to important information to be used in several contexts.

Back in 2009, a system built by Google, helped to solve health issues in the United States of America (USA) when H1N1 virus was being spread presumably as a pandemic infection, in which situation, to know about new cases as fast as possible was crucial to keep the spreading under control. Google made this by looking at people’s queries on the Internet, gathering more than a billion every day, and finding correlations between the frequency of certain search queries and the spread of flu over time and space. This system found combinations of 45 search terms among the 50 000 000 most popular searched ones, ending in a system that was able to predict a new case of
infection. That helped to slow the spread by enabling health authorities to take action in almost real-time. [83]. This is an example of the power of big data to provide society with useful insights.

A large volume of data creates the need for higher capacity storage machines and more powerful processors, as well as the concern about the quality of data in terms of completeness, consistency, uniqueness, timeliness, validity and accuracy (Veracity): the core data quality dimensions [7, 58]. Low data quality represents a problem for integration, visualisation, and analysis, which could be identified as a significant difficulty to utilise the data, and whether it is in small scale or big data, poor quality could lead to making severe and costly decisions [12].

It could be said that a large volume of data itself is not a warranty of better value than few data. The quality of data represents an important factor to be considered when making this assumption, for instance, cancer treatments include medication which has its proved effectiveness by the results of testing made on the participant patient’s DNA, so, when a doctor chooses a medication for a given patient, both have to hope that similarity between DNAs is close enough to those who participated in the drug trials [83], based on that premise, Steve Jobs decided to get a more personal treatment, by requesting his complete genetic code, not only a part of it [83] so the cancer medication could be the most precise therapy, aiming at extending his life for some years. This is an example in which, even though it was not a sample but the whole information, a single error would have had serious consequences: big data needs to be quality data.

Big data frameworks appeared as “the solution” to the big data problem, providing scalability, speedup and the possibility to compute big data despite its heterogeneity and high volume characteristics. Accompanied by its related storing technologies, big data frameworks have been considered as the most suitable option to compute large volumes of data, replacing Message Passing Interface (MPI) traditional technologies, parallel and distributed programming as former approaches. With the growth this frameworks have been presenting lately, it is of interest to analyse its application to data quality, and therefore explore if a big data framework can provide a solution for this area too.

### 1.2 Research Goals and Contributions

This research covers the design, implementation and evaluation of some of the algorithms included in the DQ²S towards increasing volume of data handled and speed
of processing using a big data framework, seeking performance improvements such as memory required, processing capacity and resource utilisation compared to its usage without the framework. The interest of the research relies on providing scalable, cost-effective, highly-automated, and optimised solutions.

The research requires redesigning the selected algorithms towards developing a version using big data technologies, so the following objectives can be met:

- Examine the potential gains in performance and scalability with the application of a big data framework.
- Explore the different parametrisations of the big data framework constructs to assess the impact on data quality algorithms.
- Propose changes to data quality methods and techniques based on the use of big data.
- Perform a literature review of the main big data frameworks and its environments, classified by purpose of usage.
- Conduct an exploratory empirical evaluation of the set of algorithms implemented using the selected big data framework, from a software engineering perspective.

This study can be used as a guide to extend data quality algorithms to increase performance and make them available to be used with larger datasets. The approach will also contribute to the literature by providing insights on key decisions to extend a sequential algorithm to benefit from parallelism using the Apache Spark Application Programming Interface (API).

1.3 Research Method

The research will be supported and developed based on the methodology known as “Design Science Research”, applicable for Information Systems Research. Considering that this study fits well into that category when recognising that, according to the United Kingdom Academy for Information Systems (UKAIS), the information processing supported by technologies is part of the definition of Information Systems (IS), having as study field the area related to those technologies that provide gains to the IS progress towards society and organisations [139]. Besides, this research is
CHAPTER 1. INTRODUCTION

Figure 1.1: The Design Science General Process

scoped as part of the domain of study of IS, under the “data, information and knowledge” domain, which comprises the “understanding and knowledge of techniques and technologies used to organise data and information and enable their effective use by individuals, groups and organisations” [139], which is what this research comprises as well, by understanding the techniques required to set a big data framework able to support the processing of a large data set using a data quality framework, DQ²-S in this case.

The “Design Science” research method, has its founding ideas in 1969 with Herbert Simon on his book, The Sciences of the Artificial [119] [56], and best described by Alan Hevner, whose contribution in Design Science Research is recognized by The American Association for the Advancement of Science (AAAS), the largest general scientific organisation [95] [43], and Samir Chatterjee, Professor and Fletcher Jones Chair of Technology Management at Claremont Graduate University’s School of Information Systems & Technology (SISAT) [140]; both founders of the Design Science Research in Information Systems and Technology Annual Conference (DESRITS), in 2006 [55], p.xiv].

Design Science offers an approach for researchers to improve IS by understanding its current state, in such a way that it leads to the design of a novel or improved condition, as represented in Figure 1.1 [55] p.x]. This dynamic adds new knowledge and increases the information available to the area in which the design was developed; the output of this process is called “artifact”, which is any tool that solves an important IS problem, giving open path to an efficient and effective management of it. [56]. An artifact can take the form of an algorithm, model, human/computer interface, language, system design methodology, etc. [55].
### 1.3. RESEARCH METHOD

#### Guideline Description

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Design as an Artifact</td>
<td>Design-science research in this study, leads to the creation of an artifact in the form of a set of algorithms implemented within the Apache Spark framework using the Python API, which comprise an extension to the DQ³S current algebra operation algorithms, described in Section 3.2.4.</td>
</tr>
<tr>
<td>2: Problem Relevance</td>
<td>This study comprises the development of technology-based solutions to the data quality requirement of big data, with its relevance mentioned in Section 1.1.</td>
</tr>
<tr>
<td>3: Design Evaluation</td>
<td>The evaluation phase supports the construction phase towards the quality of the design process of the artifact. Several evaluations were performed, according to the research method described in the present section.</td>
</tr>
<tr>
<td>4: Research Contributions</td>
<td>Contributions to knowledge will be added to the data quality area, as well as the big data IS, as presented in Section 1.2.</td>
</tr>
<tr>
<td>5: Research Rigor</td>
<td>The utility of the artifact to support the objectives of this research, as well as its efficacy handling big data, will be assessed according to the Evaluation Method, described in Chapter 4.</td>
</tr>
<tr>
<td>6: Design as a Search Process</td>
<td>As part of the iterative search process, as described in Chapter 4 and presented in Chapter 5, an optimised version was also studied and implemented.</td>
</tr>
<tr>
<td>7: Communication of Research</td>
<td>The research outcomes and results will be communicated to the community through the most suitable way according to the results, both to technology and management oriented audiences.</td>
</tr>
</tbody>
</table>

Table 1.1: The Design Science Research guidelines
CHAPTER 1. INTRODUCTION

The methodology proposes 7 guidelines that will be followed in the development of the research. A summary of the main concept of the guidelines applied to the study is shown in Table 1.1 according to the guides presented in [55] and described below.

In order to test if a big data framework offers significant gain in performance (run time, memory required, processing capacity, and resource utilisation) maintaining the same levels of data quality that an algorithm executed without it, and according to the Design Science Research Method, the DQ\textsuperscript{2}S will be extended to an implementation using Apache Spark technologies. This implementation will conform the IT artifact, created to address the need of organisations to handle big data and to have high quality data. These scalable set of algorithms will be described widely, including the constructs, models and methods applied in the development of them, enabling its implementation and application in an appropriate domain, considering that the IT artifact is the core subject matter [55].

A Design Evaluation plan was executed to demonstrate the utility, quality and efficacy of the scalable algorithms rigorously. This evaluation will be performed in phases, comprising Testing, Analytical, Experimental, and Descriptive Evaluation Methods [55].

The first evaluation will assure that the set-up of the original algorithms, configured in the machine available to develop the research is made correctly so it fulfils the expected results, according to its original model. This evaluation is expected to identify failures and misbehaviours, leading to its mending in order to have the proper output and unbiased performance measures. For the analytical evaluation, the structure of the algorithms will be examined, using pseudocode to describe them and later based on the research results, performance metrics for each algorithm will be produced. As part of the experimental evaluation, a case study within an e-Business context will be set in which the scalable algorithms will solve the case and point out its qualities in solving the case, which will be related to the usage and processing of a large data set. Finally, a descriptive evaluation will be performed, gathering the insights and analysis of the results obtained in the previous evaluations and linking them to the state-of-the-art and related work, with aim towards assessing the utility, quality and efficacy of the scalable algorithms when dealing with large datasets. In all this evaluation, it is important to keep on track with the Design Science Research, the main focus of which is to “determine how well an artifact works, not to theorize about or prove anything about why the artifact works” [55], and applied to this research means that it is required a deep understanding of why and how the algorithms works in the way they would do,
but it is not the main purpose of the research to prove why they work or not.

Part of the Design Science Research includes providing a contribution, developing the artifact using rigorous methods even in the construction of it, not only in the evaluation, by following the applicable standards to the development of algorithms; taking into consideration that this research will be an iterative search process leading to the discovery of an effective solution to the work of handling and processing of big data towards data quality.

1.4 Thesis Overview

The structure of the thesis comprises 6 chapters. In Chapter 1 the general topic and overview is introduced, followed by Chapter 2 which contains a literature review within the scope of big data and data quality, as well as the main frameworks used nowadays for big data processing. Chapter 3 includes the Design of the selected algorithms, with a wide explanation of the process of creation, technical characteristics and general behaviour of developed algorithms. Chapter 4 explains the tests made to the set of algorithms, with results are presented, and analysed in Chapter 5. Finally, Chapter 6 concludes the research and proposes future work. At the end of the document references and appendixes are shown.
Chapter 2

Background and Related Work

The purpose of this chapter is to review the literature on data quality and big data. It begins by a data cleaning, profiling and wrangling overview, then it goes to the related information and differentiation between parallel and distributed computation, and it finishes with a revision of the main big data frameworks, where Hadoop MapReduce and Apache Spark are the principal objects of study for this research.

Related work comprises those studies that worked with big data methods and techniques, implemented in data management and data quality, mainly using a big data framework which makes them relevant to this research, but with a different scope and purpose.

2.1 Data Quality: Wrangling and the Big Data Life Cycle

When talking about data quality in this document, it is done referring to the degree in which data fits to serve for its aimed purpose [113], for example, how well a medical record allows a nurse to identify the medicine that should be given to a patient, where “well” comprises, among general qualities, how accurate, complete, up to date, valid and consistent [7] is the information so that the task can be successfully achieved.

There are several processes required to assess and improve data quality, which include data extraction, data profiling, data cleansing and data integration [66], as the major ones; altogether in “the process by which the data required by an application is identified, extracted, cleaned and integrated, to yield a data set that is suitable for exploration and analysis” [105] is known as Data Wrangling. Figure 2.1 shows how each
process is placed within the big data life cycle \cite{2,66}, where data profiling is the step in which this research contributes to. Table 2.1 provides a definition of the processes mentioned above.

<table>
<thead>
<tr>
<th>Major Process</th>
<th>Process included</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA ACQUISITION</td>
<td>Data identification</td>
<td>Data has to be first generated from the real world, then converted to electrical signals so it can then be recorded in a machine. This process is called data acquisition \cite{2,84}. Identifying data means to provide useful metadata about its provenance, intended use, recording place and motivation, etc. \cite{23,90}.</td>
</tr>
</tbody>
</table>
# CHAPTER 2. BACKGROUND AND RELATED WORK

## DATA EXTRACTION AND CLEANSING

<table>
<thead>
<tr>
<th>Data extraction</th>
<th>This is the process in which data from source systems is selected and transformed into a suitable type of data according to its purpose, e.g. coordinates from a set of stored satellite images [2, 96].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data profiling</td>
<td>This refers to generating convenient metadata to support measurements against quality settings previously established, and to contribute towards “well known data”, clearing up the structure, content and/or relationships among the data. E.g. data types, data domains, timeliness, completeness, statistics, etc. [26, 71].</td>
</tr>
<tr>
<td>Data cleansing</td>
<td>Also known as cleaning or scrubbing. Requires solving errors found in invalid, inconsistent, incomplete or duplicated data so the quality of the data can be improved. To find the errors this process relies on profiling information [109, 26, 93].</td>
</tr>
</tbody>
</table>

## DATA INTEGRATION, AGGREGATION AND REPRESENTATION

| Data integration | Integrating data involves combining data from multiple sources into one, meaningful and valuable set of data [27, 53, 26]. |
Data storing

To preserve integrated data understandability, maintaining its quality, and adequacy to its purpose, it is also required to develop a suitable storage architecture design, taking into account the type of database suitability (e.g. relational, non-relational), the capacities of the database management system, etc., among all the alternatives in which data could be stored.

Data visualisation

Typically one step before analysis techniques. This process is about applying a graphical representation to the data, aimed at providing ease at future usage, transformation and understanding [40, 145, 15].

---

**MODELING AND ANALYSING**

<table>
<thead>
<tr>
<th>Statistics &amp; machine learning</th>
<th>This concerns about stating facts, in this context, from a given dataset, by interpreting data and providing a numerical picture of it, as well as using computer systems that emulate the human learning process, saving new information and outcomes, closely related to artificial intelligence (AI). Parallel statistics algorithms have been proposed to approach big data [103, 89, 15].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data mining</td>
<td>This process involves techniques to find latent valuable information, revealing patterns, cause-effect relations, implicit facts, etc., to hold up data analysis [117, 15].</td>
</tr>
</tbody>
</table>
Data analysis
A task done by the perceptual and cognitive system of an analyst, although nowadays machine learning and AI techniques can also be used. This is the ultimate phase in the process of obtaining the value from the data, by finding the significance on the insights provided by, for example, correlations or graphical representations. [145]

**INTERPRETATION**
Understand and verify results
This is the phase in which the information obtained is used and transformed into a decision that could lead to a tangible value (e.g. economic gaining, marketing advantages, scientific progress), differing each time according to the context on which it has been obtained and its purpose. This phase also comprises retracing the results obtained, verifying them and testing them in several use cases [2].

Table 2.1: Big data life cycle processes definitions.

In order to support the value characteristic of the data it is important to satisfy high quality conditions of the large data sets [12], where quality could be measured by its dimensions, having completeness, uniqueness, timeliness, validity, accuracy and consistency considered as the six core ones [7], among other identified dimensions, such as reputation, security, transactability, accessibility and interpretability [107, 86].

The processes, technologies and techniques aimed at obtaining the value from big data, are known as big data analytics [74], applied in several ambits, such as health care, social sciences, environmental and natural resources area, business and economic domains, and technology fields. Recently, big data quality has been approached from a big data analytics point of view [70, 54, 74, 75]. Some studies might conclude that data quality is not a bigger challenge than the lack of knowledge from analysts to implement
2.2. BIG DATA AND PARALLELISM

the correct methods to manage big data value[75], however, data management is an inherent phase of the big data analytics, and it involves the capacity to gather, integrate and analyse data as well, where data quality should not be considered as a separate phase.

The Data Warehousing Institute estimated low quality data cost U.S. businesses more than 600 billion USD per annum [38]. The U.S. Postal Service estimated that wrong data cost 1.5 billion USD in 2013 from mailpieces that could not been delivered to the given addresses, facing data quality problems from around 158 billion mailpieces in that single year. Data quality management strategies were recommended to increase address information accuracy, timeliness and completeness [51]. In big data, “low” error rates translate into millions of faults annually, where the risk is to lose 10-25% of the total revenues from it [38].

Big data quality requires multidisciplinary participation to progress [54] and propel the development of simple and low-cost data quality techniques, reduce the cost of poor quality data, the data error rate, and the need of data cleansing processes which involve investing not only budget, but time and effort to manage. IS research is demanded to collaborate with data wrangling insights and advances, working together with statistical experts to leverage the techniques involved, where domain specific authorities are needed to set the data analytics management, which should support the right value retrieval out of relevant problems from each area[54].

2.2 Big Data and Parallelism

The term big data itself comprises a deeper meaning, being not only a term to be defined but the name of a phenomenon and an emerging discipline [34]. The earliest mentions of big data were made, to the best of my knowledge, in 1979 by Lawrence Stone, when describing the work of cliometricians dealing with “vast quantities of data using electronic computers to process it and applying mathematical procedures” [121], and by Charles Tilly, who in 1980 describing the work done by the former, used the term “big-data people” referring to the cliometricians Stone mentioned before [138].

Then, in 1997 Michael Cox and David Ellsworth, described the term as large data sets that surpasses the available size in main memory, local disk and even remote disk [19]. Following that year, Sholom M. Weiss and Nitin Indurkhya in their book “Predictive data mining: a practical guide”, contextualize not only the term but the phenomenon, mentioning that “millions or even hundreds of millions of individual records
can lead to much stronger conclusions, making use of powerful methods to examine
data more comprehensively” and also acknowledges that analysing big data, in prac-
tice, has many difficulties [146]. Two years later, Francis X. Diebold defined big data
as “the explosion in the quantity of available and potentially relevant data, largely the
result of recent and unprecedented advancements in data recording and storage tech-
ology” [33]. By this time, it was clear that big data was not only about size, but about
the insights that a large set of data could eventually bring.

The Oxford English Dictionary, which added the term to its data base in 2013
[101], defines big data as “data of a very large size, typically to the extent that its
manipulation and management present significant logistical challenges” [102].

Nevertheless, those “logistical challenges” for one organization could be necessary
to be done when facing a smaller size of data compared to another [81], in this sense,
it seemed that relying only in the size depends on the available technology within
each organization and its capability to handle a given amount of data, so, to scope the
definition, the size of big data could be thought as the size in which using traditional
techniques to process it, is not longer an option.

The above led to define big data not only regarding size, but taking into account
another identified characteristics, known as the “V’s of big data” [58, 82]: Volume,
Velocity, Variety, Veracity and Value; where Volume refers to the data size, Velocity
evokes the high speed of change and fast generation of data, Variety relates to the
different type of data (structured, unstructured and semistructured), Veracity is the
degree of trustworthiness of the data (quality and accuracy) and Value indicates the
worth or benefit of gathering, storing, processing and analysing the data.

Because of the nature of big data, traditional approaches to managing data are not
suitable, for example, since the main aim was to handle relational data, and as previ-
ously mentioned, big data is not always relational, so, traditional techniques are not ex-
pected to work correctly [49]. When processing data, one of the main challenges faced
with big data is the large volume of it; this require scaling, and there have been two
types of scaling for big data processing: scaling-up and scaling-out, where the former
is about implementing powerful machines, with great memory and storage capacity,
as well as quick but expensive processors, and the latter refers to the usage of several
commodity machines connected as clusters, having a parallel environment, suitable to
handle large volumes of data, and an advantageous price-performance relation, com-
pared to the scaling-up approach [88, 49]. For big data frameworks, it is known that
with the proper optimisations, scaling-up performs better than scaling-out \cite{6}, however, it is still unknown exactly when is better to opt for one approach or the other, this is, the results presented were not dependable on the framework solely, but on the processed data characteristics. Nevertheless, it might be strongly preferred to utilise several smaller machines than high performance computer systems (HPC), because of the higher cost scaling-up represents, and considering the support that new paradigms provide by avoiding the necessity to communicate data, but having the processing applications running where the data is, which proposes a significant performance advantage \cite{128}, trading off performance in certain degree for a better cost-benefit ratio.

Distributed and parallel computation are two major ways of processing big data, where a distributed system communicates and coordinates their processes without a global clock, through messages, and in a component’s independence working as single system \cite{127}, and parallel computing is a term generally used to identify those processes carried out simultaneously by numerous processors, intended to reduce the runtime \cite{5}. Besides solely parallel, distributed or a mixture of both, new techniques are being developed and selected towards aiding the whole big data life cycle. Recent studies target “granular computing, cloud computing, biological computing systems and quantum computing” \cite{15} as the emergent areas, theories and techniques for big data.

2.3 Big Data Frameworks: Tools and Projects

A “big data tool”, also called “big data project’ could be conceptualised as a framework, where it implies structures, technologies, techniques, architectures, a programming model, and an environment upon which, in this case, big data processing applications can be built upon \cite{70}. There is another approach to a “big data framework” intended to propose also, a set of rules and standard concepts to provide big data a general concept, derived of the current lack of a certain and globally agreed one \cite{128,122,20}. In this research big data framework is referred using the former approach.
## Big data quality framework

<table>
<thead>
<tr>
<th>Big data quality framework</th>
<th>Description</th>
<th>Wrangling task</th>
<th>Open-source</th>
<th>Processing technology</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lavastorm</strong></td>
<td>Provides self-service data preparation with analytics functionality for data-centric applications [76].</td>
<td>Profiling, cleaning, integration, and representation.</td>
<td>No</td>
<td>Extract, transform and load processes (ETL)</td>
</tr>
<tr>
<td><strong>Datameer</strong></td>
<td>Offers both data preparation, and data analysis, with a library of over 270 functions for data preparation [24].</td>
<td>Profiling, cleaning, integration, and representation.</td>
<td>No</td>
<td>Hadoop</td>
</tr>
<tr>
<td><strong>Talend Data Quality</strong></td>
<td>Comprises profiling, cleansing, and masking of data functionality, as well as data quality monitoring over time for any format or size. [125]</td>
<td>Profiling and cleaning.</td>
<td>No</td>
<td>Hadoop</td>
</tr>
<tr>
<td><strong>Open Studio for Data Quality + Data Preparation</strong></td>
<td>The open-source solution from Talend that profiles and monitors, and a separate data preparation auto-discovery, smart suggestions, and data visualization functionality for cleansing. [126]</td>
<td>Profiling and cleaning.</td>
<td>Yes</td>
<td>Hadoop</td>
</tr>
</tbody>
</table>

Table 2.2: Big data frameworks available for data quality tasks

Several big data frameworks are based on a programming model called “MapReduce”, designed to process and generate large datasets, and parallelise them automatically, based on two functions: *map*, which works with a key/value pair and transforms it into “intermediate” pairs, and *reduce*, aimed at merging the intermediate values by
its key [1]. This is the base of the current scale-out approach, being MapReduce the model that allows the process of big data in commodity hardware [128], and altogether with open-source developments, has made the big data frameworks become widely used among industry and research [128].

Since the release of MapReduce, there have been several projects based on the programming model which have been developed, where Hadoop is the name of one of the most popular ones. An exhaustive list of all the “big data ecosystem projects” can be found in an on-going repository available at [92], which includes the names and a brief description of both open-source and proprietary projects. A review on the most popular big data frameworks is provided by [49] and [15].

Regarding data wrangling, there are some big data tools available, focused on visualisation and analytics tasks, most of them created towards powering business intelligence and being proprietary solutions, however, these frameworks require data to be already pre-processed. On the other hand, there are few frameworks available for tackling data profiling and cleansing tasks; these are described in Table 2.2. The reason of the apparent lack of frameworks designed specifically for big data quality purposes can be answered by the availability of all the other frameworks, designed as general purpose, upon which any kind of application, included those for data wrangling, can be built and designed, thus, there is an open path to the development of domain specific systems, either based on current frameworks, which could be improved to tackle any specific requirement within the data quality area, or a new framework packed with default functions and libraries for big data wrangling. Any selected path requires identification of the wrangling algorithms required, as well as knowledge on how to scale them to cope with big data.

### 2.3.1 MapReduce, Hadoop and Spark

MapReduce was presented in 2004 by Google [1], it conforms a paradigm in which commodity hardware capacities are leveraged in a hugely distributed architecture [128]. One of the attractive components of it was that by using this programming model, developers did not need to be experts on parallel and distributed systems to exploit the power of the applications coded using MapReduce style. This is because the presented runtime system solved the details of data partitioning, execution scheduling, failures and communication across machines, required to utilise resources available in large distributed systems for big data processing. MapReduce was also presented as a highly scalable in a scale-out approach, claiming to process data of the order of terabytes (TB)
Communication cost, as mentioned before, is an important issue when handling big data; MapReduce model attempts to utilise an optimum network bandwidth by storing data on the local disks of the machines that conform the cluster, dividing the data into 64MB chunks each, and distributing typically 3 copies of them on different machines across the cluster, therefore the cluster’s master schedules map tasks closer as possible to the data that the map function requires \[1\], or the closest copy of it.

Apache Spark could be simply defined as a “big data Framework”, but its official definition mentions that it is “a fast and general engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing” \[46\], claimed to be 100 times faster than the “open-source software for reliable, scalable, distributed computing” \[44\], Hadoop, another available and widely used big data Framework that works with the MapReduce programming paradigm. Presumably, Spark achieves better performance by introducing a “distributed memory abstraction” \[151\]: the resilient distributed datasets (RDDs) that can be used for “lazy evaluations”, caching intermediate results in memory across iterative computations \[152,151\]. An RDD contains five main elements, pieces of a dataset, called partitions; information about the parent RDDs or any dependence with another RDD; information on the computation needed to create the current RDD, based on its parent partitions; information about the most accessible nodes for a given partition of data; and metadata regarding the RDD scheme \[151\].

Known as the “Hadoop ecosystem”, there exists several related projects, aimed to be used for big data in different and specific purposes, examples of this platforms include Ambari (web based tool for provisioning, managing, and monitoring Apache Hadoop clusters), Avro (data serialization system), Cassandra (resilient scalable multi-master database), Chukwa (data collection system for monitoring large distributed systems), HBase (scalable, distributed database), Hive (data summarisation and ad-hoc querying capabilities), Mahout (scalable machine learning and data mining library), Pig (powerful analysis capabilities with PigLatin, a high-level language), Tez (dataflow programming framework) and Zookeeper (high-performance coordination service for distributed applications) \[44\].

One of the aims of Apache Spark is to have all the possible needs when working with big data, covered with a single solution, so, to achieve this, it allows the use of libraries (SQL and dataFrames, Spark Streaming, MLlib, GraphX, and Third-Party
2.4. **SCALABILITY OF DATA MANAGEMENT OPERATORS**

Packages) and high-level language APIs (Python, Scala, Java and R), adds the capability to run on Hadoop, Mesos, Standalone, or in the cloud and it is able to access distinct data sources including HDFS, Cassandra, HBase, S3, Alluxio (known as Tachyon), and any Hadoop data source [46].

2.4 Scalability of Data Management Operators

Previous and related work to this study is scoped to research in which scalability is the key aspect, from the one addressed using any technique, to extensions of data quality algorithms, techniques used specifically with any big data framework, and the related work that is closer to the contribution of this thesis: scalability implemented with Apache Spark.

Kalé et. al. [69] presented the “NAMD2 program”, a high-performance parallel simulator of large biomolecular systems behaviour, implemented using C++ and the machine independent parallel programming system coined as Charm++, with an interoperability framework for parallel processing called Converse. For this program the parallelisation strategy was used to support scalability and a load balancing scheme to increase performance [69]; this project is still on going as part of the Theoretical and Computational Biophysics research group from the University of Illinois at Urbana-Champaign, currently scaling beyond 500 000 cores for the largest simulations, still based on Charm++. Another publication, related to NAMD, recognises the difficulty of coding parallel programs with their used strategy and mentions that the main usage of NAMD is done in “large parallel computers” [106]. These studies then, offer the opportunity to develop a high-performance program that could be both developed with an easier programming strategy and focused on utilisation in commodity hardware, without decreasing its maximum processing capacity.

Before MapReduce, Message Passing Interface (MPI) solutions were the preferred approach to scalability, as well as parallel and distributed programming and scaling-up techniques. By the time in which Hadoop emerged, some studies within different areas began to explore its usage towards amplifying the capacity of handling large volumes of data, this is the case of a study in which genetic algorithms (GA) were implemented utilising Hadoop and compared against its MPI instances, considering number of variables as volume for the input dataset [143]. Hadoop was then presented as a solution to the issue of executing genetic algorithms in commodity hardware without failure issues, contrary to what happened with the MPI approach.
Related to data quality, two studies carried-out experimentations that explored Hadoop’s application for data cleansing processes, the first one, denominated “Big data pre-processing data quality (BDPQ) framework” [124], was presented as a set of modules to support profiling, selection and adaptation of big datasets. The data cleansing module was used to remove noise from electroencephalography (EEG) recordings. The second research mentioned, presented a method for storing and querying medical Resource Description Framework (RDF) datasets using Hadoop to assess for accuracy and validity in medical records [10], removing high costs of data upload, and improving the algorithm’s performance compared to its previous version developed using a Jena Java API for RDF approach. This study claims to be the first attempt ever done with data quality techniques for linked datasets and shows how “utilising data analytic and graph theory techniques can inform a highly optimal query solution” [10]. Future work related to this study includes testing of the approach on datasets over one billion triples. Based on the information provided, it is implied that scaling data quality algorithms is possible using a big data framework, but more work is required to investigate its full capability with larger datasets and different types of data.

Another algorithm, called PISTON [110], introduces a parallel in-memory spatio-temporal join implemented in a scalable algorithm [110] using load-balancing in a master/slave model, designed to out perform against existing algorithms with the best known performances, where PISTON algorithm addresses recognised characteristics of long execution time and poor scalability. The model implemented might behave similar to the Hadoop-base processing one, however, details on the processing architecture are not provided, limited only to inform details related to the algorithm that presents improvements in runtime and capacity to execute joins on spatio-temporal topological data, achieved due to its inherent behaviour, not because of the infrastructure used, the environment, or the technologies implemented. Masayo Ota et. al. [99] presented in the same year another study of a taxi ride-sharing scalable properties, capable of handling more than 200 million taxi trips information and calculate the optimum cost-benefit path, this algorithm was tested using Hadoop. However, similarly to PISTON algorithm’s study, the scalability was presented as obtained mainly because of the algorithm structure, not the framework tool.

Sheikh Ikhlaq and Bright Keswani [61] proposed Cloud computing as an alternative to technologies like Hadoop, but acknowledges that new methods of handling heterogeneous data without security issues and a good access to information on the Cloud are still missing [61]. Even though Cloud computing could bring an option to
2.4. SCALABILITY OF DATA MANAGEMENT OPERATORS

avoid investing on local resources, this study does not explain clearly why it considers big data frameworks as “expensive” to use, when the latter are designed to work not necessarily with super computers and are part of the available open source platforms.

Another Hadoop’s aided research is presented in the “Highly Scalable Parallel Algorithm for Maximally Informative k-Itemset Mining” \cite{115} (PHIKS) study, introduced as an algorithm that provides a set of optimisations for itemset mining, tackling execution time, communication cost and energy consumption as performance metrics. This novel algorithm was implemented on top of Hadoop using Java as programming language and compared against state-of-the-art itemset mining algorithms on its parallel form. PHIKS was tested on three different datasets, of 34GB from Amazon Reviews, 49GB from the English Wikipedia articles dataset, and the ClueWeb English articles with 1TB size. In this study, comparisons were made between different algorithms and its approaches to the same problem, in all cases using Hadoop as processing engine, however, there is no information regarding the benefits of the framework used, or the parallel techniques implemented compared to a traditional algorithm.

Apache Spark has also been used to support scalability of different algorithms, such as ”Insparq”, an API layer for efficient incremental computation over large data sets designed and implemented on top of Spark using intermediate values for each computation step within an execution, and scoped to be used in analysis of energy consumption domain with energy data from smart electricity meters \cite{16}, where an improvement to the current Spark engine processing was adapted to handle efficiently that specific kind of data. Another example is shown by Saba Sehrish et. al. \cite{116}, who executed an evaluation of Spark’s performance \cite{116} and presented within the High Energy Physics (HEP) area with a classification problem algorithm; compared against an MPI parallel programming instance, the Spark implementation turned out to have lower runtime, mainly because of the level of abstraction that Spark provides to the user regarding certain, and critical decisions, such as task assignment, or distribution and data location, but Spark still performed better when scaling. This study also provides an example on how specific domain testing is required, by pointing out that, the lack of a “high performance linear algebra library” contributed to the final runtime shown by the Spark instance, therefore, those algorithms requiring that kind of modules, are expected to look for a more convenient option than Apache Spark.

Regarding the big data frameworks, studies have been made to analyse the performance of Hadoop and Spark. A scaling-behaviour analysis when using Hadoop and the MapReduce model \cite{154} was presented identifying three types of applications as
map-, shuffle- or reduce-intensive, based on the most costly operation required by each kind of algorithm. The mentioned study provides and analysis on scalability expectations according to the volume of the input dataset, and shows that due to the nature of the algorithm, other than relying on the framework or the input size, a linear-scaling is not always present in Hadoop. Other research in which comparisons between Spark and Hadoop implementations have been evaluated include a “Max-min” scalable ant system algorithm (MMAS) [144], designed to solve the travelling salesman problem within optimisation path algorithms [144], proposing that MMAS could be used to extent neural network, logistic regression or particle swarm algorithms [144], based on its results where Spark supported good scalability. Data mining has also been an area in which related studies have been developed, showing that dense graph problems can be benefited by the technique utilised in a single-linkage hierarchical clustering algorithm using Spark (SHAS) [67], which was evaluated in comparison to an earlier instance of the same algorithm in its MapReduce based form [68] with better results shown in the latest assessment. K-means is also a data mining algorithm evaluated [52] against its equivalent using Hadoop, showing better execution time when processed with Apache Spark, and concluding that Spark will become widely used for big data processing tasks, though big data community generally agrees on that Apache Spark is not going to replace Hadoop, but work together towards merging benefits from both frameworks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type of algorithm</th>
<th>Technology used to scale</th>
<th>Size of the biggest dataset</th>
<th>Compared against</th>
</tr>
</thead>
<tbody>
<tr>
<td>[143] The ONEMAX GA (2009)</td>
<td>Domain Specific: Genetic algorithms</td>
<td>Hadoop</td>
<td>100 000 variables</td>
<td>MPI implementation</td>
</tr>
<tr>
<td>[124] BDPQ (2015)</td>
<td>Data quality</td>
<td>Hadoop</td>
<td>EEG recordings from 22 subjects</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
Table 2.3: Summary of the most relevant state-of-the-art related studies.

Different ways of leveraging scalability could be found in this literature review, where two identified approaches are shown: scalability based on algorithm design, and scalability based on platform utilised. The first approach comprises design decisions, such as heuristics implementation, architectural design (shared-nothing, shared-memory, multi-threaded), complexity, persistence, or general optimisations, whereas platform scalability comprises choices on the environment utilised regarding the selected framework, the programming language and even the cluster size and machine characteristics, as well as scaling type (out or up). This research has as main scope platform scalability.

Table 2.3 shows a summary of the related studies discussed in this chapter, and reflects that data quality domain is still under development towards scalability, which represents an open area for research where the work presented in this thesis intends to participate on by expanding data quality query systems. The aim is to explore a platform approach that could provide those systems the capability to process big data. Chapter 3 presents the designing work carried out towards the mentioned aim, where a state-of-the-art system’s engine was re-implemented to leverage its big data capabilities, preserving the characteristics of its algorithms.
Chapter 3

Designing a Scalable DQ\(^2\)S

3.1 The Data Quality Query System (DQ\(^2\)S)

The Data Quality Query System (DQ\(^2\)S) is a framework and tool that allows measuring some of the quality dimensions (completeness, timeliness, accuracy, and reputation) from a given data by profiling it \[26\]. This system reads data from a storage location, and performs database algebra operations to profile the quality of the data according to the queries applied, allowing users to get the data quality information in the same way they query if it was done using SQL commands. The DQ\(^2\)S is in fact an interface to storage and an extension of the SQL query engine that can be implemented to utilise either a relational database or a storage file system. DQ\(^2\)S provides functionality without requiring users to be aware of the stored quality-related information or the data profiling algorithms implemented at the physical level. This framework can be used with traditional data management techniques, however, the DQ\(^2\)S original implementation does not exploit parallelism and cannot cope with very large datasets \[26\].

This research covers the design, implementation and evaluation of some of the algorithms included in the DQ\(^2\)S towards increasing the data volume limit that can be handled by the DQ\(^2\)S, considering speed of processing, scalability and resource requirements. For the mentioned purpose, a big data framework was selected in order to explore the features that could be obtained for DQ\(^2\)S, and seeking for performance improvements such as memory required, processing capacity and resource utilisation compared to its usage without the framework. The focus of this research is targeted at providing scalable, cost-effective, highly-automated, and optimised solutions.
3.1. **THE DATA QUALITY QUERY SYSTEM (DQ^2S)**

### 3.1.1 The Framework

The DQ^2S is a useful contribution to the data management area, where information systems, analytical tasks based on data, and more recently, big data and business intelligence require data quality awareness, as discussed in Section 1.1. DQ^2S is a novel contribution to the data quality area, as exposed on its original publication at [26], where the related work discussion reflects DQ^2S is flexible, by allowing users to set the configuration required but without being too broad to be highly complex [26], supporting optimisation and enabling ease of use when operated by users already familiar with SQL.

DQ^2S appears as a complementary functionality to the data quality process within data management by facilitating a quality-profile of the data in use, leading to support the cleansing tasks within the data wrangling effort, as well as handling information to allow users to know their data, which is designed as one of the challenges when considering big data quality needs, and because a data profile provides a step towards a high level of confidence of the data’s fit to conduct analytical tasks [26], all of the above makes DQ^2S a viable tool within information systems aimed at data quality, however, the system requires further research and experimentation to develop a DQ^2S capable of handling efficiently larger volumes of data. Such experimentation could provide a baseline towards scaling other data quality management approaches.

The basic DQ^2S work [26] shows experimental results of the queries with two data quality dimensions tested, timeliness and completeness, where the quickest runtime obtained on an average of three runs, was around 15 minutes for a dataset of 100,000 rows, and the slowest time was an average of 76 minutes for the same dataset. An important note on the mentioned results is that two versions of the queries were executed, where one was an optimised version with an implementation of heuristics for query executions in relational databases, which reduced 29% the execution time for the Timeliness query, and 67% for the completeness query; this reduction was achieved due to a decrease on the intermediate resources size, hence, the processing of the datasets was less intense and the memory requirement was smaller, thus, paging that happened in the process was also reduced [26], this incurred on diminishing I/O costs, nevertheless, even with the optimised versions, 15 minutes were required to handle 100,000 rows, and an increasing of 70.6 times the runtime obtained from 10,000 to 100,000 rows on the best of the cases (Timeliness query with optimisation) suggest the original DQ^2S implementation is not adequate to handle larger datasets. The limitation appears on
the memory requirements, where the queries were optimised on its process, but memory handling and I/O processes are still the same for both versions (with and without heuristics), and that is not inherent to the algorithm but to the architectural environment in which the processing is carried out, this appears as an opportunity for big data frameworks, specialised on big data processing optimisation with regards to architecture models that allow machines to handle larger data than its actual memory space, discussion on big data framework approaches and details is presented in Section 2.3.

This research aims at exploring a popular and rising big data framework based on the MapReduce paradigm, towards an analysis of its suitability to extend the DQ²S for big data. Apache Spark supports in-memory processing, which reduces I/O costs, and provides a data structure that facilitates usage of intermediate results in memory, offering options to tune the data partitioning, and supporting a rich set of operations to manipulate the data, as well as resilient behaviour [152].

3.1.2 The Query

DQ²S supports the implementation of four quality dimensions applied on relational models. The dimensions include accuracy, completeness, timeliness, and reputation, which are applicable in several domains concerned to e-Business, e-Science and Information Systems, specially when dealing with geographical data [26]. Within an e-Business context, completeness and timeliness were the dimensions utilised on the evaluation of DQ²S, since the queries implemented covered several important use cases in the domain [26]. The Timeliness query is an appropriate dimension to target for improvement to the objectives of this study, since the results of the DQ²S work [26] indicated that Timeliness queries produced the slower runtimes, and furthermore, the Timeliness query includes only one profiling operator and one join, simplifying the tasks of isolating the properties of the algorithm to enable experimentation with big data frameworks, whereas the completeness algorithm is composed of two profiling operators and two joins. A single factor analysis provides accuracy on quantifying the results, and the higher runtime offers a wider opportunity for optimisation, considering that, if a big data framework can provide support to the most expensive query, it might easily support less intensive queries.

Timeliness is part of the Contextual Data Quality (CDQ) category [155], where the degree of quality depends on the context of the data’s purpose, for example, some data might be timely enough to support data driven decisions utilising weekly reports on the data, where its timeliness quality would be lesser if pretended to be used for the
3.1. THE DATA QUALITY QUERY SYSTEM (DQ2S)

Following weekly report, however, if the same data is taken on a semester based task, its timeliness degree posses a different value. Timeliness could be defined as the extent to which the data is not outdated to be useless for its intended use.

The timeliness data quality dimension, was modelled as a query for DQ2S implementing formulas to calculate the timeliness degree [26], based on timeliness calculation defined for information products [8], but applicable within a general scope for IS, as follows:

\[
\text{Timeliness}(v,s) = \max \left[ (1 - \frac{\text{Currency}(v)}{\text{Volatility}(v)}), 0 \right] \tag{3.1}
\]

Where \( v \) represents a unit of data, and \( s \) represents a control value to the sensitivity of the currency-volatility ratio, its increase or decrease will be done depending on how much impact the volatility produces, for example, if volatility is high, and thus within the given context, the timeliness is highly affected, \( s \) should increase from 0.5 (almost no impact) to a value of 2. Both \( v \) and \( s \) are given to the calculation as input, whereas currency and volatility are computed with the correspondent formulas shown in equations 3.2 and 3.3. Currency is the age of the data given by the time data was kept stored and the age it had when it was first saved, and volatility shows a measure of time upon which data is not outdated.

\[
\text{Currency}(v) = \text{DeliveryTime}(v) - \text{LastUpdateTime}(v) + \text{Age}(v) \tag{3.2}
\]

\[
\text{Volatility}(v) = \text{ExpiryTime}(v) - \text{LastUpdateTime}(v) + \text{Age}(v) \tag{3.3}
\]

Timeliness calculation is given on a 0 - 1 scale, where the scale can be seen as 0 to 100% of timeliness degree. The Timeliness query specified for the study requires to select, from an e-Business dataset, the orders that are pending and have been waiting to be validated for more than 50% of the total waiting time. Figure 3.1 shows the query expressed using the query language defined for DQ2S (called DQ2L), and Figure 3.2 shows the same query expressed with SQL, which show one of the useful aspects of DQ2S when querying on a SQL style, as well as the insight of the general algorithm process required by the query; both queries retrieve the same output. The output to the Timeliness query needs to show the order_no and the timeliness score of those pending orders that turn out to have a timeliness score lower than 0.5.
3.1.3 The Datasets

<table>
<thead>
<tr>
<th>order_no</th>
<th>customer_id</th>
<th>product_id</th>
<th>quantity</th>
<th>submit_date</th>
<th>ship_date</th>
<th>statusOrder</th>
<th>statusTimeliness_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50001</td>
<td>50002</td>
<td>50000</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>50002</td>
<td>50003</td>
<td>50001</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>3</td>
<td>50004</td>
<td>50005</td>
<td>50002</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>4</td>
<td>50003</td>
<td>50006</td>
<td>50003</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>5</td>
<td>50005</td>
<td>50007</td>
<td>50004</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>6</td>
<td>50006</td>
<td>50008</td>
<td>50005</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>7</td>
<td>50007</td>
<td>50009</td>
<td>50006</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
<tr>
<td>8</td>
<td>50008</td>
<td>50010</td>
<td>50007</td>
<td>2013-09-07</td>
<td>00:00:00</td>
<td>2013-09-15</td>
<td>00:00:00</td>
</tr>
</tbody>
</table>

The e-Business data utilised for querying with DQ^2S has five relations, each one with different information and scheme. The datasets are:

- **orderT**: Contains information about the orders made to the e-Business.
- **part**: Is the inventory of the parts the e-Business sells.
3.1. THE DATA QUALITY QUERY SYSTEM ($DQ^2S$)

- $part_supply$: Stores the inventory of the part’s suppliers.

- $part_priceTimelinessQR$: Is a relation storing quality information, called a “Quality relation”; contains timeliness information related to the part’s prices.

- $statusTimelinessQR$: A quality relation with timeliness information related to the orders.

The datasets required depend on the algorithm and the data quality model. For the Timeliness query, the input datasets are $orderT$ and $statusTimelinessQR$, both of the datasets are in a CSV format. Figure 3.3 shows a fragment of the rows in $orderT$, where:

- order_no starts in 50001
- customer_id starts in 5002
- product_id is exactly the same as customer_id
- quantity starts in 50000
- submit date is always 2013-09-07 00:00:00
- ship date is always 2013-09-15 00:00:00
- statusTimeliness_id is identical to order_no
- statusOrder contains an alternating value of pending and progressing which means there are 50% of the registers with each status

A fragment of the rows in $statusTimelinessQR$ are shown in Figure 3.4, where:

- statusTimeliness_qid starts in 50001, and is the foreign key value to statusTimeliness_id in $orderT$ dataset
- lastUpdateTime is always 2013-01-07 00:00:00
- expiryTime is always 2013-06-15 00:00:00
- deliveryTime is always 2013-12-08 00:00:00
- age is always 0
### 3.1.4 The Algorithms Included in the DQ²S Database Algebra Implementation

The algorithm that comprises the Timeliness query is the base upon which all of the DQ²S instances are created, originally modelled utilising the Java programming language, which receives the name of the Java DQ²S for the purposes of this study, and represents the existent artifact to be optimised and analysed. Table 3.1 shows a description of the operators involved in the processing of the Timeliness query, supported by Figure 3.5 showing a sequential diagram on the Timeliness execution with the Java operators, which are further represented as Java classes in Figure 3.6, both diagrams model the functionality of the Timeliness query implemented with DQ²S. The general sequence of the algorithm can be described as shown in Algorithm 1 below, representing the required Timeliness query process where each action is quoted to a class that performs the corresponding action.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimelinessQuery</td>
<td>Contains the main structure of the algorithm, utilises the query engine operators to produce the query result.</td>
<td>Names of the required datasets to complete the query and the predicates needed.</td>
<td>ID and timeliness score of the tuples with timeliness score below 0.5.</td>
</tr>
<tr>
<td>Timing</td>
<td>Gets the system time, allowing the measurement of the elapsed time by saving the system time at the beginning and at the end of the query processing.</td>
<td>None.</td>
<td>Elapsed runtime in nanoseconds.</td>
</tr>
<tr>
<td>ScanSelect</td>
<td>Loads the dataset from a CSV file and fills three array lists from Tuple Class with the appropriate information given by the dataset values.</td>
<td>Comma separated CSV dataset.</td>
<td>A tuple with three dynamic generic arrays containing the values, data types and name of the attributes from a line in the loaded CSV.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Tuple</td>
<td>Creates a data structure containing the data ingested by ScanSelect and schema information (data types and attribute names).</td>
<td>Two dynamic generic arrays, one containing the data types names and the second array formed by the attribute names. A data structure formed by three dynamic generic arrays (data values, data types, and attribute names).</td>
<td></td>
</tr>
<tr>
<td>Predicate</td>
<td>Evaluates a given predicate and returns a boolean value depending on its compliment or not, according to the operator and the evaluated values within it.</td>
<td>A predicate made of two operands and one comparison operator. A boolean value telling if the predicate was true or false after applied on a given data.</td>
<td></td>
</tr>
<tr>
<td>Join</td>
<td>Joins two dynamic generic arrays based on a given join predicate.</td>
<td>Two dynamic generic arrays and a join predicate specifying the joining statement to determine which rows need to be joined. A single dynamic generic array with joined data.</td>
<td></td>
</tr>
<tr>
<td>JoinPredicate</td>
<td>Is a support algorithm to the Join operator. Evaluates the predicate against the dynamic generic array and retrieves a Boolean value on the predicate correspondence with the data.</td>
<td>Join predicate. A boolean value telling if the join predicate was true or false after applied on a given data.</td>
<td></td>
</tr>
<tr>
<td>Timeliness</td>
<td>Calculates the timeliness score based on the formulas presented in Section 3.1.2 of this chapter.</td>
<td>A dynamic generic array containing the data required to calculate the score. The dynamic generic array utilised as input plus a new column with the correspondent calculated timeliness score.</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Input</td>
<td>Output</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>Select</td>
<td>Filters a dynamic generic array and creates a new one based on the result obtained from the Predicate class.</td>
<td>Filter predicate and a dynamic generic array to apply the filter on.</td>
<td>A dynamic generic array containing the filtered data.</td>
</tr>
<tr>
<td>Project</td>
<td>Extracts columns from a dynamic generic array based on a list of attributes given (name of the columns).</td>
<td>List of attributes and a dynamic generic array.</td>
<td>A dynamic generic array containing the data that corresponds to the required columns.</td>
</tr>
</tbody>
</table>

Table 3.1: Operators comprising the DQ²S Timeliness query.
3.1. THE DATA QUALITY QUERY SYSTEM (DQ²S)

Figure 3.5: General sequence diagram for the Timeliness query, build from the Java DQ²S instance.
 Figure 3.6: Class diagram for the Java DQ²S engine.
Algorithm 1. Part 1 - TimelinessQuery Class.

1: function MAIN( )
2:    Get the current system time ▷ Timing class
3:      Start Create first array ▷ Selects all the orders from orderT dataset with “pending” status
4:      and saves them in a dynamic array
5:      Get the name of the input dataset ← “orderT”
6:      Form the predicate to extract orders with “pending” as status ▷ Predicate class
7:      From the dataset with the given name, save the data and its schema on a
dynamic generic array applying the given predicate selection
8:      ▷ ScanSelect class
9:      End
10:     Start Create second array ▷ Selects all the registers from “statusTimelinessQR” dataset and
11:      saves them in a dynamic array
12:     Get the name of the input dataset ← “statusTimelinessQR”
13:     Create a null predicate ▷ Predicate class
14:     From the dataset with the given name, save the data and its schema on a
dynamic generic array applying the predicate selection
15:     ▷ ScanSelect class
16:     ▷ Tuple class
17:     End
18:    Start Create a Join ▷ Joins the information in the first and the second array on a given join
19:     predicate as the joining point, and saves the content in a new array
20:    Create a join predicate to join the arrays on “orderT.statusTimeliness_id” and
21:     “statusTimelinessQR.statusTimeliness_qid” ▷ JoinPredicate class
22:    Specify which two arrays should be joined
23:    Join two given arrays on the given predicate and generate a new array with the
24:    joined data ▷ Join class
25:    End
Algorithm 1. Part 2 - TimelinessQuery Class.

Start Create a Timeliness

▷ Calculates the timeliness score with the information in the joined array. Copies the content of the joined array plus a new attribute with the corresponding scores.

12: Set the name for the column to add ← “timeliness”
13: Specify the array to utilise for the Timeliness score calculation
14: Calculate the Timeliness score from the information within the given array, and add the score to a column with the given name ▷ Timeliness class

End

Start Create a Select

▷ Selects the information of the tuples which timeliness score is < 0.5 from the latest array created

15: Form the predicate to extract orders with timeliness score below 0.5 ▷ Predicate class
16: Generate a new array applying the given predicate to the latest array formed ▷ Select class

End

Start Create a Project

▷ Extracts the column with the Id and the column with the timeliness score from the latest array, and saves only the content of those attributes as the final result of the query.

17: Get the attributes name to extract from the latest array created ← “statusTimelinessQR.statusTimeliness_qid”, “timeliness.score”
18: Specify the array to utilise and the attributes to get ▷ Project class
19: Create the final array only with the columns required ▷ Project class

End

20: Print Final array
21: Get the current system time ▷ Timing class
22: Calculate the elapsed time since the beginning of the processing ▷ Timing class
23: Print Elapsed time calculated

24: end function
3.2 The Artifact

The artifact of this study is composed by the DQ$^2$S instances, which will comprise an added artifact to the existent one, the Java version. Java is a general purpose programming language, which provides an opportunity to explore DQ$^2$S performance and scalability with big data programming languages, presumably optimised to cope with large volume of data, hence, the algorithm was developed utilising different technologies, including a big data framework to provide answers to the research questions presented in Section 4.1 and support to the hypothesis validation presented in the same section. The source code of the DQ$^2$S developed instances, and the original Java code is shown in Appendix A; this section presents details on the instances development and internal composition.

3.2.1 The Java DQ$^2$S

Table 3.2 shows details on the technical aspects of components that comprise the Java DQ$^2$S, as well as information on key pieces of code utilised to develop the instance, which are important for the understanding of the technical behaviour, as this is the base for extending the algorithm to be developed for its usage with other big data technologies. Some operators did not require any special or outstanding codification than common and basic pieces of code (e.g. if statements or method calls).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Technical Description</th>
<th>Key piece of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimelinessQuery</td>
<td>Contains the main method. Creates the required objects utilising the classes involved in the timeliness query, and calls for the execution of the methods from the external classes through the created objects.</td>
<td>None.</td>
</tr>
</tbody>
</table>
### Timing

Contains three methods: `StartTime`, `stopTime`, and `durationTime`. `StartTime` gets the system time of the moment in which it is called, and saves it in a variable; the second method has the same functionality as `StartTime`, but is required to store the system time in a second moment; the third time calculates the arithmetic difference between the two variables created, and prints the value in nanoseconds. This is the elapsed time from the processing that happened between the first system time and the second.

System.nanoTime() is used to get the current value of the system timer in nanoseconds.

### ScanSelect

Gets as parameter the name of the dataset to load, loads it with a buffered reader, which allows a line per line reading. Line per line is saved into a dynamic generic array created under the custom `Tuple` data structure.

BufferedReader `rs = new BufferedReader(new FileReader(csvFile))` to read the CSV file from storage to the buffer, then `rs.readLine()` is used to read line by line from the correspondent buffer containing the data.

### Tuple

Creates a data type containing three array lists: one for the data values, filled in the `ScanSelect` class; one for the data types; and a third one for the attribute’s names (columns), which are filled within this class with a for statement.

ArrayList<Object>() to save the data, and `Add()` to add values to the array within the for statement.

### Predicate

Gets a predicate formed by one first operand, an operator, and a second operand, then evaluates both operands with the operator according to the operator’s data type, and returns true or false as the evaluation result indicates.

None.
### 3.2. THE ARTIFACT

| **Join** | Copies the data from two Tuple objects into a new one, integrating the data from both input objects. Once the new Tuple object exists, an array list is generated with the data from that new Tuple. These data is only the content that matches the given join predicate, this is, only the rows that get a true value from JoinPredicate. | Utilises `addAll()` inside a for statement to copy all the data from the Tuple objects indicated, and `add()` to copy only the rows with a true boolean value obtained from the JoinPredicate class. |
| **JoinPredicate** | Gets the position in which required operands are in a Tuple, to obtain the correspondent data type, then according to the data type, the class evaluates if the operands conform a true or false comparison with the given operator. Returns the evaluation result in boolean. | None. |
| **Timeliness** | From the input Tuple, extracts the quality information required to calculate the currency, volatility and timeliness. With the data extracted calculates the timeliness score based on the formulas 3.1, 3.2, and 3.3 presented in this chapter, then adds a new column to the original Tuple, with the timeliness score calculated for each row. | Utilises `addAll()` inside a for statement to copy all the data from the input Tuple in a new one, then adds to the new Tuple the timeliness score per row with `add()`. `get_Time()` was used to get the time in miliseconds in unix time, aslo known as epoch time. |
| **Select** | Utilises a Predicate object from the Predicate class, and a Tuple object. Whenever the Predicate class returns a true value, this class adds the evaluated Tuple to a new one. | Utilises `add()` inside a for statement to copy into a new Tuple the correspondant rows, as many times as true values are obtained from the Predicate class. |
CHAPTER 3. DESIGNING A SCALABLE DQ²S

Project Gets a list of column names and an array list with data values, then copies the data from the columns indicated into a new array list, which is later converted to Tuple object. Utilises add() inside a for statement to copy into a new Tuple the rows from the given columns.

Table 3.2: Java DQ²S technical description

3.2.2 The Python DQ²S Implementation

Scala, R, and Python are considered programming languages suitable for big data analytics. These contain classes specially designed to support data analysis tasks, such as profiling and mining large datasets [31, 87], there are other programming languages of this kind, such as Julia or Go, but those are less popular than Scala, R or Python. [31, 14]. For scientific applications, and use within enterprises and organisations, Python is on the top 10 most popular worldwide, whereas Scala is known better for web and mobile development, and still less popular than Python in those situations [32]. R is gaining popularity, getting closer to Python and gradually replacing C# for general usage [14], but around 458,000 public repositories for Python [31], makes it a clear winner in the data science landscape. Due to its growing popularity, Python was selected as the approach to test scalability and performance for DQ²S instead of other “big data languages”. Table 3.3 presents the main differences implemented from re-engineering the Java DQ²S into a Python instance, where the third columns shows pieces of code that are significantly different to the same functionality utilised in Java. The general development was done by utilising the equivalent Python code, which most of the time was only a syntax change (to Python proper syntax).
### 3.2. THE ARTIFACT

<table>
<thead>
<tr>
<th>Operator</th>
<th>Main differences with Java instance</th>
<th>Key piece of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimelinessQuery</td>
<td>Contains the main method that covers the same functionality as the Java DQ²S. The only difference is that the final printing requires a time consuming for statement to print line by line, however, the experiments for this study does not calculate the time considering the final printings too, but only the general processing time of the query.</td>
<td>None.</td>
</tr>
<tr>
<td>Timing</td>
<td>Utilises a different approach to get the system time in nanoseconds.</td>
<td>int(round(time.time() * 1e9)) is used to get the current value of the system timer in nanoseconds.</td>
</tr>
<tr>
<td>ScanSelect</td>
<td>To load the CSV content, a Python CSV reader was utilised, and the equivalent method to copy rows into an array.</td>
<td>import csv was used to utilise csv.reader(csvFile), a reader for tabular data in CSV format, and array.append() was used as equivalent to the Java method add().</td>
</tr>
<tr>
<td>Tuple, Select &amp; Project</td>
<td>The equivalent method to copy rows into an array.</td>
<td>array.append() was used as equivalent to the Java method add().</td>
</tr>
<tr>
<td>Predicate &amp; JoinPredicate</td>
<td>None.</td>
<td>None.</td>
</tr>
<tr>
<td>Join</td>
<td>The equivalent method to copy rows into an array.</td>
<td>array.append() was used as equivalent to the Java method add(), and array.extend() was used as equivalent to the Java method addAll(). The high level difference between append and extend is that extend can add more than one item each time.</td>
</tr>
</tbody>
</table>
CHAPTER 3. DESIGNING A SCALABLE DQ²S

Timeliness
Required a custom function to convert the Timestamp values into miliseconds epoch time.

utcfromtimestamp(), strftime(), and total seconds() * 1000.0 was utilised instead of the Java method get_Time().

Table 3.3: Python DQ²S technical description

3.2.3 The Optimised Python DQ²S Implementation

Python Pandas library (from panel data, “a common term for multidimensional data sets encountered in statistics and econometrics” [87]) appeared as a solution to Python lack of statistics and data analysis capabilities, where R, MatLab, Stata and SAS were dominating the area [87]. Python Pandas as a scientific library provide data structures and methods with focus on structured data utilisation, with optimised and robust solutions to manipulate those kind of data, which is a strong quality with high expectation for the DQ²S algorithm. Pandas provides a data structure called “dataFrames”, inspired by R [87], which supports SQL like manipulation. The Optimised Python DQ²S is called “Optimised” in relation to the Pandas library which is the main component utilised for this DQ²S instance. Table 3.4 presents a description of the changes required for the re-engineering of DQ²S to its version with an optimised library, where even the lines of code were significantly reduced due to the concise functionality Pandas library provide.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Main differences with Python instance</th>
<th>Key piece of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimelinessQuery</td>
<td>When utilising the Pandas library and dataFrames to store the data instead of array lists, the final printing can be done with a Pandas defined method to print dataFrames content. The rest of the codification is exactly the same within the TimelinessQuery class.</td>
<td>dataFrame.head(n). to_string() is utilised to print the content of the dataFrame containing the results, where n is the number of rows to print. This method is equivalent to printing the final results from the correspondent Tuple object in the Java and Python instances.</td>
</tr>
<tr>
<td>Timing</td>
<td>None.</td>
<td>None.</td>
</tr>
<tr>
<td>ScanSelect &amp; Tuple</td>
<td>A dataframe is a tabular data structure already containing the attributes implemented in the Java and Python versions with the Tuple class, hence, for this instance there is no Tuple class, and ScanSelect is the class responsible of providing the dataframe the information of the contents, the data type names and the attribute names. To load the CSV content into a dataframe, a Pandas CSV function is implemented instead of <code>import csv</code> Python module.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Predicate</td>
<td>The Optimised Python instance does more than returning a boolean value, the class implemented with the scientific library applies the given predicate over the given dataframe and returns a new dataframe with only the content to which the predicate applies as true. Utilises <code>dataframe.query('String predicate')</code> where the predicate is a string formed by two operands and one operator.</td>
<td></td>
</tr>
<tr>
<td>Join &amp; Join-Predicate</td>
<td>Pandas library has methods already defined to allow the join of two dataframes, which were utilised instead of an append of rows. <code>dataframe.merge</code> and <code>dataframe.join</code> were utilised instead of append, extend and JoinPredicate class. Merge method requires the join predicate as parameter, hence the JoinPredicate class is not longer required. Join is a quicker approach for joining on index, utilised within the Join class for the cases where a predicate was not given.</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 3. DESIGNING A SCALABLE DQ\textsuperscript{2}S

<table>
<thead>
<tr>
<th>Timeliness</th>
<th>The custom function to convert the timestamp values into milliseconds epoch time is different. With dataFrames, for statements are not required because the functions are applied by default to all of the rows inside the dataFrame.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\texttt{datetime.astype(np.int64) / 1e6).astype(np.uint64}) was utilised inside the function that converts from datetime objects to milliseconds epoch time.</td>
</tr>
</tbody>
</table>

| Select | In the Optimised Python, \texttt{Select} class does not realise any copy, but serves as a communication interface to the \texttt{Predicate} class, which returns the required filtered dataFrame, instead of returning a boolean value to build the array in the \texttt{Select} class. |
|        | The only code present is a parameter passing to the \texttt{Predicate} class, and a return sentence which retrieves the filtered dataFrame; the copy of the elements is carried out in the \texttt{Predicate} class. |

| Project | When utilising dataFrames, there is a straightforward way to select and extract only some columns from the dataFrame, instead of copying them explicitly. |
|         | \texttt{dataFrame[attrList]} was used to create a new dataFrame with only the column names given in the \texttt{attrList} variable. |

Table 3.4: Optimised Python DQ\textsuperscript{2}S technical description

3.2.4 The PySpark DQ\textsuperscript{2}S Implementation

The Apache Spark Generals

The Apache Spark project is part of the Apache Software Foundation(R) (ASF), which supports open-source and collaborative products among a community of users and developers all around the world [48]. Apache projects are maintained by this community, and empowered then by organisations that provide solutions based ASF’s open-source projects.

Being Spark an open-source software, supported by the ASF community, it is not surprising that there exists several and diverse documentation of the framework, such as the own Spark’s website, the Spark’s Mailing list, Spark’s wiki, information provided from Universities, line courses, professional blogs, online forums, Spark summit events repositories, and even youtube technical videos plus Spark discussion groups available on LinkedIn and Facebook, and of course published research papers and
3.2. THE ARTIFACT

books. Having all of those sources available, this study does not intend to be part of another source of Spark’s information, however, it might be valuable to clarify some basic concepts that were found rather ambiguous among all of the diversity of sources.

Spark itself is written on Scala language, which runs on the Java Virtual Machine (JVM) [18]. All of Spark’s general functionality relies on the usage of the Java Virtual Machine, taking advantage of the JVM’s memory management and garbage collection (GC) [97] that this platform-independent environment provides. The usage of JVM also preserves the possibility for Spark’s integration with other projects of the Hadoop ecosystem (refer to Section 2.3.1 for details of Hadoop ecosystem), which are JVM based. Each JVM has its own memory allocation, its own GC process, and its own threads that access its own memory area. Spark’s architecture comprises basically five concepts upon which all of its computations are developed in a cluster, described below. Table B.1 in Appendix B shows the configuration defaults of each of the concepts described, and table B.2 in the same appendix shows the parameters that can be used to modify the default values within a Spark Standalone cluster.

1. **Driver.**

   **Driver process.** Also known as the driver program, this process runs on its own JVM with 1 core and 1GB by default. The following are the activities that the driver is in charge of performing:

   - Process the main() function of the application.
   - Create the SparkContext object, which tells the driver how and where to access the master node.
   - Connect and communicate with the cluster manager process to know the cluster resource’s availability.
   - Split an application into tasks.
   - Schedule the tasks on the available executors.
   - Send the application code to the executors.

   **Driver node.** This is the node on which the driver process is running. It could be a node outside the Spark cluster (client mode), or a node inside the cluster (cluster mode).

2. **Cluster Manager.**


**Master process.** The cluster manager can also be called the master process. This process schedules the allocation of Workers and its executors within the cluster. The cluster manager can take the form of Spark’s Standalone (Spark’s own resource manager), YARN (Hadoop’s resource manager) or Mesos (a general resource and scheduling manager for clusters).

The Standalone cluster manager performs the following activities:

- Allocates resources to the worker nodes.
- Utilises the worker to create executors for the driver.
- Communicates its workers and executors availability to the driver process once the driver connects to it.

YARN and Mesos cluster managers in general perform the same activities, however, could be more involved than the Standalone manager when scheduling tasks.

**Master node.** The master node is the machine within the cluster in which the master process is running.

3. **Worker.**

**Worker process.** Also known as the Slaves process, this is also created on its own JVM. A worker is in charge of reporting to the master process its resource availability, so the master can deliver that information to the driver.

There are two important things to note about the configuration of the Workers:

- When setting number of Workers, memory per worker and cores per worker, the cluster will allocate the specified per node on the cluster, not per the whole cluster.
- The cores and memory allocated to each worker, is only to indicate the workers the resources each worker can give out to its executors, not the actual resources that the worker is going to use. The worker by default runs using 1G and 1 core, which is enough for the majority of Spark applications because of the activities it performs.

**Worker node.** A worker node is a machine within the cluster that contains one or more worker processes.
4. **Executor.** An executor is the JVM process on a worker node that processes the tasks that constitute the application. There are some noticeable aspects of an executor:

- By default one executor is created per worker process, taking all the available cores that the worker has to give out and 1GB of memory, however this setting can be changed to either have more than one executor per worker process and/or allocate more memory for the executor’s JVM.
- The executor process is the one that loads into its memory the data from storage, processes the computations and produces the results.
- Each executor loads into memory only the required data for its allocated task.
- An executor process will not start if there is not enough resources allocated by the worker, for example, if it is required 2 executors, each one with 10GB of memory but the worker has allocated 12GB, only 1 worker will start.
- Each executor computes tasks in multiple threads.

5. **Cores.** Spark cores in reality are not CPU cores but threads; when number of cores is set, this number means how many tasks should be handled simultaneously by the executor. This is the reason of why setting a larger number for the “cores” than the available CPU cores of the machine, will not trigger any error when starting a Spark cluster and submitting an application. The best practice, however, is to adhere its usage to a value equal to the number of CPU cores on the worker node.

Having more threads processed simultaneously could cause an overhead because of scheduling and communication time, resulting in more time spent on overhead computation than in the actual processing of the tasks.

For this study, from now on, when referring to cores it is done to the definition described above: **the number of tasks handled at the same time by an executor.** When referring to CPU cores, it will be done explicitly.

Spark’s elements relate to each during the processing of an application, following Spark’s cluster general operation:
1. The cluster manager (also known as the master process) has to be up before any application is submitted.

2. Once submitted and the SparkContext created, the driver program starts and asks the master for resources.

3. The worker communicates its resource availability to the master.

4. The master uses the worker to create executors. It decides where to launch them according to the resources availability reported by the workers.

5. The driver allocates tasks to the executors to process.

6. The executors can communicate only between executors of the same application.

7. Once all the tasks have been finished, results are sent back to the driver.

8. The driver then stops the SparkContext, all executor processes also stop and the cluster’s resources are released. The workers and master processes are not stopped when the driver stops.

Apache Spark as a framework for big data processing provides the following benefits [46, 151, 152]:

- Provides in-memory processing by loading data from storage to its JVM executor’s memory, allowing these to work with the data without investing too much time on storage I/O.

- Its data structures, Resilient Distributed Dataset (RDD) and dataFrames, partitions the data so it can be processed in parallel, adds information that enables to recompute a partition in case of failure, and provides Spark the ability to process large volumes of data by allowing the executors to load only the partition that it requires according to the task that was given to that executor.

- Provides fault tolerance and “speculative” processing by relocating tasks when a failure occurs or when the processing is being slow.

- Supports high level programming, with Python, Scala, Java and R Application Programming Interfaces (APIs), which gives flexibility and ease of use for a wider range of users without trading much on processing speed inherent to the language. All of the different languages run at JVM speed on Spark. Spark
3.2. THE ARTIFACT

implements sockets that allow each language to call Java Methods. A wider description on how the API implemented on this study (Python API - PySpark) is used internally with Spark can be found in the next subsection of this chapter.

- Offers built-in libraries that gives Spark its multi-purpose characteristic: SQL with a Query Optimiser, Machine Learning libraries (MLib), Streaming support and Graph computation. All of them are used on top of the Spark Core Engine.

- Supports integration with all of the major Hadoop projects, included Hadoop’s distributed file-system and Hadoop Resource Manager, which supports Spark to complement itself with other’s programs capacities and allows Spark to be an option for users that are familiar with Hadoop and its related projects (refer to Section 2.3.1).

Spark Python API

The applications developed with the Python API are called “PySpark” applications. Python API was built on top of the Java API which is on top of the Spark Core Engine written in Scala, this structure allows Python to be utilised with a Spark that internally works with JVMs. When a PySpark application is submitted, the process is the same as described earlier in this section, but there some specific details that are present with the Python API. The process of Apache Spark with Python API starts on the driver node, then PySpark utilises py4j socket to enable Python programs running in the python interpreter to utilise Java methods as if there were the same programming language. This is how from the Python programme, a Spark context JVM is launched. Then, the executor process launches a Python process (daemon.py) with as many child processes as cores were set in the Spark parameters. Next, there is a custom pipe written by the Spark engineers, capable of moving high volumes of data at high velocity between the daemon.py and the executor’s JVM. The data is loaded to the daemon.py from the JVM, but functions are serialised in the called “pickles objects” (serialised data structures of size up to 1MB each “pickle”) from the driver process, and sent to the daemon.py process, this last location is where the functions are “unpickled” (unserialised). Each child process carries out the function to its given data, however Spark’s library functions (MLib, SQL, etc.), dataFrames transformations, and shuffles are processed by the executor, thus, giving native JVM speed, not Python speed, it is nevertheless, important to note that Python processes are part of the executor in terms
of functionality, hence when referring to executors for a PySpark process it is not necessary to mention on a separate base the Python existent daemon.py processes. Finally, if data is retrieved to the driver, the data will move from the daemon.py to the executor first, then to the driver, and instead of going through the py4j, which is not designed for heavy data movement, the data goes to local storage on the driver machine, and then loaded up to the Python initial process (the process which starts the driver JVM through py4j) [39].

The PySpark DQ$^2$S utilises the SQL library, which has support to dataFrames structures. Spark SQL provides relational data support for both RDD, dataFrames and dataSets structures, where dataSets for Spark 1.6.2 are only available for Java and Scala API. DataSets provide the same functionality as dataFrames, but also additional functionalities, such as the ability to work with unstructured data. Because of the similarity of Spark dataFrames to the structure utilised to develop the Optimised Python, and because dataSets is not available for PySpark, the former was utilised for the big data framework DQ$^2$S instance. Table 3.5 presents the key differences between the code of the Optimised Python and the PySpark one.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Main differences with Optimised Python instance</th>
<th>Key piece of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimelinessQuery</td>
<td>PySpark requires many more Spark classes to be imported, and special objects to set-up the Spark communication within the elements that conform the framework. In this instance the final printing is also different and mandatory, where <code>show()</code> conforms the action required due to the lazy evaluation Spark model, and also the less time consuming for the algorithm.</td>
<td><code>dataFrame.show(n)</code> is utilised to print n rows from the final dataFrame. A key piece of code for this main class is the Spark initialisation, done with a <code>SparkContext</code> object and an <code>SQLContext</code> object to utilise the Spark SQL module.</td>
</tr>
<tr>
<td>Timing &amp; Select</td>
<td>None.</td>
<td>None.</td>
</tr>
</tbody>
</table>
### 3.2. THE ARTIFACT

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScanSelect &amp; Tuple</td>
<td>An external package was utilised to load a CSV into a DataFrame for PySpark programming, this package utilises a method which requires from the user to indicate the data type and the attributes name, similar to the approach utilised with the Tuple java class, but being DataFrames, the data structure is already designed by the package.</td>
<td>com.databricks.spark.csv is the package utilised instead of pandas.read_csv</td>
</tr>
<tr>
<td>Predicate</td>
<td>No significant differences beside that instead of utilising object as data type, PySpark assigns String data type to all of the data loaded from the CSV. The data type was overwritten in the ScanSelect class only for those elements with type different to String.</td>
<td>None.</td>
</tr>
<tr>
<td>Join &amp; Join-Predicate</td>
<td>Merge method is replaced by join method for PySpark since this has the equivalent functionality.</td>
<td>DataFrame.join was utilised for all cases in which a join or merge was required.</td>
</tr>
<tr>
<td>Timeliness</td>
<td>The custom function to convert the Timestamp values into milliseconds epoch time is different, and the final assignment of the timeliness score is made with PySpark when clause implementation.</td>
<td>SparkContext._active_spark_context and sc._jvm.functions.unix_timestamp() were utilised inside the function that converts from datetime objects to milliseconds epoch time, and withColumn(columnName, columnContent) is utilised to add a new column with the timeliness score to the existent DataFrame.</td>
</tr>
</tbody>
</table>
Project | The method utilised to extract some columns requires a sorting, otherwise the new DataFrame with the required columns is presented unordered (e.g. ID 976 will be shown first instead of ID 501).
---|---
dataFrame.select(attrList).sort(attrList[0]) was used to create a new DataFrame with only the column names given in the attrList variable and sort the elements based on the column names.

| Table 3.5: PySpark DQ²S technical description. |

### 3.3 Summary

This chapter presented insights on the artifact of this study, the DQ²S, providing details about the DQ²S as framework, the algorithm utilised and its internal composition, as well as a description of the datasets required and the desired output. This chapter also provides technical information regarding the developed DQ²S instances, seeking to gain scalability and performance improvement, having as baseline the Java DQ²S program in the original DQ²S engine. The information provided in this chapter serves as foundation for the rest of the thesis, where Chapter 4 presents how the DQ²S instances presented above were tested and compared to explore the scalability and performance capabilities of each one towards a high throughput considering different environments, also described in Chapter 4. Chapter 5 presents the results obtained from the implementation and testing of the DQ²S instances described in this chapter.
Chapter 4

Experiment design

This chapter describes and discusses the research method used to accomplish the research goals and to obtain and compare the performance results produced by the algorithms required to execute the Timeliness query, described in Section 3.1.2. The objective is to test, compare and evaluate the performance of utilising Apache Spark, towards verifying its suitability to extend the DQ^2S core engine for big data processing, and deriving insights from the experiments.

The chapter includes on its first section an overview of the objectives and scope of the experiments designed to support the research goals. To ensure the research’s reproducibility, the second section details the materials required in the experimentation, followed by the implementation details presenting information about the procedure utilised to execute the evaluation. The last part of the chapter describes the analysis and evaluation approach, which explains the designed result comparisons and presents...
the performance metrics applied, as well as the validity evaluation plan. Figure 4.1 shows a general view of this chapter’s organisation.

The principles followed in developing the different versions (instances) of DQ²S ensure a fair comparison, these comprise not re-engineering the algorithm, but utilising the equivalent syntax on each programming language that supports the same functionality, ensuring a zero difference in the input and output obtained on all the instances, and not comparing algorithms with substantial known advantages (e.g. inherent structure, default degree of parallelism) for other than only reference. Section 4.4 of this chapter presents the terms under which each DQ²S instance was implemented and analysed, and Section 4.5 discusses how the different implementations were paired to be compared, considering fair results in order to obtain useful insights from the approaches used towards the best throughput obtained, as discussed in the same section.

4.1 Objectives and Scope of the Experiment

This experiment design aims at fulfilling two main objectives: to support the research goals mentioned in Section 1.2 and to contribute to the DQ²S future work about further analysis towards the DQ²S big data processing capabilities, stated on the publication which presented the system [26]. The DQ²S future work intended to extend the system’s scalability, for which it was considered an evaluation of multiple-pass approaches, cloud computing and MapReduce processing [26]. This research presents an utilisation of the DQ²S algorithms as an “additional implementation of the data profiling algorithms” [26], and towards analysing the DQ²S scalability, this study implements a DQ²S instance for the Timeliness query under a big data framework strategy based on the MapReduce paradigm.

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Research Technique</th>
<th>Description / Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>Experiments E1 - E10 and Literature Review</td>
<td>The experiments provide support to explore the advantages of a big data framework compared with non-big data solutions by creating a baseline of the latter as well as analysing its different scalability and performance behaviours, both on a single multicore machine and on a cluster.</td>
</tr>
</tbody>
</table>
4.1. OBJECTIVES AND SCOPE OF THE EXPERIMENT

Literature provides the knowledge of known and recognised advantages inherent to big data framework’s architecture, design, and processing technique.

RQ2  Experiments: E5- E7  Experiments E5 - E7 are the ones aimed at providing results on multicore and multinode (cluster) environments.

RQ3  Algorithm design (Chapter 3)  Challenges of the development of the different DQ2S instances are presented when describing its design and construction.

Table 4.1: Research Question’s solution plan

The hypothesis of this research is that a big data framework can be used to improve performance and scalability of a data quality query system, compared to non-big data solutions. Besides verifying the hypothesis, this empirical study is intended to provide answers to the research questions below, and based on the results obtained from the experimentation, the research aims at providing evidence to support the suggestion of data quality approaches when handling big data, as discussed in Section 1.1. Table 4.1 shows how the research questions were structured and planned to be solved.

1. What are the performance and scalability advantages that big data frameworks provide to data quality management tasks?

2. What are the performance and scalability differences when using a single multicore node versus cluster computing environments for small, medium and large workloads?

3. What are the challenges of re-engineering the implementation of a non-big data quality query system to work on top of big data frameworks and platforms?

A theoretical complexity analysis of each algorithm implemented is out of the scope of this research since the work presented aims to improve on existing algorithms based on the parallel and distributed capabilities of a big data framework, however, the experimentation includes an evaluation as part of the experiment validation, and measures the results of the big data framework experiments with the performance metrics presented in Section 4.6 of this chapter.
CHAPTER 4. EXPERIMENT DESIGN

4.2 Data and Tools

The query executed in this experimentation (Timeliness query) requires two different datasets, named “Order” and “Status Timeliness QR”, both datasets are in CSV comma separated format and comprise the input data to the query. A description of its content and structure can be found in Section 3.1.3 Three different sizes of datasets were utilised as baseline: 50 000, 100 000, and 1 000 000 rows. The size in bytes of each dataset used is shown in table 4.2:

<table>
<thead>
<tr>
<th>Number of rows</th>
<th>Name of the datasets</th>
<th>Order</th>
<th>Status Timeliness QR</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 000</td>
<td></td>
<td>4.0MB</td>
<td>3.4MB</td>
</tr>
<tr>
<td>100 000</td>
<td></td>
<td>8.0 MB</td>
<td>6.8 MB</td>
</tr>
<tr>
<td>1 000 000</td>
<td></td>
<td>84.5 MB</td>
<td>68.9 MB</td>
</tr>
</tbody>
</table>

Table 4.2: Relation of number of rows and size per each input dataset utilised in the experimentation.

The experiments designed for this study were performed in two different environments. The first one is a commodity machine (desktop PC), and the second one is a 36 node cluster with a shared storage architecture. The cluster utilised in this study is the Data Processing Shared Facility at the University of Manchester (DPSF), from which this study utilises 5 of its nodes as to deploy a Spark Cluster. The DPSF cluster utilises the Sun Grid Engine (SGE) to coordinate and schedule the execution of tasks submitted by the users through batch scripting, aimed at managing the resources and processes in a fair and efficient way. The cluster is based on a Lustre file system, a scalable and optimised software commonly used for High Performance Computing (HPC), known for its usage on supercomputers around the world, with “hundreds of gigabytes per second (GB/sec) of I/O throughput” [98]. Lustre enables access to distributed data as if it was local, even when in reality, the storage disks are placed physically in further locations, and allows the operation of different processes at the same time on shared data without interference, avoiding resource contention, and providing coherency on the data if modified simultaneously; Lustre achieves concurrency support and cache coherence through its distributed lock manager (LDLM) [98]. The hardware specification for the desktop PC and the DPSF cluster is shown in Table 4.3 and a summary of the tools required for the experiments is presented in Table 4.4.
4.2. DATA AND TOOLS

Despite the desktop machine being a dual boot one, the experiments were performed on the Linux OS, a Fedora 20, with a HDD partition of 500GB. The tools and technologies selected are the common ones used for data quality and data wrangling, as well as the most popular, stable and compatible with older and newer versions at this time.

<table>
<thead>
<tr>
<th>PC &amp; DPSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of nodes</td>
</tr>
<tr>
<td>Operative System (OS)</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>No. of physical CPU cores per node</td>
</tr>
<tr>
<td>Total available CPU cores per node</td>
</tr>
<tr>
<td>RAM per node</td>
</tr>
<tr>
<td>Total Storage</td>
</tr>
</tbody>
</table>

Table 4.3: Hardware specification of the desktop machine (PC) and the cluster (DPSF) utilised in the experimentation.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Where applicable</th>
<th>Usage within the research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datasets based on the business processes of a e-Business seller company, the same data used for the testing the original DQ^2S.</td>
<td>PC &amp; DPSF</td>
<td>These are the input data to the algorithms required to execute the Timeliness query.</td>
</tr>
<tr>
<td>The original Java code from the DQ^2S, provided by the corresponding authors [26].</td>
<td>PC</td>
<td>Conforms the baseline of the performance metrics comparison.</td>
</tr>
<tr>
<td>The Java 1.7.0_71 version.</td>
<td>PC</td>
<td>Used to execute the original set of DQ^2S algorithms.</td>
</tr>
</tbody>
</table>
The last major version in the 2.x series: Python 2.7.12 PC & DPSF

The Pandas library version 0.17.1 PC & DPSF

The version 1.6.2 of the Apache Spark Framework, released on June 25, 2016, with the pre-built package for Hadoop 2.6 and later. PC & DPSF

<table>
<thead>
<tr>
<th>Tools</th>
<th>Used to execute the Python DQ$^2S$ and the Optimised Python DQ$^2S$ version of the original set of algorithms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The last major version in the 2.x series:</td>
<td>Python 2.7.12 PC &amp; DPSF</td>
</tr>
<tr>
<td>The Pandas library version 0.17.1</td>
<td>PC &amp; DPSF</td>
</tr>
<tr>
<td>The version 1.6.2 of the Apache Spark</td>
<td>Spark Framework, released on June 25, 2016, with the pre-built package for Hadoop 2.6 and later. PC &amp; DPSF</td>
</tr>
<tr>
<td>Framework, released on June 25, 2016, with</td>
<td>PC &amp; DPSF</td>
</tr>
<tr>
<td>the pre-built package for Hadoop 2.6 and</td>
<td>Used to execute the Optimised Python DQ$^2S$ version of the original set of algorithms.</td>
</tr>
<tr>
<td>later.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Tools used in the empirical evaluation of the research

4.3 Setup: Spark Cluster and Application Settings

4.3.1 The Apache Spark Local Mode

The Apache Spark big data framework was created to be utilised on a cluster architecture, but it can still it can be set in a called “Local mode”. This implementation does not require a cluster manager to schedule the work required to perform the computations, and although this Local mode is not meant to offer all of Spark’s complete configuration, yet it provides the parallel computation benefit as well as the RDD data structure, which is one of the Spark’s most valuable asset.

In Local mode, there is no such thing as workers. The driver program becomes also the executor, and the task scheduling and the tasks execution happens within the same JVM; in this mode, parallel processing is achieved by multi-threading not really by the whole cluster executors, workers and cluster manager process as it is done on a real cluster mode (either Standalone, YARN or Mesos).

To set the number of tasks that a single machine with Local mode should process at the same time, the master address needs to be set to local[n], where $n$ is the number of tasks the framework will handle simultaneously (number of threads), generally referred to as “cores”, and as mentioned in Section 3.2.4 general best practice is to set $n$ to the number of available cores.

A cluster mode can be started on a single machine, however, this is commonly
not recommended unless having a large capacity hardware because a cluster mode will cause a single machine to run several JVMs, provoking an overhead due to the JVMs initiation, generating in this way more processes than the Local mode generates, thus becoming more costly in terms of CPU usage, and therefore runtime, as the OS will need to handle more processes. When not having a cluster, or a large capacity machine, the Local mode provides an option to utilise Apache Spark taking the most of the resources available to process a Spark application with a simpler deployment. Spark Local mode was used on some of the experiments performed with the big data framework on the desktop machine described in Table 4.3, aimed at exploring the Local mode behaviour on a commodity machine for the Timeliness query.

4.3.2 The Apache Spark Standalone Mode

Another Spark mode is its implementation and usage on a cluster, which is the real focus of the framework. It is important to point out that Standalone mode cannot be called like that unless the Spark’s cluster manager is used. The role of the “cluster manager” serves a purpose within job submission on a cluster by scheduling the resources requested, utilising the workers to create executors for the driver and communicating the workers and executors availability to the driver process, even if only one user is going to utilise the cluster or only one application will be submitted, although this is not likely to be the case when Spark is used on a cluster.

For this research, to execute the Timeliness query with the PySpark code, denominated as the “application”, the TimelinessQuery class needs to be submitted to the Spark cluster. The applications submitted to Spark run through Spark’s master-slave architecture, where the set of processes corresponding to the application is coordinated by the driver program, the one containing the SparkContext object declaration, as shown in Section 3.2.4. This research uses the Standalone manager since it is the one created specifically for Spark and this investigation seeks to conduct an evaluation on the big data framework in isolation, without further aid from other technologies, in order to analyse Spark’s capabilities. This decision on the cluster manager is what gives the name to a “Standalone mode” when talking about a Spark cluster.

There are three locations in which Spark configurations can be made [129], summarised in Table 4.5:

- Application parameters can be set directly on a SparkConf instance, passed
to the SparkContext object, where the SparkConf object contains information about the application that is going to be executed within Spark, for example, the master’s node address, the application name or the home directory path.

There are three ways of setting Spark properties using the SparkContext object: embedded in the source code of an application, using flags to the pyspark-submit command used to submit an application to the framework or written on the ./spark-home-directory/conf/spark-defaults.conf file, which are taken by Spark, following that order priority.

Using flags makes possible to submit spark-defaults.conf parameters, by using the --conf flag, followed by the spark parameter. E.g. --conf spark.driver.cores=2.

- Per-machine and cluster settings can be set statically on the configurations files, where spark-env.sh is the environmental variables file, used to specify the details of the version number used for Java and Python, and set the number of executors, its number of cores to be used on each machine, as well as maximum memory. This file is needed by the cluster launching scripts found in ./spark-home-directory/sbin path, named start-master.sh, start-slaves.sh and start-all.sh, which starts the cluster and/or its slaves once submitted through Spark.

To specify where Spark should start a slave node, the slaves (without any kind of extension) file is used, taken by the start slaves launching script. All of the configuration files can be found in ./spark_home_directory/conf path.

- Logging through log4j Apache logging service, to specify how the cluster collects and presents logged data, by adding a file that should be named log4j.properties in the conf directory. Log4j is a logging and tracing API, which supports a debugging method by looking at the logs created by the application. In Spark, for example, it can be used to set the console as the place which everything must be logged to or to set the default log level.

The above mentions examples of the relevant configurations to this research, however, an exhaustive list of settings available for Spark can be found in [129].

<table>
<thead>
<tr>
<th>Configuration type</th>
<th>File / Place to utilise</th>
<th>Purpose</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Place</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.4 EXPERIMENT PROCEDURE

4.4.1 Application parameters.

SparkConf instance.

Set spark properties to access the cluster. E.g. Master’s node address, application name, or home directory path.

Source code of an application, flags in submission or ./spark.home_directory/conf/spark-defaults.conf

4.4.2 Cluster settings.

spark-env.sh

Set environmental variables. E.g. Java and Python version, number of executors, etc.

./spark.home_directory/conf/spark-env.sh

slaves

Specify address to start a slave node.

./spark.home_directory/conf/slaves

4.4.3 Logging

log4j.properties

Specify how the cluster collects and presents logged data.

./spark.home_directory/conf/log4j.properties

Table 4.5: Spark configuration locations

For this study, three files from the Spark directory were edited with different parameters on each experiment: spark-env.sh, spark-defaults.conf, and slaves. The content set to each file per experiment is described in Section 4.4 of this Chapter.

4.4 Experiment Procedure

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>From E1 to E4, will be used as a baseline to be able to compare results with the other Experiments. This will conform the analysis of how DQ²S scales and performs using non-big data implementations.</td>
</tr>
<tr>
<td>E2</td>
<td>E5 and E6 support the analysis of a big data framework’s capabilities when being used on a single multicore machine, and provides the information to compare results from E1-E4 (non-big data) with a big data solution.</td>
</tr>
<tr>
<td>E3</td>
<td>E8 supports the analysis of the big data framework behaviour in a cluster environment, and accompanied by E9, offers an overview of the advantages that a big data framework provides when processing different workload sizes.</td>
</tr>
<tr>
<td>E4</td>
<td>E9 supports the analysis of the big data framework behaviour in a cluster environment, and accompanied by E9, offers an overview of the advantages that a big data framework provides when processing different workload sizes.</td>
</tr>
</tbody>
</table>

Table 4.6: General overview of the experiment’s aim
To tackle the research questions, the testing involved the execution of the Timeliness query under different environmental settings. These settings depend on three variables: the Timeliness query instances (described in Chapter 3), the dataset file size, and the usage of a desktop machine and a cluster. The experiments were separated into nine different ones, in a way that each experiment could have its own delimited and clear set of results to be later compared and correlated altogether, these experiments are presented below. All the programming classes mentioned are described in Chapter 3. A summary of the experiment’s aims and an overview of the recorded quantity of results obtained from the nine different experiments is shown in Table 4.6 and Table 4.7 respectively. Each run indicated utilised both orderT and statusTimelinessQR datasets of the same size, for example, when indicating a run of 50 000 000 rows, both orderT and statusTimelinessQR contain 50 000 rows each one. Runtimes were recorded in milliseconds and 5 times as described in Table 4.7, the average result was utilised to assure a higher level of precision considering that executions in a system do not run

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Environment</th>
<th>Number of runtimes recorded</th>
<th>Number of averages recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>PC</td>
<td>15 (5 per input size - 5 x 3)</td>
<td>3</td>
</tr>
<tr>
<td>E2</td>
<td>PC</td>
<td>15 (5 per input size - 5 x 3)</td>
<td>3</td>
</tr>
<tr>
<td>E3</td>
<td>PC</td>
<td>15 (5 per input size - 5 x 3)</td>
<td>3</td>
</tr>
<tr>
<td>E4</td>
<td>PC</td>
<td>5 per successful input size</td>
<td>1</td>
</tr>
<tr>
<td>E5</td>
<td>PC</td>
<td>120 (one per input size per configuration - 3 x 8) (5 per single runtime - 5 x 24)</td>
<td>24</td>
</tr>
<tr>
<td>E6</td>
<td>PC</td>
<td>120 (one per input size per configuration - 3 x 8) (5 per single runtime - 5 x 24)</td>
<td>24</td>
</tr>
<tr>
<td>E7</td>
<td>PC</td>
<td>120 (one per input size per configuration - 2 x 12) (5 per single runtime - 5 x 24)</td>
<td>24</td>
</tr>
<tr>
<td>E8</td>
<td>Cluster</td>
<td>60 (one per input size per configuration - 3 x 4) (5 per single runtime - 5 x 12)</td>
<td>12</td>
</tr>
<tr>
<td>E9</td>
<td>Cluster</td>
<td>15 (5 per input size - 5 x 3)</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.7: Number of runtimes and averages collected by experiment.
4.4. *EXPERIMENT PROEDURE*

for the same exact milliseconds due to Operative System’s (OS) processes that are not user controlled.

**E1. Original Java DQ²S experiment - PC**

This experiment was executed on the desktop machine, and aims to show how the original DQ²S configuration implemented with Java scales and performs. For this experiment, each dataset size (50 000, 100 000 and 1 000 000 rows) was used as input for the Java program. The following procedure was utilised to record the results:

- (a) Open a Linux terminal.
- (b) Place to the folder containing the Java classes.
- (c) Create the classes to execute typing: `javac TimelinessQuery.java`
- (d) Execute the `TimelinessQuery` class typing: `java TimelinessQuery`
- (e) Discard the first run, known as “cold run”, in which data is loaded into the cache of the machine.
- (f) Register the runtime printed out in console, aided by the `Timing` class.
- (g) Repeat (d) and (f) five times.
- (h) Calculate the average time of the five registered times.

**E2. Python DQ²S experiment - PC**

This experiment was executed on the desktop machine, and aims to show how Python DQ²S scales and performs in comparison to the Java DQ²S instance. This was done by gathering the runtime results from the Python DQ²S executed over each dataset size (50 000, 100 000 and 1 000 000 rows), utilising the following procedure, and compared later with the results from Experiment 1 (E1 in this chapter):

- (a) Open a Linux terminal.
- (b) Locate to the folder containing the correspondent Python classes.
- (c) Execute the `TimelinessQuery` class typing: `python TimelinessQuery.py`
- (d) Discard the first run, known as “cold run”.
- (e) Register the runtime printed out in console, aided by the `Timing` class.
- (f) Repeat (c) and (e) five times.
(g) Calculate the average time of the five registered times.

E3. Optimised Python DQ²S experiment - PC

The aim of this experiment is to explore how the Optimised Python DQ²S scales and performs in comparison to the Python DQ²S and the Java DQ²S. Each dataset size (50 000, 100 000 and 1 000 000 rows) was used as input 6 times per size, collecting 5 runtimes per dataset size, discarding the first run each time. This experiment followed the same procedure as Experiment 2 (E2 in this chapter) since this is an instance developed also in Python language with the implementation of a scientific library, as described in Chapter 3.

E4. Optimised Python DQ²S maximum capacity experiment - PC

This experiment aims at finding up to how many rows the Optimised Python DQ²S can process the Timeliness query successfully on the desktop machine. Taking into account that how big is big data could be thought as the size in which using “traditional techniques” for the processing is no longer an option. For this experiment a 5 000 000 rows dataset was utilised as first attempt to process a larger dataset than the 1 000 000 rows one (baseline as described in Section 4.2). This value provided the knowledge to whether or not that was a large enough dataset, so the volume could be increased on a 5 000 000 rows basis, or decreased on 1 000 000, as follows:

(a) Open a Linux terminal.
(b) Locate to the folder containing the correspondent Python classes.
(c) Use an input data set of 5 000 000 rows.
(d) Execute the TimelinessQuery class typing: python TimelinessQuery.py
(e) Discard the first run, known as “cold run”.
(f) Repeat (d) and if the execution completes to the end successfully, add 5 000 000 rows to the input data set and repeat from (d).
(g) Once a failure is reached, subtract 1 000 000 rows to the input dataset that caused the failure.
(h) Repeat (d) and (e), and keep subtracting 1 000 000 rows until the input data size causes a successful run.
4.4. EXPERIMENT PROCEDURE

(i) After (h) is completed (a input data size was found by reducing 1 000 000 rows each time), add 500 000 rows to the last known successful dataset and repeat (d) and (e).

(j) If the input data set size of (i) was successful, execute (k), otherwise decrease by 500 000 rows the dataset and execute (d), (e) and continue with (k).

(k) Register the number of rows of the input data set.

(l) Register the runtime printed out in console, aided by the Timing class.

(m) Repeat (d) and (l) five times.

(n) Calculate the average time of the five registered times.

E5. PySpark DQ^2S Standalone mode in single machine experiment - PC

The Standalone mode was not designed to be used on a single machine, as described in Section 4.3.2, however, it is a common beginner’s mistake to implement it that way, risking at having a wrong implementation that could offer a wrong impression of a big data framework performance. Another common mistake is to create more workers in order to have more executors, since for Spark Standalone mode it is not very clear how to set more than one executor per worker, and also because on Spark’s versions below 1.4.0, it was not possible to have more than one executor per worker. This experiment sets one executor per worker to explore what would happen if a Standalone mode is implemented and intended to be used for the DQ^2S in a single multicore machine, with the circumstances mentioned (one executor per worker, constrained to physical cores). The results of this experiment will form part of the supporting information to analyse whether a cluster is mandatory to take the most of Spark, as well as contribute to the insights of the big data framework behaviour in a desktop environment (PC).

The steps followed to start-up the cluster used on this experiment are detailed in Appendix B, Section B.2, and the actions used to submit the PySpark applications can be found in Section B.3. For this experiment, three files from the Spark directory were edited: spark-env.sh, spark-defaults.conf, and slaves. All of the settings are options used by the Standalone cluster, where it was considered a total memory of 14GB, by a naive consideration of keeping 2GB for OS processes, and 4 physical cores. The number of physical cores provided a total of 8 configurations, as described below.
The submission requires several parameters to be set, including the application name, the master node address, number of workers per node, number of executors per worker and its corresponding memory and number of simultaneous tasks an executor will handle (known as “cores”). The application name and the master’s node address was defined in the code of the programs, this is not practical when using different applications or in a cluster where it might require a more dynamic update of those parameters, but for this experiment, since it was done using only one computer, it did not produce any drawback.

1. The content of the spark-env.sh file was set as follows:

```bash
export SPARK_WORKER_MEMORY={3.5,4.7,7,14}g
export SPARK_EXECUTOR_MEMORY={3.5,4.7,7,14}g
export SPARK_WORKER_INSTANCES={1,2,3,4}
export SPARK_WORKER_CORES={1,2,3,4}
export SPARK_WORKER_DIR=/local_drive/sparkdata
```

- **SPARK_WORKER_MEMORY** is used to set how much total memory must be reserved for each executor, since one executor is created per worker, according to the given parameters of the worker, as explained in Section 3.2.4. This was fixed to 3.5, 4.7, 7 and 14 GB in different times, constrained by the number of workers set. Those numbers were obtained based on the total amount of memory available and the number of workers set, hence, when the testing was executed with 2 workers, the memory was divided by 2, thus, SPARK_WORKER_MEMORY has a value of 7GB.

- **SPARK_EXECUTOR_MEMORY** allows to specify how much of the worker’s reserved memory must be given to each executor. This parameter was set to be the same value as SPARK_WORKER_MEMORY, in order to have always one executor per worker. Both SPARK_WORKER_MEMORY and SPARK_EXECUTOR_MEMORY do not accept values different to integers, therefore, to set 3.5 and 4.7GB, it was done using its equivalent integer value in MB. E.g. 3584M instead of 3.5g (Spark utilises a notation of M for MB, and g for GB in the spark-env.sh settings).
4.4. EXPERIMENT PROCEDURE

- **SPARK_WORKER_INSTANCES** defines the number of worker processes with one executor process each one. The evaluation was executed using from 1 to 4 workers each time. More than 4 workers was not tested because of the availability of 4 cores; 5 workers require at least the allocation of 5 cores, one per worker.

- **SPARK_WORKER_CORES** sets the number of simultaneous tasks to be handled per executor on each worker. In this case the value used was selected according to the number of workers used. For example, when using 1 worker, it use from 1 to 4 cores each time, but having 2 workers can only use 1 or 2 cores each since using 3 cores for 2 workers means 6 cores would be required.

- **SPARK_WORKER_DIR** is used to specify the working directory of the worker processes, in this case it was used the local drive and a directory named “sparkdata”.

2. The content of spark-defaults.conf was configured to enable visibility of the application executions log after the work is finished. The properties set were the following:

```bash
spark.eventLog.enabled true
spark.eventLog.dir //local_drive/sparkLogs
```

Where `eventLog.enable` sets to true or false to see the information after the application is finished and the `eventLog.dir` property sets the path to the folder where the logs should be saved.

3. Finally, the `slaves` file, contains only the word “localhost”, to specify that the workers should be started locally, since the implementation is deployed in the same machine.

The procedure to execute this experiment is the following one:

(a) Turn on the Standalone Spark cluster by starting the master and slaves nodes.

(b) In the files with the cluster settings, modify the number of workers, cores and allocated memory as required.

(c) Manually submit the TimelinessQuery class to the Spark cluster, using:
bin/spark-submit --packages com.databricks:spark-csv_2.11:1.3.0 \TimelinessQuery.py TimelinessRun_1 outputResults_1

Where TimelinessRun_1 is the name of the application, utilised by the
cluster deployment to identify the application, useful when the Standalone
or any other cluster mode is used by many users or to submit several ap-
plications; and outputResults_1 is the name of the folder in which Spark
will save the results, this name could be any name.

(d) Register the runtime printed out in console, aided by the Timing class.

(e) Discard the first run, known as “cold run”.

(f) Repeat (c) and (d) five times per each set of parameters.

(g) Calculate the average time of the five registered times.

E6. PySpark DQ²S Local mode in single machine experiment - PC

The aim of this experiment is to show how Spark in Local mode performs and
scales in a multicore machine. The input sizes utilised for this experiment are
also 50 000, 100 000 and 1 000 000 rows. This experiment comprises 6 execu-
tions per input data set size for the following variables, executing a total of 12
different configurations, setting from 1 to 8 threads for the n value in local[n],
where an execution without any kind of parallelisation is achieved by setting
the Spark master parameter to local[1]. This was set up to 8 because of the
available cores in the machine. This experiment and all of the ones performed
using the Local mode, utilised a driver memory of 12GB, based on Spark’s rec-
ommendation of utilising up to 75% of the total available memory [46]. The
procedure was followed as shown below:

(a) Open a Linux terminal.

(b) Locate to the Spark’s home directory.

(c) Manually submit the TimelinessQuery class to Spark using:

sbin/spark-submit TimelinessQuery.py
--packages com.databricks:spark-csv_2.11:1.3.0
--driver-memory 12g --master local[n]

with n being from 1 to 8.

(d) Register the runtime printed out in console, aided by the Timing class.
4.4. EXPERIMENT PROCEDURE

(e) Discard the first run, known as “cold run”.

(f) Repeat (d) and (e) five times per each input data size and each \( n \) value.

(g) Calculate the average time of set times comprised by five registered times each.

E7. PySpark DQ\(^2\)S Local mode maximum capacity experiment - PC

The aim of this experiment is assess how many rows Spark in Local mode performs and scales in a multicore machine and analyse how Spark manages a bigger dataset. The input size utilised for this experiment, starts from the maximum number of rows that the non-big data solution was able to process on the desktop machine, found in Experiment 4 (E4 in this chapter).

This experiment comprises an execution using the same configuration that the one set in Experiment 6 (E6 in this chapter) for \( n \) in \texttt{local}[n], plus four additional configurations: \texttt{local}[9], \texttt{local}[10], \texttt{local}[12], and \texttt{local}[16]. These four extra sets were designed to explore Spark’s behaviour regarding the general recommendation of utilising the number of threads as much as the number of available CPU cores, and to test if there is a gain when increasing the number of tasks to be handled at the same time, specially \texttt{local}[12] and \texttt{local}[16] were selected to check the results when setting three and four times respectively the number of physical cores. To perform this experiment, the following procedure was used:

(a) Open a Linux Terminal.

(b) Locate to the Spark’s home directory.

(c) Utilise the dataset that has the maximum number of rows that the Optimised Python DQ\(^2\)S could handle, revealed on E4.

(d) Manually submit the TimelinessQuery class to Spark using:

```
sbin/spark-submit TimelinessQuery.py
--packages com.databricks:spark-csv_2.11:1.3.0
--driver-memory 12g --master local[n]
```

with \( n \) being from 1 to 10, 12 and 16.

(e) Discard the first run, known as “cold run”.

(f) Register the runtime printed out in console, aided by the Timing class.
(g) Repeat (d) and (f) five times.

(h) Repeat (d) and (e) with a dataset containing 5 000 000 more rows than the last dataset.

(i) Keep adding 5 000 000 rows each time until having a failure on the execution.

(j) Once a failure is reached, subtract 1 000 000 rows and repeat (d) and (e) until the input data size runs successfully.

(k) After (g) is completed (an input data size was found by reducing 1 000 000 rows each time), add 500 000 rows to the last known successful dataset and repeat (d) and (e).

(l) If the input data set size of (k) was successful, execute (m), otherwise decrease by 500 000 rows the dataset and execute (d), (e) and continue with (m) to complete it five times.

(m) Register the number of rows of the input data set.

(n) Calculate the average time of the five registered times for both the starting number of rows and the maximum number of rows that Spark could process in a single multicore machine.

E8. PySpark DQ²S cluster Standalone mode experiment - Cluster

This experiment was designed to present how Spark in Standalone mode behaves in the cluster with 1 to 4 worker nodes, starting from the number of rows that Spark Local mode could not cope with, found in Experiment 7 (E7 in this chapter), and then doubling and tripling its size.

The configuration used was fixed to have 1 worker with 8 executors per node, each executor with 2 cores and 32GB of memory. This configuration allows the cluster to utilise the maximum capacity of each node, by having as much memory as possible with the maximum number of executors with at least 2 cores each, from a fixed availability of 16. A pre set-up test showed this configuration to be the fastest among the possible combinations of executors and cores. The best basic parametrization is utilised to compare the maximum capacity of the big data framework that could be set without further tuning. The number of workers was not set to more than one because there is no reason to have more than one considering the purpose that it covers within a cluster (Refer to Section
and the pre set-up test showed an overhead when deploying a cluster with more than one worker per node; before Apache Spark version 1.4.0, it was not possible for a worker to spawn more than one executor in the Standalone mode \[62\], therefore, to utilise as much memory as possible, with workers of no more than 32GB of memory, it was advisable to create more than one worker per node but, with this constraint removed, setting multiple executors of 32GB allows to use as much memory as required without creating duplicated processes, recalling that the executors are the processes that perform the computations on its allocated tasks, hence, its duplicity is not creating overlaid processes. The workers behaviour is deeper explored in Experiment E5. A memory of 32GB was set on all of the possible combinations, because more than that amount of memory cause the deployment to have JVMs with large heaps, which prevent a fluency on the garbage collector, that can cause pauses of few seconds that can be easily converted into minutes \[137\].

In this experiment, the difference per execution resided on the number of nodes occupied. Since 1 worker with all of the available cores and all memory available minus 1GB is the default parameter, these were not set explicitly or changed; the number of executors and its memory were set using a flag.

The procedure to perform this experiment using the cluster is as described below, and the script used to submit the application to the cluster is presented on Appendix [B] Section [B.3] (The DPSF script):

(a) Turn on the Standalone Spark cluster by starting the master and slaves nodes, requesting 32 cores of the cluster when requiring 1 node, 48 when using 2 nodes, etc., until a maximum of 80 cores for its usage on 4 nodes. On the DPSF cluster, when requesting 32 cores, a node with 16 cores is used for the driver and master processes. There is a difference for the cores requested to the DPSF cluster and the cores requested for the Spark application, the first is set within the DPSF submission script, and the second is configured through Spark flags, with the line shown in (b).

(b) Manually submit the TimelinessQuery class to the Spark cluster using a configuration to have the executors utilising all of the available cores on each node:

```
spark-submit --master=spark://MASTER_URL --verbose
--packages com.databricks:spark-csv_2.11:1.3.0
```
--executor-memory 32G --executor-cores 2 TimelinessQuery.py

Where MASTER_URL is the URL of the master node, set by the DPSF script.

(c) Register the runtime printed out in console, aided by the Timing class.

(d) Discard the first run, known as “cold run”.

(e) Repeat (b) and (c) five times per each set of parameters.

(f) Calculate the average time of the five registered times per different number of nodes.

E9. Optimised Python DQ²S cluster experiment - Cluster

This experiment shows how the optimised Python DQ²S scales and performs in the cluster, using the input data sizes utilised in Experiment 8 (E8 in this chapter). The procedure for this experiment is the same as described in Experiment 2 (E2 in this chapter), but performed in the cluster instead of in a desktop machine.

4.5 Algorithm Comparison

The results obtained from the executions defined above were analysed according to the following observations, which were designed to avoid having comparisons between the different instances of the DQ²S being advantageous one over another due to its inherent structure, power given by number of cores, or degree of parallelism, this is, having comparisons based on platform utilisation and not in algorithm design (Refer to Section 2.4 for details of platform and algorithm design scalability approaches). The purpose of the following comparisons designed is also to organise the results in such a way them could offer an insight that can support the hypothesis.

C1. The original DQ²S Java program (E1) compared with its equivalent DQ²S developed using Python (E2).

These programs support a comparison of the same algorithm written in different languages, without any kind of parallelisation implemented. This comparison was thought to offer an insight regarding Python against the original DQ²S developed with Java, with regards to its performance and scalability.

C2. The Python DQ²S program (E2) compared with the Optimised Python version of it (E3).
These instances were thought to be a basis for showing the performance obtained from implementing the data frames library without a big data framework. The results of the Optimised Python version of DQ$^2$S provide a reference for the research on scaling with big data frameworks.

C3. The Optimised DQ$^2$S Python program (E3) compared with the PySpark DQ$^2$S program in Local mode (E6).

This comparison was designed to show measurements that led to an analysis of the impact caused by the usage of a big data framework, having as baseline the results from the comparison mentioned in C2. The outcome of this comparison would be to inspect the performance obtained from the usage of Apache Spark implemented within a single multi-core machine, leading to know the actual and quantifiable impact obtained from the usage of a big data framework for this and similar use cases.

C4. The maximum number of rows processed by the Optimised DQ$^2$S Python program (E4) compared with the PySpark DQ$^2$S program in Local mode (E7).

These results were designed to explore the behaviour of the big data framework when processing the maximum data volume that the DQ$^2$S instance without parallelism could cope with, aimed at discovering if the framework can offer any advantage with regards to processing capacity, focused on the volume characteristic of big data, as well as to asses its scalability and performance comparing these two instances when processing similar workloads (volume).

C5. The PySpark DQ$^2$S program in Standalone mode (E5) compared with a PySpark DQ$^2$S in Local mode (E6), both in a single machine.

Although the Standalone mode is not designed to be utilised in a single core machine, it could be the case that it is used this way as it is possible to do it, hence, this comparison aims at exploring the differences between Local and Standalone mode implemented in a single machine and gather insights about their performance and general behaviour when processing the same datasets.

C6. The PySpark DQ$^2$S program in Local mode (E7) compared with a PySpark DQ$^2$S used in a cluster under Standalone mode (E8).

This comparison was designed to explore the extent to which a cluster environment provides significant advantages over a multicore commodity hardware.
The purpose of this comparison is to analyse if a cluster is mandatory to consider a big data framework a suitable option for big data processing.

C7. The PySpark DQ$^2$S program in Standalone mode (E8) compared with a Optimised Python DQ$^2$S version executed on the cluster (E9)

This final comparison was designed to offer a view of how a non-big data solution works under a cluster environment, therefore, to discard or include the architecture as crucial in terms of scalability and performance results.

4.6 Experiment Evaluation

The executions were monitored to assess the performance in terms of the metrics defined for parallel algorithms, which are described in detail in Section 4.6.1 and calculated using the formulas described in the same section, which require only the runtime of the execution. A second part of the experiment evaluation includes a validity assessment to support the apparent conclusions obtained from the results, as well as an description of how threats to validity were approached, described in Section 4.6.2.

4.6.1 Performance Metrics

The cluster created has a shared memory in a shared disk architecture, having available the data stored in the only one Hard Disk in use, and accessible for all the members of the cluster globally, as well as the memory available. Still a parallel architecture, it was needed to use Performance Metrics specifically designed to measure certain characteristics available for parallel programs.

The most common metrics for performance on parallel algorithms [73, 3, 28, 4, 118, 114] were used to support the performance analysis, based on the comparisons designed and described in Section 4.5. The information required by the performance metrics used for this research are the following:

- **Serial runtime (TS)**. The measurement of the time elapsed between the beginning and the end of the fastest known serial algorithm that solves the problem.

- **1 Core serial runtime (T(1))**. The measurement of the time elapsed between the beginning and the end of a parallel algorithm executed on a single core.
• **Parallel runtime (TP).** The measurement of the time elapsed between the beginning and the end of a parallel algorithm executed on more than one core. This time is composed by 3 different components: the time spent on the computation, the cores communication time, and the idle time which is when any processor is working but still has not finished.

• **Number of cores (p).** The number of processing elements on which the parallel algorithm is executed. For this study the processing elements are the cores set as described in Section 4.4.

The performance metrics considered for this study are as follows:

1. **Speedup (S).** Meant to measure the performance gain of a parallel program when solving the same problem over the sequential implementation one. In an ideal parallel system, speedup is equal to the number of cores used, however, it is usually lower than p.

There are more than one approach towards measuring speedup: Relative, Real, Absolute, Asymptotic and Asymptotic relative [114, 123]. This study occupies the ones that are valid and entirely applicable to the design and scope of the research, as follows:

• **Real Speedup (rS).** This speedup requires the sequential time TS from the fastest serial algorithm available to be compared against time obtained from the parallel algorithm over more than one core. This speedup is calculated using:

\[ rS = \frac{TS}{TP}. \]  \hspace{1cm} (4.1)

• **Relative Speedup (rtS).** Calculates the speedup regarding TS as the parallel execution time from the algorithm run over one core T(1). This is made to consider the inherent parallelism of the parallel algorithm that is being assessed. The formula utilised to calculate the relative speedup is:

\[ rtS = \frac{T(1)}{TP}. \]  \hspace{1cm} (4.2)

The aim of this calculations is to isolate the benefit from parallelism given by a single parallel processing unit and the fastest sequential algorithm.
2. **Efficiency** ($E$). Closely related to speedup, measures the resource utilisation given the number of parallel units used. Efficiency is constrained by the non-sequential operations performed by the cores, this is, how much of the entire program is parallelised. This metric is defined as the ration between the speedup value and the number of cores used.

The aimed efficiency for parallel systems is equal to one, but it is, in practice, usually between zero and one; an “scalable algorithm” tends to increase its efficiency when $p$ grows. Efficiency is calculated by:

$$E = \frac{S}{p} = \frac{TS}{TP \ast p}.$$  \hspace{1cm} (4.3)

- **Real Efficiency** ($rE$). Calculates the efficiency with the correspondent $rS$ value, as follows:

$$rE = \frac{rS}{p} = \frac{TS}{TP \ast p}.$$  \hspace{1cm} (4.4)

- **Relative Efficiency** ($rtE$). Utilises either $rtS$ or $T(1)$ for the $TS$ value in the Efficiency formula ($E$), as follows:

$$rtE = \frac{rtS}{p} = \frac{T(1)}{TP \ast p}.$$  \hspace{1cm} (4.5)

### 4.6.2 Validity Evaluation

The design of the experiment is in general a one factor with two treatments model, with some comparisons having more than two treatments. The factor is the dataset, which is the same on its content, differentiated only by its size. The treatments are the DQ$^2$S instances; an instance with different parameters is considered as another treatment, for example, a PySpark on Local mode is a different treatment to a PySpark instance on Standalone mode.

When evaluating the results validity, there are four classifications assessed [17]:
4.6. EXPERIMENT EVALUATION

- **Conclusion validity**, concerning the conclusions based on the results obtained.

- **Internal validity**, to ensure the effect of the treatment is in fact caused by the treatment and no other factor.

- **Construct validity**, to be sure the treatment is what it is supposed to be, and works as it is expected to work according to the experiment description and theory related to the treatment.

- **External validity**, to check to which extent the results obtained can lead to a generalisation and real-world application.

There are recognised threads to validity [17], correspondent to the classifications listed above. The threats applicable to this study were approached as shown in Tables 4.8, 4.9, 4.10 and 4.11.

<table>
<thead>
<tr>
<th>Threat</th>
<th>Description</th>
<th>How it was addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fishing results</td>
<td>This threat refers to the action of looking for specific or favourable results, discarding the inconvenience of some data.</td>
<td>Reproducibility details of the research offers a support to obtain the results provided, confirming data gathered was taken from the results as described, without discarding data to produce a desired outcome.</td>
</tr>
<tr>
<td>Reliability of measures</td>
<td>Level of trust in which the values that constitute the samples are accurate representations of the measured events.</td>
<td>A custom instrumentation aided by code was implemented to measure each execution runtime without human intervention or subjectivity, assuring the same mechanism for all of the experiments.</td>
</tr>
</tbody>
</table>
CHAPTER 4. EXPERIMENT DESIGN

Random interference in experimental settings

There could be factors involved in the experiment execution that alter the results, other than the known or planned.

OS processes as well as other processes could alter the runtimes if the CPU and RAM were subject to varying loads not connected with the experiment, to address this, no other applications were being used while performing the experiments besides the necessary ones (console, browser and notepad), and the same applications were used in the same state when carrying out the executions. This is also why more than one runtime was collected per instance, per dataset size and per configuration.

Random heterogeneity of subjects

Heterogeneity could lead to having abnormal results due to the heterogeneity of the datasets, rather than due to the performed and measured intervention.

This threat is addressed by having the same dataset involved in all of the executions of the experiments. To minimise the trade-off of having a reduced population, different dataset sizes were used.

Table 4.8: Conclusion validity threats applicable to the research presented. Details and addressing approach.

<table>
<thead>
<tr>
<th>Internal validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threats</td>
</tr>
<tr>
<td>History</td>
</tr>
</tbody>
</table>
### 4.6. EXPERIMENT EVALUATION

<table>
<thead>
<tr>
<th>Threat Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maturation</td>
<td>The results can be affected as time passes, presenting different behaviours the first time the experiment was applied than the subsequent times. This threat is applicable because of the cache mechanism, but it was addressed flushing the cache each time a new experiment was performed, and the collected runtimes did not take into account the first result to avoid this threat.</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>A bad design on the execution and the way of obtaining the data could provide incorrect values. This threat is aligned to the reliability of measures, addressed by implementing a runtime class, described in Chapter 3, which is homogeneous to all of the instances, does not affect or influences the results, and is implemented from the internal code, which increases accuracy of real processing time, reducing the error of measurement that could be made, for example, by counting the time including the one spent on the cluster queue.</td>
</tr>
<tr>
<td>Statistical regression</td>
<td>Classification of experimental groups based on a known given range, for example top or bottom result, which would conform a pre-disposition for further results. To avoid analysing results based on this threat, comparisons were designed to refrain from analysing results based on unequal, not comparable instances. For example, results from the cluster with a 4 node deployment is not comparable to a desktop PySpark Local mode on its worse configuration.</td>
</tr>
<tr>
<td>Ambiguity about direction of causal influence</td>
<td>An issue of unclarity about identifying the cause and the effect, where a result has the possibility of being either any of both, or only part of the cause or the effect. Homogeneity of the factor to utilise in the experimentation (the dataset) avoids variability on each of the experiments, reducing the unknown of considering if perhaps the factor, not the treatment, offered the different results.</td>
</tr>
</tbody>
</table>

Table 4.9: Internal validity threats applicable to the research presented. Details and addressing approach.
### Construct validity

<table>
<thead>
<tr>
<th>Threats</th>
<th>Description</th>
<th>How it was addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inadequate pre-operational description of constructs</td>
<td>Ambiguity and unclarity on the description of the experiment, lack of sufficient details to provide a solid base upon which results and analysis are described.</td>
<td>This research presents details on the treatments, the factor of the experiment, the procedure and its analysis method, supported by a rigorous methodology, as described in Chapter 1.</td>
</tr>
<tr>
<td>Mono-operation bias</td>
<td>When the experiment is limited to few or even only one variable, use case, factor, or treatment, which carries a risk of having a diminished scope in which results are valid. The threat becomes active when this sub-representation is not considered or known.</td>
<td>The scope of the research considers its limitations, and avoids overgeneralising results accordingly.</td>
</tr>
<tr>
<td>Mono-method bias</td>
<td>Utilising a single measure, or a single collection method, with subjectivity involved, could altogether and individually led to biased results.</td>
<td>Measurements are free of subjectivity in this research, achieved by a lack of human interpretation on the runtime recorded. The runtimes obtained from the instrumented internal code were checked against Spark’s Web User Interface, where runtime is displayed, and against human measurement only to provide a triple check on the collected results. Internal instrumentation was the only one recorded.</td>
</tr>
<tr>
<td>Restricted generalisability across constructs</td>
<td>Treatments can impact both positively and negatively on the object of study as an “unintended side effect” [17]. Both effects must be identified, otherwise conclusions become inaccurate.</td>
<td>Special care was taken when analysing the results, exploring the advantages and disadvantages presented by each one of the treatments, not only the advantages.</td>
</tr>
</tbody>
</table>

Table 4.10: Construct validity threats applicable to the research presented. Details and addressing approach.
4.7 Summary

This chapter presented the general details of the testing method and characteristics used, from the data and tools required, as well as the hardware specifications, to a description of how the testing results were analysed under certain metrics also described, and how validity threats are addressed towards its reduction, removal or control. Chapter 5 presents the results obtained from the implementation and testing of the DQ\textsuperscript{2}S instances as described in this chapter.

<table>
<thead>
<tr>
<th>Threats</th>
<th>Description</th>
<th>How it was addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interaction of selection and treatment</td>
<td>When having a sample obtained from a population not varied or not large enough, the risk appears when pretending to generalise results as if the sample was more representative of the population than it really is.</td>
<td>This threat is diminished by having a scope set, and adhering results and interpretations to the scope, without overgeneralising conclusions, as discussed in Chapter 6.</td>
</tr>
<tr>
<td>Interaction of setting and treatment</td>
<td>A serious risk to the experiment validity can become apparent when experimenting with toy problems, outdated techniques, non-reproducible or hard-reproducible experimentation, or a piece of work with no application or relevance.</td>
<td>A scope, motivation, and literature review are part of the supporting elements to the validity of the study with regards to this threat. Experimentation was performed utilising an algorithm and a system that has its application within the data quality and information technology area, utilising a framework that is widely utilised in practice (Apache Spark).</td>
</tr>
</tbody>
</table>

Table 4.11: External validity threats applicable to the research presented. Details and addressing approach.
Chapter 5

Evaluation of Experimental Results

Although performance can take into account more parameters than only execution time, such as memory usage, energy consumption, and implementation cost [73], one of the most important motivations for parallelisation is to get results within the minimum time possible [104, 3, 114, 123], thus, the experiments for this study were framed to rely on execution time, with few parameters aimed at developing a concise result set.

This chapter shows the results and analysis obtained from executing the set of experiments described in Section 4.4 utilising the average runtimes obtained from five executions of each query, and presents a discussion formed by the comparisons described in Section 4.5. The explicit values of the results are shown in Appendix C where Section C.1 shows the raw results, and Section C.2 presents the performance metrics calculated, both sections show results with a nine decimal precision.

5.1 Experiments Results

This section presents the results obtained from the execution of the experiments regarding performance and scalability. The values shown are the averages obtained from the raw results shown in Appendix C Section C.1. The results are presented in sub-sections aligned to the descriptions of the experiments shown in Section 4.4. After the first part of the results presented, comparison’s discussions are shown according to the organisation and description made in Section 4.5.
5.1. EXPERIMENTS RESULTS

R1. **E1 - Original Java DQ^2S experiment - PC**

![Runtime: Original Java DQ^2S](image)

Figure 5.1: Runtime registered in seconds for the Original Java DQ^2S processing 50 000, 100 000 and 1 000 000 rows.

The Original Java DQ^2S implementation took around 7 minutes to process the Timeliness query (described in Chapter 3, Section 3.1.2) when utilising datasets of 50 000 rows, 26 minutes with 100 000 rows datasets and 45 hours to run with 1 000 000 rows datasets in the input dataset, as illustrated in Figure 5.1. These results were consistent with the experimental results presented at DQ^2S’ original publication, Sampaio et. al. [26], where the runtime of the Timeliness query for an input of 100 000 rows was 56 minutes in a machine with half the amount of memory and CPU speed of the machine used in this research (desktop - PC machine described in Section 4.2). With the results obtained, the Original Java DQ^2S is presented as not a suitable implementation to handle more than 50 000 rows because of the relatively high runtime, considering that the size in bytes for 1 000 000 rows is 84.5MB for the OrderT dataset and 68.9MB for the statusTimelinessQR dataset. Based on these results, it is estimated that the original DQ^2S implementation would need more than 40 hours to process those 154MB with the Timeliness query. The reason why this Java instance is expensive is mainly because of the Join operations required to execute this query, where each dataset has to be fully ingested first to be saved on its own array. Moreover, the OrderT dataset is scanned row by row to separate the pending orders, and both datasets need to be joined with a method called addAll (as described in Section 3.2.1), which generates a copy of the data. Lastly, another
join operation adds the *timeliness score* to the former joined data. All those requirements cause the instance to spend a lot of time performing a sequential scan and copying each of the rows.

**R2. E2 - Python DQ²S experiment - PC**

![Runtime: Python DQ²S](image)

Figure 5.2: Runtime registered in seconds for the Python DQ²S processing 50 000, 100 000 and 1 000 000 rows.

The Python DQ²S instance presented high runtimes, with 76 minutes to complete the Timeliness query when having datasets of 50 000 rows, 5 hours for datasets of 100 000 rows and an average of 22 days to finish when processing 1 000 000 rows. Runtime of the order of days is not efficient for data management tasks, specially when handling relatively small data requires such long runtimes. Such behaviour places the Python DQ²S to be a non-suitable option to provide DQ²S the capability of handling big data.

Python is often considered an interpreted language, however, it is not a purely interpreted one \[35, 80\]. Python combines interpreting and compiling characteristics. A Python program is implicitly compiled on its execution to byte code, which is later interpreted. Regarding performance, Python is known to run slower than low level languages such as C or C++. This is because some pieces of code from Python are not fully compiled, and thus, require more execution time to be interpreted by the machine. The above represents one of the reasons why the Python DQ²S presented high runtimes, as well as the fact that Python is dynamically typed, which causes the Python interpreter to spend time inspecting each object to identify its data type to call the appropriate routines. Another drawback is Python’s object model that causes inefficient memory access due to
segmentation, compared to the object model of Python’s scientific libraries such as NumPy and Pandas [141, 142] which allow contiguous memory allocation.

R3. **E3 - Optimised Python DQ^2S experiment - PC**

![Runtime: Optimised Python DQ^2S](image)

Figure 5.3: Runtime registered in seconds for the Optimised Python DQ^2S processing 50 000, 100 000 and 1 000 000 rows.

The Optimised Python DQ^2S presented significantly better results, with runtimes of less than one second for the Timeliness query when processing 50 000 and 100 000 rows, and less than two seconds for the 1 000 000 rows dataset, as shown in Figure 5.3. The scalability of this implementation can be considered good taking into account that less than two seconds for the larger workload presented in the experiments is an acceptable runtime to work with when utilising DQ^2S instead of runtime of hours or even days. However these results appear as a baseline upon which further exploration is required, considering that 1 000 000 rows is not as large a data set according to current trends can be. Even though DQ^2S needs to be able to cope with larger volumes of data, the Optimised Python instance seems to be a promising support for bigger workloads with such small runtimes. Results of Experiment 4, shown in R4 below, present an analysis on the Optimised Python DQ^2S for larger datasets.

As described in Section 3.2.3, this DQ^2S instance was developed utilising a Python Data Analysis library called “Pandas”. The use of this library reduces the work of the Timeliness query allowing a more straight-forward and optimised usage of functionalities common to tabular data, offered by the library. A summary of the specific differences regarding code and how it decreases the
cost of some operations, such as eliminating the need for copying row by row with custom code to perform a join operation, is shown in Section 3.2.3, Table 3.4 where the table presents a relation on the main differences that the usage of a Python scientific library imply. The results of the experiment presented in this subsection (Experiment 3) shows that those simple code optimisations have an impact on the runtime.

R4. E4 - Optimised Python DQ^2S maximum capacity experiment - PC

<table>
<thead>
<tr>
<th>Dataset size (rows)</th>
<th>Dataset size</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 000 000</td>
<td>444.5MB</td>
<td>348.9MB</td>
<td>OK</td>
</tr>
<tr>
<td>10 000 000</td>
<td>894.5MB</td>
<td>689.5MB</td>
<td>OK</td>
</tr>
<tr>
<td>15 000 000</td>
<td>1.4GB</td>
<td>1.1GB</td>
<td>FAIL</td>
</tr>
<tr>
<td>14 000 000</td>
<td>1.3GB</td>
<td>982.9MB</td>
<td>FAIL</td>
</tr>
<tr>
<td>13 000 000</td>
<td>1.2GB</td>
<td>911.9MB</td>
<td>FAIL</td>
</tr>
<tr>
<td>12 000 000</td>
<td>1.1GB</td>
<td>840.9MB</td>
<td>FAIL</td>
</tr>
<tr>
<td>11 000 000</td>
<td>989.5MB</td>
<td>769.9MB</td>
<td>OK</td>
</tr>
<tr>
<td>11 500 000</td>
<td>1GB</td>
<td>805.4MB</td>
<td>OK</td>
</tr>
</tbody>
</table>

Table 5.1: Relation of the dataset sizes and results when executing Optimised Python DQ^2S in a single machine for Experiment 4.

<table>
<thead>
<tr>
<th>Dataset size (rows)</th>
<th>Dataset size</th>
<th>OrderT</th>
<th>statusTimelinessQR</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 000 000</td>
<td>1 084 457 625 bytes</td>
<td>840 891 569 bytes</td>
<td></td>
</tr>
<tr>
<td>11 500 000</td>
<td>1 036 957 625 bytes</td>
<td>805 391 569 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Size in bytes for the first dataset that caused a failure (12 000 000 rows), and the maximum number of rows processed by the Optimised Python DQ^2S (11 500 000).

The Optimised Python DQ^2S cannot be considered as a suitable option for big data processing solely based on runtime results when handling relatively small volumes of data, as shown in the results of Experiment 3 (R3 above). To evaluate the suitability of the Optimised Python implementation for processing larger datasets it is necessary to know how much data it can handle, for which this experiment was designed.
5.1. EXPERIMENTS RESULTS

Following the procedure described in Section 4.4 for Experiment E4, 11 500 000 rows appeared as the maximum size this Optimised Python DQ2S instance could process successfully, with a runtime of 21.34 seconds. The experiment, as presented in Table 5.1, showed an out of memory failure for the datasets larger than 11 500 000 rows, where 12 000 000 rows is the first dataset size that caused a failed processing. The failure appeared under the Pandas I/O API reader function called `read_csv`, which aims at reading data from a CSV format to convert it into a DataFrame object [29]; currently this is the most efficient way to ingest data from an outer file in Python programming, compared to the implementation that could be done from scratch. The Pandas reader functions are set to load data without further programming specifications but only certain parameters, such as specifying the token delimiter, date time and error handling options, data types inference or manually set, etc., which otherwise should be coded explicitly, likely involving more implementation effort and higher error rates.

The difference between the last input data size that was successfully processed, and the first that was not, as shown in Table 5.2, is 47 500 000 bytes (1 084 457 625 - 1 036 957 625) for the OrderT dataset, and 35 500 000 bytes (840 891 569 - 805 391 596) for the statusTimelinessQR dataset, this is, in sum, a difference of ~79MB which made a separation between success and failure for the query processing. The maximum input that the Optimised Python DQ2S could cope with, according to the results shown in this subsection, has a size of few bytes less than 2GB, this is the limit in which the technique utilised with the Optimised Python DQ2S instance is no longer enough for the desktop machine utilised in this study. The out of memory failure shown in Table 5.1 is presented when the query process required more memory than the available allocated to the Pandas CSV reader, hence if more main memory was available (e.g. a machine with more than 16GB of RAM) this limit could be increased, however, without adding resources, there is a possibility to increase this limit by implementing a function to read input datasets iteratively in chunks, which would increase the limit to over 11 500 000 rows (~2GB), according to performed experiments which results are shown in Table 5.1. However, this solution causes an increase in runtime because of the I/O process done in iterations in this case. The cost of each round would probably depend on the number of rows consumed per iteration (these are called chunks), as well as the content of the chunk. The exploration of workarounds to overcome this limitation is out of scope for this study.
Although the Optimised Python DQ²S offered good performance processing 11 500 000 rows in 21.34 seconds, the available Python programming library does not offer a memory-efficient ingestion approach by itself [111], and even if an amendment is made, the instance would still be a sequential implementation of the algorithm that comprises I/O traditional processes, loading data from disk to main memory and back, thus, when reaching the point in which data could not longer fit into memory, the program will not be usable for processing [87].

To summarise, the important aspects of the results gathered in this experiment are as follows:

- 11 500 000 rows is the maximum dataset size that the Optimised Python DQ²S could cope with in the desktop machine (PC) utilised in the experimentation.
- The difference between 11 500 000 rows and the first dataset size that caused a failure is $\sim$79MB.
- Above 11 500 000 rows, a failure caused by the Pandas data reader, the most efficient ingestion function for Python language, presents an out of memory error because the program tries to utilise more memory than the one that is allocated for its process.
- The failure could be amended by creating a segmentation of the data, in such a way the reader can load chunks of data instead of trying to read the whole data at once, creating a reading and loading iterative process. This can be related to the MapReduce and big data processing paradigms of utilising data segmentation and lazy data evaluations.

R5. E5 - PySpark DQ²S Standalone mode in single machine experiment - PC

The PySpark DQ²S executed using a Standalone deployment in a single desktop machine, presented its best results when configured to run with 1 worker and 4 cores (1W4C), this is, 1 executor running on the machine, handling 4 tasks simultaneously (refer to Section 3.2.4 for Spark definition of its components). This was true for all of the three different dataset sizes, as shown in Figure 5.4 where the quickest runtime with 1W4C configuration was 7.38 seconds for 50 000 rows, 7.64 seconds for 100 000 00 rows, 11.77 seconds for 1 000 000 rows. With regards to scalability, the PySpark DQ²S execution without parallelisation (this is, 1 worker and 1 Core), the execution runtime increased 1.05 times
when the input datasets were increased from 50 000 rows to 100 000 rows, and the runtime was doubled when the input dataset was increased 10 times, in number of rows (from 50 000 rows to 1 000 000). The scalability rates presented with 1W1C configuration suggest it might be easy for this implementation to reach a point in which runtime is not longer acceptable for much larger dataset sizes, because the runtime increases at a high rate when the data volume increases too. However, for runtimes obtained with 1W4C configuration, when doubling the data the execution time rate increased 1.03 times, and 1.5 times when having 1 000 000 rows, which is a more acceptable rate considering it is below doubling the runtime, and below the rates that the sequential configuration (1W1C) presented.

Figure 5.4: Runtime registered in seconds for the PySpark DQ^2S processing 50 000, 100 000 and 1 000 000 rows in Standalone mode implemented in a single machine, where W refers to workers and C to cores in the Apache Spark definition. The marked line indicates the lower runtime obtained per dataset size.

For the three different dataset sizes (50 000, 1 000 000, and 1 000 000 rows), and the experiment configured to 1 worker (with 1 Executor each), the runtime and the runtime difference between configurations, decreased when adding more cores to the configuration. Table 5.3 shows the relation of the decreasing runtime when adding more cores to 1 worker for datasets of 50 000, 100 000 and 1 000 000 rows, where results show scalability more clearly when larger datasets are processed, this is because when processing larger data sets, runtime is dominated by data processing related tasks, rather than system set up tasks. This
behaviour can probably explain the reason why the results obtained from experiments where configurations with more than one worker was used showed that, when the number of workers was set to 2 there was a runtime decrease between a configuration with 1 Core (2W1C) and 2 cores (2W2C), but the runtimes were still higher than the ones obtained with 1 worker (1W1C, 1W2C, 1W3C and, 1W4C). Another observation related to this behaviour was made when adding 3 and 4 workers, with 1 core allocated per worker (3W1C and 4W4C), where the runtimes were higher than the configurations with 1 and 2 workers, except for the larger datasets. For executions with 50 000 and 100 000, the worst runtime was obtained from the configuration with 4 workers and 1 core (4W1C). However, for 1 000 000 rows, the 4W1C configuration did not incur the highest runtime. The obtained runtime being not as good as the best runtime from executions with 1W4C configurations seem to suggest that the best configuration for executing a job should be decided based on the amount of work needed by the application and the dataset size, requiring a debug and fine tuning work to find the optimum configuration depending on the processing intensity of the Spark application given by the operators needed, or the dataset sizes within a certain use case. Even though Spark runs in a single machine in this experiment, it can be noticed that Spark able to offer the benefits that can be obtained from parallelism, tackling runtime and memory management, as well as scalability, with an added advantage of not requiring explicit parallelism to obtain such benefits.

<table>
<thead>
<tr>
<th>Seconds decreased</th>
<th>Number of rows</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50 000</td>
<td>100 000</td>
</tr>
<tr>
<td>Seconds decreased from runtime with one more core &amp; Seconds decreased from base runtime</td>
<td>10.57</td>
<td>11.20</td>
</tr>
<tr>
<td></td>
<td>2.29</td>
<td>2.48</td>
</tr>
<tr>
<td></td>
<td>0.51</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>0.39</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>3.19</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Table 5.3: Relation of seconds decreased from different configurations using 1 worker (W) and 1 to 4 cores (C) with the Standalone mode in a single machine.
5.1. EXPERIMENTS RESULTS

R6. E6 - PySpark DQ²S Local mode in single machine experiment - PC

Figure 5.5: Runtime registered in seconds for the PySpark DQ²S processing 50 000, 100 000 and 1 000 000 rows in Local mode implemented in a single machine. The marked line indicates the lower runtime obtained per dataset size.

Figure 5.6: Runtime registered in seconds for the PySpark DQ²S processing 50 000 rows in Local mode implemented in a single machine.
Figure 5.7: Runtime registered in seconds for the PySpark DQ^2S processing 100 000 rows in Local mode implemented in a single machine.

Figure 5.8: Runtime registered in seconds for the PySpark DQ^2S processing 1 000 000 rows in Local mode implemented in a single machine.
5.1. EXPERIMENTS RESULTS

### Times Slower: PySpark local[1] vs PySpark local[8]

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Runtime (seconds)</th>
<th>Times Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 000</td>
<td>6.92</td>
<td>4.62</td>
</tr>
<tr>
<td>100 000</td>
<td>7.59</td>
<td>4.99</td>
</tr>
<tr>
<td>1 000 000</td>
<td>17.69</td>
<td>9.37</td>
</tr>
</tbody>
</table>

Table 5.4: Times Slower: Relation between the runtime results of the PySpark instance with 1 (sequential) and 8 threads (best time). Where local[1]<local[8] indicates the PySpark DQ^2S in local mode with 8 threads is slower than the same implementation with 8 threads as many times as indicated.

Without any change to the PySpark code, but setting the Spark master to a Local mode (refer to Section [4.3.1](#)) for details of the Local mode concepts) the execution times were below 10 seconds for 50 000 and 100 000 rows, and below 20 seconds for 1 000 000 rows. The best runtime obtained from the 50 000 and 100 000 rows dataset was obtained when setting 8 threads, with 4.62 seconds for 50 000 rows, and 4.99 seconds for 100 000 rows, whereas for the biggest dataset size in this experiment, the best runtime was 8.73 seconds, obtained with 7 threads, as shown in Figure [5.5](#). The runtimes obtained decreased by milliseconds as more threads were added to the configuration each time, having the biggest observable reduction in Figure [5.5](#) for the set of 1 000 000 rows, where almost 6 seconds were reduced from a sequential execution (this is, when using local[1]) to 2 threads.

The ratio of how much slower is the sequential configuration (local[1]), compared to the one that offered the best performance in runtime terms (the configuration with local[8]), was around 1.5 and 1.9, as shown in Table [5.4](#), these results suggest that it is possible to reduce the runtime by almost half by adjusting the parametrisation. In terms of scalability, for the sequential configuration, which is the one set with 1 thread, the 100 000 rows execution offered 1.09 times slower results in seconds compared to the runtime presented by half of the dataset size, and 2.5 times slower for 1 000 000 rows compared to the times obtained with 50 000 rows. For the times gathered with 8 threads, the scalability results are 1.08 times slower for twice of the data volume from the smaller dataset size,
and 2.02 times slower for 1 000 000 rows. There appears to be not much difference on the scalability offered by either 1 or 8 threads when the data is relatively small (e.g. for the 50 000 rows datasets processing), and neither the runtimes are hugely different with distinct number of threads. It seems to be the case that Spark implemented in Local mode presents parallelism, but the execution of the whole processes on the same JVM causes Spark to utilise all of the resources available for one JVM, lacking a spawn of resources among processes. This resources spawning cannot happen because of the Local mode configuration that processes everything on the same JVM, thus, utilising the same configuration with few differences, the opposite to a Standalone mode where different configurations cause more impact on the internal Spark set-up. (refer to Section 4.3 for more details of the Standalone mode) This is why there are not hugely differences when setting more threads to be handled in Local mode. Figure 5.6, 5.7 and 5.8 show the runtimes registered for each of the datasets utilised in this experiment, where it is observable that the biggest runtime decrease is made when changing from processing one thread to divide the tasks in two threads.

R7. E7 - PySpark DQ²S Local mode maximum capacity experiment - PC

Figure 5.9: Runtime registered in seconds for the PySpark DQ²S processing 11 500 000 and 35 000 000 rows in Local mode implemented in a single machine. The marked line indicates the lower runtime obtained per dataset size.
5.1. EXPERIMENTS RESULTS

Figure 5.10: Runtime registered in seconds for the PySpark DQ²S processing 11 500 000 rows in Local mode implemented in a single machine.

Figure 5.11: Runtime registered in seconds for the PySpark DQ²S processing 35 000 000 rows in Local mode implemented in a single machine.

This experiment aimed at discovering the limitations of the Apache Spark Local mode implementation, by searching for the maximum data size that could be handled successfully utilising the desktop machine (PC). This experiment followed the procedure described in Section 4.4, Experiment E7, where it was obtained a successful result for up to 35 000 000 rows. When trying to process datasets with more than 35 000 000 rows, a java.lang.NegativeArray SizeException with CsvRelation error was produced when the show action
in the PySpark code triggered the whole processing of the Timeliness query. Executions of the Timeliness query for input data sets with more than 35 000 000 rows, or 3.3GB for the orderT dataset and 2.5GB for the statusTimelinessQR dataset, failed. The error might appear to be related to a difference in sizes with the schema inferred by PySpark and the rows information, or because of a faulty dataset with missing data, but it is important to revise the complete error log because there are multiple different types of error associated with the Negative Array Size Exception in Spark, making the finding of a diagnosis of the error a difficult task. For example, this error could be caused by any of the following: (i) because the buffer used to store more data than it is allowed to keep, therefore, unable to fit all of the data required, (ii) because of the serialization issue which affects the Python socket (called py4j; refer to Section 3.2.4 for details of py4j socket) when dealing with strings larger than 2GB, (iii) because an integer overflow caused by an array handling large messages, or (iv) because of large process within the code, which is the case of an error that can be produced when joining. The reason of the failure with large datasets in this case is the number (iv). The error could possibly be identified by debugging the log displayed by PySpark, but the task of diagnosing the error from the log would still be a challenging one, because Spark-generated error lots present significant similarities. 

Being a relatively new technology, the causes of run time exceptions are not always clear, and in the majority of cases, an exception could be raised by an implementation error by a programmer, or because some resource has exhausted. More than 1500 issues of that nature have been documented as unresolved, with new issues coming each month. While workarounds are suggested by the Spark community, work is still needed. Further research about Spark’s general usability for common processes, as well as the implementation of amendments to the default functions, are out of the scope of this study.

This experiment comprised the configuration of more cores than the number of CPU cores available on the machine. The hypothesis to be tested was that fixing a maximum number of cores equal to the maximum available CPU cores offers the best runtime compared to any other number of cores configured. The performed tests used from 1 to 10 cores, 12 and 16 cores to investigate what is the impact of increasing the number of cores on relation to the number of available CPU cores. Results for the PySpark DQ²S support the practice of aligning number of cores to the number of CPU cores available, based on the
fact that the best runtime was produced when the number of cores was set to 8, and for higher number of cores, the runtime began to be higher, as shown in Figure 5.9. Figures 5.10 and 5.11, show the results in separate obtained for the processing of 11 500 000 and 35 000 000 rows, where it can be observed that as more threads are added (refer to Section 4.3.1 for details of the correlation between cores and threads), the execution runtime shows a very similar curve either with the 11 500 000 rows datasets or the datasets of 35 000 000 rows, which suggests the data volume does not change the impact of the number of cores configured, this is, for any size, the number of cores configured should be the number of CPU cores available.

R8. E8 - PySpark DQ^2S cluster Standalone mode experiment - Cluster

<table>
<thead>
<tr>
<th>Dataset size (rows)</th>
<th>OrderT</th>
<th>statusTimelinessQR</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 000 000</td>
<td>3.3GB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>70 000 000</td>
<td>6.6GB</td>
<td>5GB</td>
</tr>
<tr>
<td>105 000 000</td>
<td>9.9GB</td>
<td>7.4GB</td>
</tr>
</tbody>
</table>

Table 5.5: Size in GB for the datasets sizes processed on the cluster.

Figure 5.12: Runtime registered in seconds for the processing of 35 000 000, 70 000 000, and 105 000 000 rows with PySpark DQ^2S in Standalone mode used in a cluster, utilising from 1 to 4 nodes (worker nodes). Each node was configured to contain 1 worker process with 8 executors, each executor with 2 cores and 32GB of available memory.
This experiment is the first set of tests performed in the cluster (DPSF) (described in Section 4.2). The execution was done with a Standalone deployment of Apache Spark, following the procedure described for Experiment 8 in Section 4.4. The smaller input datasets processed in this experiment have 35,000,000 rows, based on the maximum size successfully handled by the PySpark in a single machine, under the Local mode. The results from experiments performed using Spark on Local mode, namely Experiment 7 (R7 in this section), are utilised as reference to set the initial dataset size used for this experiment, rather than the results from Spark on Standalone mode showed in Experiment 5 (R5 in this section), as the reader would probably expect, because, unlike the Spark on Standalone mode running on a single machine, the Local mode allows a single machine to utilise Spark’s capabilities without trading on performance due to the limitation of not having a cluster, as discussed in Section 4.3.1. To represent an intermediate workload in this experiment, input data sets of 70,000,000 rows were chosen, doubling the size of the input data sets used in the starting workload. Finally, when tripling the starting dataset sizes, input datasets of 105,000,000 rows represent the larger workload in this experiment. The relation of number of rows and the size in GB is shown in Table 5.5.

The best runtime results for the three input data sizes described above were obtained when 4 worker nodes were used in the execution of the Timeliness query over the DPSF cluster, as Figure 5.12 shows. Note from the figure the runtimes obtained for executions of sets with 35,000,000 rows was done in few more seconds than a minute when utilising one node (71.62 seconds), and less than half a minute with 4 nodes (28.65 seconds). The medium workload required few less than two minutes when processed with one node (115.20 seconds), and still less than a minute using 4 nodes (46.01). The larger dataset utilised, took few more than 2 minutes as runtime with the Timeliness query processed in one node (145.85 seconds), and less than a minute when using 4 nodes (58.61 seconds). This confirms that, for executions on a cluster infrastructure, Spark in Standalone mode supports a DQ²S implementation producing acceptable processing capabilities on large datasets, considering that few minutes is a fair time for handling millions of rows; results also show scaling-out techniques behave well with Spark, offering parallelism reflected on quicker results each time a node is added, where each node is not adding more resources to the process by having larger capabilities (scale-up) but by adding more resources with the usage
of more nodes (scale-out).

The quick runtimes reflected on the results might be possibly achieved due to Spark’s abstraction to a parallel collection model, known as RDD (refer to Section 2.3.1 for details of RDDs), which partitions the data across nodes, and supports reutilisation without requiring data to be replicated [151], providing CPU intense applications, rather than I/O heavy load [100], which causes Spark applications to occupy main memory and CPU capabilities on an higher utilisation than the required interaction with storage.

R9. E9 - Optimised Python DQ$^2$S cluster experiment - Cluster

<table>
<thead>
<tr>
<th>Dataset size (rows)</th>
<th>Dataset size</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 000 000</td>
<td>3.3GB</td>
<td>2.5GB</td>
<td>OK</td>
</tr>
<tr>
<td>40 000 000</td>
<td>3.7GB</td>
<td>2.8GB</td>
<td>OK</td>
</tr>
<tr>
<td>50 000 000</td>
<td>4.7GB</td>
<td>3.5GB</td>
<td>FAIL</td>
</tr>
<tr>
<td>70 000 000</td>
<td>6.6GB</td>
<td>5GB</td>
<td>FAIL</td>
</tr>
<tr>
<td>105 000 000</td>
<td>9.9GB</td>
<td>7.4GB</td>
<td>FAIL</td>
</tr>
</tbody>
</table>

Table 5.6: Relation of the dataset sizes and results when executing the Optimised Python DQ$^2$S in the cluster for Experiment 9.

Running any large job on a cluster is always better than running the same job on a (single) commodity machine, having a big data platform available from the cluster or not. However, without the platform, the parallelism exploitation has to be encode within the job. The Optimised Python DQ$^2$S was executed in the DPSF cluster, providing a view of the performance that can be obtained for DQ$^2$S without a big data framework. The Optimised Python version was intended to be tested in a cluster environment aiming at getting comparable results to the ones obtained with the PySpark DQ$^2$S instance on a sequential configuration in a cluster environment, were both implementations would use only one node, hence, the same dataset sizes were tried (35 000 000, 70 000 000 and 105 000 000 rows). Results were successful only for the dataset with 35 000 000 rows, with a runtime of 83.22 seconds, as shown in Table 5.6. Larger datasets
produced an out of memory failure, also obtained in Experiment 4 (R4 in this chapter), caused by insufficient memory space for the execution of the job.

To explore the limits of the instance on the DPSF cluster, a 40 000 000 rows input datasets were processed with success taking around 112 seconds, and 50 000 000 rows, or 8GB of total input was the first size tested that produced a failure. The execution of this instance utilised a single node on the DPSF cluster, with 512GB of main memory, a much higher capacity compared to the input of 8GB. The limits on memory and the failure that appeared is dependent on the memory available for the Python process, specifically for the stack allocated by the OS, therefore, this limit is also dependant on RAM and OS, which explains the reason of a bigger limit for the Optimised Python on a cluster, compared to the limit of 2GB obtained in a desktop machine.

### 5.2 Algorithm Comparison

This section presents insights obtained from analysing the results shown in Section 5.1. Appendix C Section C.2 shows the raw results obtained for the calculation of the performance metrics.

R10. **C1 - The original DQ²S Java program (E1) compared with its equivalent DQ²S developed using Python (E2).**

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Runtime (seconds)</th>
<th>Times Slower Python &lt;Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 000</td>
<td>397.59</td>
<td>4612.62</td>
</tr>
<tr>
<td>100 000</td>
<td>1593.84</td>
<td>19019.20</td>
</tr>
<tr>
<td>1 000 000</td>
<td>160624.02</td>
<td>1974593.50</td>
</tr>
</tbody>
</table>

Table 5.7: Times Slower: Relation between Original Java and Python runtime results. Where Python <Java indicates the Python DQ²S is slower than the Java implementation as many times as indicated.

The results from the original DQ²S Java program (R1 in this section) and its
equivalent algorithm implemented in Python language (R2 in this section), presented different runtimes, where the Java program was apparently 11 to 12 times faster than the Python version, as shown in Table 5.7. There are several benchmarking studies and discussion which of Java or Python is faster [141, 85, 149, 94, 108, 153, 135]. Most of the studies present Python as more computationally expensive than Java, but they all agree that it is naive to state that a programming language is more efficient than another for all cases and situations, since the runtimes depend on several parameters, such as optimization level, the algorithm complexity, the verbosity of the code, the compiler utilised, and even the hardware involved.

For both the Python and Java DQ^2S implementations runtime quadrupled when input data sizes were increased from from 50 000 to 100 000 rows, and ~100 times when increasing the dataset size to 1 000 000 rows. These rates show both instances present a similar scalability pattern but the Java version offers better performance, which is consistent with the common view of Java being faster than Python, due to the static nature of Java and the fact that it is a compiled language rather than interpreted. Python has the option to run over different types of compilers, such as Jython, Skulpt, IronPython, ActivePython, among others, which could offer different results, and provide a research area towards possibly leveraging performance of DQ^2S on pure Python code, nevertheless, this research aims at other objectives focused on big data frameworks.

R11. **C2 - The Python DQ^2S program (E2) compared with the Optimised Python version of it (E3).**

The results of the sequential Python DQ^2S, presented in R2 of this section, and the Optimised Python version (also sequential) depicted in R3, show a significant difference between the obtained runtimes:

The non-optimised Python implementation of DQ2S incurs, as expected, longer runtimes compared to the Optimised implementation. The Optimised Python DQ^2S presented better scalability results than the Python DQ^2S. this can be interpreted considering that the Optimised Python DQ^2S required 1.7 times more seconds when the data volume was doubled from 50 000 rows to 100 000 rows, and 9.4 times more seconds when increasing the dataset size from 100 000 rows to 1 000 000, whereas the Python DQ^2S required 4.1 and 103.8 times more seconds respectively. Those results can be interpreted as providing better scalability
presented when utilising the Optimised Python DQ\textsuperscript{2}S.

The Optimised Python DQ\textsuperscript{2}S compared to the Python DQ\textsuperscript{2}S, reflects the usage of the Pandas library utilised has an impact on the DQ\textsuperscript{2}S instance, despite the programming language (Python) being the same for both instances, based on the results presented. The Python Pandas library offers (as any other library for any other language) an advantage over custom code developed, with well tested, robust, optimised for runtime functions and methods, mainly because the code is open source and so it is constantly being improved and adapted to the user’s needs, designed to efficiently manipulate data in tabular form, and support common data analysis tasks \cite{30}. Pandas is part of the spectrum of scientific libraries available for Python development, and follows Python’s orientation towards leveraging productivity and performance. Pandas library allows the usage of a special object designed for data manipulation called dataFrame, which supports a faster access to properties and functions to apply over a set of tabular data, compared to a data structure such as arrays or a group of arrays to achieve the same objectives. The library also provides readers and writers designed to handle different types of data formats, such as CSV for the case of this study. Finally another important characteristic of the library that allows the Optimised Python DQ\textsuperscript{2}S to be faster is that Pandas has SQL-like methods (e.g. merges, joins, filters, etc.) that provide ease of implementation as well as an optimised execution.

C2.1 - The sequential DQ\textsuperscript{2}S instances comparison

![Runtime (seconds) for 50 000 rows: Sequential DQ2S instances](image.png)

Figure 5.13: Comparison of the runtime in seconds obtained when processing 50 000 rows with the non-big data instances.
Figure 5.14: Comparison of the runtime in seconds obtained when processing 100 000 rows with the non-big data instances.

Figure 5.15: Comparison of the runtime in seconds obtained when processing 1 000 000 rows with the non-big data instances.
CHAPTER 5. EVALUATION OF EXPERIMENTAL RESULTS

Times Slower: Python vs Optimised Python

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Runtime (seconds)</th>
<th>Times Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Python</td>
<td>Optimised Python</td>
</tr>
<tr>
<td>50 000</td>
<td>4612.6</td>
<td>0.1</td>
</tr>
<tr>
<td>100 000</td>
<td>19019.2</td>
<td>0.2</td>
</tr>
<tr>
<td>1 000 000</td>
<td>1974593.5</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Table 5.8: Times Slower: Relation between Python and Optimised Python runtime results. Where Python < Optimised Python indicates the Python DQ_{2}S is slower than the Optimised Python implementation as many times as indicated.

Times Slower: Original Java vs Optimised Python

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Runtime (seconds)</th>
<th>Times Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Java</td>
<td>Optimised Python</td>
</tr>
<tr>
<td>50 000</td>
<td>397.59</td>
<td>0.11</td>
</tr>
<tr>
<td>100 000</td>
<td>1593.84</td>
<td>0.19</td>
</tr>
<tr>
<td>1 000 000</td>
<td>160624.02</td>
<td>1.78</td>
</tr>
</tbody>
</table>

Table 5.9: Times Slower: Relation between Original Java and Optimised Python runtime results. Where Java < Optimised Python indicates the Java DQ_{2}S is slower than the Optimised Python implementation as many times as indicated.

At this point in the analysis, the results make it possible to select the best sequential instance to be used as baseline with evidence. In the following, a summary of the results presented in R1, R2, R3, R10 and R11 of this chapter is provided.

The original Java DQ_{2}S instance was faster than the Python version. These two instances were developed without usage of any specialised library (such as Pandas), and utilised the original structure of the DQ_{2}S set of algorithms, described in Section 3.1.4. Whereas the Optimised Python version is implemented with heavy use of a scientific tabular data manipulation library. Figures 5.13, 5.14, and 5.15 provide a visual comparison of the three sequential, non-big data DQ_{2}S instances (Original Java, Python and Optimised Python) for each of the basic datasets (50 000, 100 000 and 1 000 000 rows). The presentation of these results
are made in separated graphs to ease the comparison due to the different scales that each set of results comprises. Regarding runtime, the Optimised Python, presents the best results by being at least 3 700 times faster than the Java and the Python DQ²S. Tables 5.8 and 5.9 present the relation of times slower of the Java and Python sequential instances in comparison to the Optimised Python DQ²S.

R12. C3 - The Optimised DQ²S Python program (E3) compared with the PySpark program in Local mode (E6).

![Runtime comparison graph](image)

Figure 5.16: Comparison of the runtime in seconds obtained when processing 50 000, 100 000 and 1 000 000 rows with the best non-big data instance (sequential) and its implementation with a big data instance using 1 and 8 threads.
### TIMES SLOWER: OPTIMISED PYTHON VS PYSPLARK LOCAL[1]

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Optimised Python</th>
<th>PySpark local[1]</th>
<th>Times Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 000</td>
<td>0.11</td>
<td>6.92</td>
<td>65.2</td>
</tr>
<tr>
<td>100 000</td>
<td>0.19</td>
<td>7.59</td>
<td>39.8</td>
</tr>
<tr>
<td>1 000 000</td>
<td>1.78</td>
<td>17.69</td>
<td>9.9</td>
</tr>
</tbody>
</table>

Table 5.10: Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 1 thread. Where local[1] < Optimised Python indicates the PySpark DQ²S in local mode with 1 thread is slower than the Optimised implementation as many times as indicated.

### TIMES SLOWER: OPTIMISED PYTHON VS PYSPLARK LOCAL[8]

<table>
<thead>
<tr>
<th>Dataset Size (rows)</th>
<th>Optimised Python</th>
<th>PySpark local[8]</th>
<th>Times Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 000</td>
<td>0.11</td>
<td>4.62</td>
<td>43.5</td>
</tr>
<tr>
<td>100 000</td>
<td>0.19</td>
<td>4.99</td>
<td>26.1</td>
</tr>
<tr>
<td>1 000 000</td>
<td>1.78</td>
<td>9.37</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Table 5.11: Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 8 threads. Where local[8] < Optimised Python indicates the PySpark DQ²S in local mode with 8 threads is slower than the Optimised implementation as many times as indicated.

This comparison includes the results obtained from the Optimised DQ²S Python program, which is a sequential implementation, and the results from the PySpark instance, comprising the Spark execution without parallelism (local[1]), and the best runtime obtained with 8 threads (local[8]). Spark applications, such as PySpark DQ²S can run on a single machine with implicit parallelism, dividing the tasks and data to complete the job among several threads, and can also complete a job as a sequential instance with a configuration that sets the PySpark DQ²S to run with one thread.
The aim of comparing the sequential results from the Optimised Python DQ\(^2\)S against the results obtained from executions of the PySpark implementation using a single thread is to analyse the impact of the big data framework caused solely by its usage, whereas, the aim of comparing the sequential results from the Optimised Python DQ\(^2\)S against the PySpark DQ\(^2\)S obtained with 8 threads is to analyse the best results obtained, with the purpose of avoiding a disadvantage by default for PySpark by not considering the best capability offered by it.

The Optimised Python was quicker when processing each of the three dataset sizes tested, compared against both the PySpark results running with both one and eight threads (local[1] and local[8]), as shown in Figure 5.16. Note that the PySpark instance was at least 5 times slower than the Optimised Python, as reflected in Tables 5.10 and 5.11. The runtime results could indicate that the Optimised Python instance is a better option for big data quality processing than the PySpark implementation. However, in terms of scalability, the PySpark instance presents a smaller gap when the dataset size is increased, and, as commented in R4 and R6 of Section 5.1, scalability for the Optimised Python when increasing the volume of data 20 times (from 50 000 000 to 1 000 000 rows) is 16 times more seconds whereas the scalability for the PySpark with 8 threads is 1.87 times more seconds for the same case. An overhead is observed in the PySpark instance, that is not present in the Optimised Python program, this is to set up the application processes within the framework system, to initiate the executor process and for the task scheduler to set the tasks, generate the execution plan, allocate the resources, coordinate the results, and finally turn down the deployment. Runtime is the focus of parallel performance but it is not isolated from scalability, in practical terms, there is not much gain having the quickest algorithm if it is not capable of coping with big data. This is a worthy trade-off, seeking for a fair speed but great scalability, and in this sense, PySpark has performed better than the Optimised Python, having both its default parameters, with not much tuning involved, because effort and productivity could also take part when selecting the best technology and technique to process big data if tuning is involved.
R13. C4 - The maximum number of rows processed by the Optimised DQ²S Python program (E4) compared with the PySpark program in Local mode (E7).

![Runtime (seconds) for 11 500 000 rows: Best sequential instance vs big data implementations in single machine.](image)

Figure 5.17: Comparison of the runtime in seconds obtained when processing 11 500 000 rows with the best non-big data instance and the PySpark one with 1 and 8 threads.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11 500 000</td>
<td>21.35</td>
<td>132.28</td>
<td>43.71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Times slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>local[1]&lt;Optimised Python</td>
<td>6.20</td>
</tr>
<tr>
<td>local[8]&lt;Optimised Python</td>
<td>2.04</td>
</tr>
<tr>
<td>local[1]&lt;local[8]</td>
<td>3.02</td>
</tr>
</tbody>
</table>

Table 5.12: Times Slower: Relation between the runtime results of the Optimised Python and PySpark with 1 and 8 threads for 11 500 000 rows. Where the notation implementation <implementation indicates the left implementation is slower than the implementation on the right as many times as indicated.
5.2. ALGORITHM COMPARISON

Figure 5.18: Ideal Speedup compared against the speedup results obtained from the PySpark DQ\textsuperscript{2}S Local mode in a single machine.

Figure 5.19: Ideal Efficiency compared against the speedup results obtained from the PySpark DQ\textsuperscript{2}S Local mode in a single machine.
CHAPTER 5. EVALUATION OF EXPERIMENTAL RESULTS

Figure 5.20: Performance Metrics for Parallel Algorithms: Values calculated for 11 500 000 rows processing for the PySpark Local mode DQ²S instance. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations.

The PySpark DQ²S implementation with Local mode was tested on a maximum capacity experiment because this is the mode that best fits to work within a single multicore machine, based on the results of Experiment E7 (presented in R7 in this chapter) and compared to the results from the Standalone mode presented in R5 of this chapter. The results of the PySpark DQ²S provide an analysis of how Local mode would behave if the implementation was going to be used without a cluster. This study moves later on cluster results to have a wider overview of the big data framework usage and impact on the Timeliness query.

The Optimised Python DQ²S processed successfully up to 11 500 000 rows. Figure 5.17 shows the runtime results of the PySpark instance obtained for 11 500 000 rows with a PySpark sequential configuration (local[1]) and the results obtained with the configuration that provided the quickest results among all of the configurations tested (local[8]). Comparing the results obtained from the Optimised Python DQ²S, the PySpark with 1 thread, and the PySpark with 8 threads, the Optimised Python DQ²S presented the quickest runtimes. However, the PySpark DQ²S implementation could cope with up to 35 000 000 rows, which is more than the double of the volume handled by the Optimised Python,
and furthermore, the best PySpark result is only 2.04 times slower than the Optimised Python. Note that this is a lower value than the obtained from 100 000 rows processing for the same instances, where the PySpark in Local mode was 5 times slower than the Optimised Python (as described in R12). The comparison of the number of times slower shows a decreasing rate as the volume size increases. These results suggest that, if the Optimised Python could cope with more than 11 500 000 rows using the algorithm without any amendment or further configuration, there could be a dataset size in which the PySpark runtime will not be slower than the Optimised Python could offer. Table 5.12 presents the relation of how many times slower was the PySpark DQ²S compared to the Optimised Python DQ²S implementation, and also shows a comparison between the PySpark results with 1 and 8 threads that shows local[8] configuration runs on 67% less time than the local[1] configuration when processing 11 500 000 rows. These results show the PySpark DQ²S in local mode reaches a time reduction of 17% more, if compared to the decrease of half of the runtime from local[8] to local[1] configuration when processing 1 000 000 rows (as presented in R12). Results indicate that having larger dataset sizes could present a higher decrease when setting the optimum parameters (e.g. configuring the job to work with 8 threads in this case).

The parallelisation made by the PySpark DQ²S instance in Local mode, as well as its constraints to obtain a better runtime, can be analysed based on its speedup and efficiency results. Linear scalability provides the ideal efficiency and ideal speedup [11, 148]. However, in real practice, due to overheads and different degrees of parallelisation on the algorithms, linear scaling is not common. Relative Speedup and Relative Efficiency in Figure 5.20 show the correspondent values to the behaviour of the PySpark DQ²S across itself with different number of threads, and Real Speedup and Real Efficiency show the values with respect to the fastest sequential DQ²S implementation (The Optimised Python DQ²S). The results on Speedup reflects the best performance is obtained when the number of threads (cores in Apache Spark definitions) is set to 8, and a good performance with either 7 or 8 threads with respect to the Optimised Python instance. Figure 5.18 shows the speedup obtained is below the ideal speedup, which is a common result [114]. This indicates the speedup obtained is acceptable for common practices. Figure 5.20 shows the greater speedup is achieved with 8 cores, and 7
cores offers also a high speedup. After 8 cores configured, speedup decreases although values indicate the speedup is higher than the one that could be obtained from configurations with less than 8 cores (threads). Efficiency results, shown in Figure 5.19 and Figure 5.20, are also considered normal values, where efficiency values are between ranges of 0 and 1 [114]. The results with a 9 decimal precision are shown in Appendix C, Section C.2.

R14. C5 - The PySpark program in Standalone mode (E5) compared with a PySpark Local mode (E6), both in a single machine.

This comparison comprises only the equivalent results where both modes, Local and Standalone, handle the same number of tasks simultaneously. Comparisons between results obtained with more than 1 worker were not compared to the Local mode, even though the Local mode handles, for example 4 threads, the processing is not equivalent to the Standalone mode configured with 2 workers and 2 cores, or 4 workers and 1 core. The reason of not comparing the local mode results with the Standalone with more than 1 worker is that a configuration with several workers faces a disadvantage by default due to an overhead produced by the deployment of more JVMs, as discussed in Section 4.3.1.

Figure 5.21: Runtime in seconds for the PySpark DQ^2S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 50 000 rows.
5.2. ALGORITHM COMPARISON

Figure 5.22: Runtime in seconds for the PySpark DQ²S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 100 000 rows.

Figure 5.23: Runtime in seconds for the PySpark DQ²S with Standalone and Local mode, handling from 1 to 4 tasks simultaneously for 1 000 000 rows.
Figure 5.24: Difference in seconds for 50,000, 100,000 and 1,000,000 rows when executing the PySpark DQ\textsuperscript{2}S with Standalone and Local mode, where Local mode was the quickest for all of the three dataset sizes.

The PySpark DQ\textsuperscript{2}S requires more time to process the data on a single machine when deployed in Standalone mode, on a difference of 2.11 to 3.65 seconds more than the offered by a Local mode. Figures 5.21, 5.22 and 5.23 show a comparison between these two Apache Spark modes for 50,000, 100,000, and 1,000,000 rows when processing the Timeliness query. The difference on the runtimes is not huge, the seconds added on the Standalone mode are a sum of the time required to create the worker JVM, and the master JVM, as well as the communication that occurs through isolated processes, whereas the Local mode requires only the driver JVM which acts as worker and tasks scheduler, and all the processing occurs within the same process. The greatest difference was obtained when processing the smaller dataset on a sequential configuration, as shown in Table 5.24 and the smaller difference was produced with the largest dataset and a configuration of three tasks; overall the highest time reduction is observed on the sequential configuration, this is because of the mentioned overhead on the Standalone mode, where Local mode incurs on some overhead when handling more than 1 task, however, the overhead still is lower than the Standalone one. As the dataset size increases, the gap between the two modes decreases because the processes become more intensive and the balancing of work among the tasks makes the overhead worth with respect to total runtime.
5.2. ALGORITHM COMPARISON

Figure 5.25: Speedup and Efficiency individual heat map for the Standalone and Local mode PySpark DQ\(^2\)S. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations within each set of tables.

Figure 5.26: Speedup and Efficiency global heat map for the Standalone and Local mode PySpark DQ\(^2\)S. Where green represents the higher speedup and efficiency obtained, yellow and orange represent intermediate speedup and efficiency values, and red indicates the configurations that obtained the lowest speedup and efficiency considering the values calculated for all of the configurations of all the tables.

Speedup and efficiency for 50 000, 100 000, and 1 000 000 rows, either with Local or Standalone mode presented behaviours similar to the shown in Figures 5.18 and 5.19. Figure 5.25 shows a heat map considering the values on each
column, where green colour indicates higher values, yellow signalises average values, and red presents the lower value per column; in terms of speedup and efficiency, higher values are better. Real and relative speedup increase with the number of cores, and real and relative efficiency decrease as the number of cores augment, this behaviour is observed in all of the cases presented. Figure 5.26 shows a global heat map, offering a view of the configuration and mode with the highest speedup and most efficient. Green colour frames the configuration with the higher values. The figure shows the higher speedup and efficiency were obtained with the larger dataset size, this was true for both Apache Spark modes, but comparing Standalone and Local mode, the latter presented the highest values, thus, can be considered as the most efficient, with higher speedup $DQ^2 S$ implementation.

R15. C6 - The PySpark program in Local mode single machine (E7) compared with a PySpark in a cluster using Standalone mode (E8).

![Runtime (35 000 000 rows): PySpark DQ²S best single machine vs Standalone mode in cluster](image)

Figure 5.27: Runtimes obtained from the execution of the Timeliness query with 35 000 000 rows datasets, for PySpark in a single machine with Local mode and a Standalone mode in a cluster (DPSF).

The Apache Spark Local mode might seem a better approach to use than a cluster mode, but in a single machine the Standalone mode is highly limited by the resources available, and considering that Spark was developed with a focus on clusters, results on performance in such environment is crucial to answer to which extent Apache Spark is a suitable for scalability requirements with $DQ^2 S$. The Python Spark executions with Local mode on a single machine had a limit of usage on 35 000 000 rows, with an average best runtime of 123.25
5.2. ALGORITHM COMPARISON

A cluster offers more possibilities for Apache Spark, because the resources are more diverse, thus, Spark can be widely leveraged; the number of cores and amount of memory offers more combinations to test and seek for the better parametrisation. The execution on the DPSF cluster processed 35 000 000 rows in 72.61 seconds for the highest time, on one node, and 28.64 seconds for the best time, on four worker nodes. Figure 5.27 shows the best result using the DPSF cluster was obtained with four nodes, and the worst time among the results presented was given by the Local mode implementation on the single machine. Results are not completely comparable due to the different environments and configurations involved, but the results presented can be explored and used to notice Spark’s capabilities on a higher resource environment. The results obtained with the Local mode on a single machine are useful as reference to quantify Spark’s capabilities. An important result is that Spark could cope with the double and triple of the dataset size with an increment of ∼1 minute for the larger dataset tested on the higher runtime obtained, a low amount of time, and with an instance that required no change at all to be able to produce the results.

R16. C7 - The PySpark program in cluster mode (E8) compared with a Optimised Python version executed on the cluster (E9).

Figure 5.28: Runtime in seconds for 35 000 000 rows processed in the DPSF cluster with the Optimised Python DQ²S and the PySpark instance in Standalone mode. Quicker results with more capacity of processing are not a surprise with considerably more main memory and CPU capacity, for that reason, the Optimised
CHAPTER 5. EVALUATION OF EXPERIMENTAL RESULTS

Python, which presented good runtimes and scalability up to a limit on memory, was also tested on the DPSF. This testing was done to conclude the analysis on comparing the best results from Apache Spark, with the best possible sequential results among all the DQ$^2$S instances developed. This experiment showed for the first time on this study, an Optimised Python version slower than the instance being compared to, in this case the PySpark DQ$^2$S. The difference was measured only when PySpark utilised one node, to ensure fairness and lack of bias in the comparison, since the Optimised Python instance is not able to work over more than one node. PySpark performed 13% faster than the sequential instance, as shown in Figure 5.28. Results show the Optimised Python is capable of providing a scalable solution, but the instance is limited at granting more processing capabilities, recalling the sequential instance could not cope with more than 40 000 000 rows, whereas PySpark worked successfully without further configurations when processing 105 000 000 rows datasets.

R17. Answers to the Research Questions

1. **What are the performance and scalability advantages that big data frameworks provide to data quality management tasks?**

Data quality management is part of a higher scope of information management activities within organisations and projects, comprising tasks related to the acquisition, extraction, cleansing, maintenance, and storage of data \[\text{[66, 50]}\] with the involvement of people, processes, and technology \[\text{[72, 50]}\]. Big data frameworks, if implemented, are incorporated into the architecture and technical facilities of the systems on which data quality tasks rely, and a novel aspect of them is that its development also brought a wider portfolio of database platform options for all kinds of data, specially focused on big data characteristics, such as variety, volume, and velocity, including a support to the data quality environment regarding processes and technology, and aiding people involved. Big data frameworks provide performance and scalability advantages only when the tasks processed within a framework usage requires a large resource utilisation; results show there are non-big data technologies available to provide a fair performance when processes involve large volumes of data and do not require high processing capacity from a machine. When a big data use case is present, the advantages of utilising the framework reside on the design of a solution that is
able to handle a growing workload without requiring significant re-design of existing programs, nor any hardware scaling approach, this is, either more machines or larger machines are not required compared to non-big data approaches, and performance is in general maintained, which can be translated even in a financial cost reduction. Data quality tasks benefit from this advantages in two ways: first, the task acquires the mentioned advantages inherently when a data quality task is transferred to be processed within a big data framework, and second, on a higher level as organisations, data quality tasks benefit by having a cost reduction that impacts on more areas than just the information management, and also accrues benefits on augmenting the data quality productivity due to the processing limits being increased without trading performance, thus, task time is not affected either. With big data frameworks implemented on large workloads, data quality management removes limitations on productivity and makes it possible to process higher volumes of data.

2. **What are the performance and scalability differences when using a single multicore node versus cluster computing environments for small, medium and large workloads?**

The Apache Spark big data framework provides its best performance when the number of tasks to handle simultaneously are aligned to the number of available CPU cores and the amount of data to process is large enough to keep the workers busy compared to the work done by the the task manager to schedule tasks. These two characteristics place a single machine on a limited performance, given by the number of CPU cores available, and the workload. Small workloads in a single machine are in disadvantage with non-big data framework solutions, due to the overhead Spark generates to set-up all the processes required by its computational model. Medium workloads for big data frameworks still can be in disadvantage compared with non-big data solutions, but as workload increases, overhead decreases and the gap in performance becomes smaller; this is true for data quality tasks with similar behaviour, since other kinds of algorithms, for example, iterative or machine learning processing could present a different behaviour. Scalability, however, appeared as a continuous benefit for all workload sizes processed in Apache Spark on a single machine, whereas non-big data solutions had a smaller scalability degree.
Big data frameworks are in general designed and thought to be utilised on high processing computing, the study shows Apache Spark obtained the best performance and scalability results when deployed on the DPSF cluster, and this environment was the only one in which a non-big data solution offered runtimes below the ones obtained with Apache Spark. Large workloads are the optimum use case for big data frameworks, and a cluster environment provides an optimum architecture to leverage the capabilities of high performance processing.

3. What are the challenges of re-engineering the implementation of a non-big data quality query system to work on top of big data frameworks and platforms?

Big data processing is based on the MapReduce programming model, and the majority of big data frameworks are based on or related to Hadoop (refer to Section 2.3 for further details of Hadoop relations with other big data solutions) which has been promoted as open source since 2007 [45]. The recency and ongoing development status of big data frameworks imply continuous and fast updating of their processing features and capabilities, and a wide area for progress and improvement; open source availability of the frameworks provide a massive user base, extensive documentation, a large range of utilisation within all kinds of organisations, extensive information resources, and at the same time, an ease of use once the complexity of the set up is overcome. Re-engineering the implementation of a non-big data quality query system involves achieving deep knowledge of the quality tasks required, platform characteristics and the algorithmic setup, as well as the functional requirements; a second challenge to successfully implement a non-big data solution on a big data framework, is that the latter needs to be well known on all of the possibilities a big data framework offers, from its usage to its configuration and optimisation options which most of the time require low level understanding of both the hardware and software and its involvement with the framework, which is an extra step not usually needed with the majority of the frameworks focused solely on software (e.g. Web frameworks).

Once the initial learning curve is passed through, the highest challenge a development of this kind could face is that not all the algorithms are well suited for MapReduce programming, in that sense, Apache Spark makes a
delightful work by providing implicit parallelism, dropping granular control, but without trading quality results and lack of high complexity on its APIs usage from a software engineering point of view.

5.3 Summary

This chapter presented a comprehensive account of the experimental journey used to asses extensions to the DQ$^2$S towards leveraging the Apache Spark big data framework. The main results demonstrate that big data frameworks can be used to improve performance and scalability of a data quality query system on small, medium, and large workloads, however, with intense processing requirements is the only case in which significant improvements are achieved, and even in some cases, with small workloads the overhead caused by a required set-up of the framework produce worse performance results than traditional solutions. Next chapter will draw conclusions about the impact of the work described in this thesis, insights, and future work directions.
Chapter 6

Conclusions and Future Work

6.1 Research Summary and Key Results

This research discusses the design, implementation and testing of data quality algorithms included in the DQ²S, seeking for collecting information to support the analysis and validation of the hypothesis that a big data framework can be used to improve performance and scalability of a data quality query system, compared to non-big data solutions. DQ²S was presented on its original development as a solution to data quality queries without support for big data; the study explored the extent to which the system was able to handle scalability and its performance.

To validate the hypothesis, a big data framework implementation was required, hence, the PySpark DQ²S was developed to test the algorithms and explore the results obtained. The big data DQ²S instance provide task and data parallelism, and offers a fair performance and great scalability results, compared to the best non-big data solution.

The research described in this thesis provides a detailed account of the experimental journey followed to extend DQ²S towards exploring the capabilities of a popular big data framework (Apache Spark), including the experiments used to measure scalability and usefulness of the approach. The study also provides a roadmap for researchers interested in re-purposing and porting existing information management systems and tools to explore the capabilities provided by big data frameworks. One aspect in which big data frameworks offer significant advantages is when comparing the complexity of developing a solution for parallel processing, where otherwise with mainstream parallel programming another kind of knowledge and effort is required to come up with an instance capable of processing at the level of a framework. Developing a parallel
solution with traditional techniques now appear as reinventing the wheel; there could be cases in which a finer control over the internal processes is mandatory, but those cases might have in common the need to optimise from inside, where Apache Spark is capable of delivering acceptable results with its default parameters in terms of quality and efficiency if expertise is added. This study can be used as a guide to extend data quality algorithms to increase performance and make them available to be used with larger datasets.

6.2 Research Contributions and Impact

The impact of the research can be summarised with the contributions, which include:

- The development of a scalable DQ$^2$S, for small, medium and large workloads, and its usage within either a single multicore machine or a cluster environment with a big data framework, which represents an added artifact to knowledge.

- Exhaustive optimisation work done with regards to the DQ$^2$S Timeliness query, showing the implications on usage for different environments with several parameters tested.

- A point of reference to further research on other data quality algorithms scalability and performance for big data.

- A support to extend the algorithms for Internet Query Systems (IQS) presented at [27, 25], where DQ$^2$S algorithms are utilised to support data quality tasks on web query results.

- An initial proposition of Apache Spark user level, considering exposure and expertise, which to the best of my knowledge, has not been proposed before (Appendix D).

This research has also contributed to the user documentation of the Data Processing Shared Facility at the University of Manchester (DPSF) cluster for Apache Spark [136]. Regarding the infrastructure used for big data computing linked to this research, the DPSF cluster was first provided with Apache Spark 1.6.0 version, released on January 4, 2016; later in June, the DPSF and Apache Spark had an upgrade and the cluster utilised in this study included by that time Apache Spark 1.6.2. This relatively new release was regarded as being “not yet in production” [57], and a mention of Hadoop
is done, but no Hadoop installation is available on the cluster. Before this study, no one has utilised Apache Spark before. This research tackled the initial phases on a learning curve for the new software, and left a DPSF ready to be utilised with a Standalone mode as a result of a joint work with the Research Infrastructure Team within Research IT at The University of Manchester, paving the path for future users and students who might utilise Spark installation. Difficulties arose on a basis of novice users with regards to the framework, added to the complexity of a multiuser with Lustre file system administration, which required further configuration out of the scope of solely the Spark cluster, and at the same time knowledge of the framework was required to successfully fix the details on the administrator side that finally allowed common users to work with Spark.

6.3 Future Work

Future research directions include:

- **Test with heuristics applied.** The presentation of DQ$^2$S at [26] shows experimental results of queries with two data quality dimensions tested, timeliness and completeness, where the quickest runtime obtained on an average of three runs, was around 15 minutes for a dataset of 100 000 rows, and the slowest time with an average of 76 minutes for the same dataset. An important note on the mentioned results is two versions of the queries were executed, one was an optimised version with an implementation of heuristics for query execution in relational databases, with a reduction of 29% in the execution time for the Timeliness query, and a reduction of 67% for the completeness query. The research reported in this thesis utilised the algorithms without heuristics applied to tackle the highest workload scenario involving DQ$^2$S queries. Future work can benefit from the baselines obtained on this research, to analyse and explore the algorithms with heuristics applied combined with the Optimised Python and the PySpark approach.

- **Further Apache Spark tuning.** This research was carried using default platform parameters. Spark has the possibility to tune the processing by a set of common to complex and deep parametrisation. This future work involves developing a whole set of tests to obtain the optimum settings for the given algorithm,
as well as gathering insights on its performance impact. The most common parameters applicable to the PySpark DQ²S implementation have been reviewed to provide a general overview of the extended features of Spark. The applicable tuning settings recommended by Spark [134] are as follows:

1. **Resource Allocation.** This tuning is meant to set the program to be exhaust the resources available in the cluster. In the research presented in this thesis, the number of cores, number of executors and memory for each one was set to the ones that showed the best performance in the execution of the PySpark program without further tuning. Future work might include testing on different settings with different commodity machine’s configuration, this is, different CPU cores available, and different CPU speeds, for example.

2. **Parallelism.** The level of parallelism for operations in a cluster should be high. By default, Spark sets the number of tasks to run on each file to be the total number of cores on all executor nodes if it is larger than 2, otherwise, it is set to 2. The number of tasks can be changed using `spark.default.parallelism` in the `spark-env.sh` file. The recommended number of tasks is 2-3 tasks per CPU core. In this research, the parallelism parameter was set to 3 tasks per core.

3. **Data structures.**
   (a) To utilise a Java Serialisation framework (e.g. Kryo serialiser) is generally recommended instead of preserving the default as a deserialised Java object. The intention is that shuffled data and data retrieved to disk can be changed to be used as a serialised binary representation. This can be done by setting `org.apache.spark.serializer.KryoSerializer` to `spark.serializer` in the `spark-defaults.conf` file, and registering the classes to the serialiser. However, it is recommended to try the serialiser without registering the classes [150] when using PySpark, since the Java Serialiser is likely to not produce the expected optimisation because data is already stored as a collection of byte objects, however, Python serialisers could be investigated.
   (b) With JVM in use, a flag to make data pointers be four bytes instead of eight can be set by adding `-XX:+UseCompressedOops` for `spark.executor.extraJavaOptions` in the `spark-defaults.conf` file.
Future work might include different values and approaches to tune the JVM related settings.

(c) If in any case a broadcast variable is used, spark.broadcast.compress with true as value, will make Spark compress broadcast variables before sending them through the cluster.

4. **Data format.** This optimisation comprises to use a binary format, in this case Parquet, to store data on disk, instead of using CSV/JSON.

5. **Shuffling.** A true value for spark.shuffle.compress will make Spark compress the map output files. This can be done in the `spark-defaults.conf` file.

6. **Data locality.** This refers to the closeness of a process and its required data. Having a five level location (process-local, node-local, rack-local and any location), Spark waits to schedule a task in the closest locality, if it is not available, then Spark switches to the next locality level. To get better locality scheduling the waiting time should be increased, using `spark.locality.wait.node` in the `spark-defaults.conf` file to a value above the default time of 3 seconds. An analysis on how this setting impacts on the final runtime obtained can help to identify additional performance gains.

- **Analyse impact of Spark higher versions on the DQ²S PySpark instance.**
  Spark is evolving at a high pace, since its first release in 2012 [131] there have been three major versions (0.x, 1.x, and 2.x) among 18 releases until 2016, with a time space of 1 to 3 months between each. Apache Spark presented its second major upgrade “Spark 2.0” on July 26, 2016, with changes on the API usability, SQL 2003 support, performance improvements, and more than 2500 patches from over 300 contributors [133]. Latests Apache Spark versions may have an impact on the results with further work required to identify and quantify the impact of major releases, and although significant impact is not expected, some new characteristics added to Spark over time could lead to important insights on Spark development from the user’s point of view.

- **Calculate the theoretical complexity to further analyse the scalability and performance.** Big $O$ notation requires analysis on the algorithm structure, which is a highly complex task to do for Apache Spark’s applications, because most of
6.3. **FUTURE WORK**

The functions utilised from the libraries, are formed by further wrapped algorithms. To calculate the theoretical complexity of the DQ$^2$S algorithms, each function should be analysed from its most internal structure. Theoretical information will provide a baseline and a firm limit expectation on the algorithms behaviour, which could provide another angle from where DQ$^2$S could be explored and optimised.

- **Expanding DQ$^2$S utilisation for big data with a development made for its usage on top of Apache Spark as Spark package.** Future work includes converting DQ$^2$S form being an extension to SQL queries, to become an extension to Apache Spark SQL library added to the packages repository [130]. Current state-of-the-art presents only one data quality package called “Drunken-data-quality (DDQ)”[132], which “is a small library for checking constraints on Spark data structures” [112], but it does not consider data quality dimensions, as in DQ$^2$S.

- **Contribution to formal documentation.** The development of this research found diverse and dispersed information on several formats, without a place that covers all the required, updated, information. Future work involves creating a document repository in which all the Spark related information could be found, ranked by a focus on user level, covering the major releases of Spark, on a centralised approach, without increasing the volume of information but organising the existent and filling the gaps found towards covering the needs required by each end user level.

Overall, the results reported in this research have achieved the proposed research aims and objectives, and contributed to the research community with a systematic approach towards re-purposing and porting information management systems and tools to work with big data frameworks. Extensive experimental results have also shown the advantages and limitations of the approach.
Bibliography


[22] Science Daily. Big Data, for better or worse: 90% of world’s data generated over last two years – ScienceDaily. [http://www.sciencedaily.com/releases/2013/05/130522085217.htm](http://www.sciencedaily.com/releases/2013/05/130522085217.htm), 2013. [Online; accessed 2016-01-18].


[57] Simon Hood. Big data compute and storage at UoM. *The University of Manchester. Data Science Club Meeting #4*, 2016.

[59] IBM. IBM - Data management: Tapping your most valuable asset. [Online; accessed 2016-01-20].

[60] IBM. IBM - What is big data? [Online; accessed 2016-01-18].


Appendix A

DQ²S Source code

This Appendix presents the source code utilised in the DQ²S instances. The Java code was provided by its original authors [26].

A.1 The Java DQ²S

1. The Java Timeliness Query main class.

```java
import java.util.ArrayList;

public class TimelinessQuery {
    public static void main(String[] args) {

        // Save and show the current value of the system timer (nanoseconds)
        Timing sysTime = new Timing();
        sysTime.startTime();

        // Save the pending orders from orderT in outputScan_left
        String inputTable_1 = "orderT";
        Predicate pred = new Predicate("orderT.statusOrder", ",", "pending");
        ScanSelect selScan_left = new ScanSelect(inputTable_1, pred);
        ArrayList<Object> outputScan_left = selScan_left.execute();

        // Save all data from statusTimelinessQR in outputScan_right
        String inputTable_scanRight = "statusTimelinessQR";
        ScanSelect selScan_right = new ScanSelect(inputTable_scanRight, null);
        ArrayList<Object> outputScan_right = selScan_right.execute();

        /* Join the data from statusTimelinessQR with the correspondant
         * pending orders (join outputScan_left and outputScan_right) and
         * save the joined data in outputJoin_1 */
        JoinPredicate pred_join = new JoinPredicate("orderT.statusTimeliness_id",
                                                   ",", "statusTimelinessQR.statusTimeliness_qid");
    }
}
```
A.1. THE JAVA DQ²S

Join join_1 = new Join(outputScan_left, outputScan_right, pred_join);
ArrayList<Object> outputJoin_1 = join_1.execute();

/* Calculate the timeliness value and add it to the data in */
String timelinessAttr = "statusTimelinessQR.order.statusOrder";
Timeliness timeliness = new Timeliness(outputJoin_1, timelinessAttr);
ArrayList<Object> outputTimeliness = timeliness.execute();

/* Select the rows from with a timeliness score below 0.5 and save */
Predicate pred_select_right = new
    Predicate("statusTimelinessQR.order.statusOrder.timeliness", "<", 0.5);
Select sel_right = new Select(outputTimeliness, pred_select_right);
ArrayList<Object> outputSel_right = sel_right.execute();

/* Save only the columns listed in attrList, with all of its rows */
    * and save it in outputProj */
ArrayList<Object> attrList = new ArrayList<Object>();
String[] attrListArray = {"statusTimelinessQR.statusTimeliness_qid",
    "statusTimelinessQR.order.statusOrder"};
for (int index = 0; index < attrListArray.length; index++) {
    attrList.add(attrListArray[index]);
}
Project proj = new Project(outputSel_right, attrList);
ArrayList<Object> outputProj = proj.execute();

// The following 4 lines can be uncommented to print the final output
// System.out.println("*******************************");
// System.out.println("Query result: " + outputProj.toString());
// System.out.println("Number of rows: " + outputProj.size());
// System.out.println("*******************************");

/* Return the current value of the system timer, and the total */
    * execution time (nanoseconds) */
sysTime.stopTime();
sysTime.durationTime();
} }

2. The Java Timing class.

import java.math.BigDecimal;

public class Timing {
    long startT;
    long stopT;

    // Get system time in nanoseconds and saves it in startT
    public long startTime() {
        startT = System.nanoTime();
    }
System.out.println("Process start time in nanoseconds = " + startT);
return startT;
}

// Get system time in nanoseconds and saves it in stopT
public long stopTime() {
    stopT = System.nanoTime();
    System.out.println("Process stop time in nanoseconds = " + stopT);
    return stopT;
}

// Calculate elapsed time from the point startT and stopT were obtained
public void durationTime() {
    long ExecutionDuration = (stopT - startT);
    System.out.println("Process ExecutionDuration in nanoseconds: " + ExecutionDuration);
    System.out.println("Process ExecutionDuration in seconds: " + new BigDecimal ((double) ExecutionDuration/1000000000));
}

3. The Java Predicate class.

```java
public class Predicate {
    public String operand1;
    public String comparisonOperator;

    public Predicate(String inputOperand1, String inputComparisonOperator, Object inputOperand2) {
        operand1 = inputOperand1;
        comparisonOperator = inputComparisonOperator;
        operand2 = inputOperand2;
    }

    public boolean apply(Tuple tuple) {
        if (tuple.typeAttributes.contains(operand1)) {
            int positionOfAttribute = tuple.typeAttributes.indexOf(operand1);
            String attributeType = (String) tuple.typeAttributeTypes.get(positionOfAttribute);
            if (attributeType.equals("double")) {
                double op_1 = (Double) tuple.tupleValues.get(positionOfAttribute);
                double op_2 = (Double) operand2;
                if (comparisonOperator.equals("<")) {
                    if (op_1 < op_2) {
                        return true;
                    }
                } else if(comparisonOperator.equals("<=")) {
                    if (op_1 <= op_2) {
                        return true;
                    }
                }
            }
        }
    }
```
else if (comparisonOperator.equals("==")) {
    if (op_1 == op_2)
        return true;
} else if (comparisonOperator.equals(">") ) {
    if (op_1 > op_2)
        return true;
} else if (comparisonOperator.equals(">=") ) {
    if (op_1 >= op_2)
        return true;
} else if (comparisonOperator.equals("<>") ) {
    if (op_1 != op_2)
        return true;
}

} else if (attributeType.equals("int")) {
    int op_1 = (Integer) tuple.tupleValues.get(positionOfAttribute);
    int op_2 = (Integer) operand2;
    if (comparisonOperator.equals("<") ) {
        if (op_1 < op_2) {
            return true;
        }
    } else if (comparisonOperator.equals("<=") ) {
        if (op_1 <= op_2) {
            return true;
        }
    } else if (comparisonOperator.equals("==") ) {
        if (op_1 == op_2)
            return true;
    } else if (comparisonOperator.equals(">") ) {
        if (op_1 > op_2)
            return true;
    } else if (comparisonOperator.equals(">=") ) {
        if (op_1 >= op_2)
            return true;
    } else if (comparisonOperator.equals("<>") ) {
        if (op_1 != op_2)
            return true;
    }
}

} else if (attributeType.equals("String")) {
    String op_1 = (String) tuple.tupleValues.get(positionOfAttribute);
    String op_2 = (String) operand2;
    if (comparisonOperator.equals("==") ) {
        if (op_1.equals(op_2)) return true;
    } else if (comparisonOperator.equals("<>") ) {
        if (!op_1.equals(op_2)) return true;
    }
} // End if tuple.typeAttributes
return false; // Return false for predicate
public String toString() {
    return ("operand1: " + operand1.toString() + "; comparisonOperator: " + comparisonOperator.toString() + "; operand2: " + operand2.toString()));
}
} // End class

4. The Java ScanSelect class.

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.sql.*;

public class ScanSelect {
    public String tableName;
    public Predicate predicate;
    public ArrayList<Object> output;

    public ScanSelect(String inputTableName, Predicate inputPredicate) {
        tableName = inputTableName;
        predicate = inputPredicate;
        output = new ArrayList<Object> ();
    }

    public ArrayList<Object> execute() {
        try {
            // csvFile requires the input data path
            String csvFile = "/local_drive//Ebusiness_DQ2S//"+tableName+".csv";
            String line, splitBy = ",";

            BufferedReader rs = new BufferedReader(new FileReader(csvFile));
            rs.readLine(); // Read first line (headers) to "skip" them

            boolean discardTuple = false;

            while((line = rs.readLine()) != null){
                // Create the tuple object
                Tuple tuple = new Tuple();
                if (tableName.equals("part_supply")) {
                    // Resolve tuple type
                    String[] data = line.split(splitBy);
                    String[] tupleTypeAttributes = {"part_supply.supplier_id",
                        "part_supply.part_id", "part_supply.part_price",
                        "part_supply.quantity_available", "part_supply.part_price_qid",
                        "part_supply.quantity_qid"};
                    String[] tupleTypeAttributeTypes = {"int", "int", "double", "int",
                        "int", "int"};
                }
37  tuple.createType(tupleTypeAttributes, tupleTypeAttributeTypes);
38  tuple.tupleValues.add(Integer.parseInt(data[0]));
39  tuple.tupleValues.add(Integer.parseInt(data[1]));
40  tuple.tupleValues.add(Double.parseDouble(data[2]));
41  tuple.tupleValues.add(Integer.parseInt(data[3]));
42  tuple.tupleValues.add(Integer.parseInt(data[4]));
43  tuple.tupleValues.add(Integer.parseInt(data[5]));
44  } else if (tableName.equals("part")) {
45    // Resolve tuple type
46    String[] data = line.split(splitBy);
47    String[] tupleTypeAttributes = {"part.part_id", "part.part_category",
48      "part.part_stock"};
49    String[] tupleTypeAttributeTypes = {"int", "String", "int"};
50    tuple.createType(tupleTypeAttributes, tupleTypeAttributeTypes);
51    tuple.tupleValues.add(Integer.parseInt(data[0]));
52    tuple.tupleValues.add(data[1]);
53    tuple.tupleValues.add(Integer.parseInt(data[2]));
54  } else if (tableName.equals("statusTimelinessQR")) {
55    // Resolve tuple type
56    String[] data = line.split(splitBy);
57    String[] tupleTypeAttributes = {
58      "statusTimelinessQR.statusTimeliness_qid",
59      "statusTimelinessQR.lastUpdateTime",
60      "statusTimelinessQR.expiryTime", "statusTimelinessQR.deliveryTime",
61      "statusTimelinessQR.age"};
62    String[] tupleTypeAttributeTypes = {"int", "Timestamp", "Timestamp",
63      "Timestamp", "int"};
64    tuple.createType(tupleTypeAttributes, tupleTypeAttributeTypes);
65    tuple.tupleValues.add(Integer.parseInt(data[0]));
66    tuple.tupleValues.add(Timestamp.valueOf(data[1]));
67    tuple.tupleValues.add(Timestamp.valueOf(data[2]));
68    tuple.tupleValues.add(Timestamp.valueOf(data[3]));
69    tuple.tupleValues.add(Integer.parseInt(data[4]));
70  } else if (tableName.equals("orderT")) {
71    // Resolve tuple type
72    String[] data = line.split(splitBy);
73    String[] tupleTypeAttributes = {
74      "orderT.order_no", "orderT.customer_id",
75      "orderT.product_id", "orderT.quantity", "orderT.submit_date",
76      "orderT.ship_date", "orderT.statusTimeliness_id",
77      "orderT.statusOrder"};
78    String[] tupleTypeAttributeTypes = {"int", "int", "int", "int",
79      "Timestamp", "Timestamp", "int", "String"};
80    tuple.createType(tupleTypeAttributes, tupleTypeAttributeTypes);
81    tuple.tupleValues.add(Integer.parseInt(data[0]));
82    tuple.tupleValues.add(Integer.parseInt(data[1]));
83    tuple.tupleValues.add(Integer.parseInt(data[2]));
84    tuple.tupleValues.add(Timestamp.valueOf(data[3]));
85    tuple.tupleValues.add(Timestamp.valueOf(data[4]));
86    tuple.tupleValues.add(Timestamp.valueOf(data[5]));
87    tuple.tupleValues.add(Timestamp.valueOf(data[6]));
88    tuple.tupleValues.add(Timestamp.valueOf(data[7]));
89    tuple.tupleValues.add(data[8]);
90  } else if (tableName.equals("part_priceTimelinessQR")) {
91    // Resolve tuple type
String[] data = line.split(splitBy);
String[] tupleTypeAttributes = {
    "part_priceTimelinessQR.timeliness_qid",
    "part_priceTimelinessQR.lastUpdateTime",
    "part_priceTimelinessQR.expiryTime",
    "part_priceTimelinessQR.deliveryTime", "part_priceTimelinessQR.age"};
String[] tupleTypeAttributeTypes = {
    "int", "Timestamp", "Timestamp",
    "Timestamp", "int"};
tuple.createType(tupleTypeAttributes, tupleTypeAttributeTypes);
tuple.tupleValues.add(Integer.parseInt(data[0]));
tuple.tupleValues.add(Timestamp.valueOf(data[1]));
tuple.tupleValues.add(Timestamp.valueOf(data[2]));
tuple.tupleValues.add(Timestamp.valueOf(data[3]));
tuple.tupleValues.add(Integer.parseInt(data[0]));
}

if (predicate != null) {
    discardTuple = predicate.apply(tuple);
    if (discardTuple == true) {
        output.add(tuple);
    }
} else output.add(tuple);

} // End while rs.next()
rs.close();
return output;
} // End of try block

catch (Exception e) {
    System.out.println("Driver Registration Exception: " + e.getMessage());
e.printStackTrace();
return output;
}
} // End execute method

public String toString() {
    if (predicate != null)
        return ("tableName: " + tableName.toString() + "; predicate: " + predicate.toString() + "; output: " + output.toString());
    else return ("tableName: " + tableName.toString() + "; predicate: null" + "; output: " + output.toString());
}
} // End class

5. The Java Tuple class.
public class Tuple {
    public ArrayList<Object> typeAttributes;
    public ArrayList<Object> typeAttributeTypes;
    public ArrayList<Object> tupleValues;

    public Tuple() {
        typeAttributes = new ArrayList<Object> ();
        typeAttributeTypes = new ArrayList<Object> ();
        tupleValues = new ArrayList<Object> ();
    }

    public boolean createType(String[] inputTypeAttributes, String[] inputTypeAttributeTypes) {
        for (int index = 0; index < inputTypeAttributes.length; index++) {
            typeAttributes.add(inputTypeAttributes[index]);
            typeAttributeTypes.add(inputTypeAttributeTypes[index]);
        }
        return true;
    }

    public String toString_original() {
        return ("typeAttributes: " + typeAttributes.toString() + "; typeAttributeTypes: " + typeAttributeTypes.toString() + "; tupleValues: " + tupleValues.toString());
    }

    public String toString() {
        return ("tupleValues: " + tupleValues.toString() + 
    }
}

6. The Java JoinPredicate class.

public class JoinPredicate {
    public String operand1;
    public String comparisonOperator;
    public String operand2;

    public JoinPredicate(String inputOperand1, String inputComparisonOperator, String inputOperand2){
        operand1 = inputOperand1;
        comparisonOperator = inputComparisonOperator;
        operand2 = inputOperand2;
    }

    public boolean apply(Tuple tuple) {
        if (tuple.typeAttributes.contains(operand1) &&
            (tuple.typeAttributes.contains(operand2))) {
            int positionOfAttribute_op1 = tuple.typeAttributes.indexOf(operand1);
            int positionOfAttribute_op2 = tuple.typeAttributes.indexOf(operand2);
            int positionOfComparisonOperator = tuple.typeAttributeTypes.indexOf(comparisonOperator);
        }
    }
String attributeType_op1 = (String)
    tuple.typeAttributeTypes.get(positionOfAttribute_op1);

if (attributeType_op1.equals("double")) {
    double op_1 = (Double) tuple.tupleValues.get(positionOfAttribute_op1);
    double op_2 = (Double) tuple.tupleValues.get(positionOfAttribute_op2);

    if (comparisonOperator.equals("<"){
        if (op_1 < op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("<=")) {
        if (op_1 <= op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("=")) {
        if (op_1 == op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals(">")) {
        if (op_1 > op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals(">=")) {
        if (op_1 >= op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("<>")) {
        if (op_1 != op_2) {
            return true;
        }
    }
}
else if (attributeType_op1.equals("int")) {
    int op_1 = (Integer) tuple.tupleValues.get(positionOfAttribute_op1);
    int op_2 = (Integer) tuple.tupleValues.get(positionOfAttribute_op2);

    if (comparisonOperator.equals("<"){
        if (op_1 < op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("<=")) {
        if (op_1 <= op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("=")) {
        if (op_1 == op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals(">")) {
        if (op_1 > op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals(">=")) {
        if (op_1 >= op_2) {
            return true;
        }
    }
    else if(comparisonOperator.equals("<>")) {
        if (op_1 != op_2) {
            return true;
        }
    }
}
A.1. THE JAVA DQ’S

70
71
72 }
73 }
74 else if (attributeType_op1.equals("String")) {
75   String op_1 = (String) tuple.tupleValues.get(positionOfAttribute_op1);
76   String op_2 = (String) tuple.tupleValues.get(positionOfAttribute_op2);
77   if (comparisonOperator.equals("=")) {
78     if (op_1.equals(op_2)) return true;
79   } else if (comparisonOperator.equals("<>")) {
80     if (!(op_1.equals(op_2))) return true;
81   }
82 }
83 return false; // Return false for predicate
84 } // End apply method
85
86 public String toString() {
87   return ("operand1: " + operand1.toString()
88     + "; comparisonOperator: " + comparisonOperator.toString()
89     + "; operand2: " + operand2.toString());
90 } // End class

7. The Java Join class.

1
2
3 import java.util.ArrayList;
4
5 public class Join {
6   public ArrayList<Object> inputLeft;
7   public ArrayList<Object> inputRight;
8   public JoinPredicate predicate;
9   public ArrayList<Object> output;
10
11 public Join(ArrayList<Object> inputLeft, ArrayList<Object> inputRight,
12             JoinPredicate inputPredicate) {
13   inputLeft = inputInputLeft;
14   inputRight = inputInputRight;
15   predicate = inputPredicate;
16   output = new ArrayList<Object> ();
17 }
18
19 public ArrayList<Object> execute() {
20   Tuple outputTuple = new Tuple();
21   boolean discardTuple = false;
22   // Copy contents of both left and right input into new tuple
23   for(int index_inputLeft = 0; index_inputLeft < inputLeft.size();
24     index_inputLeft++) {
25     Tuple currentTuple_left = (Tuple) inputLeft.get(index_inputLeft);
26     for(int index_inputRight = 0; index_inputRight < inputRight.size();
27       index_inputRight++) {
28       outputTuple.tupleValues.addAll(currentTuple_left.tupleValues);
outputTuple.typeAttributeTypes.addAll(currentTuple_left.typeAttributeTypes);
outputTuple.typeAttributes.addAll(currentTuple_left.typeAttributes);

Tuple currentTuple_right = (Tuple) inputRight.get(index_inputRight);
outputTuple.tupleValues.addAll(currentTuple_right.tupleValues);
outputTuple.typeAttributeTypes.addAll(currentTuple_right.typeAttributeTypes);
outputTuple.typeAttributes.addAll(currentTuple_right.typeAttributes);

if (predicate != null) {
discardTuple = predicate.apply(outputTuple);
if (discardTuple == true) {
    output.add(outputTuple);
}
} else output.add(outputTuple);
outputTuple = null;
outputTuple = new Tuple();
// End for inputRight
// End for inputLeft
return output;
// End execute method

public String toString() {
    return ("inputLeft: " + inputLeft.toString()
        + "inputRight: " + inputRight.toString()
        + "; predicate: " + predicate.toString()
        + "; output: " + output.toString());
}
// End class

8. The Java Timeliness class.

import java.util.ArrayList;
import java.sql.*;

public class Timeliness {
    public ArrayList<Object> input;
    public String timelinessAttribute;
    public ArrayList<Object> output;

    public Timeliness(ArrayList<Object> inputInput, String inputTimelinessAttribute) {
        input = inputInput;
        timelinessAttribute = inputTimelinessAttribute;
        output = new ArrayList<Object> ();
    }

    public ArrayList<Object> execute() {
        for(int index = 0; index < input.size(); index++) {
            Tuple currentTuple = (Tuple) input.get(index);
            Tuple outputTuple = new Tuple();
            Timestamp deliveryTimeValue = new Timestamp(0);
            Timestamp lastUpdateTime = new Timestamp(0);
A.1. THE JAVA DQ²S

Timestamp expiryTimeValue = new Timestamp(0);
int age;
double calculatedTimelinessScore;

// Get the timeliness attributes from the quality relation
for (int index_tupletype = 0; index_tupletype <
currentTuple.typeAttributes.size(); index_tupletype++) {
    String currentAttribute = (String)
currentTuple.typeAttributes.get(index_tupletype);
    if (currentAttribute.matches("(.*)deliveryTime") ) {
        deliveryTimeValue = (Timestamp)
currentTuple.tupleValues.get(index_tupletype);
    } else if (currentAttribute.matches("(.*)lastUpdateTime") ) {
        lastUpdateTime = (Timestamp)
currentTuple.tupleValues.get(index_tupletype);
    } else if (currentAttribute.matches("(.*)age") ) {
        age = (Integer) currentTuple.tupleValues.get(index_tupletype);
    } else if (currentAttribute.matches("(.*)expiryTime") ) {
        expiryTimeValue = (Timestamp)
currentTuple.tupleValues.get(index_tupletype);
    }
} // End for index_tupletype

/*Calculate currency and volatility for the current tuple,
 * stored in milliseconds */
long currencyLong = deliveryTimeValue.getTime() - lastUpdateTime.getTime();
long volatilityLong = expiryTimeValue.getTime() - lastUpdateTime.getTime();

// Calculate timeliness
double timelinessPartial = (1 - ((double) currencyLong/volatilityLong));

// Apply max(score, 0)
double zeroNum = 0.00;
if (timelinessPartial >= zeroNum) {
calculatedTimelinessScore = timelinessPartial;
} else calculatedTimelinessScore = zeroNum;

// Create new tuple
outputTuple.typeAttributes.addAll(currentTuple.typeAttributes);
outputTuple.typeAttributes.add(timelinessAttribute + ",timeliness");
outputTuple.typeAttributeTypes.addAll(currentTuple.typeAttributeTypes);
outputTuple.typeAttributeTypes.add("double");
outputTuple.tupleValues.addAll(currentTuple.tupleValues);
outputTuple.tupleValues.add(calculatedTimelinessScore);

output.add(outputTuple);
outputTuple = null;
} // End for index
return output;
} // End execute method
public String toString() {
    return ("input: " + input.toString()
    + "; timelinessAttribute: " + timelinessAttribute.toString()
    + "; output: " + output.toString());
}
} // End class

9. The Java Select class.

import java.util.ArrayList;
public class Select {
    public ArrayList<Object> input;
    public Predicate predicate;
    public ArrayList<Object> output;

    public select(ArrayList<Object> inputInput, Predicate inputPredicate) {
        input = inputInput;
        predicate = inputPredicate;
        output = new ArrayList<Object> ();
    }

    public ArrayList<Object> execute() {
        boolean retainTuple = false;
        for (int index = 0; index < input.size(); index++) {
            Tuple currentTuple = (Tuple) input.get(index);
            if (predicate != null) {
                retainTuple = predicate.apply(currentTuple);
                if (retainTuple == true) { // Proceed to add only tuples with predicate true value
                    output.add(currentTuple);
                }
            }
        } // End for input
        return output;
    } // End execute method

    public String toString() {
        return ("input: " + input.toString()
        + "; predicate: " + predicate.toString()
        + "; output: " + output.toString());
    }
}

10. The Java Project class.

import java.util.ArrayList;
public class Project {
    public ArrayList<Object> input;
public ArrayList<Object> attributeList;
public ArrayList<Object> output;

public Project(ArrayList<Object> inputInput, ArrayList<Object>
inputAttributeList) {
    input = inputInput;
    attributeList = inputAttributeList;
    output = new ArrayList<Object> ();
}

public ArrayList<Object> execute() {
    for(int index = 0; index < input.size(); index++) {
        Tuple currentTuple = (Tuple) input.get(index);
        Tuple outputTuple = new Tuple();
        for (int index_tupletype = 0; index_tupletype <
currentTuple.typeAttributes.size(); index_tupletype++) {
            for (int index_attrList = 0; index_attrList < attributeList.size();
                   index_attrList++) {
                if
                    (currentTuple.typeAttributes.get(index_tupletype).equals(attributeList.get(index_attrList)))
        {
            outputTuple.typeAttributes.add(currentTuple.typeAttributes.get(index_tupletype));
            outputTuple.typeAttributeTypes.add(currentTuple.typeAttributeTypes.get(index_tupletype));
            outputTuple.tupleValues.add(currentTuple.tupleValues.get(index_tupletype));
        }
        } // End for attributesList
    } // End for index_tupletype
    output.add(outputTuple);
    outputTuple = null;
    outputTuple = new Tuple();
} // End for input.size
return output;
} // End execute method

public String toString() {
    return ("input: " + input.toString()
            + "; attributeList: " + attributeList.toString()
            + "; output: " + output.toString());
}

A.2 The Python DQ²S

1. The Python Timeliness Query main class.

    from Timing import Timing
    from Predicate import Predicate
    from ScanSelect import ScanSelect
    from JoinPredicate import JoinPredicate
    from Join import Join
from Timeliness import Timeliness
from Select import Select
from Project import Project

class Query2():
    timing = Timing()
    startTime = timing.startTime()

    # Create a ScanSelect_left --> only pending orders
    inputTable_1 = 'orderT'
    predScan_left = Predicate('orderT.statusOrder', '=', 'pending')
    selScan_left = ScanSelect(inputTable_1, predScan_left)
    outputScan_left = selScan_left.execute()

    # Create a ScanSelect_right --> all from statusTimelinessQR
    inputTable_2 = 'statusTimelinessQR'
    predScan_right = None
    selScan_right = ScanSelect(inputTable_2, predScan_right)
    outputScan_right = selScan_right.execute()

    # Create a Join --> outputScan_left and outputScan_right
    predJoin = JoinPredicate("orderT.statusTimeliness_id", "=",
                            "statusTimelinessQR.statusTimeliness_qid")
    join_1 = Join(outputScan_left, outputScan_right, predJoin)
    outputJoin_1 = join_1.execute()

    # Calculate timeliness score
    timelinessAttr = "timeliness"
    timeliness = Timeliness(outputJoin_1, timelinessAttr)
    outputTimeliness = timeliness.execute()

    # Create a Select --> score < 0.5
    predSelect_right = Predicate("timeliness.score", "<", 0.5)
    sel_right = Select(outputTimeliness, predSelect_right)
    outputSel_right = sel_right.execute();

    # Creating a Project --> only columns in attrList
    attrList = []
    attrListArray =
        ["statusTimelinessQR.statusTimeliness_qid","timeliness.score"]
    for index, val in enumerate(attrListArray):
        attrList.append(attrListArray[index])
    proj = Project(outputSel_right, attrList)
    outputFinal = proj.execute()

    # Uncomment to print final output
    
    for r in outputFinal:
        print(r.valueAttributes)
        print(len(outputFinal))
    
""
2. The Python Timing class.

```python
import time

class Timing():
    def startTime(self):
        queryStartTime = int(round(time.time() * 1e9))
        print("Query start time in nanoseconds = " + str(queryStartTime))
        return queryStartTime

    def stopTime(self):
        queryStopTime = int(round(time.time() * 1e9))
        print("Query stop time in nanoseconds = " + str(queryStopTime))
        return queryStopTime

    def durationTime(self, queryStopTime, queryStartTime):
        queryExecutionDuration = (queryStopTime - queryStartTime)
        print("queryExecutionDuration in nanoseconds: " + str(queryExecutionDuration))
        print("queryExecutionDuration in seconds: " + str((queryExecutionDuration / 1000000000)))
```

3. The Python Predicate class.

```python
class Predicate():

    def __init__(self, inputOperand1, inputComparisonOperator, inputOperand2):
        self.operand1 = inputOperand1
        self.comparisonOperator = inputComparisonOperator
        self.operand2 = inputOperand2

    def apply(self, tuple1):
        if (self.operand1 in tuple1.nameAttributes):
            positionOfAttribute = tuple1.nameAttributes.index(self.operand1)
            self.attributeType = str(tuple1.typeAttributes[positionOfAttribute])

            if (self.attributeType == "double"):
                op_1 = float(tuple1.valueAttributes[positionOfAttribute])
                op_2 = float(self.operand2)

                if (self.comparisonOperator == "<"):
                    if (op_1 < op_2):
                        return True
                elif (self.comparisonOperator == "<="):
                    if (op_1 <= op_2):
                        return True
```
elif(self.comparisonOperator == "="):
    if (op_1 == op_2):
        return True
elif(self.comparisonOperator == ">"):  
    if (op_1 > op_2):
        return True
elif(self.comparisonOperator == ">="):
    if (op_1 >= op_2):
        return True
elif(self.comparisonOperator == "<>"):
    if (op_1 != op_2):
        return True
elif (self.attributeType == "int"):
    op_1 = int(tuple1.valueAttributes[positionOfAttribute])
    op_2 = int(self.operand2)
    if (self.comparisonOperator == "<"):
        if (op_1 < op_2):
            return True
    elif(self.comparisonOperator == "<="):
        if (op_1 <= op_2):
            return True
    elif(self.comparisonOperator == "="):
        if (op_1 == op_2):
            return True
    elif(self.comparisonOperator == ">"):
        if (op_1 > op_2):
            return True
    elif(self.comparisonOperator == ">="):
        if (op_1 >= op_2):
            return True
    elif(self.comparisonOperator == "<>"):
        if (op_1 != op_2):
            return True
elif (self.attributeType == "String"):
    op_1 = str(tuple1.valueAttributes[positionOfAttribute])
    op_2 = str(self.operand2)
    if (self.comparisonOperator == "="):
        if (op_1 == op_2):
            return True
    elif (self.comparisonOperator == ">%:
        if (op_1 != op_2):
            return True
else:
    return False

def __str__(self):
    toPrint = str(self.operand1)+ ';' +str(self.comparisonOperator)+ ' ' +str(self.operand2)
    return toPrint
4. The Python ScanSelect class.

```python
import csv
from Tuple import Tuple

class ScanSelect:
    def __init__(self, inputTableName, inputPredicate):
        self.tableName = inputTableName
        self.predicate = inputPredicate
        self.output = []

    def execute(self):
        # csvFile requires the input data local path
        csvFile = open('//local_drive//eBusiness//' + self.tableName + '.csv')
        csvReader = csv.reader(csvFile)
        discardTuple = False

        for row in csvReader:
            if csvReader.line_num == 1:
                continue # Skip first row
            tuple1 = Tuple()
            if (self.tableName == 'orderT'):
                data = row
                nameAttributes = ["orderT.order_no", "orderT.customer_id",
                                  "orderT.product_id", "orderT.quantity",
                                  "orderT.submit_date", "orderT.ship_date",
                                  "orderT.statusTimeliness_id", "orderT.statusOrder"]
                typeAttributes = ["int", "int", "int", "int", "Timestamp",
                                  "Timestamp", "int", "String"]
                tuple1.createType(nameAttributes, typeAttributes)
                tuple1.valueAttributes.append(int(data[0][0]))
                tuple1.valueAttributes.append(int(data[0][1]))
                tuple1.valueAttributes.append(int(data[0][2]))
                tuple1.valueAttributes.append(int(data[0][3]))
                tuple1.valueAttributes.append(str(data[0][4]))
                tuple1.valueAttributes.append(str(data[0][5]))
                tuple1.valueAttributes.append(int(data[0][6]))
                tuple1.valueAttributes.append(str(data[0][7]))
            elif (self.tableName == 'statusTimelinessQR'):
                data = row
                nameAttributes = ["statusTimelinessQR.statusTimeliness_qid",
                                  "statusTimelinessQR.lastUpdateTime",
                                  "statusTimelinessQR.expiryTime",
                                  "statusTimelinessQR.deliveryTime",
                                  "statusTimelinessQR.age"]
                typeAttributes = ["int", "Timestamp", "Timestamp",
                                  "Timestamp", "int"]
                tuple1.createType(nameAttributes, typeAttributes)
                tuple1.valueAttributes.append(int(data[0][0]))
                tuple1.valueAttributes.append(str(data[0][1]))
                tuple1.valueAttributes.append(str(data[0][2]))
```

```python
tuple1.valueAttributes.append(str(data[0][3]))
tuple1.valueAttributes.append(int(data[0][4]))

if (self.predicate != None):
    self.discardTuple = self.predicate.apply(tuple1);
if (self.discardTuple == True):
    self.output.append(tuple1)
else:
    self.output.append(tuple1)

return self.output

def __str__(self):
    toPrint = 'TableName: ' + str(self.tableName) + '
Predicate: ' + str(self.predicate) + '
Output: ' + str(self.output)
    return toPrint
```

5. The Python Tuple class.

```python
class Tuple():

    def __init__(self):
        self.nameAttributes = []
        self.typeAttributes = []
        self.valueAttributes = []

    def createType(self, inputNameAttributes, inputTypeAttributes):
        for index, val in enumerate(inputNameAttributes):
            self.nameAttributes.append(inputNameAttributes[index])
            self.typeAttributes.append(inputTypeAttributes[index])
        return True

    def __str__(self):
        toPrint = str(self.nameAttributes) + ';' + str(self.typeAttributes) + ';
        +str(self.valueAttributes)
        return toPrint
```

6. The Python JoinPredicate class.

```python
class JoinPredicate():

    def __init__(self, inputOperand1, inputComparisonOperator, inputOperand2):
        self.operand1 = inputOperand1
        self.comparisonOperator = inputComparisonOperator
        self.operand2 = inputOperand2
```

def apply(self, tuple1):
    if (self.operand1 in tuple1.nameAttributes and self.operand2 in
tuple1.nameAttributes):
        positionOfAttribute_op1 =
        tuple1.nameAttributes.index(self.operand1)
        positionOfAttribute_op2 =
        tuple1.nameAttributes.index(self.operand2)

        self.attributeType_op1 = str
        (tuple1.typeAttributes[positionOfAttribute_op1])

        if (self.attributeType_op1 == "double"):
            op_1 = Double(tuple1.valueAttributes[positionOfAttribute_op1])
            op_2 = Double(tuple1.valueAttributes[positionOfAttribute_op2])

            if (self.comparisonOperator == "<"):
                if (op_1 < op_2):
                    return True
            elif(self.comparisonOperator == "<="):
                if (op_1 <= op_2):
                    return True
            elif(self.comparisonOperator == ":="):
                if (op_1 == op_2):
                    return True
            elif(self.comparisonOperator == ">"):
                if (op_1 > op_2):
                    return True
            elif(self.comparisonOperator == ">="):
                if (op_1 >= op_2):
                    return True
            elif(self.comparisonOperator == "<>"):
                if (op_1 != op_2):
                    return True

        elif (self.attributeType_op1 == "int"):
            op_1 = int(tuple1.valueAttributes[positionOfAttribute_op1])
            op_2 = int(tuple1.valueAttributes[positionOfAttribute_op2])

            if (self.comparisonOperator == "<"):
                if (op_1 < op_2):
                    return True
            elif(self.comparisonOperator == "<="):
                if (op_1 <= op_2):
                    return True
            elif(self.comparisonOperator == ":="):
                if (op_1 == op_2):
                    return True
            elif(self.comparisonOperator == ">"):
                if (op_1 > op_2):
                    return True
            elif(self.comparisonOperator == ">="):
                if (op_1 >= op_2):
                    return True
            elif(self.comparisonOperator == "<>"):
                if (op_1 != op_2):
                    return True
elif (self.comparisonOperator == "<>"):
    if (op_1 != op_2):
        return True

elif (self.attributeType_op1 == "String"):
    op_1 = str(tuple1.valueAttributes[positionOfAttribute_op1])
    op_2 = str(tuple1.valueAttributes[positionOfAttribute_op2])
    if (self.comparisonOperator == ":="):
        if (op_1 == op_2):
            return True
    elif (self.comparisonOperator == "<>"):
        if (op_1 != op_2):
            return True
    else:
        return False

def __str__(self):
    toPrint = str(self.operand1) + ';' + str(self.comparisonOperator) + ';
    toPrint += str(self.operand2)
    return toPrint

7. The Python Join class.

from Tuple import Tuple
class Join():
    def __init__(self, inputInputLeft, inputInputRight, inputPredicate):
        self.inputLeft = inputInputLeft
        self.inputRight = inputInputRight
        self.predicateJoin = inputPredicate
        self.output = []

def execute(self):
    outputTuple = Tuple()
    discardTuple = False

    for indexInput_left, val in enumerate(self.inputLeft):
        currentTuple_left = self.inputLeft[indexInput_left]

        for indexInput_right, val in enumerate(self.inputRight):
            outputTuple.valueAttributes.extend(currentTuple_left.valueAttributes)
            outputTuple.typeAttributes.extend(currentTuple_left.typeAttributes)
            outputTuple.nameAttributes.extend(currentTuple_left.nameAttributes)

            currentTuple_right = self.inputRight[indexInput_right]
            outputTuple.valueAttributes.extend(currentTuple_right.valueAttributes)
            outputTuple.typeAttributes.extend(currentTuple_right.typeAttributes)
            outputTuple.nameAttributes.extend(currentTuple_right.nameAttributes)

            if currentTuple_left.valueAttributes == currentTuple_right.valueAttributes:
                outputTuple.valueAttributes.extend(currentTuple_right.valueAttributes)
                outputTuple.typeAttributes.extend(currentTuple_right.typeAttributes)
                outputTuple.nameAttributes.extend(currentTuple_right.nameAttributes)
A.2. THE PYTHON DQ²S

if self.predicateJoin != None:
    discardTup = self.predicateJoin.apply(outputTuple);
    if discardTup == True:
        self.output.append(outputTuple)
else:
    self.output.append(outputTuple)
outputTuple = None;
outputTuple = Tuple()
return self.output;

def __str__(self):
    toPrint = str(self.inputLeft) + ';' + str(self.inputRight) + ';
    + str(self.predicate) + ';' + str(self.output)
return toPrint

8. The Python Timeliness class.
from Tuple import Tuple
import re
import time
from datetime import datetime

class Timeliness():
    def __init__(self, inputInput, inputTimelinessAttr):
        self.input = inputInput
        self.timelinessAttr = inputTimelinessAttr
        self.output = []

def execute(self):
    # Convert timestamp to epoch time in milliseconds
    def to_unix_time(timestamp):
        epoch = datetime.utcfromtimestamp(0)
        my_time = datetime.strptime(timestamp, '%Y-%m-%d %H:%M:%S')
        delta = my_time - epoch
        return delta.total_seconds() * 1000.0

    for index, val in enumerate(self.input):
        currentTuple = self.input[index]
        outputTuple = Tuple()

        for index_tupletype, val in enumerate(currentTuple.nameAttributes):
            currentAttribute =
            str(currentTuple.nameAttributes[index_tupletype])

            if re.match(".*deliveryTime", currentAttribute) != None:
                deliveryTime_str =
                currentTuple.valueAttributes[index_tupletype]
            elif re.match(".*lastUpdateTime", currentAttribute) != None:
                lastUpdateTime_str =
                currentTuple.valueAttributes[index_tupletype]
elif re.match("\(.*)age\", currentAttribute) != None:
    age = int(currentTuple.valueAttributes[index_tupletype])
elif re.match("\(.*)expiryTime\", currentAttribute) != None:
    expiryTime_str =
currentTuple.valueAttributes[index_tupletype]

# Calculate Timeliness Score
currencyLong = (to_unix_time(deliveryTime_str) -
to_unix_time(lastUpdateTime_str))
volatilityLong = (to_unix_time(expiryTime_str) -
to_unix_time(lastUpdateTime_str))
timelinessPartial = (1 - (currencyLong/volatilityLong))

# Apply max(score,0)
zeroNum = 0.00
if timelinessPartial >= 0.00:
    timelinessScore = timelinessPartial
else:
    timelinessScore = zeroNum

# Create new tuple
outputTuple.nameAttributes.extend(currentTuple.nameAttributes)
outputTuple.nameAttributes.append(self.timelinessAttr + ".score")
outputTuple.typeAttributes.extend(currentTuple.typeAttributes)
outputTuple.typeAttributes.append("double")
outputTuple.valueAttributes.extend(currentTuple.valueAttributes)
outputTuple.valueAttributes.append(timelinessScore)
self.output.append(outputTuple)
outputTuple = None

return self.output

def __str__(self):
    toPrint = str(self.input)+ ';' + str(self.timelinessAttr) + ';'
    + str(self.output)
    return toPrint

9. The Python Select class.

from Tuple import Tuple

class Select():
    def __init__(self, inputInput, inputPredicate):
        self.input = inputInput
        self.predicate = inputPredicate
        self.output = []

    def execute(self):
        retainTuple = False
        for index, val in enumerate(self.input):
            if self.predicate:
currentTuple = Tuple()
currentTuple = self.input[index]
if self.predicate != None:
    retainTuple = self.predicate.apply(currentTuple)
    if retainTuple == True:
        self.output.append(currentTuple)
return self.output

def __str__(self):
    toPrint = str(self.input) + ';' + str(self.predicate) + '
    return toPrint

10. The Python Project class.

from Tuple import Tuple
class Project():

def __init__(self, inputInput, inputAttrList):
    self.input = inputInput
    self.attrList = inputAttrList
    self.output = []

def execute(self):
    for index, val in enumerate(self.input):
        currentTuple = self.input[index]
        outputTuple = Tuple()
        for index_tupletype, val in enumerate(currentTuple.nameAttributes):
            for index_attrList, val in enumerate(self.attrList):
                if currentTuple.nameAttributes[index_tupletype] ==
                    self.attrList[index_attrList]:
                    outputTuple.nameAttributes.append
                        (currentTuple.nameAttributes[index_tupletype])
                    outputTuple.typeAttributes.append
                        (currentTuple.typeAttributes[index_tupletype])
                    outputTuple.valueAttributes.append
                        (currentTuple.valueAttributes[index_tupletype])
        self.output.append(outputTuple)
    self.outputTuple = None
    outputTuple = Tuple();
    return self.output

def __str__(self):
    toPrint = str(self.input) + ';' + str(self.attrList) + '
    return toPrint
A.3 The Optimised Python DQ$^2$S

1. The Optimised Python Timeliness Query main class.

```python
from Timing import Timing
from Predicate import Predicate
from ScanSelect import ScanSelect
from Join import Join
from Timeliness import Timeliness
from Select import Select
from Project import Project
import numpy as np
import pandas as pd

class Query2:
    timing = Timing()
    startTime = timing.startTime()

    # Creating a ScanSelect_left --> Only pending orders
    inputTable_1 = 'orderT'
    predScan_left = Predicate('statusOrder', '=', 'pending')
    selScan_left = ScanSelect(inputTable_1, predScan_left)
    outputScan_left = selScan_left.execute()

    # Creating a ScanSelect_right --> All from statusTimelinessQR
    inputTable_2 = 'statusTimelinessQR'
    predScan_right = None
    selScan_right = ScanSelect(inputTable_2, predScan_right)
    outputScan_right = selScan_right.execute()

    # Creating a Join --> Join outputScan_left and ouputScan_left
    predJoin = ('statusTimeliness_id', '=', 'statusTimeliness_qid')
    join_1 = Join(outputScan_left, outputScan_right, predJoin)
    outputJoin_1 = join_1.execute()

    # Creating Timeliness --> Calculate timeliness score
    timelinessAttr = 'timeliness'
    timeliness = Timeliness(outputJoin_1, timelinessAttr)
    outputTimeliness = timeliness.execute()

    # Creating a Select --> Select those with Timeliness < 0.5
    predSelect_right = Predicate('timeliness_score', '<', 0.5)
    sel_right = Select(outputTimeliness, predSelect_right)
    outputSel_right = sel_right.execute()

    # Creating a Project --> Short the DF to have only two columns
    attrList = ['statusTimeliness_qid', 'timeliness_score']
    proj = Project(outputSel_right, attrList)
    outputFinal = proj.execute()

    # Uncomment to print final output
```
A.3. THE OPTIMISED PYTHON DQ²S

'''

n = len(outputFinal.index)
print(outputFinal.head(n).to_string())
print("Project Output = ")
print(n)
'''

stopTime = timing.stopTime()
timing.durationTime(stopTime, startTime)

2. The Optimised Python Timing class.

```python
import time

class Timing():
    def startTime(self):
        queryStartTime = int(round(time.time() * 1e9))
        print("Query start time in nanoseconds = " + str(queryStartTime))
        return queryStartTime

    def stopTime(self):
        queryStopTime = int(round(time.time() * 1e9))
        print("Query stop time in nanoseconds = " + str(queryStopTime))
        return queryStopTime

    def durationTime(self, queryStopTime, queryStartTime):
        queryExecutionDuration = (queryStopTime - queryStartTime)
        print("queryExecutionDuration in nanoseconds: " + str(queryExecutionDuration))
        print("queryExecutionDuration in seconds: " + str((queryExecutionDuration/1000000000)))
```

3. The Optimised Python Predicate class.

```python
class Predicate():
    def __init__(self, inputOperand1, inputComparisonOperator, inputOperand2):
        self.operand1 = inputOperand1
        self.comparisonOperator = inputComparisonOperator
        self.operand2 = inputOperand2

    def apply(self, dfInput):
        operand1_true = 0
        nameAttributes = dfInput.columns
        typeAttributes = dfInput.dtypes
        if (self.operand1 in nameAttributes):
            positionOfAttribute = nameAttributes.get_loc(self.operand1)
            attributeType = str(typeAttributes[positionOfAttribute])
            operand1_true = 1

        if (operand1_true != 0):
```
if (attributeType != "object"):
    self.operand1 = str(self.operand1)
    op_2 = str(self.operand2)
    if (self.comparisonOperator == "<"):
        filterClause = "" + self.operand1 + "<" + op_2 + ""
        print(filterClause)
        dfOutput = dfInput.query(filterClause)
    elif(self.comparisonOperator == "<="):
        filterClause = "" + self.operand1 + "<=" + op_2 + ""
        dfOutput = dfInput.query(filterClause)
    elif(self.comparisonOperator == "="):
        filterClause = "" + self.operand1 + "=" + op_2 + ""
        dfOutput = dfInput.query(filterClause)
    elif(self.comparisonOperator == ">"):
        filterClause = "" + self.operand1 + ">" + op_2 + ""
        dfOutput = dfInput.query(filterClause)
    elif(self.comparisonOperator == ">="):
        filterClause = "" + self.operand1 + ">=" + op_2 + ""
        dfOutput = dfInput.query(filterClause)
    elif(self.comparisonOperator == "<>"):
        filterClause = "" + self.operand1 + "!='" + op_2 + ""
        dfOutput = dfInput.query(filterClause)
    return dfOutput
else:
    return dfInput

4. The Optimised Python ScanSelect class.

import csv
import numpy as np
import pandas as pd
from datetime import datetime

class ScanSelect():
    def __init__(self, inputTableName, inputPredicate):
        self.tableName = inputTableName
        self.predicate = inputPredicate
        self.output = []
    
    def execute(self):
        if (self.tableName == 'orderT'):
A.3. **THE OPTIMISED PYTHON DQ²S**

```python
nameAttributes = ["order_no", "customer_id", "product_id", "quantity", "submit_date", "ship_date", "statusTimeliness_id", "statusOrder"]

elif (self.tableName == 'statusTimelinessQR'):
    nameAttributes = ["statusTimeliness_qid", "lastUpdateTime", "expiryTime", "deliveryTime", "age"]

# dfInput requires the input data local path
dfInput = pd.read_csv('//local_drive//eBusiness//'+self.tableName+'.'csv', names=nameAttributes, header=0, encoding='latin-1')

if (self.predicate != None):
    discardTuple = self.predicate.apply(dfInput)
    dfOutput = discardTuple

else:
    dfOutput = dfInput

return dfOutput
```

5. **The Optimised Python Join class.**

```python
class Join():

def __init__(self, inputInputLeft, inputInputRight, inputPredicate):
    self.inputLeft = inputInputLeft
    self.inputRight = inputInputRight
    self.predicateJoin = inputPredicate
    self.output = []

def execute(self):
    if self.predicateJoin != None:
        columnName1 = self.inputLeft.columns
        columnName2 = self.inputRight.columns
        operand1 = self.predicateJoin[0]
        comparisonOperator = self.predicateJoin[1]
        operand2 = self.predicateJoin[2]
        operands = [operand1,operand2]
        if (operand1 in columnName1 and operand2 in columnName2 and comparisonOperator == "=":
            dfOutput = self.inputLeft.merge(self.inputRight, left_on=operand1, right_on= operand2, how="left")
        else:
            dfOuput = self.inputLeft.join(self.inputRight)
    else:
        dfOuput = self.inputLeft.join(self.inputRight)
    return dfOutput
```

6. **The Optimised Python Timeliness class.**

```python
import re
import time
from datetime import datetime
import numpy as np
import pandas as pd
```
class Timeliness:

def __init__(self, inputInput, inputTimelinessAttr):
    self.input = inputInput
    self.timelinessAttr = inputTimelinessAttr
    self.output = []

def execute(self):
    # Convert timestamp to epoch time in milliseconds
    def unixtime(datetime):
        epoch = (datetime.astype(np.int64) / 1e6).astype(np.uint64)
        return epoch

    lastUpdateTime_dt = pd.to_datetime(self.input['lastUpdateTime'])
    deliveryTime_dt = pd.to_datetime(self.input['deliveryTime'])
    expiryTime_dt = pd.to_datetime(self.input['expiryTime'])

    #Calculate Timeliness Score
    timelinessPartial = (1 - (unixtime(deliveryTime_dt) - unixtime(lastUpdateTime_dt) + self.input['age']) / (unixtime(expiryTime_dt) - unixtime(lastUpdateTime_dt))) + self.input['age']

    attrName = str(self.timelinessAttr) + '_score'

    # Add timeliness score as new column
    df = (self.input.assign(added=timelinessPartial))
    df.ix[df.added < 0.00, "added"] = 0.00
    # Rename column containing final timeliness score
    df1 = df.rename(columns={"added": attrName})

    return df1

class Select:

def __init__(self, inputInput, inputPredicate):
    self.input = inputInput
    self.predicate = inputPredicate
    self.output = []

def execute(self):
    if (self.predicate != None):
        dfOutput = self.predicate.apply(self.input)
        return dfOutput
    else:
        return self.input

7. The Optimised Python Select class.
A.4. THE PYSPARK DQ²S

8. The Optimised Python Project class.

```python
class Project():
    def __init__(self, inputInput, inputAttrList):
        self.input = inputInput
        self.attrList = inputAttrList
        self.output = []
    def execute(self):
        dfRes = self.input[self.attrList]
        return dfRes
```

A.4 The PySpark DQ²S

1. The PySpark Timeliness Query main class.

```python
from __future__ import print_function
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.sql.dataframe import DataFrame
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext, Row
from pyspark.sql.functions import *
from pyspark.sql.types import *
from datetime import datetime
import sys
import math
from Timing import Timing
from Predicate import Predicate
from ScanSelect import ScanSelect
from Join import Join
from Timeliness import Timeliness
from Select import Select
from Project import Project

# USAGE: bin/spark-submit --packages com.databricks:spark-csv_2.11:1.3.0
   TimelinessQuery.py appName outputFolderName

# URL must be replaced by the master address
# Used for utilisation in local machine or when having full control
# Commented when used in cluster without admin control
sc = SparkContext("spark://URL:7077", sys.argv[1],"$SPARK_HOME")

# Uncommented when used in cluster without admin control
# sc = SparkContext()

sqlContext = SQLContext(sc)
timing = Timing()
```
```
33  startTime = timing.startTime()
34
35  # Create a ScanSelect_left --> only pending orders
36  inputTable_1 = 'orderT'
37  predScan_left = Predicate('statusOrder', '=', 'pending')
38  selScan_left = ScanSelect(inputTable_1, predScan_left, sqlContext)
39  dfOutputScan_left = selScan_left.execute()
40
41  # Create a ScanSelect_right --> All from statusTimelinessQR
42  inputTable_2 = "statusTimelinessQR"
43  predScan_right = None
44  selScan_right = ScanSelect(inputTable_2, predScan_right, sqlContext)
45  dfOutputScan_right = selScan_right.execute()
46
47  # Create a Join --> Joining outputScan_left and outputScan_right
48  predJoin = ("statusTimeliness_id", ",", "statusTimeliness_qid")
49  dfJoin_1 = Join(dfOutputScan_left, dfOutputScan_right, predJoin)
50  dfOutputJoin_1 = dfJoin_1.execute()
51
52  # Create Timeliness (add a new column with the calculated timeliness
53  timelinessAttr = "timeliness"
54  timeliness = Timeliness(dfOutputJoin_1, timelinessAttr)
55  dfOutputTimeliness = timeliness.execute()
56
57  # Create a Select (extracting only that are true with the condition)
58  predSelect_right = Predicate("timeliness_score", ",", 0.5)
59  sel_right = Select(dfOutputTimeliness, predSelect_right)
60  dfOutputSel_right = sel_right.execute()
61
62  # Create a Project (shorting the DF to have only two columns)
63  attrList = ["statusTimeliness_qid", "timeliness_score"]
64  proj = Project(dfOutputSel_right, attrList)
65  dfOutputFinal = proj.execute()
66
67  # Uncomment next four lines to print full output
68  # nrows = dfOutputFinal.count()
69  # dfOutputTimeliness.show(nrows, truncate=False)
70  # print("Project Output= ")
71  # print(nrows)
72
73  # An action is mandatory to trigger full processing
74  # due to Spark’s lazy evaluation.
75  # Show is the less costly action.
76
77  dfOutputFinal.show(1)
78
79  stopTime = timing.stopTime()
80  timing.durationTime(stopTime, startTime)
```

2. The PySpark Timing class.
import time

class Timing():

    def startTime(self):
        queryStartTime = int(round(time.time() * 1e9))
        print("Query start time in nanoseconds = \" + str(queryStartTime))
        return queryStartTime

    def stopTime(self):
        queryStopTime = int(round(time.time() * 1e9))
        print("Query stop time in nanoseconds = \" + str(queryStopTime))
        return queryStopTime

    def durationTime(self, queryStopTime, queryStartTime):
        queryExecutionDuration = (queryStopTime - queryStartTime)
        print("queryExecutionDuration in nanoseconds: \" + str(queryExecutionDuration))
        print("queryExecutionDuration in seconds: \" + str((queryExecutionDuration / 1000000000)))

3. The PySpark Predicate class.

class Predicate(object):

    def __init__(self, inputOperand1, inputComparisonOperator, inputOperand2):
        self.operand1 = inputOperand1
        self.comparisonOperator = inputComparisonOperator
        self.operand2 = inputOperand2

    def apply(self, dfInput):
        operand1_true = 0
        columnName = dfInput.schema.names
        typeAttributes = [f.dataType for f in dfInput.schema.fields]
        if (self.operand1 in columnName):
            positionOfAttribute = dfInput.schema.names.index(self.operand1)
            attributeType = str(typeAttributes[positionOfAttribute])
            operand1_true = 1
        if (operand1_true != 0):
            if (attributeType != "StringType"):
                self.operand1 = str(self.operand1)
                op_2 = str(self.operand2)
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">="):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">"):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">="):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">"):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">="):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">"):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">="):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">"):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">="):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
            elif (self.comparisonOperator == ">"):
                filterClause = "+self.operand1+"+op_2++
                dfOutput = dfInput.filter(filterClause)
APPENDIX A. DQ$^2$S SOURCE CODE

```python
dfOutput = dfInput.filter(filterClause)
elif(self.comparisonOperator == ">"):
    filterClause = ""+self.operand1+">"+op_2+""
    dfOutput = dfInput.filter(filterClause)
elif(self.comparisonOperator == ">="):
    filterClause = ""+self.operand1+">=\"+op_2+""
    dfOutput = dfInput.filter(filterClause)
elif(self.comparisonOperator == "<>"):
    filterClause = ""+self.operand1+"!\"+op_2+""
    dfOutput = dfInput.filter(filterClause)

e elif (attributeType == "StringType"):
    op_2 = str(self.operand2)
    if(self.comparisonOperator == "="):
        filterClause = ""+self.operand1+"\"\"+op_2+\""\""
        dfOutput = dfInput.filter(filterClause)
    elif(self.comparisonOperator == "<>"):
        filterClause = ""+self.operand1+"\"\"+op_2+\""\""
        dfOutput = dfInput.filter(filterClause)
    return dfOutput
else:
    return dfInput
```

4. The PySpark ScanSelect class.

```python
class ScanSelect():
    def __init__(self, inputTableName, inputPredicate, inputSqlContext):
        self.tableName = inputTableName
        self.predicate = inputPredicate
        self.sqlContext = inputSqlContext

    def execute(self):
        # loadFile requires input data to be in the same directory as ScanSelect
        loadFile = self.tableName+".csv"
        dfInput =
            self.sqlContext.read.format("com.databricks.spark.csv").option("header", "true").load(loadFile)
        if (self.tableName == 'orderT'):
            dfInput = dfInput.select(
                dfInput.order_no.cast("int").alias("order_no"),
                dfInput.customer_id.cast("int").alias("customer_id"),
                dfInput.product_id.cast("int").alias("product_id"),
                dfInput.quantity.cast("int").alias("quantity"),
                dfInput.submit_date.cast("timestamp").alias("submit_date"),
                dfInput.ship_date.cast("timestamp").alias("ship_date"),
                dfInput.statusTimeliness_id.cast("int").alias("statusTimeliness_id"),
                dfInput.statusOrder)
        elif (self.tableName == 'statusTimelinessQR'):
            dfInput = dfInput.select(
```
A.4. THE PYSPARK DQ^2S

```python
dfInput.statusTimeliness_qid.cast("int").alias("statusTimeliness_qid"),
dfInput.lastUpdateTime.cast("timestamp").alias("lastUpdateTime"),
dfInput.expiryTime.cast("timestamp").alias("expiryTime"),
dfInput.deliveryTime.cast("timestamp").alias("deliveryTime"),
dfInput.age.cast("int").alias("age"))

if (self.predicate != None):
    dfDiscardTuple = self.predicate.apply(dfInput)
    dfOutput = dfDiscardTuple
else:
    dfOutput = dfInput

return dfOutput
```

5. The PySpark Join class.

```python
from pyspark.sql.functions import col
class Join(object):
    def __init__(self, inputInputLeft, inputInputRight, inputPredicate):
        self.inputLeft = inputInputLeft
        self.inputRight = inputInputRight
        self.predicateJoin = inputPredicate

    def execute(self):
        if self.predicateJoin != None:
            columnName1 = self.inputLeft.schema.names
            columnName2 = self.inputRight.schema.names
            operand1 = self.predicateJoin[0]
            comparisonOperator = self.predicateJoin[1]
            operand2 = self.predicateJoin[2]
            if (operand1 in columnName1 and operand2 in columnName2 and
                comparisonOperator == "="):
                dfOutput = self.inputLeft.join(self.inputRight,
                    col(str(self.predicateJoin[0])) == col(str(self.predicateJoin[2])),
                    "left_outer")
            else:
                dfOutput = self.inputLeft.join(self.inputRight)
        else:
            dfOutput = self.inputLeft.join(self.inputRight)

        return dfOutput
```

6. The PySpark Timeliness class.

```python
import re
import time
from datetime import datetime
from pyspark import SparkConf, SparkContext
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.sql import functions as F
```
class Timeliness():
    def __init__(self, inputInput, inputTimelinessAttr):
        self.input = inputInput
        self.timelinessAttr = inputTimelinessAttr

    def execute(self):
        # Convert timestamp to epoch time in milliseconds
        def unix_timestamp(timestamp=None, format='yyyy-MM-dd HH:mm:ss'):
            sc = SparkContext._active_spark_context
            if timestamp is None:
                return Column(sc._jvm.functions.unix_timestamp())
            return Column(sc._jvm.functions.unix_timestamp(_to_java_column(timestamp), format))

        # Calculate Timeliness Score
        timelinessScore = (1 - 
            ((unix_timestamp(self.input.deliveryTime) - unix_timestamp(self.input.lastUpdateTime)) + self.input.age) / 
            ((unix_timestamp(self.input.expiryTime) - unix_timestamp(self.input.lastUpdateTime)) + self.input.age))
        attrName = QString(str(self.timelinessAttr)) + "_score"
        # Apply max(score,0)
        dfOutput = self.input.withColumn(attrName, F.when(timelinessScore >= 0.00, timelinessScore).otherwise(0.00))
        return dfOutput

class Select(object):
    def __init__(self, inputInput, inputPredicate):
        self.input = inputInput
        self.predicate = inputPredicate

    def execute(self):
        if (self.predicate != None):
            dfOutput = self.predicate.apply(self.input)
            return dfOutput
        else:
            return dfInput

class Project():
    def __init__(self, inputInput, inputAttrList):
        self.input = inputInput
        self.attrList = inputAttrList

    def execute(self):
dfRes = self.input.select(self.attrList).sort(self.attrList[0])
return dfRes
Appendix B

Apache Spark set up

B.1 Configuration of Spark’s Basic Elements

Five concepts are required to understand the general Spark architecture. These elements are the ones on which an application submitted to Spark is processed, as described in Chapter 4. Table B.1 shows the configuration defaults of each of the concepts described in the mentioned chapter, and Table B.2 shows the parameters that can be used to modify the default values within a Spark Standalone cluster.

<table>
<thead>
<tr>
<th>Process</th>
<th>Memory</th>
<th>Cores</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>1GB</td>
<td>1</td>
<td>1 per cluster</td>
</tr>
<tr>
<td>Master</td>
<td>1GB</td>
<td>1</td>
<td>1 per cluster</td>
</tr>
<tr>
<td>Worker</td>
<td>1GB</td>
<td>1</td>
<td>1 per node</td>
</tr>
</tbody>
</table>

Worker’s resources to give out

<table>
<thead>
<tr>
<th>Process</th>
<th>spark-defaults.conf</th>
<th>spark-env.sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>–driver-cores</td>
<td>SPARK_DRIVER_CORES</td>
</tr>
<tr>
<td></td>
<td>–driver-memory</td>
<td>SPARK_DRIVER_MEMORY</td>
</tr>
<tr>
<td>Master</td>
<td>NA</td>
<td>SPARK_DAEMON_MEMORY</td>
</tr>
</tbody>
</table>
B.2. SINGLE MACHINE DEPLOYMENT

The following steps were followed to set up Spark in Standalone mode on a single machine:


2. Save the package in the desired location.

3. Extract the package using:

   cd locationOfThePackage
   tar -xf nameOfThePackage.tgz

4. Optional: Rename folder to Spark or a shorter name to facilitate usage. e.g. SPARK instead of spark-1.6.2-bin-hadoop2.6.tgz

5. Open a terminal and go to the SPARK/conf location using cd command.

6. Create spark-env.sh file from spark-env.sh.template and add the needed configurations, as follows:

   cp spark-env.sh.template spark-env.sh

7. Add the corresponding content (configurations), for example:

---

<table>
<thead>
<tr>
<th>Worker</th>
<th>-worker-instances</th>
<th>NA</th>
<th>SPARK_DAEMON_MEMORY</th>
<th>SPARK_WORKER_INSTANCES</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Worker's resources to give out</th>
<th>-cores</th>
<th>spark.worker.cores</th>
<th>SPARK_WORKER_CORES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-memory</td>
<td>spark.worker.memory</td>
<td>SPARK_WORKER_MEMORY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Executor</th>
<th>-executor-cores</th>
<th>spark.executor.cores</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-executor-memory</td>
<td>spark.executor.memory</td>
<td></td>
</tr>
</tbody>
</table>

Table B.2: Apache Spark processes’ general configuration parameters and location, where NA stands for “Not Applicable”.

---
export SPARK_EXECUTOR_MEMORY=7g
export SPARK_WORKER_INSTANCES=2
export SPARK_WORKER_CORES=2
export SPARK_WORKER_DIR=/home/usr/work/sparkdata

8. Create the `conf` file using the `slaves.template` file, as follows:

   `cp slaves.template slaves`

9. Add the IP address and port of the slaves, for example “localhost” at the bottom of the text inside the `slaves` file (with `gedit` command).

10. Locate to the main Spark folder using the `cd` command from `SPARK/conf`.

11. Type `sbin/start-master.sh` to start the master.

12. Type `sbin/start-slaves.sh` to start the executors on the cluster.

13. Open a browser and go to localhost:8080

14. Copy the URL shown at the top of the page displayed, this is the IP of the Spark master, which should look similar to `spark://ipAddress:7077`. This IP should be used as the `master` parameter to the SparkContext through the SparkConf object, either with the `--master` flag on submission, or directly on the code, for example, in the place of the `masterIP` in `sc = SparkContext(masterIP, appName, sparkDir)`.  

15. Submit the application to Spark with the following code:

   `bin/spark-submit --packages com.databricks:spark-csv_2.11:1.3.0 \`sourceCode.py appName outputFolderName`

To utilise **Spark in Local mode** on a single machine, steps 1 to 4 above remain the same. After step 4, the procedure is as follows:

1. Locate to the main Spark folder with `cd` command.
2. Submit the application to Spark, typing:

```
sbin/spark-submit TimelinessQuery.py --packages com.databricks:spark-csv_2.11:1.3.0
--driver-memory 12g --master local[n]
```

Where \( n \) is the number of threads that Spark will handle simultaneously to process the tasks required to complete the processing.

### B.3 PySpark Submission to the DPSF Cluster

To submit a PySpark program to the Apache Spark Standalone mode in the Data Processing Shared Facility (DPSF) at The University of Manchester, the following steps are required:

1. Once logged in a session on the DPSF, the initial point on which the user is places is called “the login node”.

2. The login node contains a “scratch area” which is a disk storage area on a Lustre file system. All of the PySpark classes and the datasets must be stored in this area.

3. Place inside the scratch area, this can be done from the login node with `cd scratch`.

4. Create a submission script, with the information displayed on the Data Processing Shared Facility official user’s documentation for Apache Spark usage at [136]. The script utilised for this research, was set as follows, where comments within the script are the ones displayed at [136]:

```bash
#!/bin/bash
#$ -S /bin/bash
#$ -cwd
#$ -V
#$ -o outputfile.log

#$ -l mem512
#$ -pe orte-16.pe nC
```
# Generate a config file and temporary directories for the spark cluster
spark-hydra-config.sh

# Inform Spark where the config file is
export SPARK_CONF_DIR='spark-hydra-config.sh -d'

# Set up the environment (note that there is a . then a space
# at the start of the line)
. ${SPARK_CONF_DIR}/spark-env.sh

# Start the Spark cluster
start-all.sh

# Submit our spark app to the spark cluster
spark-submit --master=$MASTER_URL --verbose \ 
--packages com.databricks:spark-csv_2.11:1.3.0 \ 
--executor-memory 32G --executor-cores 2 "$1"

# Stop the Spark cluster
stop-all.sh

# OPTIONAL: Cleanup any temporary worker directories
# in the user’s scratch area
spark-hydra-config.sh -cw

Where:

(a)  
-o outfile.log indicates the script to generate a file with the given
   name, which will contain the information that would be printed out in con-
   sole.

(b)  
-l mem512 indicates the request of nodes with 512GB.

(c)  
nC needs to be replaced by the number of cores requested to the DPSF clus-
    ter. In the research nC was substituted by 32, 48, 64, and 80, as described
    in Section 4.4, Experiment E8.

(d)  
$MASTER_URL is a variable set by a custom script that allows this submission
    script to get the master IP required. The custom script utilised on the DPSF
    was created by the administrators of the cluster to allow the users utilise
Apache Spark within the facilities, and its completely transparent to the users.

(e) "$1" receives the class name as parameter, given by the submission line shown in step 5 of this listing. In this case the parameter is “TimelinessQuery.py”

5. Submit the script from the scratch area, typing:

```
qsub pyspark.sh TimelinessQuery.py
```

6. Once the job has finished, collect the runtime shown as output by the timing class (Refer to Section 3.2 for details of the timing class). The output from the PySpark classes is found on the file called `outputfile.log`.

An important note when utilising the cluster is to be aware of the defaults set by Spark and its boundaries. As shown in Section B.1, Spark offers one worker per node by default, and one executor per worker. A worker gets an allocation of all the available memory minus 1GB, recalling that this allocation comprises the resources the worker has to spawn on its executors, not its own daemon resources; the worker also gets a number of cores equal to the number available CPU cores on the node, which are 16 in the DPSF case. The executor on a cluster deployment gets 1GB of main memory, and all of the cores available on the worker. The settings are taken by node, this is, all of the nodes will have as many workers and executors as set on the configuration files, or 1 worker and 1 executor if the defaults are not changed. The number of workers can be changed as shown in the submission script [136]:

```
spark-hydra-config.sh -e SPARK_WORKER_INSTANCES=2 -e \  
SPARK_WORKER_CORES=8 -e SPARK_WORKER_MEMORY=127G
```

The number of executors can be changed by dividing the resources that the worker has. For example, to have two executors from a worker with 500GB of memory and 16 cores, the submission script should be:

```
spark-submit --master=$MASTER_URL --verbose --executor-memory 32G \  
--executor-cores 8 myapp.py arg1 arg2
```
Which will make the application to use two executors, with a total utilisation of 64GB and 16 cores. This can be useful to give the application more resources on a large capacity node, without occurring on overhead by setting several workers, which was the only option before version 1.4.0 of Spark, as mentioned in Section 4.3.2.

A final note on some limits on the memory configuration are that the executor memory cannot be set to have more memory than the memory available in the worker, otherwise the executor will not start. The executor will also fail if the worker is set to have more memory than the real memory; for example, on the DPSF with nodes of 512GB, if the worker memory is set to be of 800GB, and the executor settings indicate 800GB, the executor will cause a failure, however, the worker will not trigger any error by itself with regards to the memory. A summary of the usage above mentioned is presented in [136].
Appendix C

Testing Results

C.1 Raw Results

The following results were obtained by executing the set of experiments presented in Chapter 4 with the procedure specified in Section 4.4. These results conform the actual runtime obtained with full nine-decimal precision, separated in two parts for its presentation: the first part presents the raw results from each of the testing’s executions made, as well as its calculated average (Appendix C.1), and the second part (Appendix C.2) contains the values obtained from applying the performance metrics formulas specified in Section 4.6.1; the formulas were applied utilising the average values, not the raw ones.

E1. Original Java DQ²S experiment - PC

Table C.1.A shows the execution times in nanoseconds obtained when executing the Original Java DQ²S with 50 000, 10 000 and 1 000 000 rows. The results in seconds are shown in Table C.1.B.

E2. Python DQ²S experiment - PC

Table C.2.A shows the execution times in nanoseconds obtained when executing the Python DQ²S with 50 000, 10 000 and 1 000 000 rows. The results in seconds are shown in Table C.2.B.

E3. Optimised Python DQ²S experiment - PC

Table C.3.A shows the execution times in nanoseconds obtained when executing the Optimised Python DQ²S with 50 000, 10 000 and 1 000 000 rows. The results in seconds are shown in Table C.3.B.
E4. **Optimised Python DQ\textsuperscript{2}S maximum capacity experiment - PC**

Table \[\text{C.4}\] shows the execution times in seconds and nanoseconds obtained when executing the Optimised Python DQ\textsuperscript{2}S on a single Desktop machine, to process 11 500 000 rows.

E5. **PySpark DQ\textsuperscript{2}S Standalone mode in single machine experiment - PC**

Tables \[\text{C.5.A}\] to \[\text{C.12.A}\] show the runtimes obtained from the execution of the Timeliness query using PySpark DQ\textsuperscript{2}S instance over different configurations in a single machine with a Standalone mode, as specified in Chapter 4.

Table \[\text{C.5.A}\] shows the execution times in nanoseconds obtained from 1 worker with 1 executor, and 1 core configuration, with 50 000, 100 000 and 1 000 000 rows and 14GB of memory for the executor. Table \[\text{C.5.B}\] shows the correspondent results expressed in seconds.

Table \[\text{C.6.A}\] shows the execution times in nanoseconds obtained from 1 worker with 1 executor, 2 cores for the executor and 14GB allocated, having 50 000, 100 000 and 1 000 000 rows as input. Table \[\text{C.6.B}\] shows the correspondent results expressed in seconds.

Table \[\text{C.7.A}\] shows the execution times in nanoseconds obtained from testing the algorithm over 1 worker with 1 executor, using 3 cores and 14GB for the executor, with 50 000, 100 000 and 1 000 000 rows. Table \[\text{C.7.B}\] shows the correspondent results expressed in seconds.

Table \[\text{C.8.A}\] shows the execution times in nanoseconds obtained when using 1 worker with 1 executor, and 4 cores, with 50 000, 100 000 and 1 000 000 rows, and 14GB allocated for the executor. Table \[\text{C.8.B}\] shows the correspondent results expressed in seconds.

Table \[\text{C.9.A}\] shows the execution times in nanoseconds obtained when using 2 workers with 1 executor each. Each executor had 1 core and 7GB of memory. The results shown were obtained when processing 50 000, 100 000 and 1 000 000 rows. Table \[\text{C.9.B}\] shows the correspondent results expressed in seconds.

Table \[\text{C.10.A}\] shows the execution times in nanoseconds obtained when using 7GB for each of the 2 executors under 1 worker each one, with 2 cores each executor, and 50 000, 100 000 and 1 000 000 rows. Table \[\text{C.9.B}\] shows the correspondent results expressed in seconds.
Table C.11.A shows the execution times in nanoseconds obtained when using 1 core per each executor; one per one of the 3 workers, and having 4.7GB of memory being used for each executor. The results are shown separated for 50 000, 100 000 and 1 000 000 rows. Table C.11.B shows the correspondent results expressed in seconds.

Finally, table C.12.A shows the execution times in nanoseconds obtained when using 3.5GB for each of the 4 executors, one per each of the 4 workers, with 1 core each executor, and 50 000, 100 000 and 1 000 000 rows. Table C.12.B shows the correspondent results expressed in seconds.

E6. **PySpark DQ³S Local mode in single machine experiment- PC**

Tables C.13.A to C.20.B show the runtimes obtained from the execution of the Timeliness query using PySpark DQ³S instance in Local mode over different number of threads set in a single machine, as specified in Chapter 4.

Table C.13.A shows the execution times in nanoseconds obtained when using 1 thread for 50 000, 100 000 and 1 000 000 rows. Table C.13.B shows the correspondent results expressed in seconds.

Table C.14.A shows the execution times in nanoseconds obtained when using 2 threads for 50 000, 100 000 and 1 000 000 rows. Table C.14.B shows the correspondent results expressed in seconds.

Table C.15.A shows the execution times in nanoseconds obtained when using 3 threads for 50 000, 100 000 and 1 000 000 rows. Table C.15.B shows the correspondent results expressed in seconds.

Table C.16.A shows the execution times in nanoseconds obtained when using 4 threads for 50 000, 100 000 and 1 000 000 rows. Table C.16.B shows the correspondent results expressed in seconds.

Table C.17.A shows the execution times in nanoseconds obtained when using 5 threads for 50 000, 100 000 and 1 000 000 rows. Table C.17.B shows the correspondent results expressed in seconds.

Table C.18.A shows the execution times in nanoseconds obtained when using 6 threads for 50 000, 100 000 and 1 000 000 rows. Table C.18.B shows the correspondent results expressed in seconds.

Table C.19.A shows the execution times in nanoseconds obtained when using
7 threads for 50 000, 100 000 and 1 000 000 rows. Table C.19.B shows the correspondent results expressed in seconds.

Table C.20.A shows the execution times in nanoseconds obtained when using 8 threads for 50 000, 100 000 and 1 000 000 rows. Table C.20.B shows the correspondent results expressed in seconds.

E7. PySpark DQ²S Local mode maximum capacity experiment - PC

Table C.21.A shows the execution times in nanoseconds obtained when executing the PySpark DQ²S in Local mode in a single desktop machine, using from 1 to 6 cores to process 11 500 000 rows; table C.22.A shows the results produced when using 7 to 10, 12, and 16 cores. Table C.21.B and C.22.B shows the correspondent results expressed in seconds.

Table C.23.A shows the execution times in nanoseconds obtained when executing the PySpark DQ²S in Local mode in a single desktop machine, using from 1 to 6 cores to process 35 000 000 rows; table C.24.A shows the results produced when using 7 to 10, 12, and 16 cores. Table C.23.B and C.24.B shows the correspondent results expressed in seconds.

E8. PySpark DQ²S cluster Standalone mode experiment - Cluster

Table C.25.A shows the execution times in nanoseconds obtained when executing the PySpark DQ²S in Standalone mode in a cluster, using from 1 to 4 worker nodes, with 1 worker process and 8 executors each worker node, where each executor had 2 cores, to process 35 000 000 rows. Table C.25.B shows the correspondent results expressed in seconds.

Table C.26.A shows the execution times in nanoseconds obtained when executing the PySpark DQ²S in Standalone mode in a cluster, using from 1 to 4 worker nodes, with the same configuration of worker processes and executors as the results above but in this case to process 70 000 000 rows. Table C.26.B shows the correspondent results expressed in seconds.

Table C.27.A shows the execution times in nanoseconds obtained when executing the PySpark DQ²S in Standalone mode in a cluster, using from 1 to 4 worker nodes with the same configuration utilised on the testing above, but in this case to process 105 000 000 rows. Table C.26.B shows the correspondent results expressed in seconds.
E9. Optimised Python DQ\textsuperscript{2}S cluster experiment - Cluster

Table [C.28] shows the execution times in seconds and nanoseconds obtained when executing the Optimised Python DQ\textsuperscript{2}S with 11 500 000 rows on a cluster.

### Original Java DQ\textsuperscript{2}S experiment - PC

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>395.6086666884</td>
<td>1575.236880335</td>
<td>15975.6756062903</td>
</tr>
<tr>
<td></td>
<td>396.421228093</td>
<td>1578.989301930</td>
<td>161491.273971825</td>
</tr>
<tr>
<td></td>
<td>399.666318041</td>
<td>1605.963309986</td>
<td>161307.4351881</td>
</tr>
<tr>
<td></td>
<td>399.479739027</td>
<td>1613.074351881</td>
<td>1595.949558224</td>
</tr>
<tr>
<td>Average</td>
<td>397.591378952</td>
<td>1593.842680471</td>
<td>160624.015017364</td>
</tr>
</tbody>
</table>

Table C.1.A Original Java DQ\textsuperscript{2}S execution times in nanoseconds.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td>395.6086666884</td>
<td>1575.236880335</td>
<td>15975.6756062903</td>
</tr>
<tr>
<td></td>
<td>396.421228093</td>
<td>1578.989301930</td>
<td>161491.273971825</td>
</tr>
<tr>
<td></td>
<td>399.666318041</td>
<td>1605.963309986</td>
<td>161307.4351881</td>
</tr>
<tr>
<td></td>
<td>399.479739027</td>
<td>1613.074351881</td>
<td>1595.949558224</td>
</tr>
<tr>
<td>Average</td>
<td>397.591378952</td>
<td>1593.842680471</td>
<td>160624.015017364</td>
</tr>
</tbody>
</table>

Table C.1.B Original Java DQ\textsuperscript{2}S execution times in seconds.
### APPENDIX C. TESTING RESULTS

#### Python DQ^2S experiment - PC

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong>&lt;br&gt;(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4572926146048</td>
<td>18446547727616</td>
<td>197459350493748</td>
</tr>
<tr>
<td></td>
<td>4558974070016</td>
<td>20086998425856</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4520499258112</td>
<td>19525056004096</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4868713852928</td>
<td>18321093747968</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4541966257920</td>
<td>18716327816960</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>4612615917005</td>
<td>1901920474499</td>
<td>197459350493748</td>
</tr>
</tbody>
</table>

Table C.2.A Python DQ^2S execution times in nanoseconds.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong>&lt;br&gt;(seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4572.926146048</td>
<td>18446.547727616</td>
<td>1974593.50493748</td>
</tr>
<tr>
<td></td>
<td>4558.974070016</td>
<td>20086.998425856</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4520.499258112</td>
<td>19525.056004096</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4868.713852928</td>
<td>18321.093747968</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4541.966257920</td>
<td>18716.327816960</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>4612.615917005</td>
<td>19019.20474499</td>
<td>1974593.50493748</td>
</tr>
</tbody>
</table>

Table C.2.B Python DQ^2S execution times in seconds.

#### Optimised Python DQ^2S experiment - PC

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong>&lt;br&gt;(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>104900864</td>
<td>188679168</td>
<td>1784650240</td>
</tr>
<tr>
<td></td>
<td>107708928</td>
<td>193017088</td>
<td>1772024832</td>
</tr>
<tr>
<td></td>
<td>104930048</td>
<td>189856256</td>
<td>1778906880</td>
</tr>
<tr>
<td></td>
<td>107182080</td>
<td>190298880</td>
<td>1786706176</td>
</tr>
<tr>
<td></td>
<td>105730816</td>
<td>191415040</td>
<td>1793463040</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>106090547</td>
<td>190653286</td>
<td>1783150234</td>
</tr>
</tbody>
</table>

Table C.3.A Optimised Python DQ^2S execution times in nanoseconds.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong>&lt;br&gt;(seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.104900864</td>
<td>0.188679168</td>
<td>1.784650240</td>
</tr>
<tr>
<td></td>
<td>0.107708928</td>
<td>0.193017088</td>
<td>1.772024832</td>
</tr>
<tr>
<td></td>
<td>0.104930048</td>
<td>0.189856256</td>
<td>1.778906880</td>
</tr>
<tr>
<td></td>
<td>0.107182080</td>
<td>0.190298880</td>
<td>1.786706176</td>
</tr>
<tr>
<td></td>
<td>0.105730816</td>
<td>0.191415040</td>
<td>1.793463040</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.106090547</td>
<td>0.190653286</td>
<td>1.783150234</td>
</tr>
</tbody>
</table>

Table C.3.B Optimised Python DQ^2S execution times in seconds.
### Optimised Python DQ$^2$S maximum capacity experiment - PC

<table>
<thead>
<tr>
<th>Execution Time (11 500 000 rows)</th>
<th>nanoseconds</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>21291235840</td>
<td>21.291235840</td>
<td></td>
</tr>
<tr>
<td>21564015872</td>
<td>21.564015872</td>
<td></td>
</tr>
<tr>
<td>21319143936</td>
<td>21.319143936</td>
<td></td>
</tr>
<tr>
<td>21351192832</td>
<td>21.351192832</td>
<td></td>
</tr>
<tr>
<td>21205413632</td>
<td>21.205413632</td>
<td></td>
</tr>
</tbody>
</table>

**Average** 21346200422 21.346200422

Table C.4: Optimised Python DQ$^2$S execution times in nanoseconds and seconds for 11 500 000 rows.

### PySpark DQ$^2$S Standalone mode in single machine - 1W1C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>10662479104</td>
<td>11138216960</td>
<td>20729656832</td>
</tr>
<tr>
<td></td>
<td>10515385088</td>
<td>11112353024</td>
<td>20709535232</td>
</tr>
<tr>
<td></td>
<td>10487711744</td>
<td>11247056128</td>
<td>20715038976</td>
</tr>
<tr>
<td></td>
<td>10559790080</td>
<td>11291499008</td>
<td>20873278976</td>
</tr>
<tr>
<td></td>
<td>10610558208</td>
<td>11187699968</td>
<td>20823687168</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>10567184845</td>
<td>11195365018</td>
<td>20770239437</td>
</tr>
</tbody>
</table>

Table C.5.A Execution times in nanoseconds obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 1 core.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td>10.662479104</td>
<td>11.138216960</td>
<td>20.729656832</td>
</tr>
<tr>
<td></td>
<td>10.515385088</td>
<td>11.112353024</td>
<td>20.709535232</td>
</tr>
<tr>
<td></td>
<td>10.487711744</td>
<td>11.247056128</td>
<td>20.715038976</td>
</tr>
<tr>
<td></td>
<td>10.559790080</td>
<td>11.291499008</td>
<td>20.873278976</td>
</tr>
<tr>
<td></td>
<td>10.610558208</td>
<td>11.187699968</td>
<td>20.823687168</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>10.567184845</td>
<td>11.195365018</td>
<td>20.770239437</td>
</tr>
</tbody>
</table>

Table C.5.B Execution times in seconds obtained from PySpark DQ$^2$S Standalone mode in a single machine over 1 worker and 1 core.
### PySpark DQ^2S Standalone mode in single machine - 1W2C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8156534016</td>
<td>8737822976</td>
<td>14619853056</td>
<td></td>
</tr>
<tr>
<td>8256544768</td>
<td>8716681984</td>
<td>14629060864</td>
<td></td>
</tr>
<tr>
<td>8372070144</td>
<td>8642941952</td>
<td>14512630016</td>
<td></td>
</tr>
<tr>
<td>8278275840</td>
<td>8684564992</td>
<td>14365971968</td>
<td></td>
</tr>
<tr>
<td>8360314880</td>
<td>8824982784</td>
<td>14301459968</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>8284747930</td>
<td>8721398938</td>
<td>14485795174</td>
</tr>
</tbody>
</table>

**Table C.6.A** Execution times in nanoseconds obtained from PySpark DQ^2S Standalone mode in a single machine over 1 worker and 2 cores each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.156534016</td>
<td>8.737822976</td>
<td>14.619853056</td>
<td></td>
</tr>
<tr>
<td>8.256544768</td>
<td>8.716681984</td>
<td>14.629060864</td>
<td></td>
</tr>
<tr>
<td>8.372070144</td>
<td>8.642941952</td>
<td>14.512630016</td>
<td></td>
</tr>
<tr>
<td>8.278275840</td>
<td>8.684564992</td>
<td>14.365971968</td>
<td></td>
</tr>
<tr>
<td>8.360314880</td>
<td>8.824982784</td>
<td>14.301459968</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>8.284747930</td>
<td>8.721398938</td>
<td>14.485795174</td>
</tr>
</tbody>
</table>

**Table C.6.B** Execution times in seconds obtained from PySpark DQ^2S Standalone mode in a single machine over 1 worker and 2 cores each one.

### PySpark DQ^2S Standalone mode in single machine - 1W3C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7745988864</td>
<td>7973724928</td>
<td>12726220800</td>
<td></td>
</tr>
<tr>
<td>7787958016</td>
<td>7960871936</td>
<td>12421887744</td>
<td></td>
</tr>
<tr>
<td>7811255040</td>
<td>7848456960</td>
<td>12290481664</td>
<td></td>
</tr>
<tr>
<td>7710612992</td>
<td>7957158144</td>
<td>1222852864</td>
<td></td>
</tr>
<tr>
<td>7812652032</td>
<td>7915779840</td>
<td>12260795136</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>7773693389</td>
<td>7931198362</td>
<td>1238444764</td>
</tr>
</tbody>
</table>

**Table C.7.A** Execution times in nanoseconds obtained from PySpark DQ^2S Standalone mode in a single machine over 1 worker and 3 cores each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.745988864</td>
<td>7.973724928</td>
<td>12.726220800</td>
<td></td>
</tr>
<tr>
<td>7.787958016</td>
<td>7.960871936</td>
<td>12.421887744</td>
<td></td>
</tr>
<tr>
<td>7.811255040</td>
<td>7.848456960</td>
<td>12.290481664</td>
<td></td>
</tr>
<tr>
<td>7.710612992</td>
<td>7.957158144</td>
<td>12.22852864</td>
<td></td>
</tr>
<tr>
<td>7.812652032</td>
<td>7.915779840</td>
<td>12.260795136</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>7.773693389</td>
<td>7.931198362</td>
<td>12.38444764</td>
</tr>
</tbody>
</table>

**Table C.7.B** Execution times in seconds obtained from PySpark DQ^2S Standalone mode in a single machine over 1 worker and 3 cores each one.
### C.1. RAW RESULTS

#### PySpark DQ²S Standalone mode in single machine - 1W4C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7369091072</td>
<td>7608481024</td>
<td>11512333056</td>
</tr>
<tr>
<td></td>
<td>7349709056</td>
<td>7576862976</td>
<td>11952416000</td>
</tr>
<tr>
<td></td>
<td>7364735744</td>
<td>7625577856</td>
<td>11885402880</td>
</tr>
<tr>
<td></td>
<td>7365112064</td>
<td>7629357056</td>
<td>11787275008</td>
</tr>
<tr>
<td></td>
<td>7426702080</td>
<td>7760252928</td>
<td>11710218752</td>
</tr>
<tr>
<td>Average</td>
<td>7357410560</td>
<td>7639706368</td>
<td>11769529139</td>
</tr>
</tbody>
</table>

Table C.8.A Execution times in nanoseconds obtained from PySpark DQ²S Standalone mode in a single machine over 1 worker and 4 cores each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.369091072</td>
<td>7.608481024</td>
<td>11.512333056</td>
</tr>
<tr>
<td></td>
<td>7.349709056</td>
<td>7.576862976</td>
<td>11.952416000</td>
</tr>
<tr>
<td></td>
<td>7.364735744</td>
<td>7.625577856</td>
<td>11.885402880</td>
</tr>
<tr>
<td></td>
<td>7.365112064</td>
<td>7.629357056</td>
<td>11.787275008</td>
</tr>
<tr>
<td></td>
<td>7.426702080</td>
<td>7.760252928</td>
<td>11.710218752</td>
</tr>
<tr>
<td>Average</td>
<td>7.357410560</td>
<td>7.639706368</td>
<td>11.769529139</td>
</tr>
</tbody>
</table>

Table C.8.B Execution times in seconds obtained from PySpark DQ²S Standalone mode in a single machine over 1 worker and 4 cores each one.

#### PySpark DQ²S Standalone mode in single machine - 2W1C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10471302912</td>
<td>10722405120</td>
<td>16472211968</td>
</tr>
<tr>
<td></td>
<td>10219045120</td>
<td>10681270784</td>
<td>16221628248</td>
</tr>
<tr>
<td></td>
<td>10336449280</td>
<td>10786972928</td>
<td>16420616192</td>
</tr>
<tr>
<td></td>
<td>10162857984</td>
<td>10617496064</td>
<td>16653227264</td>
</tr>
<tr>
<td></td>
<td>10332693760</td>
<td>10810214912</td>
<td>16090560000</td>
</tr>
<tr>
<td>Average</td>
<td>10304469811</td>
<td>10723671962</td>
<td>16371575654</td>
</tr>
</tbody>
</table>

Table C.9.A Execution times in nanoseconds obtained from PySpark DQ²S Standalone mode in a single machine over 2 workers and 1 core each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.471302912</td>
<td>10.722405120</td>
<td>16.472211968</td>
</tr>
<tr>
<td></td>
<td>10.219045120</td>
<td>10.681270784</td>
<td>16.221628248</td>
</tr>
<tr>
<td></td>
<td>10.336449280</td>
<td>10.786972928</td>
<td>16.420616192</td>
</tr>
<tr>
<td></td>
<td>10.162857984</td>
<td>10.617496064</td>
<td>16.653227264</td>
</tr>
<tr>
<td></td>
<td>10.332693760</td>
<td>10.810214912</td>
<td>16.090560000</td>
</tr>
<tr>
<td>Average</td>
<td>10.304469811</td>
<td>10.723671962</td>
<td>16.371575654</td>
</tr>
</tbody>
</table>

Table C.9.B Execution times in seconds obtained from PySpark DQ²S Standalone mode in a single machine over 2 workers and 1 core each one.
## APPENDIX C. TESTING RESULTS

### PySpark DQ^2S Standalone mode in single machine - 2W2C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9165194240</td>
<td>9250158080</td>
<td>13457061888</td>
<td></td>
</tr>
<tr>
<td>9152016128</td>
<td>9248869120</td>
<td>13512259072</td>
<td></td>
</tr>
<tr>
<td>9085235712</td>
<td>9415583744</td>
<td>13250208768</td>
<td></td>
</tr>
<tr>
<td>9009185792</td>
<td>9312792064</td>
<td>13469066752</td>
<td></td>
</tr>
<tr>
<td>9051820800</td>
<td>9383379968</td>
<td>13320818944</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>9092690534</td>
<td>9358156595</td>
<td>13401883085</td>
</tr>
</tbody>
</table>

Table C.10.A Execution times in nanoseconds obtained from PySpark DQ^2S Standalone mode in a single machine over 2 workers and 2 cores each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.165194240</td>
<td>9.25015808</td>
<td>13.457061888</td>
<td></td>
</tr>
<tr>
<td>9.152016128</td>
<td>9.42886912</td>
<td>13.512259072</td>
<td></td>
</tr>
<tr>
<td>9.085235712</td>
<td>9.41558374</td>
<td>13.250208768</td>
<td></td>
</tr>
<tr>
<td>9.009185792</td>
<td>9.31279206</td>
<td>13.469066752</td>
<td></td>
</tr>
<tr>
<td>9.051820800</td>
<td>9.38337996</td>
<td>13.320818944</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>9.09269053</td>
<td>9.358156595</td>
<td>13.401883085</td>
</tr>
</tbody>
</table>

Table C.10.B Execution times in seconds obtained from PySpark DQ^2S Standalone mode in a single machine over 2 workers and 2 cores each one.

### PySpark DQ^2S Standalone mode in single machine - 3W1C

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10721553920</td>
<td>11476656896</td>
<td>15787690752</td>
<td></td>
</tr>
<tr>
<td>10942905088</td>
<td>11374004992</td>
<td>15779695104</td>
<td></td>
</tr>
<tr>
<td>10598299904</td>
<td>11213497088</td>
<td>16031830016</td>
<td></td>
</tr>
<tr>
<td>10985516032</td>
<td>11401945856</td>
<td>15731942912</td>
<td></td>
</tr>
<tr>
<td>10821647872</td>
<td>11572036096</td>
<td>15983724800</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>10813984563</td>
<td>11407628186</td>
<td>15862976717</td>
</tr>
</tbody>
</table>

Table C.11.A Execution times in nanoseconds obtained from PySpark DQ^2S Standalone mode in a single machine over 3 workers and 1 core each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.721553920</td>
<td>11.476656896</td>
<td>15.787690752</td>
<td></td>
</tr>
<tr>
<td>10.942905088</td>
<td>11.374004992</td>
<td>15.779695104</td>
<td></td>
</tr>
<tr>
<td>10.598299904</td>
<td>11.213497088</td>
<td>16.031830016</td>
<td></td>
</tr>
<tr>
<td>10.985516032</td>
<td>11.401945856</td>
<td>15.731942912</td>
<td></td>
</tr>
<tr>
<td>10.821647872</td>
<td>11.572036096</td>
<td>15.983724800</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>10.813984563</td>
<td>11.407628186</td>
<td>15.862976717</td>
</tr>
</tbody>
</table>

Table C.11.B Execution times in seconds obtained from PySpark DQ^2S Standalone mode in a single machine over 3 workers and 1 core each one.
### C.1. RAW RESULTS

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PySpark DQ^2S Standalone mode in single machine - 4W1C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>12171860992</td>
<td>12360322048</td>
<td>16538985728</td>
</tr>
<tr>
<td></td>
<td>12321287168</td>
<td>12490898176</td>
<td>16732512768</td>
</tr>
<tr>
<td></td>
<td>12258028032</td>
<td>12256569856</td>
<td>16582315008</td>
</tr>
<tr>
<td></td>
<td>12100891904</td>
<td>12700909312</td>
<td>16402367232</td>
</tr>
<tr>
<td></td>
<td>12105985280</td>
<td>12240269312</td>
<td>16489517824</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>12191610675</td>
<td>12409793741</td>
<td>16549139712</td>
</tr>
</tbody>
</table>

Table C.12.A Execution times in nanoseconds obtained from PySpark DQ^2S Standalone mode in a single machine over 4 workers and 1 core each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PySpark DQ^2S Local mode using 1 core in a single machine</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td>12.171860992</td>
<td>12.360322048</td>
<td>16.538985728</td>
</tr>
<tr>
<td></td>
<td>12.321287168</td>
<td>12.490898176</td>
<td>16.732512768</td>
</tr>
<tr>
<td></td>
<td>12.258028032</td>
<td>12.256569856</td>
<td>16.582315008</td>
</tr>
<tr>
<td></td>
<td>12.100891904</td>
<td>12.700909312</td>
<td>16.402367232</td>
</tr>
<tr>
<td></td>
<td>12.105985280</td>
<td>12.240269312</td>
<td>16.489517824</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>12.191610675</td>
<td>12.409793741</td>
<td>16.549139712</td>
</tr>
</tbody>
</table>

Table C.12.B Execution times in seconds obtained from PySpark DQ^2S Standalone mode in a single machine over 4 workers and 1 core each one.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PySpark DQ^2S Local mode in single machine - local[1]</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>6873299968</td>
<td>7561091840</td>
<td>17818112768</td>
</tr>
<tr>
<td></td>
<td>6982749952</td>
<td>7544175872</td>
<td>1762320976</td>
</tr>
<tr>
<td></td>
<td>6865008896</td>
<td>7562930688</td>
<td>17520658944</td>
</tr>
<tr>
<td></td>
<td>6953572096</td>
<td>7614562048</td>
<td>17765099008</td>
</tr>
<tr>
<td></td>
<td>6918232832</td>
<td>7667899136</td>
<td>17745416704</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>6918572749</td>
<td>7590131917</td>
<td>17694503680</td>
</tr>
</tbody>
</table>

Table C.13.A Execution times in nanoseconds obtained from PySpark DQ^2S Local mode using 1 core in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PySpark DQ^2S Local mode using 1 core in a single machine</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td>6.873299968</td>
<td>7.561091840</td>
<td>17.818112768</td>
</tr>
<tr>
<td></td>
<td>6.982749952</td>
<td>7.544175872</td>
<td>17.62320976</td>
</tr>
<tr>
<td></td>
<td>6.865008896</td>
<td>7.562930688</td>
<td>17.520658944</td>
</tr>
<tr>
<td></td>
<td>6.953572096</td>
<td>7.614562048</td>
<td>17.765099008</td>
</tr>
<tr>
<td></td>
<td>6.918232832</td>
<td>7.667899136</td>
<td>17.745416704</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>6.918572749</td>
<td>7.590131917</td>
<td>17.694503680</td>
</tr>
</tbody>
</table>

Table C.13.B Execution times in seconds obtained from PySpark DQ^2S Local mode using 1 core in a single machine.
### APPENDIX C. TESTING RESULTS

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50,000 rows</th>
<th>10,000 rows</th>
<th>1,000,000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time (nanoseconds)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Local mode in single machine - local[2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54,529,840,64</td>
<td>58,028,349,44</td>
<td>115,430,172,16</td>
<td></td>
</tr>
<tr>
<td>54,129,617,92</td>
<td>58,361,218,56</td>
<td>116,203,202,56</td>
<td></td>
</tr>
<tr>
<td>55,334,440,96</td>
<td>58,903,587,84</td>
<td>117,737,349,12</td>
<td></td>
</tr>
<tr>
<td>55,268,567,04</td>
<td>59,443,240,96</td>
<td>118,859,648,00</td>
<td></td>
</tr>
<tr>
<td>54,587,568,64</td>
<td>57,520,522,24</td>
<td>117,639,331,84</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>54,770,007,04</td>
<td>58,451,383,81</td>
<td>117,173,940,74</td>
</tr>
</tbody>
</table>

Table C.14.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 2 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50,000 rows</th>
<th>10,000 rows</th>
<th>1,000,000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time (seconds)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Local mode in single machine - local[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.452,984,06</td>
<td>5.802,834,94</td>
<td>11.543,017,216</td>
<td></td>
</tr>
<tr>
<td>5.412,961,79</td>
<td>5.836,121,85</td>
<td>11.620,320,256</td>
<td></td>
</tr>
<tr>
<td>5.334,440,96</td>
<td>5.890,358,78</td>
<td>11.773,734,912</td>
<td></td>
</tr>
<tr>
<td>5.268,567,04</td>
<td>5.944,324,09</td>
<td>11.885,964,80</td>
<td></td>
</tr>
<tr>
<td>5.487,568,64</td>
<td>5.752,052,22</td>
<td>11.763,933,18</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>5.477,000,70</td>
<td>5.845,138,38</td>
<td>11.717,394,07</td>
</tr>
</tbody>
</table>

Table C.14.B Execution times in seconds obtained from PySpark DQ²S Local mode using 2 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50,000 rows</th>
<th>10,000 rows</th>
<th>1,000,000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time (nanoseconds)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Local mode in single machine - local[4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50,075,491,84</td>
<td>53,383,759,36</td>
<td>108,723,092,48</td>
<td></td>
</tr>
<tr>
<td>50,621,742,08</td>
<td>53,248,409,60</td>
<td>98,712,812,94</td>
<td></td>
</tr>
<tr>
<td>50,734,661,12</td>
<td>52,988,500,48</td>
<td>100,217,390,08</td>
<td></td>
</tr>
<tr>
<td>50,128,332,80</td>
<td>53,601,502,72</td>
<td>103,051,461,12</td>
<td></td>
</tr>
<tr>
<td>50,490,600,96</td>
<td>53,170,106,88</td>
<td>102,932,341,76</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>50,410,165,76</td>
<td>53,278,455,81</td>
<td>102,724,229,28</td>
</tr>
</tbody>
</table>

Table C.15.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 3 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50,000 rows</th>
<th>10,000 rows</th>
<th>1,000,000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time (seconds)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Local mode in single machine - local[4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.007,549,18</td>
<td>5.338,375,93</td>
<td>10.872,309,24</td>
<td></td>
</tr>
<tr>
<td>5.062,174,20</td>
<td>5.324,840,96</td>
<td>9.871,282,94</td>
<td></td>
</tr>
<tr>
<td>5.073,466,11</td>
<td>5.298,850,04</td>
<td>10.021,739,08</td>
<td></td>
</tr>
<tr>
<td>5.012,833,28</td>
<td>5.360,150,27</td>
<td>10.305,146,12</td>
<td></td>
</tr>
<tr>
<td>5.049,060,96</td>
<td>5.317,010,68</td>
<td>10.293,234,17</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>5.041,016,57</td>
<td>5.327,845,58</td>
<td>10.272,742,29</td>
</tr>
</tbody>
</table>

Table C.15.B Execution times in seconds obtained from PySpark DQ²S Local mode using 3 cores in a single machine.
### PySpark DQ<sup>2</sup>S Local mode in single machine - local[4]

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4945903104</td>
<td>5133988096</td>
<td>8970474752</td>
<td></td>
</tr>
<tr>
<td>4948933120</td>
<td>5181114112</td>
<td>8793011968</td>
<td></td>
</tr>
<tr>
<td>4965520896</td>
<td>5096628992</td>
<td>9088845056</td>
<td></td>
</tr>
<tr>
<td>4910916864</td>
<td>526072688</td>
<td>870254920</td>
<td></td>
</tr>
<tr>
<td>4972674048</td>
<td>5172325120</td>
<td>8819279872</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>4948789606</td>
<td>5168955802</td>
<td>8874831514</td>
</tr>
</tbody>
</table>

Table C.16.A Execution times in nanoseconds obtained from PySpark DQ<sup>2</sup>S Local mode using 4 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.945903104</td>
<td>5.133988096</td>
<td>8.970474752</td>
<td></td>
</tr>
<tr>
<td>4.948933120</td>
<td>5.181114112</td>
<td>8.793011968</td>
<td></td>
</tr>
<tr>
<td>4.965520896</td>
<td>5.096628992</td>
<td>9.08845056</td>
<td></td>
</tr>
<tr>
<td>4.910916864</td>
<td>5.26072688</td>
<td>8.70254920</td>
<td></td>
</tr>
<tr>
<td>4.972674048</td>
<td>5.172325120</td>
<td>8.819279872</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>4.948789606</td>
<td>5.168955802</td>
<td>8.874831514</td>
</tr>
</tbody>
</table>

Table C.16.B Execution times in seconds obtained from PySpark DQ<sup>2</sup>S Local mode using 4 cores in a single machine.
### APPENDIX C. TESTING RESULTS

#### PySpark DQ²S Local mode in single machine - local[5]

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>5198445056</td>
<td>548340032</td>
<td>8917495040</td>
</tr>
<tr>
<td></td>
<td>5130334720</td>
<td>558270032</td>
<td>8968935168</td>
</tr>
<tr>
<td></td>
<td>5182439936</td>
<td>5432392960</td>
<td>8896977920</td>
</tr>
<tr>
<td></td>
<td>5116220928</td>
<td>5463419136</td>
<td>9013767936</td>
</tr>
<tr>
<td></td>
<td>5130665984</td>
<td>5461233152</td>
<td>8936705792</td>
</tr>
<tr>
<td>Average</td>
<td>5151621325</td>
<td>5484617062</td>
<td>8946776371</td>
</tr>
</tbody>
</table>

Table C.17.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 5 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td>5.198445056</td>
<td>5.48340032</td>
<td>8.917495040</td>
</tr>
<tr>
<td></td>
<td>5.130334720</td>
<td>5.58270032</td>
<td>8.968935168</td>
</tr>
<tr>
<td></td>
<td>5.182439936</td>
<td>5.432392960</td>
<td>8.896977920</td>
</tr>
<tr>
<td></td>
<td>5.116220928</td>
<td>5.463419136</td>
<td>9.013767936</td>
</tr>
<tr>
<td></td>
<td>5.130665984</td>
<td>5.461233152</td>
<td>8.936705792</td>
</tr>
<tr>
<td>Average</td>
<td>5.151621325</td>
<td>5.484617062</td>
<td>8.946776371</td>
</tr>
</tbody>
</table>

Table C.17.B Execution times in seconds obtained from PySpark DQ²S Local mode using 5 cores in a single machine.

#### PySpark DQ²S Local mode in single machine - local[6]

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>4805391104</td>
<td>5094859776</td>
<td>8835384064</td>
</tr>
<tr>
<td></td>
<td>4737410304</td>
<td>5095909888</td>
<td>8789655808</td>
</tr>
<tr>
<td></td>
<td>4778916096</td>
<td>5063123712</td>
<td>8925155328</td>
</tr>
<tr>
<td></td>
<td>4782843904</td>
<td>5035049984</td>
<td>8858992128</td>
</tr>
<tr>
<td></td>
<td>4723966976</td>
<td>5088356864</td>
<td>8657932032</td>
</tr>
<tr>
<td>Average</td>
<td>4765705677</td>
<td>5075460045</td>
<td>8813423872</td>
</tr>
</tbody>
</table>

Table C.18.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 6 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td>4.805391104</td>
<td>5.094859776</td>
<td>8.835384064</td>
</tr>
<tr>
<td></td>
<td>4.737410304</td>
<td>5.095909888</td>
<td>8.789655808</td>
</tr>
<tr>
<td></td>
<td>4.778916096</td>
<td>5.063123712</td>
<td>8.925155328</td>
</tr>
<tr>
<td></td>
<td>4.782843904</td>
<td>5.035049984</td>
<td>8.858992128</td>
</tr>
<tr>
<td></td>
<td>4.723966976</td>
<td>5.088356864</td>
<td>8.657932032</td>
</tr>
<tr>
<td>Average</td>
<td>4.765705677</td>
<td>5.075460045</td>
<td>8.813423872</td>
</tr>
</tbody>
</table>

Table C.18.B Execution times in seconds obtained from PySpark DQ²S Local mode using 6 cores in a single machine.
### C.1. RAW RESULTS

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PySpark DQ²S Local mode in single machine - local[7]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 000 rows</td>
<td>4663134208</td>
<td>5044144896</td>
<td>8720456960</td>
</tr>
<tr>
<td>4644529920</td>
<td>501870032</td>
<td>8878997760</td>
<td></td>
</tr>
<tr>
<td>4660646144</td>
<td>497775104</td>
<td>8612283136</td>
<td></td>
</tr>
<tr>
<td>4693438976</td>
<td>509901320</td>
<td>8710627840</td>
<td></td>
</tr>
<tr>
<td>4659883008</td>
<td>5034482176</td>
<td>8714520064</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4664326451</td>
<td>5038823066</td>
<td>8727377152</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PySpark DQ²S Local mode in single machine - local[8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 000 rows</td>
<td>4.663134208</td>
<td>5.044144896</td>
<td>8.720456960</td>
</tr>
<tr>
<td>4.644529920</td>
<td>5.01870032</td>
<td>8.878997760</td>
<td></td>
</tr>
<tr>
<td>4.660646144</td>
<td>4.97775104</td>
<td>8.612283136</td>
<td></td>
</tr>
<tr>
<td>4.693438976</td>
<td>5.09901320</td>
<td>8.710627840</td>
<td></td>
</tr>
<tr>
<td>4.659883008</td>
<td>5.034482176</td>
<td>8.714520064</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4.664326451</td>
<td>5.038823066</td>
<td>8.727377152</td>
</tr>
</tbody>
</table>

Table C.19.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 7 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PySpark DQ²S Local mode in single machine - local[8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 000 rows</td>
<td>4607827200</td>
<td>4969145088</td>
<td>9088124928</td>
</tr>
<tr>
<td>4634355712</td>
<td>498546976</td>
<td>9099651072</td>
<td></td>
</tr>
<tr>
<td>4603010816</td>
<td>5042586112</td>
<td>9532989828</td>
<td></td>
</tr>
<tr>
<td>4640083968</td>
<td>5011299840</td>
<td>9506847744</td>
<td></td>
</tr>
<tr>
<td>4615563776</td>
<td>4959327744</td>
<td>9632914944</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4620168294</td>
<td>4993553152</td>
<td>9372105523</td>
</tr>
</tbody>
</table>

Table C.19.B Execution times in seconds obtained from PySpark DQ²S Local mode using 7 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PySpark DQ²S Local mode in single machine - local[8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 000 rows</td>
<td>4.607827200</td>
<td>4.969145088</td>
<td>9.088124928</td>
</tr>
<tr>
<td>4.634355712</td>
<td>4.98546976</td>
<td>9.099651072</td>
<td></td>
</tr>
<tr>
<td>4.603010816</td>
<td>5.042586112</td>
<td>9.532989828</td>
<td></td>
</tr>
<tr>
<td>4.640083968</td>
<td>5.011299840</td>
<td>9.506847744</td>
<td></td>
</tr>
<tr>
<td>4.615563776</td>
<td>4.959327744</td>
<td>9.632914944</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4.620168294</td>
<td>4.993553152</td>
<td>9.372105523</td>
</tr>
</tbody>
</table>

Table C.20.A Execution times in nanoseconds obtained from PySpark DQ²S Local mode using 8 cores in a single machine.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>50 000 rows</th>
<th>10 000 rows</th>
<th>1 000 000 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PySpark DQ²S Local mode in single machine - local[8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 000 rows</td>
<td>4.607827200</td>
<td>4.969145088</td>
<td>9.088124928</td>
</tr>
<tr>
<td>4.634355712</td>
<td>4.98546976</td>
<td>9.099651072</td>
<td></td>
</tr>
<tr>
<td>4.603010816</td>
<td>5.042586112</td>
<td>9.532989828</td>
<td></td>
</tr>
<tr>
<td>4.640083968</td>
<td>5.011299840</td>
<td>9.506847744</td>
<td></td>
</tr>
<tr>
<td>4.615563776</td>
<td>4.959327744</td>
<td>9.632914944</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4.620168294</td>
<td>4.993553152</td>
<td>9.372105523</td>
</tr>
</tbody>
</table>

Table C.20.B Execution times in seconds obtained from PySpark DQ²S Local mode using 8 cores in a single machine.
### APPENDIX C. TESTING RESULTS

#### PySpark DQ\(^2\)S Local mode maximum capacity - local[1-6]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time 11 500 000 rows (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131589269248</td>
<td>7.3765188096</td>
<td>6.078119936</td>
<td>5.0822084096</td>
<td>4.7598737920</td>
<td>4.5034638080</td>
<td></td>
</tr>
<tr>
<td>132989397184</td>
<td>5.4388361984</td>
<td>5.9746337388</td>
<td>5.0732338176</td>
<td>4.7743496960</td>
<td>4.4813570048</td>
<td></td>
</tr>
<tr>
<td>132248972800</td>
<td>7.5507175880</td>
<td>5.7896614976</td>
<td>5.0493220096</td>
<td>4.6637328604</td>
<td>4.5086447872</td>
<td></td>
</tr>
<tr>
<td>132398460016</td>
<td>7.573619968</td>
<td>6.0409650944</td>
<td>4.9736456304</td>
<td>4.7446155264</td>
<td>4.4683508928</td>
<td></td>
</tr>
<tr>
<td>132176488192</td>
<td>7.7783117824</td>
<td>5.7602428160</td>
<td>5.0325387776</td>
<td>4.6466738324</td>
<td>4.4617302016</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>132278962688</td>
<td>7.536347590</td>
<td>5.9279237581</td>
<td>5.0405019034</td>
<td>4.717371366</td>
<td>4.4847133389</td>
</tr>
</tbody>
</table>

Table C.21.A Execution times in nanoseconds obtained from PySpark DQ\(^2\)S Local mode using 1 to 6 cores in a single machine for 11 500 000 rows.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time 11 500 000 rows (seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43831034880</td>
<td>43.774910976</td>
<td>44.017587986</td>
<td>44.87532928</td>
<td>44.712393984</td>
<td>45.03220376</td>
<td></td>
</tr>
<tr>
<td>44153880832</td>
<td>43.759703040</td>
<td>44.57311388</td>
<td>44.55742800</td>
<td>44.39171840</td>
<td>45.02331494</td>
<td></td>
</tr>
<tr>
<td>43866841856</td>
<td>44.747205120</td>
<td>44.671670784</td>
<td>44.533266944</td>
<td>44.352089088</td>
<td>45.061735184</td>
<td></td>
</tr>
<tr>
<td>43853100896</td>
<td>44.91389920</td>
<td>44.665667284</td>
<td>44.111430880</td>
<td>44.247562976</td>
<td>44.25866928</td>
<td></td>
</tr>
<tr>
<td>43825918208</td>
<td>43.71859152</td>
<td>44.837411968</td>
<td>44.095053412</td>
<td>44.282157824</td>
<td>44.64708032</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>43.869631334</td>
<td>43.710804838</td>
<td>44.402970214</td>
<td>44.434947533</td>
<td>44.406692966</td>
<td>44.838683904</td>
</tr>
</tbody>
</table>

Table C.21.B Execution times in seconds obtained from PySpark DQ\(^2\)S Local mode using 1 to 6 cores in a single machine for 11 500 000 rows.

#### PySpark DQ\(^2\)S Local mode maximum capacity - local[7-10, 12, 16]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time 11 500 000 rows (nanoseconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.831034880</td>
<td>43.774910976</td>
<td>44.017587986</td>
<td>44.87532928</td>
<td>44.712393984</td>
<td>45.03220376</td>
<td></td>
</tr>
<tr>
<td>44.153880832</td>
<td>43.759703040</td>
<td>44.57311388</td>
<td>44.55742800</td>
<td>44.39171840</td>
<td>45.02331494</td>
<td></td>
</tr>
<tr>
<td>43.866841856</td>
<td>44.747205120</td>
<td>44.671670784</td>
<td>44.533266944</td>
<td>44.352089088</td>
<td>45.061735184</td>
<td></td>
</tr>
<tr>
<td>43.853100896</td>
<td>44.91389920</td>
<td>44.665667284</td>
<td>44.111430880</td>
<td>44.247562976</td>
<td>44.25866928</td>
<td></td>
</tr>
<tr>
<td>43.825918208</td>
<td>43.71859152</td>
<td>44.837411968</td>
<td>44.095053412</td>
<td>44.282157824</td>
<td>44.64708032</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>43.869631334</td>
<td>43.710804838</td>
<td>44.402970214</td>
<td>44.434947533</td>
<td>44.406692966</td>
<td>44.838683904</td>
</tr>
</tbody>
</table>

Table C.22.A Execution times in nanoseconds obtained from PySpark DQ\(^2\)S Local mode using 7 to 10, 12, and 16 cores in a single machine for 11 500 000 rows.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time 11 500 000 rows (seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.831034880</td>
<td>43.774910976</td>
<td>44.017587986</td>
<td>44.87532928</td>
<td>44.712393984</td>
<td>45.03220376</td>
<td></td>
</tr>
<tr>
<td>44.153880832</td>
<td>43.759703040</td>
<td>44.57311388</td>
<td>44.55742800</td>
<td>44.39171840</td>
<td>45.02331494</td>
<td></td>
</tr>
<tr>
<td>43.866841856</td>
<td>44.747205120</td>
<td>44.671670784</td>
<td>44.533266944</td>
<td>44.352089088</td>
<td>45.061735184</td>
<td></td>
</tr>
<tr>
<td>43.853100896</td>
<td>44.91389920</td>
<td>44.665667284</td>
<td>44.111430880</td>
<td>44.247562976</td>
<td>44.25866928</td>
<td></td>
</tr>
<tr>
<td>43.825918208</td>
<td>43.71859152</td>
<td>44.837411968</td>
<td>44.095053412</td>
<td>44.282157824</td>
<td>44.64708032</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>43.869631334</td>
<td>43.710804838</td>
<td>44.402970214</td>
<td>44.434947533</td>
<td>44.406692966</td>
<td>44.838683904</td>
</tr>
</tbody>
</table>

Table C.22.B Execution times in seconds obtained from PySpark DQ\(^2\)S Local mode using 7 to 10, 12, and 16 cores in a single machine for 11 500 000 rows.
## C.1. RAW RESULTS

### PySpark DQ^2S Local mode maximum capacity - local[1-6]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>387859.98994</td>
<td>226735.00024</td>
<td>169494.21318</td>
<td>144895.91475</td>
<td>131405.68846</td>
<td>125550.75572</td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td>389712.54144</td>
<td>218825.73872</td>
<td>170425.14112</td>
<td>143335.79092</td>
<td>131355.748038</td>
<td>125242.98340</td>
</tr>
<tr>
<td>Average</td>
<td>388221.75708</td>
<td>21096.5992522</td>
<td>181715.21292</td>
<td>143575.68777</td>
<td>131355.748038</td>
<td>125242.98340</td>
</tr>
</tbody>
</table>

Table C.23.A Execution times in nanoseconds obtained from PySpark DQ^2S Local mode using 1 to 6 cores in a single machine for 35 000 000 rows.

### PySpark DQ^2S Local mode maximum capacity - local[7-10, 12, 16]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>1239626.01728</td>
<td>1227843.25072</td>
<td>12372497.8176</td>
<td>1237295.1032</td>
<td>12372376.040</td>
<td>12547467.1872</td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td>1241234.02112</td>
<td>1229473.2006</td>
<td>123591.26572</td>
<td>12411196.723</td>
<td>12551147.856</td>
<td>12560631.232</td>
</tr>
<tr>
<td>Average</td>
<td>1240532.92762</td>
<td>1232501.7075</td>
<td>123643.004365</td>
<td>123643.004365</td>
<td>123643.004365</td>
<td>123643.004365</td>
</tr>
</tbody>
</table>

Table C.24.A Execution times in nanoseconds obtained from PySpark DQ^2S Local mode using 7 to 10, 12, and 16 cores in a single machine for 35 000 000 rows.

### PySpark DQ^2S Local mode maximum capacity - local[7-10, 12, 16]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (nanoseconds)</td>
<td>1239626.01728</td>
<td>1227843.25072</td>
<td>12372497.8176</td>
<td>1237295.1032</td>
<td>12372376.040</td>
<td>12547467.1872</td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td>1241234.02112</td>
<td>1229473.2006</td>
<td>123591.26572</td>
<td>12411196.723</td>
<td>12551147.856</td>
<td>12560631.232</td>
</tr>
<tr>
<td>Average</td>
<td>1240532.92762</td>
<td>1232501.7075</td>
<td>123643.004365</td>
<td>123643.004365</td>
<td>123643.004365</td>
<td>123643.004365</td>
</tr>
</tbody>
</table>

Table C.24.B Execution times in nanoseconds obtained from PySpark DQ^2S Local mode using 7 to 10, 12, and 16 cores in a single machine for 35 000 000 rows.
### PySpark DQ²S Standalone mode on a cluster - 35 000 000 rows

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35 000 000 rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66954494208</td>
<td>40034502144</td>
<td>36861419008</td>
<td>2967743872</td>
<td></td>
</tr>
<tr>
<td>71653293056</td>
<td>40291827968</td>
<td>31768627968</td>
<td>28998573568</td>
<td></td>
</tr>
<tr>
<td>74397865216</td>
<td>39341289984</td>
<td>31986951936</td>
<td>28249470976</td>
<td></td>
</tr>
<tr>
<td>77074788864</td>
<td>39420195072</td>
<td>32379415040</td>
<td>28117914112</td>
<td></td>
</tr>
<tr>
<td>73011555840</td>
<td>39189282816</td>
<td>33078972160</td>
<td>28194881024</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>72618399437</td>
<td>39655419597</td>
<td>33215077222</td>
<td>28647716710</td>
</tr>
</tbody>
</table>

Table C.25.A Execution times in nanoseconds obtained from PySpark DQ²S Standalone mode processing 35 000 000 rows with 1 to 4 worker nodes on a cluster.

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35 000 000 rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66954494208</td>
<td>40.034502144</td>
<td>36.861419008</td>
<td>29.67743872</td>
<td></td>
</tr>
<tr>
<td>71653293056</td>
<td>40.291827968</td>
<td>31.768627968</td>
<td>28.998573568</td>
<td></td>
</tr>
<tr>
<td>74397865216</td>
<td>39.341289984</td>
<td>31.986951936</td>
<td>28.249470976</td>
<td></td>
</tr>
<tr>
<td>77074788864</td>
<td>39.420195072</td>
<td>32.379415040</td>
<td>28.117914112</td>
<td></td>
</tr>
<tr>
<td>73011555840</td>
<td>39.189282816</td>
<td>33.078972160</td>
<td>28.194881024</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>72.618399437</td>
<td>39.655419597</td>
<td>33.215077222</td>
<td>28.647716710</td>
</tr>
</tbody>
</table>

Table C.25.B Execution times in seconds obtained from PySpark DQ²S Standalone mode processing 35 000 000 rows with 1 to 4 worker nodes on a cluster.

### PySpark DQ²S Standalone mode on a cluster - 70 000 000 rows

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70 000 000 rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(nanoseconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101095668992</td>
<td>68086117888</td>
<td>54591626240</td>
<td>48209958144</td>
<td></td>
</tr>
<tr>
<td>134146179072</td>
<td>66611641856</td>
<td>5330426880</td>
<td>49384262912</td>
<td></td>
</tr>
<tr>
<td>104077149184</td>
<td>67483824896</td>
<td>58199030272</td>
<td>4552409024</td>
<td></td>
</tr>
<tr>
<td>105101277184</td>
<td>70298240000</td>
<td>58156339200</td>
<td>4414183040</td>
<td></td>
</tr>
<tr>
<td>131581689088</td>
<td>70488784896</td>
<td>54432688896</td>
<td>42770220032</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>115200392704</td>
<td>68593721907</td>
<td>56542022298</td>
<td>4600006630</td>
</tr>
</tbody>
</table>

Table C.26.A Execution times in nanoseconds obtained from PySpark DQ²S Standalone mode processing 70 000 000 rows with 1 to 4 worker nodes on a cluster.

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70 000 000 rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101.095668992</td>
<td>68.086117888</td>
<td>54.591626240</td>
<td>48.209958144</td>
<td></td>
</tr>
<tr>
<td>134.146179072</td>
<td>66.611641856</td>
<td>57.330426880</td>
<td>49.384262912</td>
<td></td>
</tr>
<tr>
<td>104.077149184</td>
<td>67.483824896</td>
<td>58.199030272</td>
<td>45.52409024</td>
<td></td>
</tr>
<tr>
<td>105.101277184</td>
<td>70.298240000</td>
<td>58.156339200</td>
<td>44.14183040</td>
<td></td>
</tr>
<tr>
<td>131.581689088</td>
<td>70.488784896</td>
<td>54.432688896</td>
<td>42.770220032</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>115.200392704</td>
<td>68.593721907</td>
<td>56.542022298</td>
<td>46.000066630</td>
</tr>
</tbody>
</table>

Table C.26.B Execution times in seconds obtained from PySpark DQ²S Standalone mode processing 70 000 000 rows with 1 to 4 worker nodes on a cluster.
### PySpark DQ\(^2\)S Standalone mode on a cluster - 105 000 000 rows

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>157113010176</td>
<td>88414319104</td>
<td>78251291136</td>
<td>60394453248</td>
</tr>
<tr>
<td>(nanoseconds)</td>
<td>144317761024</td>
<td>85405874176</td>
<td>73666785972</td>
<td>63932439040</td>
</tr>
<tr>
<td></td>
<td>141386390784</td>
<td>87008677120</td>
<td>73601081088</td>
<td>55310124032</td>
</tr>
<tr>
<td></td>
<td>142545898752</td>
<td>87754412288</td>
<td>73675163136</td>
<td>55662246912</td>
</tr>
<tr>
<td>Average</td>
<td>145854936525</td>
<td>86404121958</td>
<td>75324774246</td>
<td>58609512038</td>
</tr>
</tbody>
</table>

Table C.27.A Execution times in nanoseconds obtained from PySpark DQ\(^2\)S Standalone mode processing 105 000 000 rows with 1 to 4 worker nodes on a cluster.

<table>
<thead>
<tr>
<th>Number of worker nodes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>157.113010176</td>
<td>88.414319104</td>
<td>78.251291136</td>
<td>60.394453248</td>
</tr>
<tr>
<td>(seconds)</td>
<td>144.317761024</td>
<td>85.405874176</td>
<td>73.666785972</td>
<td>63.932439040</td>
</tr>
<tr>
<td></td>
<td>141.386390784</td>
<td>87.008677120</td>
<td>73.601081088</td>
<td>55.310124032</td>
</tr>
<tr>
<td></td>
<td>142.545898752</td>
<td>87.754412288</td>
<td>73.675163136</td>
<td>55.662246912</td>
</tr>
<tr>
<td>Average</td>
<td>145.854936525</td>
<td>86.404121958</td>
<td>75.324774246</td>
<td>58.609512038</td>
</tr>
</tbody>
</table>

Table C.27.B Execution times in seconds obtained from PySpark DQ\(^2\)S Standalone mode processing 105 000 000 rows with 1 to 4 worker nodes on a cluster.

### Optimised Python DQ\(^2\)S maximum capacity experiment - Cluster

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>nanoseconds</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>(35 000 000 rows)</td>
<td>82839184128</td>
<td>82.839184128</td>
</tr>
<tr>
<td></td>
<td>84307152128</td>
<td>84.307152128</td>
</tr>
<tr>
<td></td>
<td>82913083136</td>
<td>82.913083136</td>
</tr>
<tr>
<td></td>
<td>82871251968</td>
<td>82.871251968</td>
</tr>
<tr>
<td></td>
<td>83172557056</td>
<td>83.172557056</td>
</tr>
<tr>
<td>Average</td>
<td>83220645683</td>
<td>83.220645683</td>
</tr>
</tbody>
</table>

Table C.28: Optimised Python DQ\(^2\)S execution times in nanoseconds and seconds for 35 000 000 rows.
APPENDIX C. TESTING RESULTS

C.2 Performance Metrics Results

This section presents different calculations with a nine-decimal precision measured with the average runtime obtained from the tables above. All the labels from the tables are correspondent to the formulas (4.1) to (4.5), explained in Section 4.6.1.

Table C.29 shows the performance metrics calculated using the information from the processing of 11 500 000 rows with the PySpark in Local mode, implemented in a single machine. The serial runtime (TS) is the runtime obtained from the Optimised Python, conforming the best sequential runtime known; Serial runtime with 1 core (T(1)) is the runtime obtained from the Local mode with local[1], the parallel runtime (TP) correspond to the runtimes obtained with the parametrisation with local[n], where n is a number from 2 to 10, 12 and 16. The relative speedup (rtS), and relative efficiency (rtE) are calculated considering the runtimes from T(1), whereas the real speedup (rS) and real efficiency (rE), are calculated with the values from TS, to produce values with respect to the best sequential time known.

Table C.30.A shows the performance metrics calculated using the information from the processing of 50 000 rows. In this case, the serial runtime (TS) is the runtime obtained from the Optimised Python with the same number of rows as input. The Serial runtime with 1 core (T(1)), utilised to calculate the relative speedup (rtS) and relative efficiency (rtS), is the parallel runtime (TP) obtained from 1 worker and 1 Core for 50 000 rows, with default parameters from the PySpark DQ\textsuperscript{2}S instance.

Tables C.30.B and C.30.C show the information in the same way as C.30.A but using its correspondent value for TS, T(1) and TP, based on the average runtimes obtained from 100 000 and 1 000 000 rows.

Tables C.31.A, C.31.B, and C.31.C show the performance metrics calculated for the values obtained from the PySpark in Local mode, executed on a single machine with 50 000, 100 000, and 1 000 000 rows as corresponds.

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.34620042</td>
<td>132.279</td>
<td>132.279</td>
<td>0.161373</td>
<td>1</td>
<td>0.161373</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21.34620042</td>
<td>132.279</td>
<td>75.36365</td>
<td>0.283243</td>
<td>1.755209</td>
<td>0.141621</td>
<td>0.877605</td>
</tr>
<tr>
<td>3</td>
<td>21.34620042</td>
<td>132.279</td>
<td>59.27924</td>
<td>0.360096</td>
<td>2.231455</td>
<td>0.120032</td>
<td>0.743818</td>
</tr>
<tr>
<td>4</td>
<td>21.34620042</td>
<td>132.279</td>
<td>50.40502</td>
<td>0.423494</td>
<td>2.624321</td>
<td>0.105873</td>
<td>0.65608</td>
</tr>
<tr>
<td>5</td>
<td>21.34620042</td>
<td>132.279</td>
<td>47.17377</td>
<td>0.452501</td>
<td>2.804079</td>
<td>0.0905</td>
<td>0.560816</td>
</tr>
<tr>
<td>6</td>
<td>21.34620042</td>
<td>132.279</td>
<td>44.84713</td>
<td>0.475977</td>
<td>2.949552</td>
<td>0.079329</td>
<td>0.491592</td>
</tr>
</tbody>
</table>
### C.2. PERFORMANCE METRICS RESULTS

<table>
<thead>
<tr>
<th>W</th>
<th>C</th>
<th>Mem</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>14G</td>
<td>10.56718445</td>
<td>0.017029662</td>
<td>1.000000000</td>
<td>0.08209626</td>
<td>1.000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4.7G</td>
<td>1.783150234</td>
<td>0.106353286</td>
<td>1.000000000</td>
<td>0.09548234</td>
<td>1.000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.29: Performance Metrics for Parallel Algorithms: Values calculated for 11 500 000 rows processing from PySpark Local mode in single machine execution.

<table>
<thead>
<tr>
<th>W</th>
<th>C</th>
<th>Mem</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>14G</td>
<td>10.56718445</td>
<td>0.106090547</td>
<td>10.567184845</td>
<td>0.010039623</td>
<td>1.000000000</td>
<td>0.010039623</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4.7G</td>
<td>1.783150234</td>
<td>0.106090547</td>
<td>10.567184845</td>
<td>0.010039623</td>
<td>1.000000000</td>
<td>0.010039623</td>
<td></td>
</tr>
</tbody>
</table>

Table C.30.A Performance Metrics for Parallel Algorithms: Values calculated for 50 000 rows processing from PySpark Standalone mode in single machine execution.

<table>
<thead>
<tr>
<th>W</th>
<th>C</th>
<th>Mem</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>14G</td>
<td>10.56718445</td>
<td>0.106353286</td>
<td>1.000000000</td>
<td>0.09548234</td>
<td>1.000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4.7G</td>
<td>1.783150234</td>
<td>0.106353286</td>
<td>1.000000000</td>
<td>0.09548234</td>
<td>1.000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.30.B Performance Metrics for Parallel Algorithms: Values calculated for 100 000 rows processing from PySpark Standalone mode in single machine execution.

<table>
<thead>
<tr>
<th>W</th>
<th>C</th>
<th>Mem</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>14G</td>
<td>10.56718445</td>
<td>0.106090547</td>
<td>10.567184845</td>
<td>0.010039623</td>
<td>1.000000000</td>
<td>0.010039623</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4.7G</td>
<td>1.783150234</td>
<td>0.106090547</td>
<td>10.567184845</td>
<td>0.010039623</td>
<td>1.000000000</td>
<td>0.010039623</td>
<td></td>
</tr>
</tbody>
</table>

Table C.30.C Performance Metrics for Parallel Algorithms: Values calculated for 1 000 000 rows processing from PySpark Standalone mode in single machine execution.
### PySpark DQ^2S Local mode in single machine (50 000 rows)

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.106090547</td>
<td>6.918572749</td>
<td>6.918572749</td>
<td>0.015334167</td>
<td>1.000000000</td>
<td>0.015334167</td>
<td>1.000000000</td>
</tr>
<tr>
<td>2</td>
<td>0.106090547</td>
<td>5.477000704</td>
<td>5.477000704</td>
<td>0.019370190</td>
<td>1.263204648</td>
<td>0.009685095</td>
<td>0.631602324</td>
</tr>
<tr>
<td>3</td>
<td>0.106090547</td>
<td>5.041016576</td>
<td>5.041016576</td>
<td>0.021045467</td>
<td>1.372455862</td>
<td>0.007015156</td>
<td>0.457485287</td>
</tr>
<tr>
<td>4</td>
<td>0.106090547</td>
<td>4.948789606</td>
<td>4.948789606</td>
<td>0.021437676</td>
<td>1.39803317</td>
<td>0.005359419</td>
<td>0.349508329</td>
</tr>
</tbody>
</table>

Table C.31.A Performance Metrics for Parallel Algorithms: Values calculated for 50 000 rows processing from PySpark Local mode in single machine execution.

### PySpark DQ^2S Local mode in single machine (100 000 rows)

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.190653286</td>
<td>7.590131917</td>
<td>7.590131917</td>
<td>0.025118573</td>
<td>1.000000000</td>
<td>0.025118573</td>
<td>1.000000000</td>
</tr>
<tr>
<td>2</td>
<td>0.190653286</td>
<td>5.845138381</td>
<td>5.845138381</td>
<td>0.032617412</td>
<td>1.298557592</td>
<td>0.016308706</td>
<td>0.649268796</td>
</tr>
<tr>
<td>3</td>
<td>0.190653286</td>
<td>5.327845581</td>
<td>5.327845581</td>
<td>0.035784312</td>
<td>1.424615598</td>
<td>0.011928104</td>
<td>0.474871866</td>
</tr>
<tr>
<td>4</td>
<td>0.190653286</td>
<td>5.168955802</td>
<td>5.168955802</td>
<td>0.036884294</td>
<td>1.468407200</td>
<td>0.009221074</td>
<td>0.367101800</td>
</tr>
</tbody>
</table>

Table C.31.B Performance Metrics for Parallel Algorithms: Values calculated for 100 000 rows processing from PySpark Local mode in single machine execution.

### PySpark DQ^2S Local mode in single machine (1 000 000 rows)

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>TS</th>
<th>T(1)</th>
<th>TP</th>
<th>rS</th>
<th>rtS</th>
<th>rE</th>
<th>rtE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.783150234</td>
<td>17.694503680</td>
<td>17.694503680</td>
<td>0.100774244</td>
<td>1.000000000</td>
<td>0.100774244</td>
<td>1.000000000</td>
</tr>
<tr>
<td>2</td>
<td>1.783150234</td>
<td>11.717394074</td>
<td>11.717394074</td>
<td>0.152179761</td>
<td>1.510105709</td>
<td>0.076089881</td>
<td>0.755052854</td>
</tr>
<tr>
<td>3</td>
<td>1.783150234</td>
<td>10.272742298</td>
<td>10.272742298</td>
<td>0.173580742</td>
<td>1.722471290</td>
<td>0.057860247</td>
<td>0.574157097</td>
</tr>
<tr>
<td>4</td>
<td>1.783150234</td>
<td>8.874831514</td>
<td>8.874831514</td>
<td>0.200922151</td>
<td>1.993784744</td>
<td>0.050230538</td>
<td>0.498446186</td>
</tr>
</tbody>
</table>

Table C.31.C Performance Metrics for Parallel Algorithms: Values calculated for 1 000 000 rows processing from PySpark Local mode in single machine execution.
Appendix D

Apache Spark User Level Proposal

The work done to carry out this study faced the experience to realise there could be classifications of users when dealing with big data frameworks, from naive to expert with regards to expertise level, accompanied by a classification of novice to experienced users based on exposition to the framework, this classification was set according to general division studied for computer users\cite{37, 41, 147}, which could be extended for its usage within the case this study presents.

![Proposed big data frameworks user classification with regards to experience and expertise.](image)

Figure D.1: Proposed big data frameworks user classification with regards to experience and expertise.
The user’s level of exposure can be classified on three: novice, intermediate and advanced, assuming a quality exposure where the user was involved in gaining skills and knowledge. Quality on the exposure is measured on a range of naive, competent and expert. With the designation mentioned, a novice user might have from 0 to 12 months of exposure to the framework, with naive expertise; whereas an intermediate user might have from 6 to 24 months of exposure and a competent expertise; advanced users should have more than 24 months of exposure with a competent expertise, or around 18 months with expert skills. The temporal classification proposed is represented in Figure [D.1] There is a larger gap between being competent and expert, and intermediate to advanced, than the gap from making a step out of being a naive, novice user, this is because once the basic elements are understood, the common tasks are successfully implemented, and the general knowledge is acquired, there is a wide path to specialisation with big data frameworks; in Apache Spark, the users could be classified according to the following:

1. **Novice and naive**
   - Basic knowledge of big data.
   - Entry involvement on big data frameworks.
   - Basic understanding of MapReduce paradigm.
   - Capable of processing Apache Spark applications without any understanding of Spark’s processing model.
   - Development of Spark applications without any built-in library.
   - Unaware of the available parametrisation that can be done to Spark’s jobs.
   - With little to zero involvement on community discussion forums.
   - Able to provide answers to questions regarding Spark’s general characteristics.
   - Unable to provide answers to general technical and low level questions regarding Spark’s details.
   - Unable to provide thorough answers to any question regarding Spark.

2. **Intermediate and competent**
   - Knowledge about big data and one specialisation (quality, security, management, visualisation, etc.).
• Familiar with the big data framework ecosystem.
• Competent understanding of MapReduce.
• Use of Apache Spark with knowledge on its internal processes, both on software and hardware level.
• Clear knowledge of the elements involved in the processing of Apache Spark.
• Capable of utilise at least one Spark built-in library, and general knowledge about the main libraries.
• Able to understand on high level Apache’s different modes and available cluster managers.
• Able to configure and utilise at least one cluster manager.
• Aware of the parametrisation that can be done to Spark’s jobs.
• Able to set common parameters without in-depth understanding of its function.
• Able to perform basic tuning to Spark’s jobs.
• Usually actively involved on community discussion forums.
• Able to provide answers to questions regarding Spark’s general characteristics with detail.
• Capable of providing answers to general technical questions regarding Spark.
• Able to provide thorough answers to some questions regarding Spark.

3. **Advanced and expert**

• Experience and knowledge on more than one big data framework.
• Strong knowledge on big data, and knowledge on more than one specialisation or strong knowledge on one.
• Solid understanding of MapReduce paradigm.
• Competent knowledge on Spark’s available libraries, and strong skills to utilise at least one.
• Able to configure and utilise more than one cluster manager.
• Able to integrate and utilise other open source projects with Spark.
• Able to perform tuning to Spark’s jobs for different kinds of algorithms and use cases.
• Actively involved on community discussion forums.
• Able to provide thorough answers to questions regarding Spark’s general characteristics with detail.
• Able to provide thorough answers to a wide number of questions regarding Spark, from technical aspects to high level details.

Some users might already have expertise and involvement on big data frameworks, and an advantage on MapReduce and big data understanding if Hadoop was previously known. The list above is not extensive and does not pretend to over generalise, considering that users could present different behaviours, backgrounds and levels of exposure; each case could be hugely different but, this is a proposed high level classification, comprising the basic elements each novice, intermediate and advanced big data framework user should consider.