Experimenting with Big Data Computing for Scaling Data Quality-Aware Query Processing

DOI:
https://doi.org/10.1016/j.eswa.2021.114858

Document Version
Accepted author manuscript

Experimenting with Big Data Computing for Scaling Data Quality-Aware Query Processing

Sonia Cisneros-Cabrera[a], Anna-Valentini Michailidou[b], Sandra Sampaio[a,*], Pedro Sampaio[c], Anastasios Gounaris[b]

[a]*Department of Computer Science, The University of Manchester, UK*
[b]*Department of Informatics, Aristotle University of Thessaloniki, Greece*
[c]*Alliance Manchester Business School, The University of Manchester, UK*

Abstract

Combining query processing techniques with data quality management approaches enables enforcement of quality constraints, such as timeliness, accuracy and completeness, as part of *ad-hoc* query specification and execution, improving the quality of query results. Despite the emergence of novel data quality processing tools, there is a dearth of studies assessing performance and scalability in the execution of data quality assessment tasks during query processing. This paper reports on an empirical study aiming to investigate the extent to which a big data computing framework (Spark) can offer significant gains in performance and scalability when executing data quality querying tasks over a range of computational platforms including a single commodity multi-core machine and a cluster-based platform for a wide range of workloads. Our results show that substantial performance and scalability gains can be obtained by using optimized data science libraries combined with the parallel and distributed capabilities of big data computing. We also provide guidelines on choosing the appropriate computational infrastructure for executing DQ-aware queries.

Keywords: Data Quality-Aware Queries, Big Data Computing, Empirical Evaluation

*Corresponding author
Email addresses: sonia.cisneroscabrera@manchester.ac.uk (Sonia Cisneros-Cabrera), annavalen@csd.auth.gr (Anna-Valentini Michailidou), s.sampaio@manchester.ac.uk (Sandra Sampaio), p.sampaio@manchester.ac.uk (Pedro Sampaio), gounaria@csd.auth.gr (Anastasios Gounaris)

1. Introduction

Big data computing frameworks, discussed in (Kune et al., 2016), such as Hadoop and Apache Spark are gradually becoming the mainstream choice for parallel and distributed processing of very large data sets in data intensive applications. Despite several empirical studies available in the literature reporting on the performance of big data computing frameworks in a variety of application domains, such as (Singh & Bawa, 2017; Harnie et al., 2017; Li et al., 2016; Tous et al., 2015, October; Veiga et al., 2016, December; Cheng et al., 2016, March; Zhang & Sakr, 2013, May; Shahrivari & Jalili, 2016), there is only a small set of studies assessing the performance implications of big data computing frameworks in supporting Data Quality (DQ) management tasks, for example, (Schelter et al., 2018; He et al., 2016, July; Taleb et al., 2015, June; Khayyat et al., 2015, June).

As key activities in DQ management, querying and data profiling (DP) allow the collection of statistics and value assessments to build a profile for a data set and use quality-based constraints as part of the querying process. Profiles typically consider data properties relating to the content, structure and quality of data, obtained via data exploration and used to unveil analysis-relevant information about the data, such as value frequencies, formats, patterns, defects, outliers, etc. The information may involve multiple attributes and/or entities of the target data set, relevant to a multitude of DQ dimensions, as reported in (Pipino et al., 2002), such as timeliness, completeness, and accuracy. Similar to any activity involving automation of data analysis, DQ-aware query processing is a computationally demanding process, with tasks involving application of aggregation operations, checking of dependencies between attributes, combination of data using join and union operations, application of arithmetic operations, evaluation of boolean expressions, etc. Moreover, such tasks can be executed over complex structured and semi-structured data, frequently found in big data applications. The integration of big data computing with DQ management ap-

proaches should be an important part of the data quality technology stack, as suggested in (Loshin, 2014), since the efficient computation and scalability of DQ processing tasks is paramount to performing data science at scale.

Recent research in DQ management for big data has focused on the following topics: (1) *automation* of DQ assessment and processing, aiming at minimizing the need for manual tasks (Schelter et al., 2018; Khayyat et al., 2015, June; Heidari et al., 2019, June); (2) assessment of the *effectiveness* of automated DQ processing (Zhang et al., 2019, April; Milani et al., 2019, April), where machine learning approaches have had a significant impact; and (3) *scaling* of existing DQ processing tools and systems for coping with a variety of big data loads (Schelter et al., 2018, 2019, April). Topics (1) and (2) concentrate the bulk of DQ for big data research to date, with topic (3) recently gaining prominence, due to the need to ensure scalability of applications.

Popular and/or commercial DQ management tools such as Metanome for data profiling, described in (Papenbrock et al., 2015), and Trifacta for data wrangling, described in (Trifacta, 2017), have been developed with native support for execution on big data computing frameworks; however, to the best of our knowledge, comprehensive empirical performance studies in relation to these tools have yet to be published. Recently, both Apache and Databricks have provided their own solutions for DQ management for big data, namely *Griffin*[1] and *drunken*[2], respectively, but these solutions have not been empirically assessed from a performance and scalability point of view either. Ultimately, the success of these tools will depend on scalability and performance characteristics.

In this paper, we investigate the scalability and performance issues associated with a set of DQ-aware query processing tasks over large, sensor-collected traffic data sets. Time series data sets have been selected for the empirical study due to their prevalence in data science and the associated DQ management challenges.

Given the absence of DQ management application benchmarks focused on

[1] https://griffin.apache.org/

[2] https://github.com/FRosner/drunken-data-quality

specific DQ dimensions, we have chosen DQ assessment and profiling algorithms addressing the most commonly used DQ dimensions in practice, namely timeliness, completeness and accuracy, evaluating boolean expressions involving multiple attributes, arithmetic and aggregate operations, and combining large data sets using join operations. The algorithms are executed over one of the most widely used big data computing platforms, Apache Spark, using one of the most popular libraries for fast data manipulation in data science (i.e., Python Pandas), varying data types, data set sizes and computing infrastructures (e.g., commodity PCs and parallel computing clusters). The algorithms are implemented in DQ^{2}S - the Data Quality Query System - which supports a SQL-based query language and an extensible set of algebraic operators associated with DQ dimensions, allowing data to be queried based on user-imposed quality thresholds, described in (Sampaio et al., 2015).

As part of our empirical investigation, we perform a number of experiments based on the DQ^{2}S code base, addressing the following issues:

- Assessing the performance and scalability of a set of DQ management queries, ranging from simple queries involving one dimension of quality and one input data collection, to complex queries involving the collection of information regarding two DQ dimensions.

- Exploring two performance improvement technologies: Pandas, a programming language library for fast tabular data manipulation in memory, and Apache Spark, a big data computing platform for parallel execution.

- Experimenting with complex, real-world, time series traffic data sets collected from road-side sensors, with different types of data, e.g., numeric, categorical and miscellaneous, ranging from small to large, input to a variety of DQ-aware queries deployed on multiple infrastructure settings, including a commodity PC and a cluster of PCs.

In addition to performance and scalability insights, the results discussed in this paper also show a breaking point at which commodity multi-core PCs

running DQ^{2}S without the support of big data computing frameworks are no longer able to handle very large volumes of data. This result is relevant towards identifying data set sizes where it is necessary to apply the capabilities of big data computing frameworks. In our experience in wrangling traffic data, reported in (Sampaio et al., 2019), conventional data management tools running on commodity PCs have significant scalability limitations when handling very large data sets (e.g., the file size limit of 250MB in Web Office 365 Excel, or data sets of more than 4 gigabytes in OpenRefine, described in (OpenRefine-Organization, 2019)). Moreover, we show up to which point using Spark on a single multi-core PC is more beneficial than running it on a cluster.

This study can also be used as a roadmap for those interested in re-purposing and porting existing DQ management systems and tools to possibly increase scalability and performance by exploring the features provided by the big data frameworks (Apache Spark and Pandas). The research also contributes to the literature by providing insights into the requirements (i.e., data volume, architecture, level and type of parallelism) over which a big data computing framework starts and ceases to provide benefits, and the challenges associated with the porting of non-parallel "small data" applications to scale and fully explore the capabilities of big data frameworks deployed on a parallel computing infrastructure. Additionally, the code base used for the experiments is publicly available in a repository to enable third parties to extend and/or re-run our experiments.[3].

The remainder of this paper is organized as follows: Section 2 provides a literature review within the scope of big data quality processing. Section 3 describes background on the DQ^{2}S system and the three main variations or instances of DQ^{2}S compared in the experiments. Section 4 presents the research questions motivating the work, as well as the settings for all the experiments and the evaluation criteria used in the discussion and analysis of results. Section 5 presents the experiments and corresponding results. Section 6 discusses key

[3]https://github.com/annavalentina/Data-Quality-Query-System

insights arising from this research. Section 7 concludes the article and outlines future work.

2. Related Work

There are three major topics of related work covered in this literature review: (1) research addressing *automation (AUT)* of DQ assessment and management tasks, involving the development of languages and functionality for end-users to express DQ requirements and constraints, minimizing the need for manual tasks; (2) research on assessing the *effectiveness (EFF)* of automated DQ processing, for example, by injecting errors into datasets and evaluating error fixing functionality; (3) research on *scalability (SCA)* of existing DQ processing tools and systems for coping with big data loads.

Table 1 provides a summary of the papers included in the literature review.

(Schelter et al., 2018) focus on automation of data quality verification. Declarative definition of data quality constraints is supported via a number of provided logic syntax language constructs, and these can be combined with user-defined code via calls to external functions. The user-defined constraints in (Schelter et al., 2018) are translated into aggregation queries, which run over Spark.

Topics	**Approach**	**Evaluation**	**Source**
AUT, EFF, SCA	DQ constraints in logic-based notation compiled into Spark code.	Empirical evaluation on Spark using real-world data.	(Schelter et al., 2018, 2019, April)
AUT, EFF	Unsupervised ML for error detection. DQ extension layered on tools such as Excel, without big data capabilities.	Empirical evaluation using real-world data.	(Wang & He, 2019, June)

AUT, EFF	Accuracy constraints expressed as data exploration queries complying with privacy requirements. APEx Data exploration tool supporting the accuracy dimension via translation of queries and accuracy bounds into differential private preserving algorithms.	Empirical evaluation using real-world data.	(Ge et al., 2019, June)
AUT	Application of ML for error detection.	Empirical evaluation using real-world data; focus on effectiveness of error detection (recall and precision).	(Heidari et al., 2019, June)
EFF	Application of ML and conditional functional dependencies for detecting missing values	Empirical evaluation using real-world data; focus on the effectiveness of the completeness approach.	(Zhang et al., 2019, April)
EFF	Application of ML for detection of timeliness DQ issues.	Empirical evaluation using real-world data; focus on the effectiveness of the timeliness approach.	(Milani et al., 2019, April)
AUT, EFF	Application of ML for selection and configuration of error detection algorithms.	Empirical evaluation with a focus on coverage and effectiveness of error detection.	(Mahdavi et al., 2019, June)
AUT, SCA	SQL-based DQ extension for error repair layered on top of OpenRefine.	Scalability experiments on a MacBook Pro machine scaling to 5 million rows, without performing tests on a big data computing framework.	(He et al., 2016, July)
AUT	Rule-based data profiling framework focusing on data cleaning layered on top of Hadoop.	Not available.	(Taleb et al., 2015, June)
AUT, SCA	Execution of SPARQL queries encoding DQ validation tasks.	Empirical evaluation on a 8-node cluster using Hadoop and SPARQL queries.	(Bonner et al., 2015, October)

AUT, SCA	Encoding of DQ constraints both in declarative and procedural ways; complementary DQ component to general data processing platforms, such as PostgreSQL.	Empirical evaluation using both synthetic and real-world data.	(Khayyat et al., 2015, June)

Table 1: Summary of Related Work.

(He et al., 2016, July) propose an approach to data cleaning that uses SQL as declarative language to repair errors. The data cleaning system interacts with users to gradually validate a possible set of minimal SQL update queries to repair data. The system relies on user data exploration for error identification and manual repair. Once a user repair is made, the system is able to "guess" a set of possible queries to fix the identified error. Optimization techniques to prune the search space composed of possible update queries are offered, as well as the use of multi-hop search algorithms. The system was designed as an extension to the OpenRefine data wrangling tool, described in (OpenRefine-Organization, 2019), and uses PostgreSQL as underlying DBMS. The experiments assess scalability on a MacBook Pro machine scaling to 5 million rows, without performing tests on a big data computing framework.

(Taleb et al., 2015, June) propose a framework for big data quality management that suggests techniques for overcoming or minimising data quality issues arising from different tasks belonging to the data preparation step in the big data management lifecycle, with a focus on data cleaning. The proposed data cleaning module uses a profile of the target data set as guide to the cleaning process. This profile encompasses a list of rules to be applied over the data to address specific data quality issues relating to a particular quality request. The selection of rules, which is the core activity of the profile generation process, is performed based on samples extracted from the target data set, but details about how this activity is carried out are not included. Moreover, the authors claim that the module has been tested using the Hadoop implementation of MapReduce. However, experimental details about the accuracy of their results and overall system performance are not provided.

(Bonner et al., 2015, October) focus on the identification of errors in patient monitoring data, by executing data validation queries over RDF medical data sets. To achieve scalability when dealing with large data sets, the authors proposed a method for storing and querying RDF data sets using the Hadoop implementation of MapReduce, by which parallelism is exploited in the execution of SPARQL queries over the data, allowing better scalability than previous approaches using the MapReduce programming model. Performance improvements are achieved through mainstream optimisation techniques applied to RDF and SPARQL, such as SPARQL joining strategies, data caching and the amalgamating of queries that share common join elements to avoid re-computation of joins. Empirical results using an eight node cluster suggest that accuracy validation of RDF data requires domain knowledge that is likely to be encoded in both queries and RDF data.

(Khayyat et al., 2015, June) propose the BigDansing data cleaning system to enable error detection and fixing in large datasets. The system supports encoding of data quality constraints both in declarative and procedural ways, and was designed as a complementary layer to general data processing platforms. Cleaning involves (1) specifying data quality rules; (2) detecting violations based on specified rules; and (3) repairing the dataset until there are no violations to the specified rules. Users are expected to express logic syntax rules to define DQ constraints. The experimental work was conducted using the TPC-H benchmark data, both in single-node and multi-node settings, using Shark SQL and the Spark big data computing framework, scaling to 10 million rows. The experiments show significant performance improvements in the execution of repair algorithms compared to baseline systems such as PostgreSQL.

(Wang & He, 2019, June) propose UniDetect, a unified framework to automatically detect diverse types of errors in tabular data, which can be integrated with other tools, such as Excel. To create the error detection model, unsupervised machine learning and multiple data sets are used. The focus of their evaluation is on the achieved level of automation and precision of error detection when applied to common types of errors.

(Mahdavi et al., 2019, June) address the problem of selecting appropriate error detection algorithms along with suitable configurations for each algorithm, taking into consideration the data set at hand. As a solution, they propose Raha, a new configuration-free error detection system, which uses a semi-supervised machine learning approach that does not require users to provide configurations for error detection algorithms, and limits the set of all possible algorithms/configurations with heuristics. The focus of their evaluation is on the coverage of the different types of error detection algorithms and the effectiveness of error detection.

(Heidari et al., 2019, June) address the problem of improving machine learning models for automatic error detection in data sets, by showing that data augmentation, considered as a form of weak supervision, can be used to improve model training. Data properties that impact on the quality of the model, such as heterogeneity in types of error, imbalance in the number of errors, which account for under-representation of certain types of errors, are addressed in their solution. The focus of their evaluation is on the effectiveness of error detection, i.e., precision and recall.

(Ge et al., 2019, June) address the problem of accuracy in query answers when the exploration of sensitive data is performed by (externally hired) data analysts. As a solution, they propose APEx, a system that allows data analysts to submit sequences of queries along with accuracy bounds for data exploration. By translating queries and accuracy bounds into differentially private algorithms, APEx returns query answers to the data analyst that meet the accuracy bounds. The focus of their evaluation is on the level of privacy loss in data exploration for entity resolution with high accuracy, and under reasonable privacy settings.

(Zhang et al., 2019, April) address the problem of completeness of numerical values in sensor-collected and heterogeneous data sets, by using conditional dependencies that hold over certain tuples rather than the whole relation. These dependencies are used for learning an individual regression model for each complete tuple together with its neighbor, which helps determining a missing value

based on regression results by the individual models. The focus of their evaluation is on the accuracy of their approach compared to other existing data imputation approaches.

(Milani et al., 2019, April) address the problem of data decay (timeliness) by capturing space and time update patterns in a database, from query logs, and by inferring values that are possibly out-of-date using a spatio-temporal probabilistic model. The focus of their evaluation is on the effectiveness of their technique to identify out-of-date values over real data, considering the accuracy of the repaired values compared to other existing techniques.

(Schelter et al., 2019, April) address the problem of automatic DQ verification by proposing an open source library, based on Apache Spark, that allows users to explicitly specify DQ constraints to be verified in a declarative way, and that provides mechanisms for the automatic validation of these constraints on partitions of data. Optimization strategies to avoid re-reading previously processed partitions following individual partition updates, and to verify selected combinations of partitions, are also proposed and compared against non-optimized DQ verification jobs. Their evaluation is performed on Spark, with a focus on the efficiency of their optimization strategies with regard to runtime and reduction on the number of Spark jobs.

(Schelter et al., 2018) is the closest work to the one reported in this paper, enabling declarative definition of DQ constraints. While in (Schelter et al., 2018) this is done via provided logic syntax constructs, in DQ^2S this is done via extensions to SQL with a special DQ syntax clause. In (Schelter et al., 2018), external functions are combined with the provided constructs through calls, whilst, in DQ^2S, an extensible set of operations are available to all users of the system, allowing operators to be chained to compose a solution, in a manner resembling the combination of operators included in a relational algebra query plan. The design of DQ^2S allows the application of heuristics for automatic performance and quality-based optimizations. The user-defined constraints in (Schelter et al., 2018) are ultimately translated into aggregation queries, which run over Spark. In our work, we focus on the assessment of performance and

scalability of different implementations of DQ^{2}S, with and without the use of Spark or Pandas to manipulate data in memory, store data in disk, and provide opportunities for parallelization. We also experiment with different amounts of memory, infrastructure settings that impact on the level of parallelism, and the level of complexity of data quality/profiling tasks and their impact on scalability and performance of the system. The tested algorithms are described in detail and implement objective dimensions of data quality, beyond accuracy. In contrast to (Schelter et al., 2018), we also include an empirical comparison between DQ^{2}S and an emerging big data DQ system, namely *drunken*.

3. The Data Quality Query System (DQ^{2}S)

DQ^{2}S, described in (Sampaio et al., 2015), is a query processing and data quality constraint enforcement system designed to facilitate the profiling and data quality assessment of large data sets, which combines traditional data management and data profiling techniques for data cleansing. DQ^{2}S includes a data model that seamlessly allows users to associate quality related properties with data stored in a database or file system, by providing facilities for modeling and storing those properties. As a result, the model enables the querying of not only data, but also quality-related information associated with the data, as well as the generation of data profiles via a declarative, SQL-like, query language and an interface that isolates the user from the complexities of the profiling algorithms and the data model constructs. DQ^{2}S also allows users to request data quality to be measured according to an extensible set of algorithms, applied over the available quality-related information, and the ability to construct and apply filters when querying the data based on both data quality measures and quality-related information. Since the profiling algorithms are implemented as algebraic operators, optimisation can be achieved in DQ^{2}S as part of query processing.

The expression below shows a query expressed in the query language defined within the DQ^{2}S system (called DQ^{2}L) that is equivalent to the following text

fragment representing a DQ profiling request: *"Select all purchase orders that have 'pending' as status and have been waiting to be validated for more than 50% of the maximum waiting time"*. Note that DQ^{2}L extends SQL with the *WITH QUALITY AS* syntax clause, allowing users to specify data quality related requests. Also note the use of data quality related functions in both the *SELECT* and *WITH QUALITY AS* clauses, such as *TIMELINESS*, which measure the quality of a table, tuple or attribute according to underlying implementations of data quality dimensions.

```
SELECT order_No, TIMELINESS(status)
FROM Order
WHERE status = 'Pending'
WITH QUALITY AS TIMELINESS(status) <= 0.5
```

Figure A.5 in Appendix A illustrates a query plan for the DQ^{2}L query shown above, based on a combination of both relational algebra operators, such as *join*, *select* and *project*, and data quality related operators, such as *timeliness*. As the *timeliness* operator adds a data quality score as attribute to individual tuples, a *select* operator to filter tuples based on this data quality score follows operator *timeliness*. Appendix A presents a description of the operators and other language constructs that comprise the DQ^{2}S Timeliness query. Quality-aware optimisation is one of the types of optimisation that the framework is capable of. For example, if multiple implementations of the *timeliness* operator are available, the optimiser is able to select the most appropriate operator for the task at hand, based on stored information about the operator and its uses and/or collected statistics about operator choices by different groups of users.

Whilst the semantics of the relational operators is similar to the semantics of relational algebra, the computation associated with this particular implementation of the *timeliness* operator is based on the data quality properties of currency and volatility, as defined in (Ballou et al., 1998) and described in Equations 1, 2 and 3. In Equation 1, v represents a unit of data, and s represents a control variable associated with the sensitivity of the currency-volatility ratio, which should be calibrated according to the level of impact that volatility has over the

result. Calculations of currency and volatility are expressed in Equations 2 and 3, respectively. Currency represents the current age of the data based on the time the data was stored and the age it had at that point in time, and volatility shows a measure of time during which data is not outdated. The timeliness result is given on a 0 - 1 scale, where the scale can be seen as 0 to 100% of timeliness degree. The logical operator query plan for the Timeliness query in Figure A.5 shows the *order_no* and the *timeliness score* of each pending order that has a timeliness score lower than 0.5.

$$Timeliness(v, s) = max[(1 - \frac{Currency(v)}{Volatility(v)}), 0]^{s} \quad (1)$$

$$Currency(v) = DeliveryTime(v) - LastUpdateTime(v) + Age(v) \quad (2)$$

$$Volatility(v) = ExpiryTime(v) - LastUpdateTime(v) + Age(v) \quad (3)$$

Although DQ^{2}S was designed to support data quality management functionality seamlessly integrated with mainstream query processing, the original DQ^{2}S system engine did not explore parallelism opportunities or the capabilities of big data computing frameworks and parallelism. Preliminary experiments using the original engine included elapsed times for complex data profiling queries over small scale data sets using a single *Intel core i5* machine with 8GB of RAM and clock speed of 2.30 GHz, described in (Sampaio et al., 2015). Elapsed times for the data profiling tests using a data set of 100 000 records were above 15 minutes, as reported in (Sampaio et al., 2015), which is extremely slow when performing interactive data analytics operations. As the major performance bottleneck identified in the original DQ^{2}S system related to I/O processes, the adaptation of the original algorithms towards execution over a big data computing platform provided an opportunity to increase performance and scalability of DQ^{2}S.

In this work, three different instances of the DQ^{2}S engine are used to assess and explore the capabilities of big data computing for data quality management. A description of the implementation instances is provided as follows:

(I) Non-parallel or Optimised Python DQ^{2}S: The Optimised Python DQ^{2}S is called "Optimised" since it makes intensive use of the functionality packaged in the Python Pandas library (Augspurger et al., 2015), which supports data structures and methods optimised for data manipulation, specially data in tabular form. This instance is used as a baseline for performance comparisons involving the two parallel instances described below. Other non-parallel DQ^{2}S implementations were tested and compared against the Optimised Python instance (outside the scope of this paper), namely, a Java instance described in (Sampaio et al., 2015), and a Python instance that does not use Pandas. Previous experiments have shown that the Optimised Python instance is more than 3 orders of magnitude faster than the other non-parallel instances, in the following order, from the fastest to the slowest: Optimised Python, Java and Python. Thus, the Optimised Python instance is the non-parallel instance that performs best and was therefore chosen as baseline for comparison against the parallel instances considered in this paper.

(II) Parallel I or PySpark DQ^{2}S: Developed to be used under the Apache Spark big data computing framework, the PySpark DQ^{2}S instance utilises the SQL Apache Spark library and the Python API, which offer capabilities to explore parallelism, either on a single machine by using multi-threading or on a cluster of machines. This instance is used to show performance gains obtained solely from the exploitation of parallelism without the help of special functions for optimising data manipulation.

(III) Parallel II or PySpark+Pandas DQ^{2}S: This instance differs from the previous one in that it employs Pandas along with PySpark to enable not only parallelism exploitation but also data manipulation optimisations. As such, it facilitates the investigation of performance gains obtained from the Pandas-optimised data manipulations in a parallel setting.

4. Research Questions, Experiment Settings and Evaluation Criteria

This empirical study intends to provide answers to the research questions described as follows. These questions are associated with gaps found in past work involving big data frameworks, some of which were described in the related work (Section 2), which mostly do not identify the circumstances (e.g., data volume, machine architecture, level and type of parallelism) at which these frameworks start and cease to provide any benefit and what programming burdens are associated with the re-purposing of non parallel DQ components into systems that can be executed to exploit big data computing frameworks.

1. What performance and scalability differences can be observed when using the Non-parallel, Parallel I and Parallel II instances in *Local mode*[4]? Also, how much data can each instance efficiently handle?

2. What performance and scalability differences can be observed when using a single core of a machine versus when using all of its cores for the Parallel I and Parallel II instances?

3. What performance and scalability differences can be observed when using a single multi-core machine versus cluster-based computing environments for small, medium and large workloads?

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

5. What are the challenges of re-engineering the implementation of a DQ-aware query processing system to work on top of big data computing framework?

In the following sections, the environment, settings and evaluation criteria for the experiments performed to address the research questions are presented. Later, we present results from experiments, in Section 5, and we answer the above questions in Section 6.

[4]Local mode in this context means execution of jobs using a single server. In other words, performance benefits are obtained from parallelisation across all the cores in a server, but not across several servers.

4.1. Datasets, Queries and Experimental Settings

To answer the research questions previously described, the queries described in Table 2 were chosen so that a broad range of DQ profiling tasks of varying levels of complexity, and varying resource requirements is covered. It is worth pointing out that each query uses one or more data profiling operators, designed to provide data quality assessment considering completeness, timeliness and accuracy. And they are named after the quality dimensions they focus on.

Query	**Description**
Completeness (C)	This query computes a completeness score for each record/row regarding the fields/columns specified by the user. To establish whether a value is missing, apart from blanks, the operator checks the content of the corresponding cells against special symbols used to denote missing values (e.g., 0 for some numeric fields). Additionally, the Completeness query encompasses functionality to read files and to structure the final query output by applying a Project operator. An illustration of the execution plan for this query is shown in Figure B.7 in Appendix B.
Timeliness (T)	This query computes a Timeliness score for each input record, based on available data quality information associated with the record. This calculation is performed by means of a Timeliness operator, described in detail in Section 3. The association between data records and their corresponding quality information is materialised by means of join, making the Timeliness query generally more costly than the C query described above. An illustration of the execution plan for this query is shown in Figure B.8 in Appendix B.
Accuracy (A)	This query calculates an Accuracy score for each input record by means of an Accuracy operator. This operator adds a score to each record, indicating its level of accuracy regarding a boolean expression defined by the user. Examples of such expressions include the following: "`shipDate > submitDate`", " `TrafficVolume = VolumeLane1 + VolumeLane2`", etc., where names in typewriter font denote field names in a record. Similarly to the C query, it does not contain any expensive join operations. An illustration of the execution plan for this query is shown in Figure B.6 in Appendix B.

Timeliness-Completeness (T+C)	This is a more complex query that combines two data quality operators, namely Timeliness and Completeness, incurring the application of two potentially expensive join operations involving three input files. It is built using the same operators and inputs as queries T and C. An illustration of the execution plan for this query is shown in Figure B.10 in Appendix B.
Timeliness-Accuracy (T+A)	This is another complex query that, similarly to the T+C query described above, combines two data profiling operators, Accuracy and Timeliness, incurring two join operations and three potentially large input files. An illustration of the execution plan for this query is shown in Figure B.9 in Appendix B.

Table 2: DQ Queries.

To be able to observe limitations or advantages in the level of scalability associated with the target big data framework, four sizes for each of three different datasets were chosen to be used in the experiments described in this paper, namely 100MB, 512MB, 1GB and 10GB, covering a reasonably broad range of sizes, from small to large. The three datasets chosen for the experiments, described in Table 3, provide us with the opportunity to observe changes in performance relating to the data manipulation and resource allocation made by the framework implementation for different types of data. The datasets describe traffic data collected via inductive loop sensors planted on the roads of the city of Manchester in the UK[5]. While one of the datasets contain miscellaneous data types, the second dataset contains numeric fields only, and the third data set contains categorical fields.

Data Set	**Description**

[5]The data files were provided by the Greater Manchester Traffic Authority, TfGM.

Traffic	This is a real-world dataset describing traffic information collected at real-time, including event recording timestamp, vehicle count per lane of a road fragment within a particular time period, average speed of vehicles detected within a time period, individual count of vehicles of a given class detected within a particular time period (e.g., motorcycles), etc. As such, a single record contains a variety of data types, including categorical, numeric, timestamp/date, string, etc. In total, each record contains 17 fields.
Numeric	This is a dataset derived from the Traffic dataset, by removing and mapping non-numeric values into numbers. In total, each record contains 14 fields.
Categorical	This is a dataset derived from the Traffic dataset, by having ranges of field values mapped into defined categories. In total, each record contains 14 fields.

Table 3: Data Sets.

Each of the datasets is composed of three files, one of which is a data file describing traffic-related events recorded at real-time (*File 1*), a second file describes Timeliness related quality information, as described in Section 3 (*File 2*), and the third file contains latitude and longitude information associated with each road fragment in consideration (*File 3*). As indicated in Table 4, each of queries *A*, *C*, *T*, *T+A*, and *T+C* uses a different subset of these three files, and so, different sizes of the files are used in different experiments, making sure that each experiment involving 10GB, for instance, uses in fact only 10GB of data, having roughly even fractions of the 10GB of data coming from each file involved in the query. In Table 4, the sizes and number of rows of the datasets used in each query for the experiments with 100MB of data are presented. Tables C.8, C.9 and C.10 in Appendix C present the same information for the data used in the experiments with 512MB, 1GB and 10GB, respectively. Note that the number of rows used in the 10GB dataset exceed **100M**, as in (Schelter et al., 2018).

Finally, three settings are considered in terms of the computational infrastructure tested, described as follows:

- A *Commodity PC* with the following specifications: 6 core processor, 8GB

Dataset	*Traffic*			*Numeric*			*Categorical*		
/Query	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*
C, A	100.5MB 1268589 rows	-	-	100.5MB 1533332 rows	-	-	100.1MB 1226892 rows	-	-
T	52.2MB 734328 rows	50.6MB 734330 rows	-	50MB 76998 rows	50.6MB 734330 rows	-	50.6MB 622313 rows	50.6MB 734331 rows	-
T+C	31.2MB	41.2MB	32MB	30.5MB	40.6MB	31.3MB	30.5MB	40.6MB	31.3MB
T+A	389999 rows	589999 rows	750000 rows	468999 rows	589999 rows	750000 rows	375000 rows	750001 rows	590000 rows

Table 4: File sizes (in MB) and number of rows for each query used in the experiments with 100MB of data.

of RAM. In terms of Software, this Linux machine uses Python 3.6.7, Spark 2.3.1, Scala 2.11.8, Java 1.8.0_201 and Pandas 0.24.2.

- A commodity PC, part of the cluster described below, *Cluster-1node*, with the following specifications: 6 core processor, 64GB of RAM. This is also a Linux machine, with the following: Python 3.5.3, Spark 2.3.2, Scala 2.11.8, Java 1.8.0_191 and Pandas 0.24.2.

- A shared-nothing cluster with four nodes, *Cluster-4nodes*, with the following specifications: 6 core processor, 64GB of RAM; 8 core processor, 32GB of RAM; 4 core processor, 32GB of RAM; and 8 core processor, 64GB of RAM.

Each node is a Linux machine, with the following: Python 3.5.3, Spark 2.3.2, Scala 2.11.8, Java 1.8.0_191 and Pandas 0.24.2.

4.2. Experiment Evaluation

The performance of five executions of each of the five test queries, varying input data, input data size and machine architecture, is measured and evaluated using well established metrics based on execution runtime, commonly used to measure the performance of parallel algorithms. The metrics and corresponding formulas are described in Table 5. Since it is a parallel architecture, the following metrics for measuring the performance of parallel algorithms, described in (Kwiatkowski, 2001, September; Alecu, 2007; Demmel, 1995; Alexandrov, 2013;

Andrews, 2000; Sahni & Thanvantri, 1996), were used to support the performance analysis described in Section 5. The following information is required for understanding the performance metrics used in this research:

Metric	Description
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. In our work, this corresponds to the runtime of the Non-parallel solution.
One 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. We explore this metric dealing with both Parallel I and Parallel II implementation instances.
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 of three different components: *(i)* the time spent on computation, *(ii)* the cores communication time, and *(iii)* the idle time, which corresponds to the periods of time when a processor is waiting for input/output (I/O) or is in low-power idle mode.
Number of cores (p)	The number of processing units on which the parallel algorithm is executed. We experiment with settings of 1, 6 and 26 cores. These settings cover the maximum capacity of a single node and the overall capacity of the entire cluster.
Speedup (S)	The main performance metric considered in this study. It measures the performance gain of a parallel program when solving the same problem over the corresponding sequential program. In an ideal parallel system, speedup is equal to the number of cores (**p**) used, however, it is usually lower than **p**.

Table 5: Metrics Description.

There is more than one approach towards measuring speedup: Relative, Real, Absolute, Asymptotic and Asymptotic relative, as described in (Sahni & Thanvantri, 1996; Sun & Gustafson, 1991). This study applies 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

the following formula:

$$rS = \frac{TS}{TP}. \tag{4}$$

- **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 the following:

$$rtS = \frac{T(1)}{TP}. \tag{5}$$

4.3. Evaluation Methodology

The evaluation methodology adopted in this paper (illustrated in Figure D.11 of Appendix D) includes experiments involving *all* combinations of the following: the three instances of the DQ^{2}S system described in Section 3, among which *Parallel I* and *Parallel II* are able to explore both intra- and inter-node parallelisms, while *Non-parallel* serves as a baseline; the three computational infrastructure settings described in Section 4.1, all composed of multi-core machines and, among which, *Cluster-1node* and *Cluster-4nodes* represent sets of machines in a cluster; the five DQ management queries described in Section 4.1, distinguishable by DQ dimension, complexity and resource requirements; the three types of datasets described in Section 4.1, namely *Traffic*, *Numeric* and *Categorical*; and the four sizes for each dataset, described in Section 4.1, ranging from 100MB to 10GB. The only exceptions are the combinations involving the *Non-parallel* instance of DQ^{2}S and the *Cluster-4node* infrastructure setting, because this particular instance does not include functionality for parallel execution, and so it is not possible to run it over multiple machines.

Additionally, to observe performance gains obtained from intra-node parallelism, experiments involving all cores, as well as a single core, of each multi-core machine were performed for each infrastructure setting. Results are discussed with a focus on the metrics described in Section 4.2 and presented in Section 5. A total of 45 plots, organised as 3 groups of 15 plots, are shown, being each group associated with a dataset type. In each group of plots, each individual

plot associates a DQ management task (i.e., a query) with an infrastructure setting. In each plot, the horizontal axis is used to show dataset size variations, while bars are used to show the obtained execution times for each instance of DQ^{2}S, in seconds. As a consequence, each plot contains 4 groups of bars, one for each dataset size. It is worth pointing out that experimental results involving the *Parallel II* instance and the *Cluster-4node* infrastructure setting do not appear in the plots, due to the fact that this instance does not scale well with the increase in number of cluster nodes, as discussed in Section 5.

Moreover, in Section 5, additional experiments comparing DQ^{2}S with the *drunken* system (as explained in Section 2) are reported, as well as the main observations obtained from all other experiments. In Section 6, a discussion of results, in the context of the research questions posed in Section 4, is provided.

5. Experiments Description and Evaluation of Results

Although performance can take into account more parameters than just run (or execution) time as, for example, memory usage, energy consumption, and implementation cost, as suggested in (Kwiatkowski, 2001, September), one of the most important motivations for parallelisation is to compute results within the minimum possible time, as described in (Pancake, 1996; Alecu, 2007; Sahni & Thanvantri, 1996; Sun & Gustafson, 1991). Thus, the experiments for this study are mainly focused on execution time and are described in this section.

More specifically, this section presents the performance and scalability results obtained from several sets of experiments. We first assess our solutions in the context explained earlier and then we compare them against the Databricks *drunken* data quality approach. We do not compare our system against Apache Griffin, since it employs the Hive infrastructure, while we use HDFS to store input files.

5.1. Evaluation of solutions

Each experiment was executed five times and the average running time is presented. The results are shown in Figures 1, 2 and 3 for each of the three

datasets, respectively. Each figure contains a 5 X 3 matrix of subfigures, where each row corresponds to a different query (defined in Section 4.1) and each column corresponds to a different computational infrastructure setting. Note that the experiments with the Parallel I and the Parallel II implementation instances encompass cases where only a single node of the cluster (or the commodity PC) is used, and, in this node, a single core is allocated for the execution. These experiments are denoted by symbol *[1]*. There are also experiments where all cores available within a node are allocated, indicated by symbol *[*]*. This notation is used in the first and second columns of the matrices in Figures 1, 2 and 3. Also, note from the figures that two special symbols are used, namely X and ∞. The first one indicates that the experiment could not be performed due to memory issues (i.e., memory exhaustion). The second one represents an experiment that needs more than five hours to finish execution. It is also worth pointing out that experimental results involving the Parallel II implementation performed over the four-node cluster infrastructure were similar to the Parallel II results obtained over the one-node cluster infrastructure, mainly because *the combination of PySpark and Pandas does not scale well on a multi-node infrastructure*, as described in Section 6. For this reason, the Parallel II results over the four-node cluster are not shown on the bar plots in Figures 1, 2 and 3. In the third column of the matrices in the figures, the bar associated with 'Parallel I[*]-1 node' corresponds to the bar associated with 'Parallel I[*]' in the second column; and the bar associated with 'Parallel I[1]-1 node' in the third column corresponds to the bar associated with 'Parallel I[1]' in the second column.

The main observations from the experiments are as follows:

I. The Parallel I (PySpark) solution outperforms the Non-parallel (Pandas) and the Parallel II (PySpark+Pandas) solutions in all cases, except for queries T, A and C on the 100MB datasets. As discussed in more detail in Section 6, we have observed that Pandas, by default demands abundant use of memory for efficient data manipulation, causing our Pandas-based implementations to perform worse than our PySpark implementation (where Pandas is not used),

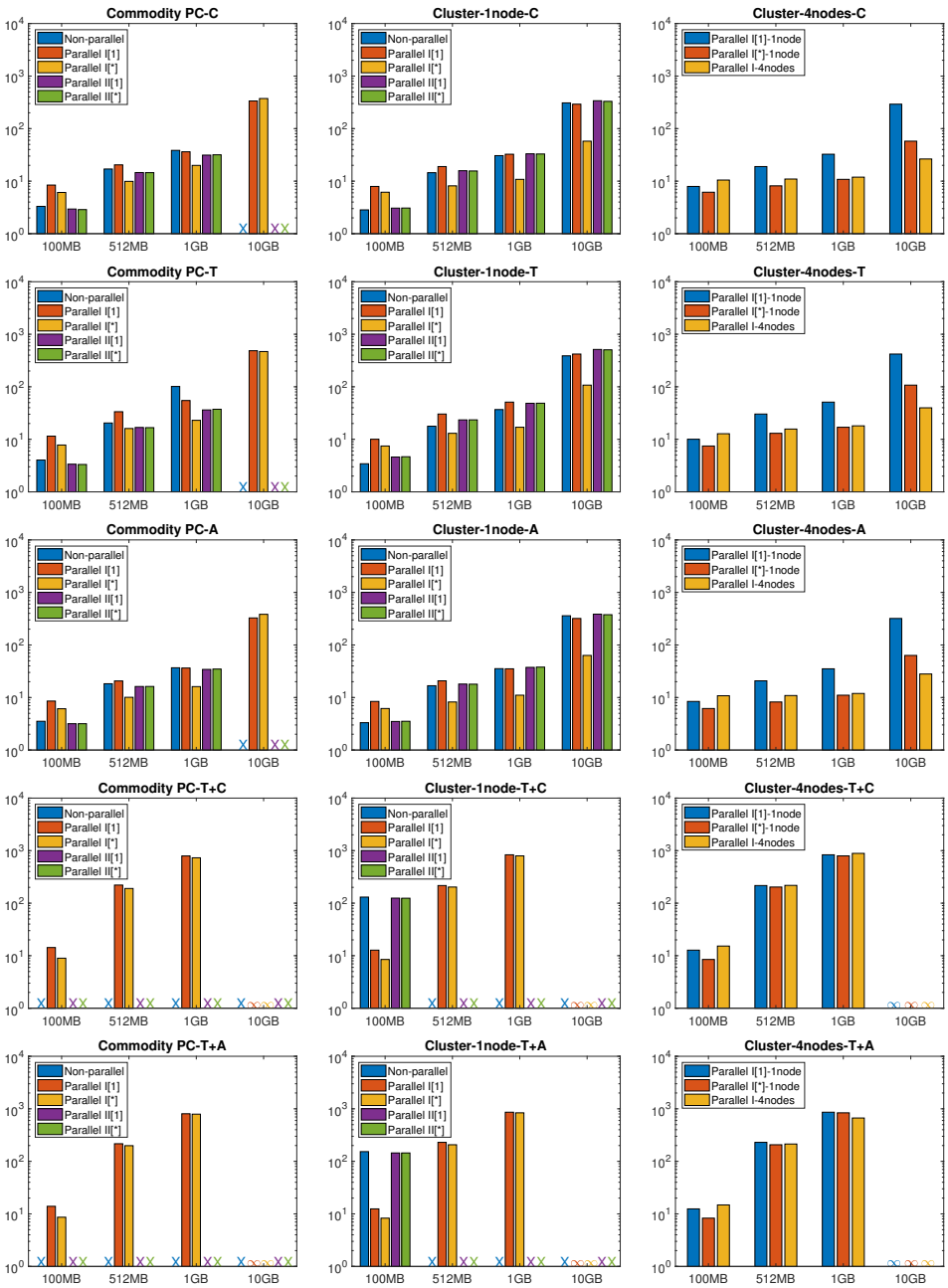

Figure 1: Traffic Dataset Related Execution Times in Seconds.

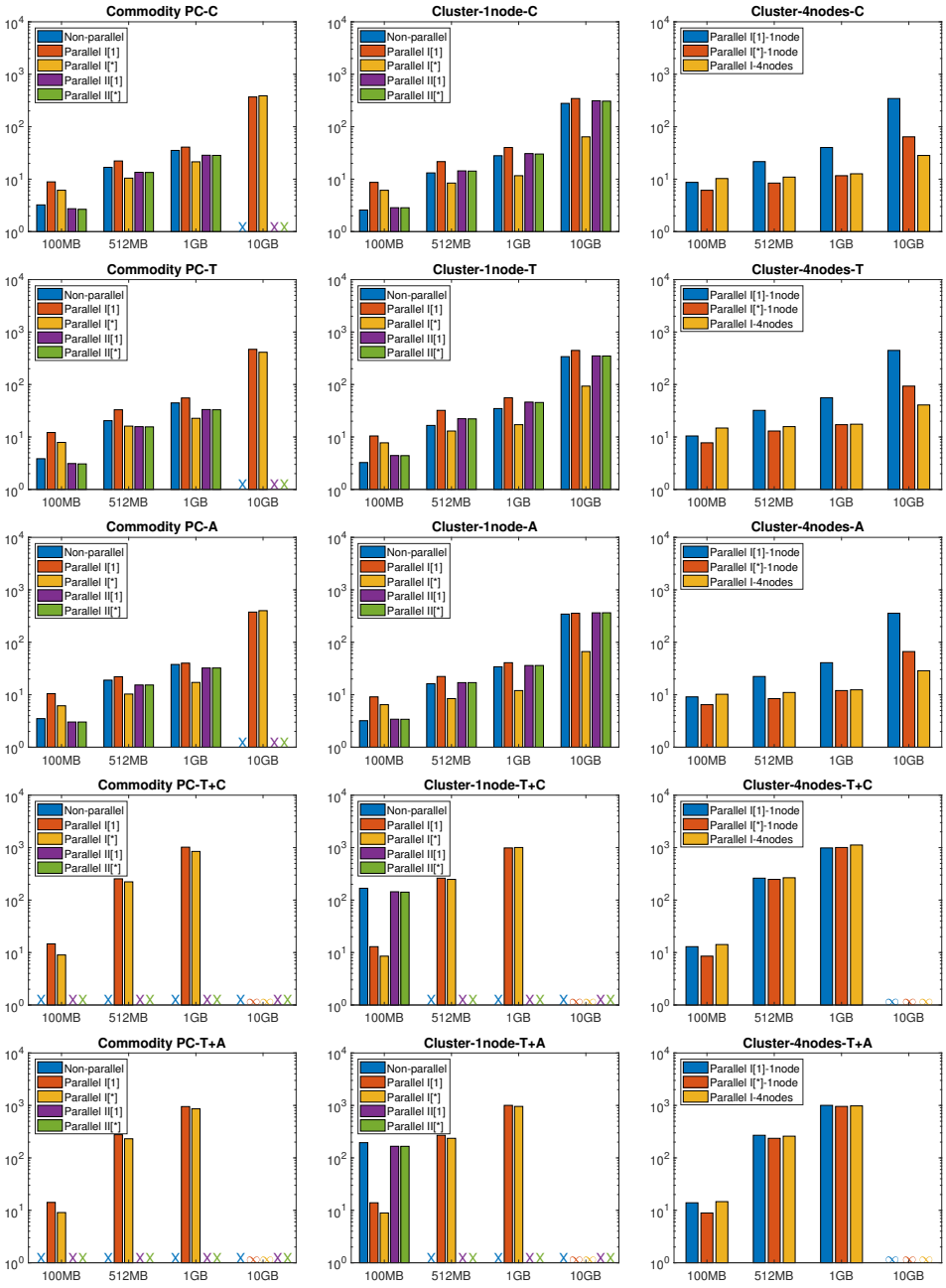

Figure 2: Numeric Dataset Related Execution Times in Seconds.

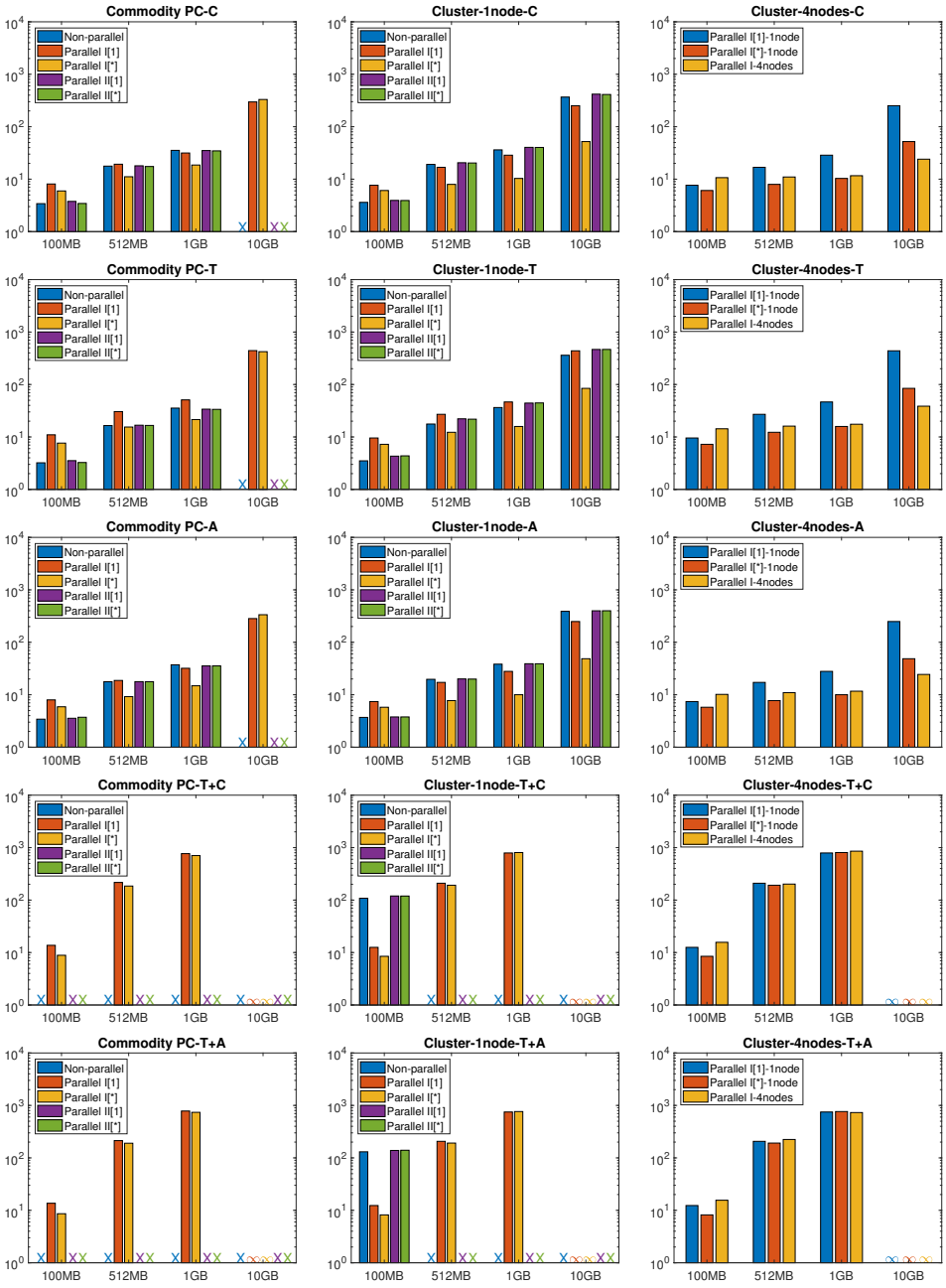

Figure 3: Categorical Dataset Related Execution Times in Seconds.

having as exceptions only the cases where small datasets are input to queries that are not significantly demanding in terms of memory.

II. In general, we observe super linear scalability when increasing the data size for all implementation instances. For example, in the Cluster-1node setting, a 100X-increase in data size (from 100MB to 10GB) leads to increases in running time of no more than 22X, while keeping values for all the other variables the same for the T, A and C queries. The execution overhead associated with these queries is then outweighed by performance benefits as dataset sizes are increased. Nevertheless, the queries involving two joins (T+A, T+C) exhibit quadratic complexity in dataset size, when they manage to complete, indicating that the level of scalability associated with these queries is significantly lower.

III. The Non-parallel instance cannot effectively run the T+A and T+C queries for the datasets larger than 100MBs, largely due to memory exhaustion during the execution of the first of two join operations, causing Pandas to terminate the execution. It is worth pointing out that memory problems are better handled in Spark when Pandas is not used, because it is able to run memory demanding queries for long periods of time without terminating execution, even when memory is scarce.

IV. The benefits of using the full cluster may be exhibited only when dealing with the datasets of 10GB; for the 1GB datasets, employing the full cluster reveals slightly worse performance than when employing a single machine, with exceptions being observed for the T+A query only, which is probably the most demanding query in terms of memory and processing. For smaller datasets, the associated overheads clearly outweigh any parallelism benefits.

V. In summary, when the datasets are no larger than 1GB, the first choice is to employ the Parallel I implementation instance on a single machine; for larger datasets, there is a need to resort to a cluster.

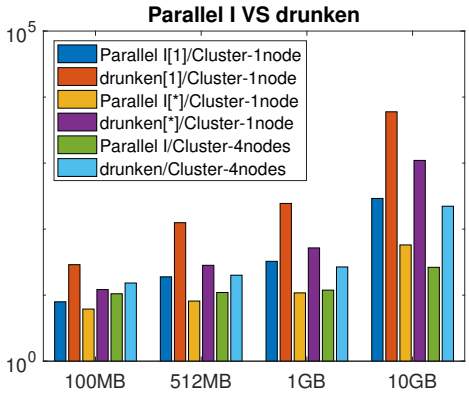

Figure 4: Comparison of Parallel I and drunken using the C query and the traffic dataset.

5.2. Comparison against drunken

In this section, we compare the DQ^{2}S Parallel I instance against the Drunken's data quality solution. The rationale for this comparison is the fact that Drunken's data quality approach presents similarities with the DQ^{2}S approach in its set of specific data quality operators, as described in Section 2. To compare the two solutions, our Parallel I results for query C are compared against Drunken's Parallel I query C results for each data size (i.e, 100MB, 512MB, 1GB and 10GB) of the Traffic dataset, running on one node of the four-node cluster and also on all nodes of the four-node cluster, as shown in Figure 4. Drunken's query C uses Drunken's implementation of the Completeness operator, which generally checks the presence of null values in a column of a table, but, unlike DQ^{2}S, Drunken does not attach a score to each data record in order to facilitate error detection and repair. Note that Drunken's query C uses all other DQ^{2}S operators that compose query C, so that any differences in the obtained runtimes from the two approaches relate solely to the execution of the Completeness operator. The version of Drunken used in these experiments is v4.1.1. As it can be seen from Figure 4, the DQ^{2}S approach is superior in performance, achieving over an order of magnitude faster response time. Based on these findings, the DQ^{2}S implementation can be deemed as an optimized version of the functionality provided by Drunken.

6. Discussion of Results

In this section, we expand on the results presented in Section 5 in the form of answers to the research questions raised in Section 4:

1. What performance and scalability differences can be observed when using the Non-parallel, Parallel I and Parallel II instances in Local mode? Also, how much data volume can each instance handle?

For the commodity PC, with a total of 6 cores and only 8GB of RAM, Parallel I can effectively run queries T+A and T+C. However, for the Non-parallel and Parallel II instances, the execution of queries T+A and T+C does not successfully terminate, as Pandas aborts the execution of these queries due to lack of sufficient memory space. Parallel I can also process the 10GB dataset for all five queries, which is not the case for the other two instances. On average, the Non-parallel instance exhibits 1.92X higher execution time for the T, C and A queries on the 512MB and 1GB datasets than the other instances; in the worst case, it runs 4.4X slower. Parallel II performs better than Non-parallel, but is still 1.57X slower than Parallel I for the T, C and A queries on the 512MB and 1GB datasets.

An exception is noticed when using the commodity PC and the 100MB traffic and numeric datasets, where Parallel II is up to 25% faster than the second most efficient instance (Parallel I) for queries T, A and C; however, in absolute times, the difference is less than half a second due to the small size of the datasets. For the small (100MB) categorical dataset, however, the Non-parallel solution is faster. *These exceptions involving small datasets reflect the advantages of using Pandas when memory space is sufficient for its memory demanding data manipulations.*

When comparing the Non-parallel and Parallel II instances, considering the queries that the Non-parallel instance can handle, i.e., T, C and A for datasets $\leq$ 1GB, we can observe that Non-parallel is, on average, 33% slower for the traffic dataset, 24% slower for the numeric dataset and equally fast for the categorical dataset. *This reflects the advantages of the combination of Pandas with*

the Spark framework for medium sized time series datasets under constrained memory resources.

For the Cluster-1node setting, with a total of 6 cores and 64GB of RAM, the Non-parallel and Parallel II instances are able to effectively execute the T, A and C queries for the 10GB dataset, as well as the T+C and T+A queries for the 100MB dataset, but the performance obtained from these implementations is still inferior to that of Parallel I. More specifically, Non-parallel is 2.3X slower than Parallel I for the 512MB and 1GB datasets and 5.24X slower for the 10GB dataset on average. It is worth pointing out that, moving the execution of the A query for the 10GB dataset from Non-parallel to Parallel I would allow a speed-up higher than 8X on average. Interestingly, by doing the same for the 512MB dataset, one would obtain an average speed-up of more than 17.3X for queries T+C and T+A. In this setting, Parallel II performs even more poorly, being 2.54X slower for the 512MB and 1GB datasets and 5.78X for the 10GB dataset. Note that, in general, the Parallel II instance is 11% slower than the Non-parallel instance in this infrastructure setting. *These observations, once more, confirm the superior performance of Spark instances that are not based on Pandas, such as Parallel I, in the presence of memory constraints; the positive impact that the availability of larger memory spaces can make on Pandas-based implementations; and also the evidence of better performances from the Non-parallel instance, compared to the Parallel II instance, when more memory resources are available.* It is possible that, if running on a machine with hundreds of GBs of memory, Pandas would show faster running times than Parallel I, but up to a certain dataset size.

2. What performance and scalability differences can be observed when using a single core of a machine versus when using all of the cores of the same machine considering the Parallel I and Parallel II instances?

The most noteworthy observation is that Parallel II does not benefit from a multi/many-core setting in a significant way; i.e., the difference between running

Parallel II on a single core and running it on all cores hardly exceeds 2%. This is not the case for Parallel I. Using all 6 cores of the commodity PC for the Parallel I instance can speedup the execution of the T, C and A queries up to 2.43X. However, the speedups are much lower for the two-join queries, i.e., between 1.7% and 62.5%. We believe the reason why Parallel II does not scale in a significant way when running on a multi/many-core infrastructure setting is due to the use of continuous blocks of memory by Pandas that supports its column-oriented approach for fast analytics, preventing Spark from making a more distributed memory allocation.

It is also worth pointing out that, for the 10GB dataset and on the commodity PC, the C and A queries run faster on a single core, probably because these queries lack any expensive join operation and, consequently, in these cases the overheads associated with a multi-core execution on a machine with limited resources cannot be compensated. As observed from our experiments, using all 6 cores of a powerful cluster machine either yields benefits or exhibits similar performance. For the T, C and A queries the maximum speedup is 5.37X (close to linear speed-up, observed for the A query on the 10GB numeric dataset). For the two-join queries, the speedups are still relatively small, i.e., between 2.7% and 56.4%.

Note that the above mentioned speed-ups are relative speedups. Since Parallel I outperforms the Non-parallel instance, even when using a single core, the real speedups are much higher. For example, for the commodity PC, it is up to 4.41X and, for the cluster-1node, it is over 21.9X (which corresponds to super linear speedup) without considering the cases that Non-parallel cannot execute.

3. What performance and scalability differences can be observed when using a single multi-core machine versus cluster-based computing environments for small, medium and large workloads?

From the experiments, we can observe that employing the full cluster does not incur performance benefits, unless the dataset to be processed is the largest one. For example, for the 10GB dataset, increasing capacity from 6 to 26

cores (i.e., a 4.33-fold increase in the number of cores) leads to performance speed-ups of 2.26X. *Thus, the overhead associated with the Spark executor inter-communication can outweigh any parallelism benefits when processing smaller datasets.* Also, regarding the smaller datasets, it can be seen from the experiments that the use of a single cluster machine is enough, as no signs of memory contention were noticed when processing these datasets.

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

Data quality management applications, in most cases, deal with very large datasets that stress the resources of a commodity machine. Technologies for efficient data manipulation, such as Pandas, can offer fast response times, to some extent, but generally fail to effectively process datasets that are not relatively small (with a few hundred megabytes). In such cases, big data computing frameworks, such as Spark, are the sensible choice, as they can handle larger datasets, regardless of whether the execution is on a single multi-core machine or on a cluster. When using a cluster of machines to run the application, the performance gain is even more significant for large datasets at the order of several GBs. Based on the experiments, a Pandas-based implementation is generally up to 33% slower than the Parallel I implementation for the dataset sizes that the former can cope with. In a machine that has enough main memory resources, a Pandas-based implementation can handle larger datasets with more than a few hundred megabytes, but is still up to 8X slower than the Spark-based solutions when they both execute successfully.

5. What are the challenges of re-engineering the implementation of a DQ-aware query processing system to work on top of big data computing framework?

Each of the technologies used, Spark and Pandas, applies its own internal optimization techniques and, thus, has its own independently implemented set of functions. This means it is not possible to realise some of the functionality of Pandas in Pyspark, due to the different data storing strategies that these

technologies rely on; i.e., whilst Pandas makes use of continuous blocks in main memory, Pyspark is used in distributed scenarios. Due to this main difference, each of the DQ management queries described in this paper was carefully designed and implemented to use and benefit from the underpinning technologies, individually. In the following, we discuss the main technical issues faced in this implementation.

Datatype Optimisations. When analyzing big data, it is important to set data types to the relevant data columns appropriately. In some cases, Pandas treats numerical values as strings and performs string operations on them, using more memory than necessary to store these values. This can also happen when dealing with boolean and datetime columns, or with relatively small numerical values, by using a larger number of bytes to store these values than actually needed. Due to this phenomenon, it is left to the developer the job to explicitly define the data types of various data columns, which, if carefully done, can result in significant data storage reduction by an order of magnitude in most cases. Since Pandas stores data in main memory, this reduction can be crucial for the successful execution of data quality operators, when facing limitations in memory availability. In all our experiments, the Pandas-based solutions exploited data type optimization benefits to the largest possible extent.

Implementation of operators for different data types. The accuracy operator used in the A and A+T queries evaluates an accuracy expression that depends on the data types used within the input dataset. For example, in the experiments with the Traffic dataset, the used accuracy expression is the following: "`Volume = Class1Volume + ... + Class6Volume`" where `Volume` is an integer that indicates the total volume of traffic for a road fragment considering a given time period, and `Class`i`Volume` represents the number of detected vehicles, belonging to class i, that were detected on the same road fragment and during the same time period. Examples of classes of vehicles include cars, trucks, motorcycles, etc. The level of accuracy of traffic data can, therefore, be assessed by checking whether the total volume of traffic is the same as the sum

of the individual volumes of traffic associated with each class of vehicle. On the other hand, because categorical data represents ranges of values, e.g., High, Medium or Low volume of traffic, for the Categorical dataset, the accuracy expression evaluated within the accuracy operator is the following: "`Volume = Class2Volume`", which checks how the level of the total traffic volume compares to the volume of cars, which represent the vast majority of detected vehicles for the road fragment in consideration. If the expression evaluates to true, then the level of accuracy of volume of traffic is reasonable. To check if the different accuracy expressions significantly differ in execution time and impact on the experimental results, we compared the runtimes resulting from the evaluation of the two accuracy expressions using the A query and the 512MB Categorical and Traffic datasets. We observed that, when using the Traffic dataset and Pandas, query A evaluating the first accuracy expression (i.e., the one that involves the sum of values in multiple columns) takes 18 seconds to finish; and when using the Categorical dataset and Pandas, the second accuracy expression (i.e., the one that involves a simple equality comparison between two values is separate columns) takes 17 seconds to finish. Likewise, when using Pyspark, the obtained runtimes are 21 seconds and 20 seconds, respectively; thus, the impact on the overall performance is not high. As a second check, we also examined whether the use of different data types in boolean expressions would, in general, affect query runtime; for example, we measured the runtime resulting from the evaluation of an expression involving a comparison between two Integers against a similar expression involving a comparison between two Strings. And so, we executed 164,000,000 iterations of each comparison in Python, as this number of iterations is approximately the same as the number of rows in our 10GB categorical dataset, and, consequently, the number of times a comparison expression is evaluated when processing this dataset, and then we compared the obtained runtimes. The difference is of approximately 1 second, with the "`100==100`"-type comparison taking 11 seconds, and the "`high`"=="`high`"-type, taking 10 seconds. Again, this difference is relatively small and does not significanly affect the overall runtime of the queries. And so, we concluded that the use of differ-

ent accuracy expressions in the same query for different input datasets would not invalidate our experiment results and interpretations (as shown also in the repository).

Column Removal. When using the Pandas framework, one is able to isolate the data columns that are relevant to the application and discard any other. For example, for the C query, it is possible to discard all the irrelevant columns and deal only with the columns that have been selected to be checked for completeness, due to the fact that the Pandas framework operates in main memory using continuous blocks for column contents and offers a variety of built-in and ready to use data manipulation functions. On the other hand, in Spark, data structure manipulation/extension cannot be easily done using built-in, optimized functions, as Spark allocates memory to objects in a distributed manner. Therefore, to be able to effectively discard irrelevant columns using Spark without Pandas, the data would need to be indexed and/or parsed per record. Due to the overhead this extra task generates, we decided not to discard data columns in these experiments and execute the queries over the whole Spark DataFrame.

Koalas. Due to the limitation of Pandas regarding scalability for big data, the need for technology to enable Pandas to effectively run on distributed settings arises. Koalas[6] addresses this need by augmenting the Pyspark's DataFrame API to make it compatible with Pandas. In other words, this new package combines the scalability of Pyspark with the execution speed of Pandas. Although the project is quite young and needs time to mature, the future of DQ^{2}S seems quite promising, as Koalas will, most likely, remove the need to decide whether to use Pandas or Pyspark, depending on the application, size of dataset, etc., and will, most likely, also enable benefits from each individual technology to be combined by the merge of functionalities from the two technologies. However, at the time of the writing of this work, not all of the Pandas functions are implemented in Koalas. Due to this fact, we were not able to implement the same

[6]https://github.com/databricks/koalas

queries using this new package.

As discussed previously, the benefits of using the full cluster (Cluster-4nodes) mainly appear when dealing with the largest datasets used in this empirical study (10GB). This result was expected since, at this volume of data, resources from one machine alone for handling and processing datasets may not be sufficient when deploying the workload on non-cluster based infrastructures or single cluster nodes. For the 1GB datasets, employing the full cluster reveals slightly worse performance than when employing a single machine, with exceptions being observed for the T+A query only, which is the most demanding query in terms of memory and processing. For smaller datasets, the associated overheads clearly outweigh any parallelism benefits. These observations may hold for a few more years, since, given trends such as Moore's law, single machine hardware capabilities will continue to increase and allow larger datasets to be handled efficiently when performing data quality processing tasks. Table 6 summarises other key insights obtained from the experiments with a focus on *Cluster-1 node*. Insights obtained from the experiments with this infrastructure generalise well the insights obtained from all experiments, particularly because the *Cluster-1node* infrastructure setting shares important properties with the other two settings, by having the same machine specification as one of the nodes of the *Cluster-4nodes* infrastructure, and by being composed of a single node, as the *Commodity PC* infrastructure, helping to simplify our explanations. Also, with this infrastructure, the most complete set of results were obtained for all scenarios.

In particular, in Table 6, we describe our main remarks about how different dataset types impact on the execution time of each query. As dataset size and DQ^2S instance also impact on execution time, these are included in our remarks, emphasising how the different technologies explored in the implementation of each instance (e.g., Pandas and/or Spark) impact on execution time.

Query	**Key Insights**

C	The experiments with query *C* and the three datasets have shown that, in the *Non-parallel* DQ^{2}S instance, *Numeric* is the dataset with the lowest execution times. The *Traffic* dataset has the second best performance, while, for the *Categorical* dataset, execution times are about 30 to 45% longer than those obtained for the *Numeric* dataset. The factors that have most contributed to these differences in execution time are the following: the sensitivity of the Pandas library to availability of memory space, which benefits the dataset with the lowest sized data record (in bytes), i.e., the *Numeric* dataset; and the evaluation cost of the Completeness expression of query *C*, which is at its lowest when applied over the *Numeric* dataset, since it involves simple equality comparisons of small integer values. These differences in execution time, however, cannot be clearly observed when executing query *C* in the *Parallel I* and *Parallel II* DQ^{2}S instances, using the 1GB, the 512MB and the 100MB dataset sizes. However, for the dataset size of 10GB and the same instances, the times obtained from the processing of the *Categorical* dataset were, approximately, 23% lower than those obtained for the other two datasets. On one hand, the negative impact of diminished memory spaces on the Pandas library, when processing larger datasets in the *Parallel II* instance; and, on the other hand, the positive impact of the Spark framework on both *Parallel I* and *Parallel II* instances, relating to its better handling of larger datasets, have contributed to lower the times for processing the *Categorical* dataset, which contains the largest sized records.
A	Of all five queries, query *A* presents the lowest execution times, along with query *C*, considering all instances, datasets and dataset sizes. However, when processing the *Categorical* dataset, query *A* shows, on average, a 10% advantage over not only query *C*, but also its own processing of the other two datasets. In particular, the Accuracy expression evaluated in query *A* over the *Numeric* dataset incurs a slightly higher cost, as it performs arithmetic operations involving data values in six different columns of the input file, while the Accuracy expression applied over the *Categorical* dataset evaluates a single Boolean expression involving data values in only two different columns.
T	Compared to the other two queries that involve a single DQ dimension (i.e., *C* and *A*), query *T* is the one with the highest execution times (50% higher, on average), considering all instances, datasets and dataset sizes, largely due to its expensive join operation. Although less consistency is observed in the impact of dataset type on the execution times of query *T*, in a considerable number of cases, query *T* shows lower execution times when processing the *Categorical* dataset. The fact that this dataset is the one with the smallest total number of records (compared to *Traffic* and *Numeric*) has a positive impact on the execution cost of the join operation.

T+C	Along with query *T+A*, query *T+C* is the most expensive query, considering all instances, datasets and dataset sizes, largely due to the presence of two expensive join operations. When processing the *Numeric* dataset, it presents its longest execution times (by up to 31%) considering all instances (note that the *Non-parallel* instance is able to process only the 100MB dataset size for this query). The fact that the *Numeric* dataset is the one with the largest total number of records (compared to Categorical and *Traffic*) has a negative impact on the cost of the join operations. However, from the experimental results, it cannot be clearly observed which of the *Categorical* and *Traffic* datasets would come in second place.
T+A	The insights for query *T+A* are similar to the ones for query *T+C*.

Table 6: Key Insights.

7. Conclusions and Future Work

The research described in this paper provides a detailed account of an experimental journey to explore the capabilities of a popular big data computing framework (Apache Spark) and a library for fast data manipulation (Pandas) to improve performance and scalability of DQ-aware query processing. The study also provides empirical insights for researchers interested in re-purposing and porting existing data quality management and data wrangling systems and tools to explore the capabilities provided by big data computing frameworks.

The issue of porting existing DQ management tools to run on top of big data computing frameworks to exploit parallelism in the execution of DQ management tasks is of considerable importance given that the majority of tools available to support DQ management tasks were designed and implemented without native support for big data computing. From the perspective of big data research, this issue is also an important and under-explored dimension, as most work in this area focuses on the variety, volume and velocity of data, with limited contributions addressing the veracity (data quality) dimension (Saha & Srivastava, 2014, March; Fan, 2015).

We believe the main motivation for the so called "post-system development parallelization" in the age of Machine Learning lies in the fact that, even though algorithms induced from training data sets may have higher accuracy in general,

these algorithms may not scale well, since they are not designed with parallelism exploitation in mind. Therefore, research in the future will increasingly involve porting ML-trained algorithms into parallel and distributed computing infrastructures and, thus, face similar challenges as the ones reported in this study.

To be able to port an existing DQ management tool onto a big data framework, some level of re-engineering may be necessary, to design the solution in terms of the data structures and functions that the framework is able to use for the purposes of parallelising the execution. This re-engineering needs to be carefully planned if significant performance and scalability improvements are to be achieved, by avoiding the combination of functionality for parallelism exploitation, provided within the framework, with other functionality, provided elsewhere, which can prevent effective parallelization of the execution by the framework. This case was noted when Pandas and PySpark were combined to explore intra-node as well as inter-node parallelism. In other words, the empirical study outlined in this paper has shown that substantial pay-offs can be obtained by using optimized data science libraries combined with the parallel and distributed capabilities of big data computing; however, the combination of libraries and the usage of a cluster should be done with care.

It is also noted that, even though the size of the input data sets is the most obvious factor impacting on the decision as to which big data platform and infrastructure settings to adopt, data transformations performed during execution and characteristics of the tasks have also a significant impact on the level of scalability and performance that can be obtained from the framework and infrastructure. For example, for the cases where medium data sets of 512MB and 1GB were input to the T+A query, a single multi-core machine execution performed poorly compared to the same job executing on the cluster, unlike all other queries when processing data sets of these sizes. It has been observed that this is largely due to the fact that query T+A generates larger intermediate results than the other queries, and performs some extra work to transform row-based data manipulations into column-based manipulations that the Python functions are optimized to perform.

The following summarises our main observations, based on the discussion of experimental results provided in Section 6:

- The combination of Spark and Pandas did not yield significant benefits when moving from a single-node cluster to a multi-node cluster, due to conflicting memory allocation strategies of these two technologies, as Pandas tends to make use of continuous blocks of memory in support of its column-oriented approach for fast analytics, making it difficult, to some extent, for Spark to make a more distributed memory allocation (refer to the discussion on Koalas in Section 6).

- The porting of DQ management tools, originally built without native support for big data computing, to a big data computing platform can have a significant impact on data processing performance. We have been able to observe this when comparing results obtained before the porting was performed, reported in (Sampaio et al., 2015), with the results presented in this paper, where, for instance, data volumes that are three orders of magnitude larger were processed within similar time periods.

- A key guideline for choosing infrastructure setting and engine implementation when processing medium to small datasets (e.g., of size 1GB or less) involves the use of a single, powerful machine to exploit the performance benefits of a big data computing framework, such as Spark, without the exhaustive use of libraries for fast in-memory data manipulation, such as Pandas, as observed from results involving the Parallel I implementation instance and the Cluster-1node setting. Even though experiments with small data sizes of 100MB have shown us the advantages of Pandas, the provided additional benefits of the combination Pandas-Spark are still modest, with execution times just a few seconds shorter. For larger datasets, running Parallel I on a cluster has been observed to yield benefits.

- The combination of multiple DQ dimensions in a single DQ management

task, due to requiring multiple join operations, has a tangible negative effect on scalability.

- From the successful execution of complex DQ management tasks in all engine instances of the target DQ processing tool, it has been observed that speedups were of multiple orders of magnitude.

- On any infrastructure setting, the impact of dataset type on execution time can only be observed if physical data properties, such as number of records, and other job characteristics, such as specific data management task, engine implementation and dataset size, are also taken into account.

Future research directions include: (i) The investigation of the performance of an extended library of low-level constructs in line with the work in (Schelter et al., 2018) and the user requirements described in (Krishnan et al., 2016, June); the latter survey emphasizes also the lack of benchmarks to assess the performance of different data quality solutions. (ii) The investigation of further Apache Spark tuning, e.g., in line with the work in (Gounaris & Torres, 2018); we also plan to apply Machine Learning for setting hyper-parameters towards optimizing the choice when a portfolio of algorithms can be used for executing a given task (Morar et al., 2017).

Acknowledgements

Financial support has been provided by the National Council of Science and Technology (abbreviated CONACYT) to the first author (agreement n. 411896). The research of the authors from the Aristotle University has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code:T1EDK-01944) Financial support for this research was also provided by the Alliance Manchester Business School Big Data Forum research programme.

References

Alecu, F. (2007). Performance analysis of parallel algorithms. *Journal of Applied Quantitative Methods*, *2(1)*, 129–134.

Alexandrov, V. (2013). Parallel scalable algorithms - performance parameters. `https://www.bsc.es/sites/default/files/public/computer_science/extreme_computing/parallel_algorithms_bcn_prace_l1_part2_2013.pdf`. Accessed: 2019-11-15.

Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

Augspurger, T., Ayd, W., Bartak, C., Battiston, P., Cloud, P., Garcia, M., Hayden, A., Horikoshi, M., Hoyer, S., McKinney, W., Mendel, B., Reback, J., Roeschke, M., Schendel, J., She, C., den Bossche, J. V., & 7.Young (2015). Python data analysis library. `http://pandas.pydata.org/`. Accessed: 2019-11-22.

Ballou, D., Wang, R., Pazer, H., & Tayi, G. K. (1998). Modeling information manufacturing systems to determine information product quality. *Management Science*, *44(4)*, 462–484.

Bonner, S., McGough, A. S., Kureshi, I., Brennan, J., Theodoropoulos, G., Moss, L., Corsar, D., & Antoniou, G. (2015, October). Data quality assessment and anomaly detection via map/reduce and linked data: A case study in the medical domain. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)* (pp. 737–746). doi:`10.1109/BigData.2015.7363818`.

Cheng, D. X., Golab, W., & Ward, P. A. S. (2016, March). Efficient incremental smart grid data analytics. In *Workshop Proceedings of the EDBT/ICDT 2016 Joint Conference on CEUR-WS.org*.

Demmel, J. (1995). Measuring Performance of Parallel Programs. http://people.eecs.berkeley.edu/{~}demmel/cs267-1995/lecture04.html. Accessed: 2019-11-15.

Fan, W. (2015). Data quality: From theory to practice. *ACM SIGMOD Record*, *44(3)*, 7–18. doi:10.1145/2854006.2854008.

Ge, C., He, X., Ilyas, I. F., & Machanavajjhala, A. (2019, June). Apex: Accuracy-aware differentially private data exploration. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)* (pp. 177–194). doi:10.1145/3299869.3300092.

Gounaris, A., & Torres, J. (2018). A methodology for spark parameter tuning. *Big Data Research*, *11*, 22–32. doi:10.1016/j.bdr.2017.05.001.

Harnie, D., Saey, M., Vapirev, A. E., Wegner, J. K., Gedich, A., Steijaert, M., Ceulemans, H., Wuyts, R., & Meuter, W. D. (2017). Scaling machine learning for target prediction in drug discovery using apache spark. *Future Generation Computer Systems*, *67*, 409 – 417. doi:http://dx.doi.org/10.1016/j.future.2016.04.023.

He, J., Veltri, E., Santoro, D., Li, G., Mecca, G., Papotti, P., & Tang, N. (2016, July). Interactive and deterministic data cleaning. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)* (pp. 893–907). doi:10.1145/2882903.2915242.

Heidari, A., McGrath, J., Ilyas, I. F., & Rekatsinas, T. (2019, June). Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)* (pp. 829–846). doi:10.1145/3299869.3319888.

Khayyat, Z., Ilyas, I. F., Jindal, A., Madden, S., Ouzzani, M., Papotti, P., Quiané-Ruiz, J.-A., Tang, N., & Yin, S. (2015, June). Bigdansing: A system for big data cleansing. In *Proceedings of the International Conference on*

Management of Data (SIGMOD) (pp. 1215–1230). doi:10.1145/2723372.2747646.

Krishnan, S., Haas, D., Franklin, M. J., & Wu, E. (2016, June). Towards reliable interactive data cleaning: A user survey and recommendations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA)* (pp. 9:1–9:5). doi:10.1145/2939502.2939511.

Kune, R., Konugurthi, P. K., Agarwal, A., Chillarige, R. R., & Buyya, R. (2016). The anatomy of big data computing. *Software Practice and Experience*, *46(1)*, 79–105. doi:10.1002/spe.2374.

Kwiatkowski, J. (2001, September). Evaluation of parallel programs by measurement of its granularity. In *Proceedings of the International Conference in Parallel Processing and Applied Mathematics* (pp. 145–153). doi:10.1007/3-540-48086-2_16.

Li, X., Song, J., Zhang, F., Ouyang, X., & Khan, S. U. (2016). Mapreduce-based fast fuzzy c-means algorithm for large-scale underwater image segmentation. *Future Generation Computer Systems*, *65*, 90 – 101. doi:http://dx.doi.org/10.1016/j.future.2016.03.004.

Loshin, D. (2014). Understanding big data quality for maximum information usability. https://www.sas.com/content/dam/SAS/en_us/doc/whitepaper1/understanding-big-data-quality-107113.pdf. Accessed: 2019-11-22.

Mahdavi, M., Abedjan, Z., Fernandez, R. C., Madden, S., Ouzzani, M., Stonebraker, M., & Tang, N. (2019, June). Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)* (pp. 865–882). doi:10.1145/3299869.3324956.

Milani, M., Zheng, Z., & Chiang, F. (2019, April). Currentclean: Spatio-temporal cleaning of stale data. In *Proceedings of the 35th IEEE International*

Conference on Data Engineering (ICDE) (pp. 172–183). doi:10.1109/ICDE.2019.00024.

Morar, T., Knowles, J., & Sampaio, S. (2017). Initialization of bayesian optimization viewed as part of a larger algorithm portfolio. https://pdfs.semanticscholar.org/6655/7923e7538f6c52fa5a4da07a1a75056d423c.pdf?_ga=2.258824386.917927128.1574450387-1580165763.1571312984. Accessed: 2019-11-22.

OpenRefine-Organization (2019). OpenRefine. https://openrefine.org/. Accessed: 2019-11-22.

Pancake, C. (1996). Is parallelism for you? *IEEE Computational Science and Engineering*, *3*, 18–37. doi:10.1109/99.503307.

Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., & Naumann, F. (2015). Data profiling with metanome. *Proceedings of the VLDB Endowment*, *8*, 1860–1871. doi:10.14778/2824032.2824086.

Pipino, L. L., Lee, Y. W., Wang, R. Y., Lowell Yang Lee, M. W., & Yang, R. Y. (2002). Data quality assessment. *Communications of the ACM*, *45*, 211–218. doi:10.1145/505248.506010.

Saha, B., & Srivastava, D. (2014, March). Data quality: The other face of big data. In *Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE)* (pp. 1294–1297). doi:10.1109/ICDE.2014.6816764.

Sahni, S., & Thanvantri, V. (1996). Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology: Systems & Technology*, *4*, 43–56.

Sampaio, S., Aljubairah, M., Permana, H. A., & Sampaio, P. (2019). A conceptual approach for supporting traffic data wrangling tasks. *The Computer Journal*, *62*, 461–480. doi:10.1093/comjnl/bxy113.

Sampaio, S., Dong, C., & Sampaio, P. R. F. (2015). DQ2S - A framework for data quality-aware information management. *Expert Systems with Applications*, *42*, 8304–8326. doi:10.1016/j.eswa.2015.06.050.

Schelter, S., Grafberger, S., Schmidt, P., Rukat, T., Kießling, M., Taptunov, A., Bießmann, F., & Lange, D. (2019, April). Differential data quality verification on partitioned data. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)* (pp. 1940–1945). doi:10.1109/ICDE.2019.00210.

Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, *11*, 1781–1794. doi:10.14778/3229863.3229867.

Shahrivari, S., & Jalili, S. (2016). Single-pass and linear-time k-means clustering based on mapreduce. *Information Systems*, *60*, 1–12. doi:http://dx.doi.org/10.1016/j.is.2016.02.007.

Singh, H., & Bawa, S. (2017). A mapreduce-based scalable discovery and indexing of structured big data. *Future Generation Computer Systems*, *73*, 32–43. doi:http://dx.doi.org/10.1016/j.future.2017.03.028.

Sun, X.-H., & Gustafson, J. L. (1991). Toward a better parallel performance metric. *Parallel Computing*, *17*, 1093–1109. doi:http://dx.doi.org/10.1016/S0167-8191(05)80028-6.

Taleb, I., Dssouli, R., & Serhani, M. A. (2015, June). Big data pre-processing: A quality framework. In *Proceedings of the 2015 IEEE International Congress on Big Data* (pp. 191–198). doi:10.1109/BigDataCongress.2015.35.

Tous, R., Gounaris, A., Tripiana, C., Torres, J., Girona, S., Ayguad, E., Labarta, J., Becerra, Y., Carrera, D., & Valero, M. (2015, October). Spark deployment and performance evaluation on the marenostrum supercomputer. In *Proceedings of the 2015 IEEE International Conference on Big Data*

(Big Data) (pp. 299–306). doi:http://dx.doi.org/10.1109/BigData.2015.7363768.

Trifacta (2017). Trifacta data wrangling for hadoop: Accelerating business adoption while ensuring security & governance. http://pages.trifacta.com/rs/172-KJH-591/images/Trifacta-White-Paper-Accelerating-Adoption-Ensuring-Governance.pdf. Accessed: 2019-11-22.

Veiga, J., Expósito, R. R., Pardo, X. C., Taboada, G. L., & Tourifio, J. (2016, December). Performance evaluation of big data frameworks for large-scale data analytics. In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)* (pp. 424–431).

Wang, P., & He, Y. (2019, June). Uni-detect: A unified approach to automated error detection in tables. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)* (pp. 811–828). doi:10.1145/3299869.3319855.

Zhang, A., Song, S., Sun, Y., & Wang, J. (2019, April). Learning individual models for imputation. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)* (pp. 160–171). doi:10.1109/ICDE.2019.00023.

Zhang, F., & Sakr, M. (2013, May). Dataset scaling and mapreduce performance. In *Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)* (pp. 1683–1690). doi:10.1109/IPDPSW.2013.143.

Appendix A. DQ²S Constructs Description

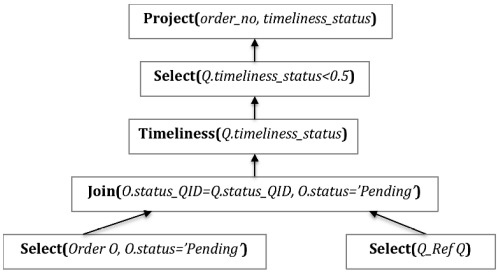

Figure A.5: Query Plan for Timeliness query expressed in DQ²L (Sampaio et al., 2015).

Operator	Description	Input	Output
TimelinessQuery	Contains the main structure of the algorithm, utilises the query engine operators to produce the query result.	Names of the required datasets to complete the query and the predicates needed.	ID and timeliness score of the tuples with timeliness score below 0.5.
Timing	Obtains 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.	None.	Elapsed runtime in nanoseconds.
ScanSelect	Loads the dataset from a CSV file and fills three array lists from Tuple Class with the appropriate information given by the dataset values.	Comma separated CSV dataset.	A tuple with three dynamic generic arrays containing the values, data types and name of the attributes from a row in the loaded CSV.
Tuple	Creates a data structure containing the data ingested by ScanSelect and schema information (data types and attribute names).	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).
Predicate	Evaluates a given predicate and returns a boolean value depending on the result of the evaluation.	A predicate made of two operands and one comparison operator.	Boolean value telling if the predicate evaluated to True or False after applied on the input data.
Join	Joins two dynamic generic arrays based on a given join predicate.	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.
JoinPredicate	Is a support algorithm to the Join operator. It evaluates the join predicate against the input data and returns a boolean value indicating whether the predicate evaluation resulted in True or False.	Join predicate.	Boolean value telling if the join predicate was true or false after applied on a given data.
Timeliness	Calculates the timeliness score based on the formulas presented in Section 3 of this paper.	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 corresponding calculated timeliness score.
Select	Filters a dynamic generic array and creates a new one based on the result obtained from the Predicate class.	Filter predicate and a dynamic generic array to apply the filter on.	A dynamic generic array containing the filtered data.

Project	Extracts columns from a dynamic generic array based on a list of attributes given (name of the columns).	List of attributes and a dynamic generic array.	A dynamic generic array containing the data that corresponds to the required columns.

Table A.7: Operators comprising the DQ^{2}S Timeliness query.

Appendix B. DQ²S Query Execution Plans

Figures B.6, B.7, B.8, B.9 and B.10 show the five query plans for each of the queries described in Section 4.1.

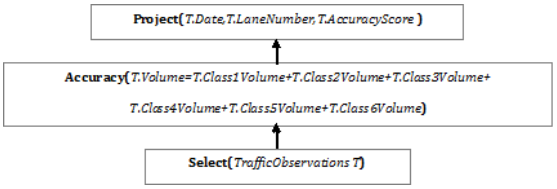

Figure B.6: Execution Plan for the Accuracy (A) Query.

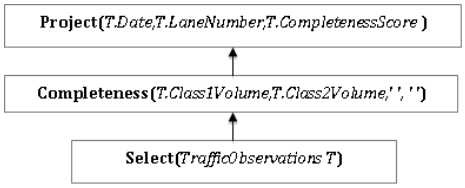

Figure B.7: Execution Plan for the Completeness (C) Query.

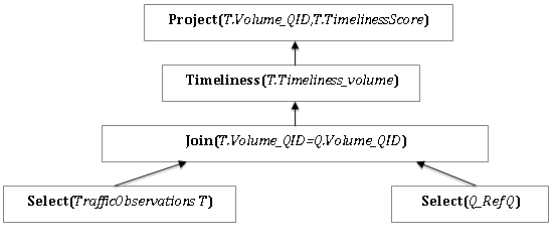

Figure B.8: Execution Plan for the Timeliness (T) Query.

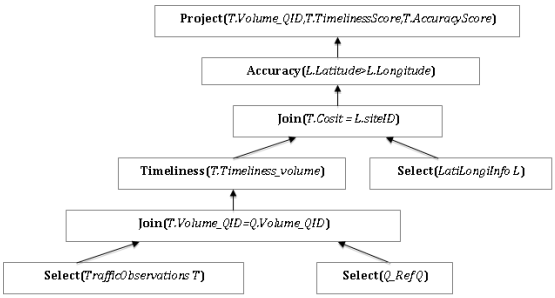

Figure B.9: Execution Plan for the Timeliness-Accuracy (T+A) Query.

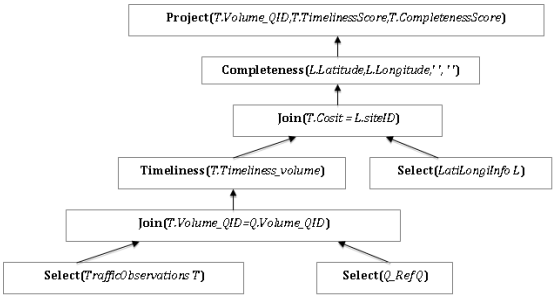

Figure B.10: Execution Plan for the Timeliness-Completeness (T+C) Query.

Appendix C. Files Sizes Used in Each Query

Tables C.8, C.9 and C.10 show the sizes and number of rows of the datasets used in each query for the experiments with 512MB, 1GB and 10GB, respectively.

Dataset	*Traffic*			*Numeric*			*Categorical*		
/Query	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*
C, A	512.6MB 6411566 rows	-	-	512.3MB 7750000 rows	-	-	512.5MB 6223635 rows	-	-
T	258.5MB 3240000 rows	252.9MB 3627098 rows	-	260.8MB 3959999 rows	252.7MB 3627098 rows	-	260.1MB 3164581 rows	252.7MB 3627098 rows	-
T+C	165.2MB	170.3MB	177.7MB	165.0MB	170.3MB	177.7MB	165.3MB	170.3MB	177.7MB
T+A	2075000 rows	2450000 rows	4261145 rows	2509999 rows	2450000 rows	4261145 rows	2016199 rows	2450000 rows	4261145 rows

Table C.8: File sizes (in MB) and number of rows for each query used in the experiments with 512MB of data.

Dataset	*Traffic*			*Numeric*			*Categorical*		
/Query	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*
C, A	1.0GB 13381964 rows	-	-	1.0GB 16320160 rows	-	-	1.0GB 12243304 rows	-	-
T	588.6MB 7360082 rows	514.0MB 7360083 rows	-	511.8MB 7750001 rows	514.0MB 7360083 rows	-	512.1MB 6218419 rows	514.0MB 7360083 rows	-
T+C	300.3MB	300.1MB	400.6MB	300.4MB	306.1MB	400.6MB	300.3MB	306.1MB	400.6MB
T+A	3800000 rows	4390000 rows	9604913 rows	4554999 rows	4390000 rows	9604913 rows	3653090 rows	4390000 rows	9604913 rows

Table C.9: File sizes (in MB) and number of rows for each query used in the experiments with 1GB of data.

Dataset	*Traffic*			*Numeric*			*Categorical*		
/Query	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*	*File 1*	*File 2*	*File3*
C, A	10.8GB 133819637 rows	-	-	10.9GB 163201601 rows	-	-	10.3GB 124016365 rows	-	-
T	5.9GB 73600820 rows	5.2GB 73600820 rows	-	4.9GB 73600820 rows	5.2GB 73600820 rows	-	5.1GB 61334001 rows	5.2GB 73600820 rows	-
T+C	3.3GB	3.1GB	4.7GB	3.0GB	3.0GB	4.0GB	3.0GB	3.0GB	4.0GB
T+A	37400000 rows	43000000 rows	96074333 rows	44999999 rows	43000000 rows	96074333 rows	37028562 rows	43000000 rows	96074333 rows

Table C.10: File sizes (in GB) and number of rows for each query used in the experiments with 10GB of data.

Appendix D. Summary of the Evaluation Methodology

Figure D.11 shows a summary of the adopted evaluation methodology. Note that experiments were performed for all combinations of DQ^2S instances, infrastructure settings, queries, dataset types and dataset sizes described in Section 4.2, the only exception being experiments involving the *Non-parallel* instance and the *Cluster-4nodes* infrastructure setting, as this is the only instance that cannot be made to execute on a cluster with multiple nodes. As a consequence, each dot in the figure represents all dataset types considered in this work (i.e., Numeric, Categorical and Traffic).

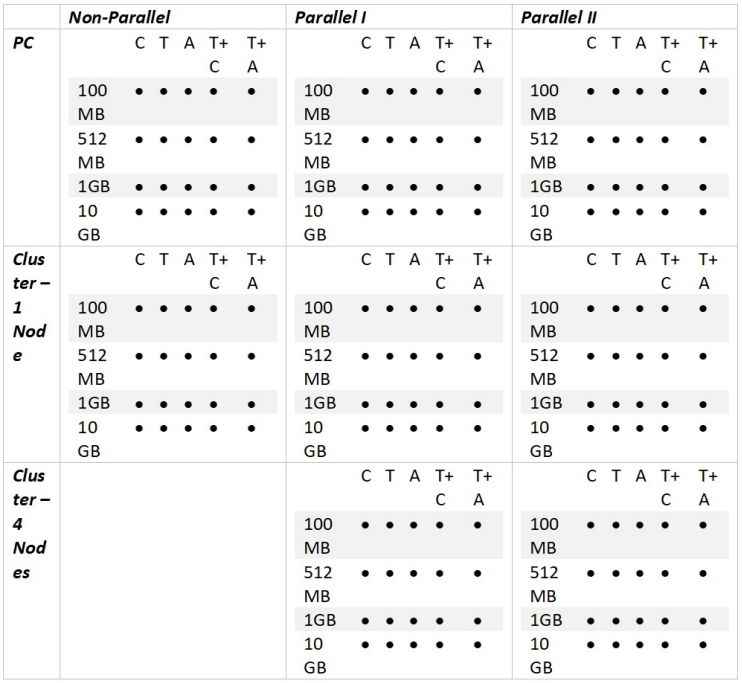

Figure D.11: Summary of the Evaluation Methodology.