POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master Degree in Computer Engineering

Master Thesis

Data Quality for streaming applications

Advisor
prof. Paolo Garza

Candidate:
Andrei Robert Zannelli

Company tutors
AGILE LAB S.R.L
Ing. Paolo Platter

April 2021

Acknowledgements

Thanks to my family for having allowed this goal through their support, to Antonella for having always believed in me and to my friends and companions in this adventure. I would like to thank Roberto Coluccio, for the teachings (not only academic) given to me during the first phase of the thesis. Last but not least, to me for never giving up.

Abstract

The topic of big data has become highly sought after in recent years and with it all the problems that they entail. The ability to analyze large amounts of data in an innovative way has made it possible to facilitate the development and the enormous production of data from countless sources such as social media, sensors, industrial machines or simply server logs has certainly encouraged the development of the big data field. With the increase in the production speed of all these types of data, we have begun to speak of streaming data, to indicate the production of data in near real time. Of course, with the acceleration of production, the need to hasten their analysis rose too and there were many answers proposed by the top players in the sector, such as Apache Spark and its two components dedicated to streaming, DStreams and Structured Streaming. One of the fundamental problems of all this data is its quality, as it is often used by large companies to decide the best business choices to implement in response to the data in their possession. The latter two themes are the protagonists of this thesis, namely data quality in streaming environments. This thesis rests its foundations on the Data Quality framework, an open source software produced by AgileLab s.r.l., which aims to transport the academic concepts of data quality into a practical and everyday reality. The objective set by this thesis was to integrate the aforementioned framework with the possibility of managing the analysis of streaming sources, without having to change the very nature of the project and solving all the problems that streaming carries with it. The analysis and design of the solution started from highlighting the main problems, the necessary abstractions and the work carried out in academic fields by the community of researchers and developers to enter the world of streaming data, passing through the technologies and methods used by this research. Subsequently, the current structure of the aforementioned framework is passed under the microscope, analyzing the various modules and entities, to highlight its functioning and capabilities. The changes made to allow integration with the most commonly used formats and file types within the streaming data circle are reported immediately after, also highlighting both the conceptual and implementation problems. The research ends with the analysis and comparison of the standard model and the one created in this thesis, using a common database for both as benchmark and highlighting how the various components work together to obtain the same result in both cases. Finally it is exposed what can be implemented to continue the optimization of the project and preparing the most interesting improvements.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Big Data

Nowadays, the amount of data generated has grown exponentially, mainly because the manufacturers of these data have grown rapidly in numbers since data is now collected by the most disparate variety of devices such as mobile devices, cheap and numerous information-sensing Internet of things devices, aerial (remote sensing), software logs, cameras, microphones, radio-frequency identification (RFID) readers and wireless sensor networks.

Clearly, size is the first characteristic that comes to mind considering the question "what is big data?" However, other characteristics of big data have emerged recently. For instance, Laney et al. (2001)[1] suggested that Volume, Variety, and Velocity (or the Three V's) are the three dimensions of challenges in data management. The Three V's have emerged as a common method to describe big data[2]. To these three V's, we can also add Veracity and Value:

- Volume: scale of the data. In this context, we are talking of ZettaBytes (1024^7 Bytes). Is estimated that from 2009 to 2020, we had an increase factor of 44 in the amount of data produced.
- Variety: different forms of data as for formats, types and structure (numerical, video, text, etc.)[3] that is being produced.
- Velocity: analysis of streaming data. A fast generation rate leads to having the right tools to perform a quick analysis, in order to keep up with the amount of data produced.
- Veracity: uncertainty of the data and its quality.
- Value: exploit information provided by data that translates into business advantage.

The last V is the one of interest for this thesis, that is the quality of the data we are working on. Therefore, having a good quality in the data automatically leads to a greater value of information that we can extract, consequently the need emerges to have the right tools to assess the quality of the analyzed data.

1.2 Data Quality

The literature comprises various techniques for defining, assessing, and improving data quality. However, requirements for data and their quality vary between organizations. Due to this variety, choosing suitable methods that are advantageous for the data quality of an organization or in a particular context can be challenging[4]. The idea has been addressed mainly in the scope of the use case that was being analyzed, so there are many standards for what data quality should be. From biology to telecommunications, from scientific data to medicine, is clear that we are still away from a specific standardization of how should be defined and what should respect data quality standards. Naturally, an attempt was made to create a common basis for most of the fields of interest which surrounds the big data landscape and the main dimensions have been defined as follows:

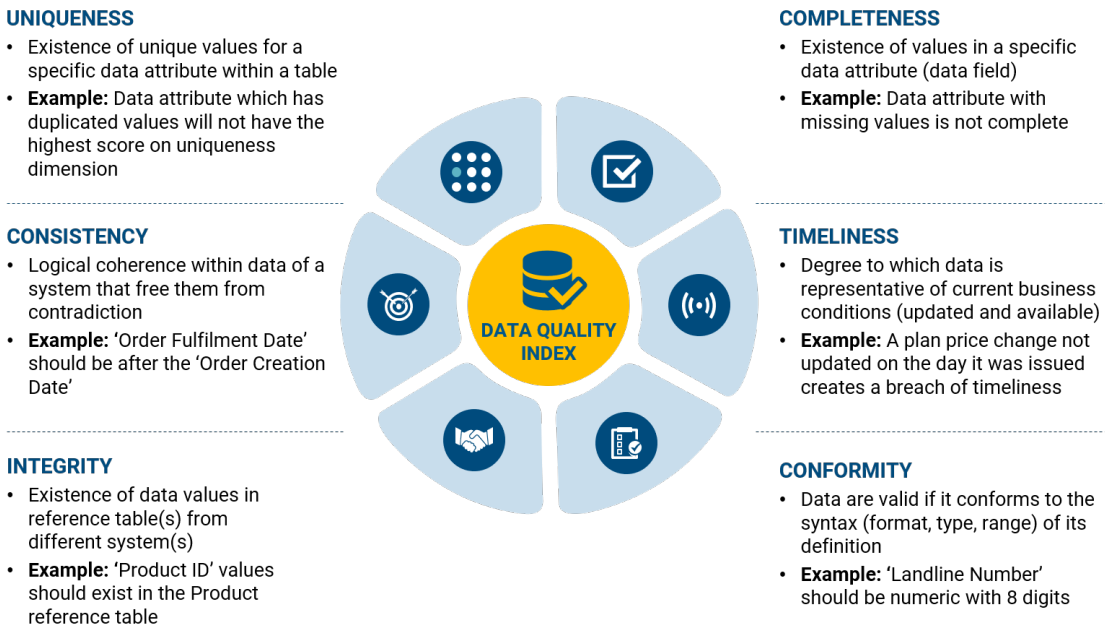


Figure 1.1: The data quality dimensions[5]

- Completeness: existence of values in a specific data attribute.

- Timeliness: degree to which data is representative of current business conditions.
- Conformity: data are valid if it conforms to the syntax of its definition.
- Uniqueness: existence of unique values for a specific data attribute within a table.
- Consistency: logical coherence within data of a system that free them from contradictions.
- Integrity: existence of data values in reference table(s) from different system(s).

When we talk about big data analysis, we can identify two distinct phases: the first consists in verifying and validating the quality of the data being analyzed and the second consists in the actual analysis. Being the first step into the analysis of big data, the quality of the former should be one of the most important pillar in any pipeline which aims to deal with this technology. The main methods that we are going to expose in chapter 2 present subjective validation tools: this choice is dictated by the fact that the quality of the data must first of all be an attribute directly perceived by all those people and entities that interface with the data, both to make business choices (such as stakeholders), both to take advantage of the information they bring within themselves. Knowing that you have data of dubious quality or unsuitable for the task can lead to incorrect choices and actions, which involve various repercussions for the company itself. For this reason, and due to the exponential increase in data production in recent decades, efforts towards this sector have enhanced considerably, heightening awareness of the sensitivity of the data quality area.

The study of this thesis will start from the work done from the academic point of view by other researchers in the field of data quality, reporting what they have tried to standardize as components and dimensions of the quality problem. Subsequently the major technological tools used in this sector will be reported, also explaining their correlation with the development of this thesis. Subsequently, the framework from which the analysis of the problem started, provided by Agile Lab S.r.l, will be presented and the changes made to achieve the objective set by the research will be shown.

Chapter 2

Related Works

This chapter will begin with a report on what was accomplished by others, both from a theoretical and a practical point of view, who have tried to structure and resolve similar problems to the one addressed by this thesis. Then, focus will shift on the tools, frameworks and notions that have been used in order to accomplish the task. Background and basic notions will be provided to the reader in order to help through the reading of this thesis.

2.1 Academic papers

In this first section we will present a series of Data Quality frameworks and methodologies, to highlight what we have focused on so far and how the parameters in the field of data quality have been defined. Some of the methods are presented and evaluated by[6]. Due to the very nature of the problem, it is easy to find methodologies proposed in academic environments but very difficult to find open source implementations, since most of the companies that engage in this problem tend to offer ad-hoc solutions for their customers and with proprietary software solutions. All the DQ dimensions reported in each methodology are defined by how the authors of the methodology itself intended them.

2.1.1 AIMQ: a methodology for information quality assessment

AIMQ[7] it is a methodology that seeks to report how an IQ assessment is made and provide the basis for testing its effective quality. Its main components are: a Data Quality categorization model, an information quality assessment instrument, a bench-marking gap analysis and a role gap analysis. The measurement techniques used in this method could be applied to both structured and unstructured data. The DQ dimensions used are:

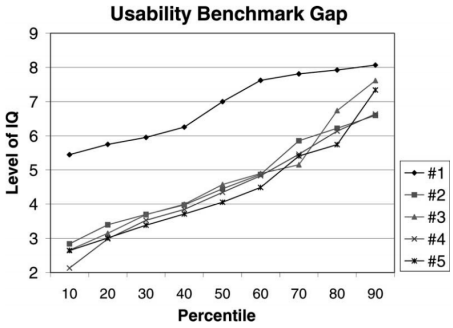
- Accessibility: how accessible a given information is, also in terms of speed when we need it.
- Appropriate amount: this information is of sufficient volume for our needs.
- Believability: how credible the information is.
- Completeness: the information contains all the values we need.
- Consistent representation: the representation of this information is compact and concise.
- Ease of operation: the simplicity in manipulating information so that we can achieve our goal.
- Free-of-error: this information is correct, accurate and reliable.
- Interpret-ability: it is easy to interpret what this information means.
- Objectivity: this information was objectively collected and presents an impartial view.
- Relevancy: this information is useful to our work.
- Reputation: this information comes from good sources.
- Security: access to information is only possible for those who hold the right.
- Timeliness: this information is sufficiently up-to-date for our work.
- Understand-ability: the meaning of this information is easy to understand.

These dimensions are grouped by the authors into four main categories, intrinsic IQ, accessibility IQ, representational IQ, and contextual IQ.

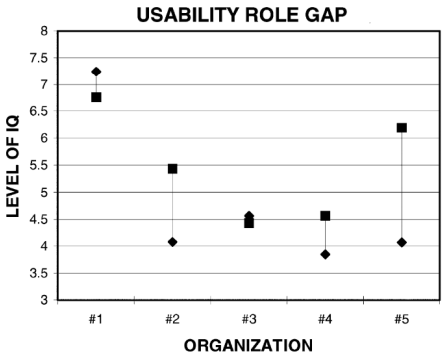
By intrinsic IQ the authors mean that the data have an inherent quality of information, that is, that the information itself already carries a sort of quality. With contextual IQ, the focus is on the need to consider the IQ within the context in which the current task is performed: this translates into requiring that the IQ is complete, relevant, timely and exhaustive from the point of view of quantity. The last two categories, that is representation and accessibility, highlight the weight that the computer system carries, which has the purpose of storing and providing access to the information itself. In other words, this translates into a system that allows information to be presented in an interpretable, easy to understand and concise way. Of course, it is also intended that the aforementioned system is accessible, but only by those who have the necessary authorization to do so, therefore there is a secure system.

	Conforms to specifications	Meets or exceeds consumer expectations
Product Quality	Sound information IQ dimensions Free-of-error Concise representation Completeness Consistent representation	Useful information IQ dimensions Appropriate amount Relevancy Understandability Interpretability Objectivity
Service Quality	Dependable information IQ dimensions Timeliness Security	Usable information IQ dimensions Believability Accessibility Ease of operation Reputation

(a) The 2x2 model from [7].



(b) An example of the benchmark gap analysis from [7].



(c) An example of the role gap analysis from [7].

Figure 2.1: Three simple graphs

The Data Quality categorization model is represented as a rectangle divided into four main parts, and is intended to show what IQ means both from the point of view of the consumers of this information, and from the point of view of the managers. Within the model, the information is inserted in a certain quadrant and the positioning depends on how it is considered: product or service. The other two quadrants take into account the improvements applicable to quality: the evaluation

can take place against a formal specification or from the point of view of customer expectations.

The information quality assessment instrument consists of a simple questionnaire, which has the task of measuring the quality over the dimensions of IQ that are considered important by both consumers and information managers. Most of these dimensions are used to measure quality for each quadrant of the aforementioned rectangular model. This tool can be used to validate the quality of information within organizations, having it filled in by subjects who, more than others, interface with the data.

The last two components of the model, benchmarking gap analysis and the role gap analysis, consist of analysis techniques with the task of interpreting the findings obtained from the questionnaire.

The first analysis compares the result obtained by the organization that is performing it with an organization in the same sector, which is considered to be one of the best in the IQ.

The second, on the other hand, measures the distance between the assessments made by different stakeholders of an information system. This distance is plotted on a graph, as shown in Figure 2.1c. The AIMQ is a framework that relies on subjective measurements only, thru the use of the IQA instrument, a questionnaire including several items that help with measuring data quality. Overall, the AIMQ focuses on prioritizing areas for data quality improvement.

The model reported in figure 2.1a represents an example that extrapolated from the research document. The four quadrants indicate the fundamental aspects of IQ on which to make improvement decisions.

In figure 2.1b we can observe an example of IQ evaluation. the ordinate axis represents the level of quality, which can vary between zero and ten. The abscissa axis represents the percentage of those who answered the questionnaire. The graph shows the results of five companies, where the number one indicates the company considered the best in the IQ. As we can see, there is a substantial gap between the best and the other four companies considered by the authors' example. The elements to be considered during this phase are the position of the gap and their size.

Figure 2.1c is an example of the Role Gap analysis. on the abscissa axis we find the five organizations mentioned previously and, on the ordinate axis, we find the level of data quality, just like in the previous graph. Diamonds represent the average quality level expressed by customers, while squares represent the average quality level according to information systems professionals. Again, the things to be aware of are the actual size of this difference and the direction. If the difference is very large, as we can see for company number five, it means that information systems managers are not at all aware of the problems that consumers of information are facing when they deal with the data. The reverse is identified if it starts from the managers (squares) and ends in the consumers (diamonds), and translates into a

higher quality assessment by the managers than that provided by consumers.

2.1.2 Data Quality Assessment

DQA[8] is a method which describes the principles that can help organizations develop usable data quality metrics. In this article, they describe the subjective and objective assessments of data quality, and present three functional forms for developing objective data quality metrics. The authors present an approach that combines the subjective and objective assessments of data quality, and illustrate how it has been used in practice.

Subjective assessments are the one manifested by the users and generators of the data: for example, we can obtain subjective assessment by asking to complete a questionnaire regarding how the individuals (like stakeholders) feels about the data quality dimensions that we want to analyze.

Objective assessments can be task-independent or task-dependent. Task-independent metrics reflect states of the data without the contextual knowledge of the application, and can be applied to any data set, regardless of the tasks at hand. Task-dependent metrics, which include the organization's business rules, company and government regulations, and constraints provided by the database administrator, are developed in specific application contexts. A comparison of the previously performed objective and subjective measurements where discrepancies are identified is suggested in combination with a root cause analysis. The findings should then be processed by taking the necessary measures for data quality improvement. The DQ dimensions analyzed by the authors of this paper are:

- Accessibility: how easy it is to have access to the data, especially in cases where time is of vital importance.
- Appropriate amount of data: the extent to which the volume of data is appropriate for the task at hand.
- Beliavability: measures the level of credibility and truthfulness of the data.
- Completeness: measures the level of completeness of the data, or to what extent it can be considered free from any deficiency.
- Concise representation: measures the level of compactness in the representation of the data.
- Consistent representation: measures the level of consistency of the data, that is how much the data is represented with the same format.
- Ease of manipulation: measures the level of ease in manipulating the data and how easily it is possible to apply it to different tasks.

- Free-of-error: it measures in what quantity the analyzed data is correct and reliable.
- Interpretability: measures how much the data is expressed in an interpretable way, using the correct symbols and units.
- Objectivity: measure of how objective the data is, or how much it is without prejudice and presented in an impartial way.
- Relevancy: measures the level of applicability of the data with respect to the current task.
- Reputation: how much reputation the data has, taking into account its origin and content.
- Security: the extent to which access to data is restricted appropriately to maintain its security.
- Timeliness: the extent to which the data is sufficiently up-to-date for the task at hand.
- Understandability: the extent to which data is easily comprehended.
- Value/added: the extent to which data is beneficial and provides advantages from its use.

The authors state that when a company tries to calculate and define metrics for the aforementioned data quality dimensions, it essentially always runs into one of the following functional forms:

- Simple Ratio: ration of desired outcomes to total outcomes.
- Min or Max Operation: minimum or maximum value among normalized individual data quality indicator values.
- Weighted Average: Assigning weighting factors to represent the importance of the variables to the evaluation of a dimension.

The DQA measures the stakeholder expectations and defines quantitative metrics, by the functional forms, in order to measure the data quality in an objective way. These functional forms are intended to define a metric by which the various dimensions of data quality can be measured. There are several refinements applicable to these forms, such as adding parameters for sensitivity. Using the simple ratio, the dimensions of free-of-error, completeness, consistency, concise representation, relevancy and ease of manipulation can be calculated.

By free-of-error, the authors mean that the data under consideration is correct. The measure is defined as the division between the number of data that is in an

error state (i.e. that is not considered complete) and the total number of data available. The dimension of completeness can be easily measured by calculating the ratio of the number of incomplete data to the total number of data. Consistency is also measured with a ratio, but this time the dividend is the number of breaches of a given consistency type and the divisor is the total number of consistency checks.

The minimum and maximum operations are considered by the authors of the paper as conservative. This is because they assign an aggregate value to the dimension which is equal to the lowest quality indicator, which is normalized to belong to the range between zero and one. The maximum operation is used if a liberal interpretation is warranted. The maximum operator is used only if a free interpretation of the quality indicators is required. The latter is more useful when dealing with more complex metrics, such as those concerning the dimension of data accessibility and their timeliness. An example of using the minimum metrics is to be found at the dimension concerning the appropriate quantity. Each of these dimensions are then rated on a scale from 0 to 1, and overall believability is then assigned as the minimum value. For example, lets assume that the believability of a data source is rated as 0.6; believability against a common standard is 0.8; and believability based on experience is 0.7. The overall believability rating is then 0.6 (the lowest number). An alternative is to compute the believability as a weighted average of the individual components. A working definition of the appropriate amount of data should reflect the data quantity being neither too little nor too much. Timeliness represents how up-to-date the data is with respect to the task it's used for. For the multivariate case, an alternative to the min operator is a weighted average of variables. If a company has a good understanding of the importance of each variable to the overall evaluation of a dimension, for example, then a weighted average of the variables is appropriate. To insure the rating is normalized, each weighting factor should be between zero and one, and the weighting factors should add to one. Regarding the believability example mentioned earlier, if the company can specify the degree of importance of each of the variables to the overall believability measure, the weighted average may be an appropriate form to use.

What this methodology tries to do is to propose a path, which contains both subjective and objective entities, to be followed in order to be able to answer the following question: How good is a company's data quality? The ultimate goal is to identify data quality problems and mitigate them, so as to improve a company's data status. This road involves the following steps:

- Performing subjective and objective data quality assessments;
- Comparing the results of the assessments, identifying discrepancies, and determining root causes of discrepancies;
- Determining and taking necessary actions for improvement;

2.1.3 Heterogenous Data Quality Methodology for Data Quality

HDQM[9] is a methodology which aims to assess and improve data quality of all types of data managed in an organization. Its main components are: state reconstruction, quantitative evaluation of data quality problems and selection of appropriate improvement activities. HDQM is one of the few methodologies that considers structured, unstructured and semi-structured data.

Structured data consists in a generalization or aggregation of items described by elementary attributes defined within a domain items, like a relational database table. Semi-structured data is characterized by the lack of a rigid, formal structure. Typically, it contains tags or other types of markup to separate textual content from semantic elements, like an XML file. Unstructured data can be found in different forms: from web pages to emails, from blogs to social media posts, etc. 80% of the data we have is known to be unstructured. Regardless of the format used for storing the data, the most common format are textual documents made of sequences of words.

The main DQ dimensions analyzed in this method are accuracy and currency. The accuracy used inside this method is the so called Syntactic accuracy which is measured by means of comparison functions that evaluate the distance between a value v and another value x belonging to the domain, which is considered as the correct representation of the real-life phenomenon that v aims to represent. The currency used is the normalized currency, defined as the ratio between Actual and Optimal currency. Currency is usually defined as the "temporal difference between the date in which data are used and the date in which data have been updated". Therefore, Normalized currency is the ratio between the minimum time span that data have become old (Optimal currency) and the Actual currency of these data. More specifically, Normalized currency concerns how promptly data are updated with respect to how promptly they should be to users needs and the main domain constraints.

HDQM aims to provide indications on the optimal DQ improvement program that an organization should undertake with respect to its peculiar needs and constraints. HDQM consists of three main phases and each of them is composed of a number of steps. In particular, the main phases are:

1. State reconstruction, which aims to reconstruct all the relevant knowledge regarding the organizational units, processes, resources and conceptual entities involved in the organization.

2. Assessment, which aims to obtain a quantitative evaluation of data quality problems. DQ dimensions are measured in order to assess the current level of data quality and to set the new DQ targets that must be reached at the end of the DQ improvement program

3. Improvement, where improvement activities are selected by evaluating their effects in terms of DQ dimensions/cost ratio.

State reconstruction is a complex phase that encompasses a preliminary task of problem identification and three tasks of reconstruction. The problem identification task aims to identify the most relevant data quality problems as they are perceived by all the actors involved in the business processes. This means to focus on the most important data only, the so called master data and on those data that are involved in some organizational shortcoming. The subjective perception given by internal and external actors is quantified by means of focused interviews and survey questionnaires.

The assessment phase of the HDQM consists of two steps. Firstly it starts by ranking the resources that have been identified during the previous step, in order to establish feasibility and risk for the subsequent improvement phase. Secondly, the actual quantitative measurement of data quality is performed. In this assessment, the relevant dimensions are measured by applying appropriate metrics. The data quality improvement phase starts with an analysis of data quality requirements which in this model is performed using a process-oriented approach.

It follows a selection of activities for data quality improvement. This is done by using both a data-driven and process-driven strategy in order to produce a Resource/Improvement Activity matrix, starting from a data quality requirement definition, passing through a selection of the data quality improvement activity and ending with the selection and evaluation of the most adequate improvement processes. Finally, an improvement process is chosen and evaluated based on this matrix. The selected process should incorporate all relevant dimensions and resources.

The HDQM provides a novel approach to data quality cost considerations. Apart from stating quantitative methods to cost-benefit analysis, the model proposes to qualitatively compare costs with benefits. First, a Resource/Activity matrix is used to identify candidate improvement processes followed by an evaluation of costs for each of the candidate processes. The qualitative approach consists of categorizing costs (very low, low, medium, high, very high) and subsequently, comparing the values along with their effects on dimensions (data quality dimension/cost ratio) in order to find the appropriate improvement process.

2.1.4 Task Based Data Quality

TBDQ[10] is a method which uses subjective and/or objective measurements to assess DQ. The method attempts to improve DQ by analyzing and modifying organizational processes (which are considered as a sequence of tasks) which potentially create DQ problems. Is a process-driven method. Re-engineering the current process in an organization could prove very expensive hence, modifying the processes in TBDQ does not mean altering the current tasks, but to add new improving tasks

to counter the effects of risky tasks. The main components are: planning and evaluating assessment, evolution and execution of improvement. The DQ dimensions treated are: accuracy, completeness, consistency and timeliness.

The TBDQ performs an initial assessment by means of survey questionnaires followed by the objective assessment using metrics such as a simple ratio. The assessment phase of the TBDQ consists of two steps. Firstly, the goals and scope of data quality for the business are defined in the planning phase. This includes specifying a minimum level of data quality, defining and assigning weight to the dimensions and identifying those tasks within the process that can lead to data quality issues. In the subsequent evaluation step, weights are assigned to the data quality problems by using a pair-wise comparison matrix of the analytical hierarchical process[11]. A questionnaire-based approach is suggested for subjective assessments, combined with a simple ratio metric for objective assessments of data quality. Based on this, the data quality issues receive value. It is also suggested that subjective and objective assessment results are compared as a form of validation

The improvement phase according to the TBDQ consists of two steps. Firstly, prioritization of data units is suggested and data improvement tasks such as data correction or notification designed and proposed in the evolution step. The decision on a suitable task is based on an "award system" comparing the different tasks based on execution costs and level of improvements. In the execution steps, the tasks and modified process units are performed. In addition, the execution is analyzed in terms of scope and achieved amount of improvement. A cost-benefit analysis is considered from a qualitative perspective. The model uses the award system in order to choose the improvement tasks. The processes are evaluated on the basis of their execution costs and the level of improvement. This is done by means of a Time-Driven Activity-Based Costing as proposed in[12]. The cost considerations are made towards budget constraints, non-quality costs, improvement costs and cost-benefit analysis.

2.2 Technologies

In this section we will talk about all the technologies used to achieve the goal set by this thesis, reporting the key concepts and justifying the choices made.

2.2.1 Scala

Scala[1] is a general purpose programming language. It was created and developed by Martin Odersky and was officially released on January 20, 2004. The reasons for the birth of this language are to be found in the need to have a better version of Java.

[1] https://www.scala-lang.org/

This led its creator to make a new language, dealing with the missing opportunities of Java but which had the same basic idea, such as Java Virtual Machines. During compilation, Scala file translates to Java byte code and runs on JVM (Java Virtual machine). Scala was designed to be both object-oriented and functional. It is a pure object-oriented language in the sense that every value is an object and functional language in the sense that every function is a value. The name of Scala is derived from word scalable which means it can grow with the demand of users. Is statically typed but has a lightweight syntax, is fully inter-operable with Java, so that libraries written in either language may be referenced directly in Scala or Java code, and is a perfect fit for Domain Specific Language: is a programming language with a higher level of abstraction optimized for a specific class of problems[13]. Scala has many features of functional programming languages like Scheme, Standard ML and Haskell, including currying, immutability, lazy evaluation, and pattern matching.

The choice to use Scala was trivial, since all the source code of Apache Spark™, the tool protagonist of this thesis, has been written in Scala.

2.2.2 Apache Hadoop

Apache Hadoop[2][14] is a collection of open-source software utilities which have the purpose of simplifying the use of a cluster in order to solve problems concerning the analysis and processing of huge amounts of data, exploiting the enormous computational capacity made available by the cluster itself. It also contains a framework dedicated to distributed storage systems which provides process support for big data, applying the MapReduce[15] programming model. The concept behind the design of the individual components of Hadoop is that hardware failure is the practice and must be automatically handled by the system itself, without diverting the user's attention and efforts.

Currently, Hadoop is a top-level project of Apache, used and maintained by a big community of contributors and users. Published with the open-source license Apache 2.0, it is used by the largest computer companies including Yahoo!, Facebook, Adobe, EBay, IBM, LinkedIn and Twitter. Before Hadoop, data processing was carried out by High Performance computing and Grid Computing systems. However, Hadoop offers a set of easy-to-use libraries and takes advantage of data replication on individual nodes to improve data access times, avoiding, if possible, to transfer them over the network. The idea behind Hadoop was born from the need to find a new methodology to be able to process huge amounts of data quickly and effectively. Hadoop queries often require reading a large amount of data (Gigabytes or even Terabytes), unlike queries on traditional systems where it is often required to read a single or a few records in a table.

[2] https://hadoop.apache.org/

The core of Apache Hadoop is composed of a part that deals with storage, commonly called Hadoop Distributed File System (HDFS), and a part that deals with processing, following the MapReduce paradigm. The storage system divides the largest files into fixed-size blocks and distributes them to all the nodes that make up the cluster. Subsequently, the code written by the user is packaged and also sent on the aforementioned nodes, so as to be able to process the data in a parallelizable way: this method takes advantage of the concept of data location, where the individual nodes of the cluster applies the user's code to those blocks that are in their possession. This allows an acceleration in data processing, also due to the efficiency obtained by exploiting this so-called locality. All these expedients allows us to obtain the results in such short times that is not remotely comparable with what we would have from a conventional architecture composed of a single mainframe.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities that supports other Hadoop modules. it is the main core of the Apache framework;
- Hadoop Distributed File System (HDFS) – a distributed file-system that stores the data used in the processing part, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – a platform responsible for managing computing resources in the clusters;
- Hadoop MapReduce – an implementation of the MapReduce programming model for large-scale data processing.
- Hadoop Ozone – an object store for Hadoop;

The Hadoop system is a highly reliable one, as it can run on commodity hardware clusters and has been designed to continue to function even if one or more cluster nodes fail. The system is highly scalable as nodes can be added or removed to the cluster on necessity. The ability to exploit distributed architectures to its own advantage is absolutely not to be underestimated and, Hadoop allows us to focus on the data processing part rather than the underlying structure.

2.2.3 Apache Kafka®

Apache Kafka®[3][16] is a scalable, fault-tolerant, and highly available distributed streaming platform that can be used to store and process data streams. The platform aims to provide its users with high troughput and low latency guarantees in

[3] https://kafka.apache.org/

order to manage data streams in real time. It consists of three main elements: the cluster, the Streams API and the Connect API. The first element has the task of storing the data flows, which are represented within Kafka as sequences of messages continuously produced by applications and sequentially consumed by others. The Connect API is used to insert data into Kafka and extract it to external systems, such as HDFS, databases, etc. The Streams API, on the other hand, has the task of allowing developers to create complex processing structures to read input streams from Kafka. Each single message is saved within Kafka with a key-value pair: the data can be divided into different partitions within multiple topics. Within each of these partitions, messages are sorted by their offset, which represents the position of the message within the partition itself. The messages are then indexed and stored with a timestamp. The processes that have the task of reading these messages are called consumers. A Kafka cluster is composed of several servers, which take the name of brokers, and the partitions, previously mentioned, are distributed on all the nodes of the cluster: this allows Kafka to deliver an enormous amount of data flows guaranteeing its users fault tolerancy.

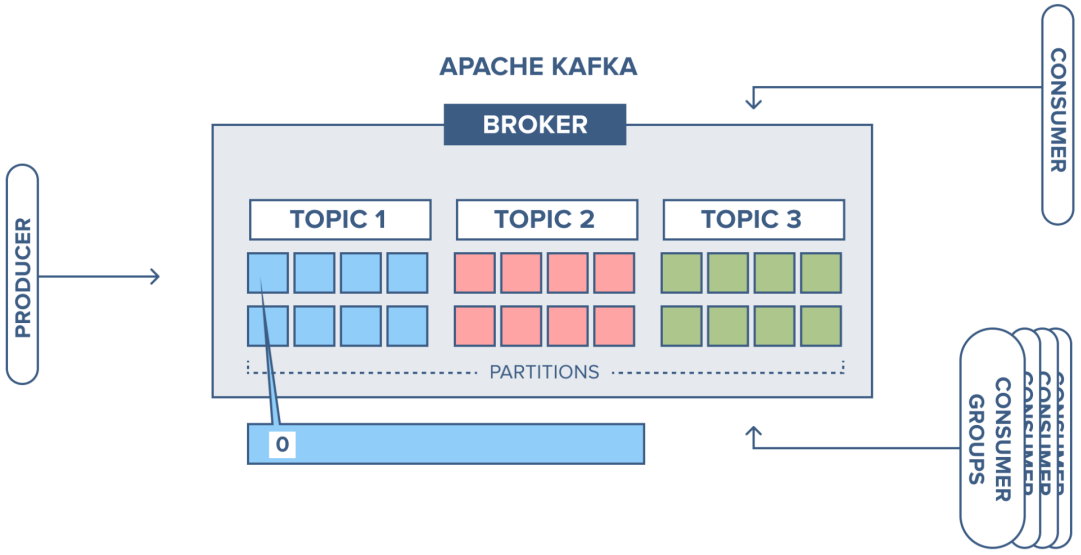


Figure 2.2: The architecture of Apache Kafka®

2.2.4 Apache Spark

Apache Spark[4] is a open-source unified analytics engine for large-scale data processing. Spark provides its users with a series of APIs, which allows them to program clusters while maintaining guarantees on the tolerance of failures and exploiting the parallelism of data. The project has its origins at the University of California, as it was the subject of academic research and, subsequently, it was donated to the Apache foundation, with the aim of making it open-source and accessible. The motivation behind the born of Apache Spark was that using MapReduce for complex iterative jobs or multiple jobs on the same data involves lots of disk I/O and since the cost of main memory was decreasing, the solution proposed by Spark was that of keeping more data in main memory instead on disks: data is read only once from the external storage and is shared across the main memory of each server which is part of the cluster, thus improving the speed of the jobs.

Spark is based on a basic component (Spark Core) that is exploited by all the high-level data analytics components. Consequently, when the efficiency of the core component is increased also the efficiency of the other high-level components increases.

Spark allows its users to implement both iterative programs and perform exploratory data analysis. As for the latency, the comparison with MapReduce does not hold up either, as the difference lies within several orders of magnitude. In order to function, Apache Spark needs two basic components: a cluster manager and a distributed storage system. As for the former, support for YARN, Mesos and Kubernetes is already integrated, along with standalone mode, which is Spark's native cluster. For the latter, Spark can tie into a large set of systems, such as Alluxio, HDFS, Amazon S3, or even a custom solution, just to name a few. Often, for testing and debugging purposes only, Spark can be run locally: in this case, the executors are the CPU cores it is running on and the distributed system is that of the local machine.

The structure of a Spark program is composed of a Driver, which contains the main method, defines the flow of the application and accesses Spark through the use of the SparkContext object (starting from Spark 2.0, the entry point has become the SparkSession object if we want to take advantages of the abstractions offered by DataFrames and DataSets). The Driver also defines the Resilient Distributed Datasets (RDDs), which are allocated to the cluster nodes, and the parallel operations to be applied to the latter are invoked.

These parallel operations applicable on RDDs can be divided into two main categories: Transformations and Actions. A transformation is applied to an RDD and produces a new RDD. An action is a function that is applied on an RDD and produces a value. Transformations are called "lazy", as they are only performed if

[4] https://spark.apache.org/

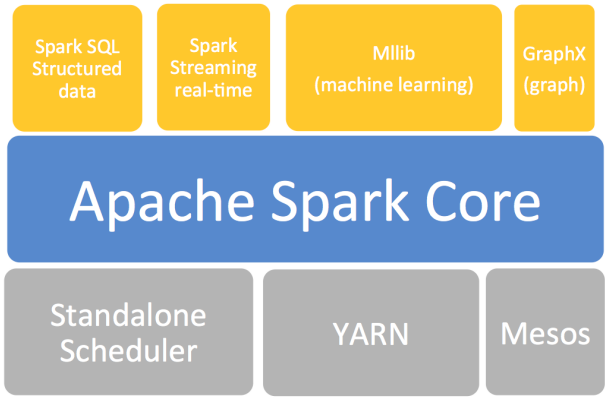


Figure 2.3: The full Apache Spark™ stack.

an action is called on the resulting RDD. This means that if no action is called on the RDD, all transformations will never be performed, as the developer does not require explicitly the framework to produce a result.

The SparkContext (or SparkSession) object allows us to create RDDs and submit executors that perform specific operations in parallel on the RDDs. The worker nodes of the cluster execute the application through the executors: each executor runs the operations specified in the Driver on its RDD partition. The great advantage of Spark is that it also allows local execution, on a single node or computer: threads are used to parallelize and run the application on the RDDs. This is very useful for developing and testing applications before deploying them on a cluster.

Before continuing with the definitions, it is necessary to report the terminology used in the Spark application development environment.

- Application: we refer to a program created by the user based on Spark. It is composed of a driver and executors, which belong to the cluster.
- Application jar: a jar[5] which contains the Spark application.
- Driver Program: it is considered as the entry point of the Spark program: here we find all the functions that have the purpose of managing and transforming the RDDs, as well as creating the SparkContext.
- Cluster Manager: often an external service, such as YARN, which has the task of managing the resources located on the cluster.

[5] https://docs.oracle.com/javase/tutorial/deployment/jar/index.html

- Deploy Mode: it is used to define where the driver will actually run.
  - Cluster mode - the driver runs within the cluster itself, together with the various executors.
  - Client mode - the driver is not inside the cluster.
- Worker Node: this term refers to any node in the cluster capable of executing code within the cluster.
- Executor: it is a cluster node that has the task of executing the portion of code it receives and applying it to the data block in its possession.
- Task: with task we indicate the smallest possible unit of work within spark: each executor has the purpose of executing one task at a time.
- Stage: with stage we indicate a grouping of tasks that are directly dependent on each other, that is, we cannot divide them into separate entities.
- Job: it consists of a series of stages and represents the largest unit of work within Spark. Usually a job corresponds to an action on RDDs and is made up of a series of stages.

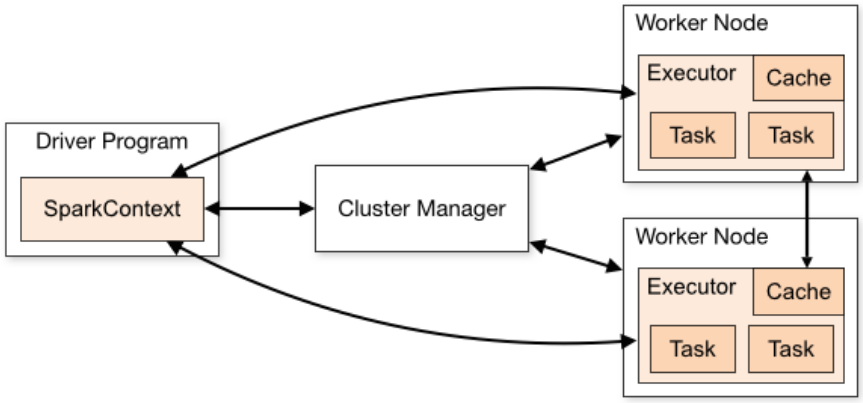


Figure 2.4: architecture overview of a spark cluster launched in cluster mode.

RDD Abstraction

The data is represented as resilient distributed dataset[17], a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. RDDs are partitioned collections of objects spreaded across the nodes of a cluster, stored in main memory and only if they do not fit, stored on local

disks. A Spark program is written in terms of operations or these RDDs, that are built and manipulated thru a set of parallel transformations and actions. RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other RDDs (its lineage) to compute its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure. For each RDD, Spark maintains a set of metadata, known as Directed Acyclic Graph (DAG), which represents its lineage: this is used for two main reasons: first, RDD are "lazy" which means that the chain of transformations are applied on the input RDD only when an action is required by the programmer (like saving the RDD on external HDFS-compliant); second, since Spark is built on the same idea of Hadoop (hardware fault is a common thing), when an RDD which was on a node that failed (or maybe was too slow and the calculation was moved on another node), all the necessary informations for the recreation of the RDD are inside the DAG. Another key factor of RDD is that they are immutable: once an RDD is created, it cannot be modified, only a new RDD can be obtained by applying transformations and actions that are exposed through Spark. The lazy evaluation brings a lot of benefits, like:

- Increased manageability: by lazy evaluation, users can organize their Apache Spark program into smaller operations, reducing the number of passes on data by grouping operations.
- Computation savings and increased speed: since only necessary values get compute, it saves the trip between driver and cluster, thus speeding up the process.
- Reduced Complexity: the two main complexities of any operation are time and space. Using Apache Spark lazy evaluation we can overcome both, since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.

Recently, a further abstraction was conceived, trying to overcome that provided by RDDs. In the first version of Spark, RDDs were the main API but, starting from the second version, the Datasets API was recommended as the primary replacement, without however deprecating that of RDDs, in fact even now, with the third iteration of the framework, RDD abstraction is present as a lower layer of the entire API dedicated to Datasets.

Spark Core

Contains the basic functionalities of Spark exploited by all components. It provides distributed task dispatching, memory management, fault recovery, scheduling and basic I/O functionalities, all exposed through an Application Programming Interface. This interface mimics the functional programming paradigm: a driver program

has the task of evoking all the operations applied on the RDDs in parallel, sending functions to Spark, which has the task of scheduling them and executing them in parallel on the cluster that has available. These operations have an RDD as input and produce a new RDD as an output. Fault tolerance guarantees are maintained by keeping track of all the transformations that an RDD undergoes, from start to finish. This mapping is called lineage, and is used in case of failures to reconstruct the RDD to its pre-failure state. An RDD can contain any type of Scala, Java, or Python object.

Spark SQL

SparkSQL[18] is the Spark component used for structured data processing. It takes advantage of the abstraction made available by the Dataset API and behaves as if it were an SQL engine. It integrates support for the Hive query language and, of course, can interface with tables on it. It allows us to have more detailed information on the data structure and to execute SQL queries. It also features further optimization, which has the task of improving and optimizing all the queries applied to both datasets and RDD, called Catalyst.

Datasets are a distributed collection of structured data which provide both the benefits of RDDs and the optimization of the execution of the SparkSQL engine, which uses information on the nature and structure of the data to calculate the best execution plan before running the actual code. The DataFrame are a particular case of Datasets that are organized in named columns, conceptually they can be represented as relational database tables. A DataFrame is a Dataset of Row objects. A DataFrame can be built from different sources, such as structured textual data files (CSV or JSON for example), existing RDDs, Hive tables or external relational databases. Datasets are generalized structures with respect to DataFrames: they are collections of objects associated with the same class, whose schema corresponds with the attributes of the class itself. They are very similar to RDDs but contain advanced optimizations, such as Encoders, which have the task of serializing objects to be able to transmit them over the network to the executors. These Encoders allow Spark to perform operations such as filtering, sorting and hashing without having to deserialize the object. A dataframe can be considered as the natural evolution of the RDD, since we have two optimizations already present for RDDs but improved and adapted to the context of the dataframe:

- Custom Memory Management: takes the name of Tungsten Project and allows us to save a lot of memory since storage takes place in memory outside the heap and is in binary format. This allows us to avoid the overhead produced by having a garbage collector and avoid java serialization.
- Optimized Execution plan: before executing any query, an execution plan is created and optimized for the best possible execution.

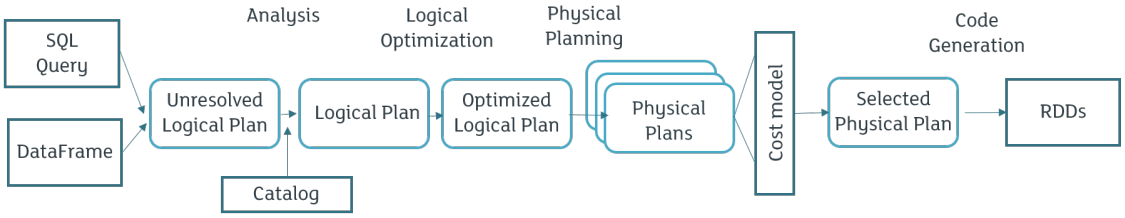


Figure 2.5: The Catalyst query optimizer pipeline.

Spark Streaming

The birth of Apache Spark Streaming was more a necessity than anything else, as the production of data in almost instant times required, once again, the need to create a new method to be able to analyze them, also respecting the requirements in terms of latency and processing speed that entails. Just think of the fact that in 2014[6], 500 million tweets were sent daily.

Spark Streaming was added as part of the Apache Spark suite in 2013 to provide scalable, fault-tolerant, and high-throughput stream processing to handle real-time data streams. It allows us to connect to different sources that produce data streams, such as Kafka, Amazon Kinesis or Apache Flume and the processing itself takes place through the use of complex algorithms, created with the help of high-level functions, such as those made available by Scala. Finally, the results can be sent to external file-systems and databases. The nature of the birth of Spark Streaming is to be found within the various problems that arose when Hadoop was used to manage and process streaming data: the batch approach of Hadoop is not recommended, as it involves a great latency that it is not at all suitable for real-time processing. Spark Streaming allows us to overcome these barriers, offering support for both batch and streaming processes, thus conveying a single platform with common tools to problems of a completely different nature. This feature has undoubtedly accelerated the spread and adoption of Spark Streaming compared to its competitors: often, to add streaming support, a programmer had to make simple and small changes to analysis pipelines already oriented to batch processing. The integration does not stop at the simple modification, in fact it is also possible to combine data of a static nature with data of a streaming nature, all while maintaining the optimizations already provided by the Spark platform, such as the Apache Spark SQL component. The main abstraction at stake is the Discretized Stream[7], which is an architecture that takes advantage of the rich library and the guarantees of fault tolerance already made available by Spark.

[6] https://blog.twitter.com/official/en_us/a/2014/the-2014-yearontwitter.html

[7] usually abbreviated with DStream

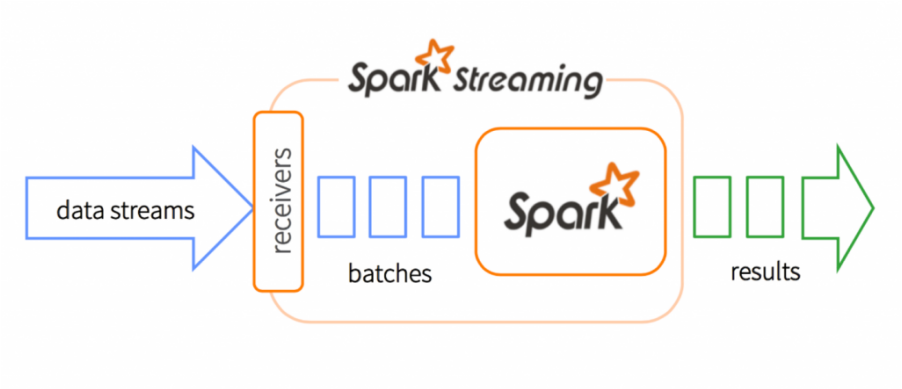


Figure 2.6: High-level architecture of how Apache Streaming works

All the optimizations mentioned above, such as data location and task allocation based on available resources, are still being exploited thanks to the conversion done by DStreams. All this allows for an excellent load balancing and a really fast recovery from machine failures. DStreams are the main abstraction of Apache Streaming: they are a continuous sequence of RDDs (of the same type) representing a continuous stream of data. DStreams can either be created from live data (such as data from HDFS, Kafka or Flume) or can be generated by transforming existing DStreams using operations such as map, window and reduceByKeyAndWindow. While a Spark Streaming program is running, each DStream periodically generates a RDD. The main idea behind DStreams is to represent an infinite stream of data as a sequence of finite batches, which can be treated as if they were static data frames. Therefore it is possible to use all the implementations and optimizations already present for analyzing continuous streams of data.

Apache Spark Structured Streaming

Often referred to as Apache Spark Streaming 2.0, Structured Streaming[8] can be seen as the natural evolution of how Spark handles streaming data. Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. Internally, Structured Streaming queries are processed using a micro batch, which is composed of an engine that processes a defined series of small jobs sequentially, in such a way as to ensure latencies of the order of 100 milliseconds and providing guarantees of tolerance to failures. With the arrival of version 2.3, a new analysis engine has been introduced, still in the planning phase,

[8] https://spark.apache.org/docs/2.4.7/structured-streaming-programming-guide.html

which can even guarantee latencies in the order of milliseconds, but guaranteeing at-least-once processing. The main difference with Spark Streaming (often also called Spark Streaming Legacy) is that there is no batch concept here: with each trigger, new data arrives, which can be continually hung on an infinite table.

The semantic guarantee of the end-to-end exactly-once was the basis on which the Structured Streaming designers focused more: in order to achieve this, every single component was designed in such a way to be able to trace in a reliable and accurate the entire progress of the processing, in order to manage any type of failure by starting the process from the exact point of interruption or restarting it entirely. Each source is supposed to have its own internal offset, such as Kafka does, in order to allow tracking of the reading within the stream. The engine uses ad-hoc techniques, such as checkpointing and write-ahead logs to record the offset of the data processed in each single trigger.

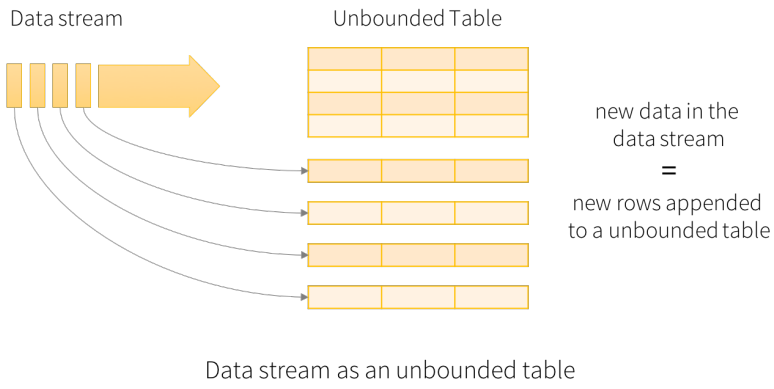


Figure 2.7: Visual representation of how the result of a query in Structured Streaming looks like [19].

To better understand and appreciate these properties offered by Spark Structured Streaming, allow us to do a little digression on the meaning of the concepts aforementioned, such as At-least-once, At-most-once and Exactly-once: lets suppose we have, upstream, a source that produces a series of messages in real time and a downstream application that has to perform some kind of calculations on these messages.

- At-least-once: the guarantee is given to process the data at least once by all the entities forming the process. This translates into the fact that a data can be re transmitted from the source in the event that it is lost before being processed. Since we have this re transmission, it could happen that the same data is processed more than once.

- At-most-once: is translated into a best-effort. The processing of a data is guaranteed at least once by all the entities of the process, i.e. we do not have any re transmission in case of failures.
- Exactly-once: even if the source retries sending a message, it leads to the message being delivered exactly once to the streaming application. Exactly-once semantics is the most desirable guarantee, but also a poorly understood one. This is because it requires a cooperation between the source system itself and the application consuming the messages.

As for the input data, Structured Streaming has a few types of sinks already implemented:

- File Source: reads files written in a directory as a stream of data. The formats that are supported are simple TXT files, parquet files, JSON files, CSV and ORC files. It is necessary to put every single file inside the target folder, which in most file systems, can be achieved by file move operations, in order to trigger the read from Spark Structured Streaming. This source is also fault-tolerant, which means that once the analysis has begun, in the event that any problem arises that interrupts the execution of the program, Spark is able to resume exactly where it left off.
- Kafka source: reads data from Kafka. It's compatible with Kafka broker versions 0.10.0 or higher. This is also fault-tolerant, mainly due to the nature of kafka itself.
- Socket source: reads UTF-8 text data from a socket connection. Note that, as mentioned by the official documentation, this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.
- Rate source: generates data at the specified number of rows per second, each output row contains a timestamp and value. This source is intended for testing and bench-marking. This source is fault-tolerant.

In the context of streaming, the time event is certainly essential for the correct functioning of an application that analyzes data in real time. In these cases, what matters most is the time when the event was generated (such as a log file to check the status of a server) rather than when this information reached Spark. For this reason, Spark Structured Streaming offers the possibility to manage the arrival of delayed data, through the use of the watermark. This function allows us to manage the data arriving late to our pipeline. By default, Spark updates all the data (also the late ones) without any defined policy. Since by the very nature of how the results of a streaming query are calculated, Spark needs to keep a series of extra information for each input data, in order to create and maintain aggregates. Through the use of watermarking we can reduce the burden of this

overhead. By applying the watermark, we are going to communicate to Spark to keep the intermediate states of the data for a fixed time (watermark parameter), in order to eliminate them once this period of time is over.

Window aggregation operations based on the time event of the input data are easily obtainable and often used a lot in Spark Structured Streaming. Usually, the data is grouped in order to calculate different aggregates on them, obtaining fundamental information to make the best business decisions. The introduction of the watermark in the equation simplifies and greatly optimizes the analysis of streaming data and allows us to have queries that run for entire weeks, if not even months.

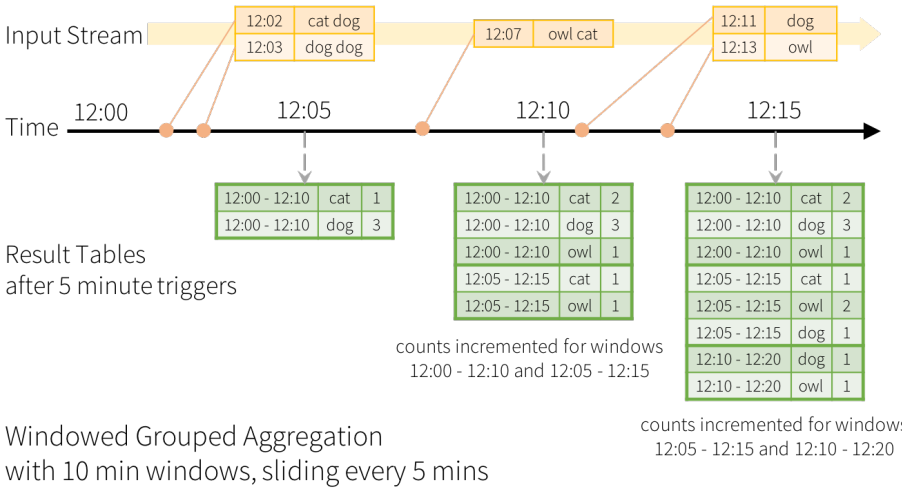


Figure 2.8: Visual representation of how window aggregations work in Spark Structured Streaming [20].

Another useful function, made available by Spark Structured Streaming, is the possibility of de-duplicating the data we are dealing with. To do this, we need to have a unique identifier present within the data, more precisely a column that plays this role. Spark will keep a small internal state to be able to delete all that data we have already encountered. Here too, we have the option of using watermarking to eliminate those aggregate states that have exceeded a certain threshold. In case we don't decide to use the watermark, the aggregate status will continue to increment, as Spark needs this data to be able to distinguish new data from old data.

Once we have defined all our application logic on what we want to do with the data we have taken as input, we must go and define how to extract the results we have obtained. First we need to specify the details about the output, such as the data format, location etc. Next we also need to specify the output mode, the query

name (optional), the trigger interval (also optional) and the checkpoint location. The latter is used for output sinks that can guarantee end-to-end fault-tolerance. Output mode is how the query result will be saved. There are three types of output modes and they are:

- Append: as the word itself says, in this mode only the new rows that were generated by the last trigger will be added. This method is supported only for those type of queries that do not modify already produced rows. This is also the default way.

- Complete: the entire result table will be output after every single trigger. This method is supported for aggregation queries.

- Update: in this mode, only the rows that have changed since the last trigger will be outputted by the application.

2.2.5 Apache Hive

Apache Hive[21][22] is an open-source relational database system for analytic big-data workloads. Hive focused mainly on Extract-Transform-Load (ETL) or batch reporting workloads that consisted of reading huge amounts of data, executing transformations over that data (e.g., data wrangling, consolidation, aggregation) and finally loading the output into other systems that were used for further analysis. Hive structures data into the well-understood database concepts like tables, columns, rows, and partitions. It supports all the major primitive types – integers, floats, doubles and strings , as well as complex types such as maps, lists and structs. The latter can be nested arbitrarily to construct more complex types. In addition, Hive allows users to extend the system with their own types and functions. The query language is very similar to SQL and therefore can be easily understood by anyone familiar with SQL. While the tables are logical data units in Hive, table metadata associates the data in a table to HDFS directories. The primary data units and their mappings in the HDFS name space are as follows:

- Tables: a table is stored in a directory in HDFS.

- Partitions: a partition of the table is stored in a subdirectory within a table's directory.

- Buckets: a bucket is stored in a file within the partition's or table's directory depending on whether the table is a partitioned table or not.

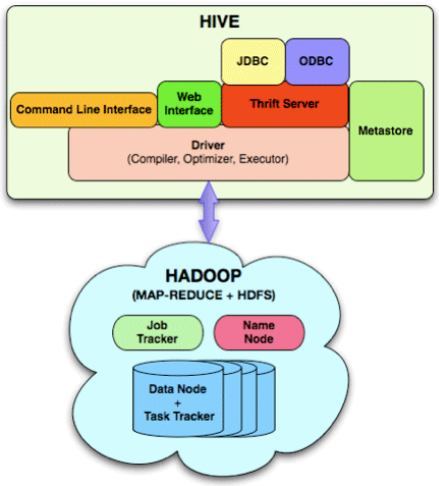


Figure 2.9: Hive System Architecture

2.2.6 Apache HBase

HBase[23][24] is the abbreviation of Hadoop Database and it runs on top of the Hadoop as a scalable big data store system. Take advantage of of Hadoop's distributed file system and MapReduce model by default. It is referred as the columnar database because, in contrast to a relational database which stores data in rows, HBase stores data in columns. HBase is modeled after the Google's BigTable project, so it provides distributed data storage capabilities like the BigTable[25] on HDFS. Some of the key features of HBase are listed below:

- Horizontally scalable.
- Fault tolerant storage capability for sparse data.
- Supports parallel processing, HDFS and MapReduce.
- High adaptable data model.
- Ability to host large tables.
- Real-time lookups.
- Automatic load balancing.
- Supports block cache.
- JAVA API for clients.

2.2.7 PostgreSQL

PostgreSQL[9] is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986. The initial concepts for the system were presented in[26], and the definition of the initial data model appeared in[27]. The design of the rule system at that time was described in[28]. The rationale and architecture of the storage manager were detailed in[29].

[9] https://www.postgresql.org/

Chapter 3

Main core

In this chapter we will report what was the core of this thesis. First of all we will explain how the framework developed by Agilelab currently works, highlighting the main components and their functions. Subsequently we will go to expose what have been the changes applied to allow the aforementioned framework to support streaming data.

3.1 Data-Quality Framework

DQ is a framework developed by Agilelab and published under the GNU GPLv3[1] license as an open-source project on GitHub[2]. The framework was created in order to create parallel and distributed quality checks in big data environments. It can be used to calculate metrics and perform checks on structured and unstructured data and is based entirely on Spark. Compared to other data quality products, this framework performs checks at a very low level of the data, without applying or exploiting any kind of SQL abstraction. The main purpose of this framework is to apply a series of calculations and checks on large amounts of data in a single pass, without having to repeatedly analyze the input data, since very often, this phase is only the beginning of a pipeline much more complex that has the task of performing different types of calculations and transformations on the aforementioned data, in order to apply and extract business strategies and decisions. A typical framework execution flow consists in loading various sources on which we want to carry out data quality analyzes, calculate a series of metrics (3.1.2) and apply a series of checks (3.1.3) on the latter to verify the values obtained. Finally, the framework loads the results (metrics, composed metrics, column checks, file checks, load checks) on an

[1] https://www.gnu.org/licenses/lgpl-3.0.en.html

[2] https://github.com/agile-lab-dev/DataQuality

external database and also offers the possibility to save them on the local filesystem (3.1.4).

Doing this in a single pass is a fairly fundamental concept, as data is very often in raw form and available in huge quantities. This framework aims to analyze the various dimensions that characterize data qualities, reporting objective values on which decisions can be made subsequently.

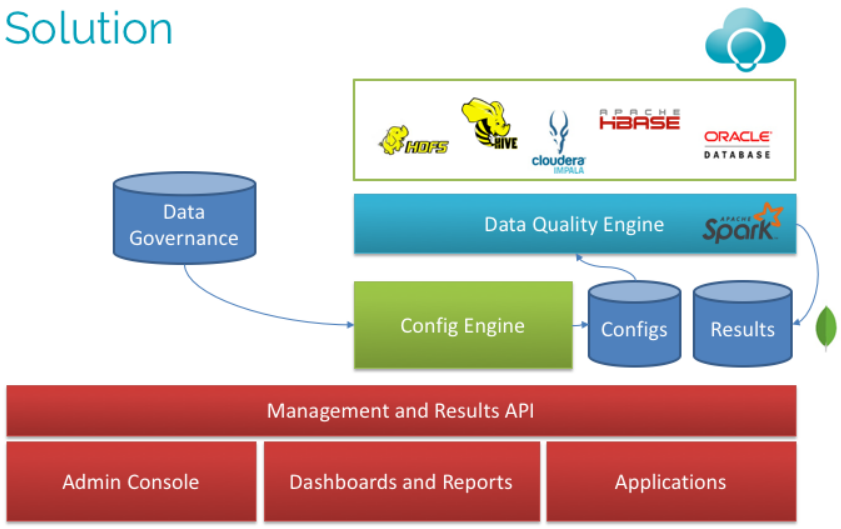


Figure 3.1: The architecture of the Data Quality framework proposed by Agilelab

In the Figure 3.1 we can see how the framework is structured from an architectural point of view. The data governance part indicates all those business decisions and strategies that will be applied to the data being analyzed through the use of a configuration engine, which has the purpose of conducting the analysis through entities that will be defined later in this chapter. The latter will produce a series of configuration files that will start the analysis by the framework, which in turn will produce results that can be saved on an external database for further analysis.

The Data Quality Engine block can subsequently be divided into five main subcomponents, which we will analyze in detail in the following sub-chapters. The Data Quality framework is split in tow main modules:

- Config engine: a web application that helps writing data quality configuration files and validating them.
- Core engine: the main module of the data quality framework, which runs the Spark Application, calculates all that is defined in the configuration files. This component can be ran locally on a single machine or in client mode on a cluster manager (YARN).

All the main logic of the program is defined through the use of Typesafe[3] configuration files. In order to successfully launch the application, we need to create and fill two Typesafe configuration files: an application configuration and a run configuration.

In the first file we are going to specify the name of the application and the version of execution, the paths where we are going to save the various types of results (such as checks, metrics, virtual sources, etc.). Furthermore, the type and main parameters of the external database on which the results obtained are to be saved must also be specified here. Postgres, SQLite and Oracle are currently supported. In addition, we can specify an email address to send notifications and error reports, in case we cannot wait for the execution to finish and we need to know when this happens.

In the second configuration file, we must specify all the sources that will be the protagonist of our analysis and all the metrics and checks that we want to be performed on the aforementioned elements. The entities that we can configure are the following:

- Sources: are considered as the entry point to the application. They represent files and tables to be processed inside the framework itself.
- Virtual sources: leverages plain SQL queries in order to apply transformations on the defined sources.
- Metrics: metrics are formulas that can be applied to a table or a column. For each workflow we can define a set of metrics to process in one-pass run.
- Composed metrics: additional metrics that are made of other base metrics.
- Checks: checks are the control unit of the application, can be applied to metric results in order to check if it fits defined user-constraint (greater/lesser than or equal to some constant value or other metric result). Returns "Boolean" values, which can be easily evaluated.
- Targets: set of files and alarms to which will be passed results of previous modules in order to save or/and notify by email.
- Postprocessing: set of transformations to finalize any results and prepare reports.

3.1.1 Source

In the Data Quality batch job the workflow specification is defined by a sequence of steps, the first of which is to define the data sources.

[3] https://github.com/lightbend/config

```
Sources: [
  {
    id        = "GOT_B"
    type      = "HDFS"
    path      = "./docs/examples/data/battles.csv"
    delimiter = ","
    header    = true
    fileType  = "csv"
    keyFields = ["name","year","defender_king"]
  },
...
```

Listing 3.1: Example of a Typesafe configuration for an HDFS-compliant file source.

The listing 3.1 contains the entity that has to be written in the configuration file in order to define a source. In this listing, we are defining an HDFS-compliant source, labelled with GOT_B and which is found at the path in a CSV format.

There are two macro data source types, **SOURCE** (listing 3.1) and **VIRTUAL SOURCE** (listing 3.2). For the first type, the framework supports Hadoop Distributed File System with formats CSV (delimiters must be provided), AVRO (schema must be provided), Parquet (schema is inferred) and fixed length formats (schema must be provided explicitly). Support is also provided for relational database tables, such as Oracle, Postgres and SQLite. there is also support for sources from Hive and HBase.

The virtual sources are generated starting from conventional sources to which SQL-like operations are applied, actually creating new sources, which will be treated during the course of the application like the "real" sources. Taking as example the listing 3.2, that virtual source is created from the join of two sources: "GOT_B" and "GOT_D". The type field is used by the application to know ho to treat this virtual source and the sql field contains the query used to perform the join. These SQL-like operations are as follows:

- Filter-SQL: this label tells the framework to apply a sql query to a parent source and use the result as a virtual source.
- Join-SQL: perform a full-outer-join between the given sources and related columns.
- Join: perform any other supported join between the given sources and related columns.

```
VirtualSources: [
  {
    id             = "DFB"
```

```
4       type           = "JOIN-SQL"
5       parentSources  = ["GOT_B","GOT_D"],
6       sql            = "select␣avg(length(d.Name))␣as␣
           NAMES_LENGTH␣from␣GOT_D␣as␣d␣join␣GOT_B␣as␣b␣on␣
           d.Allegiances=b.attacker_1"
7       save           = true
8     },
9   ...
```

Listing 3.2: Example of a Typesafe configuration for creating virtual sources.

In this section we must provide the information necessary for the application so that it can correctly load the input data. Among the data that we must provide in the configuration file (as shown by listing 3.1) we find the following:

- id: this is the identifier with which the program will refer to this source.
- path: the full path to the file location in the filesystem.
- fileType: currently, the framework supports reading CSV, Parquet, Avro and fixed length files.
- header: boolean value to indicate if the file has the header or not. It is used to ignore the first line (in the case of a CSV file for example) and extract the schema starting from the fields present there.
- date: this parameter can be optional, as if absent[4] it is taken from the application configuration file.
- delimiter: the separator character between two values on the same line is highlighted. If absent, the default "," is used.
- quote: it is used to indicate the possible quotation character, if the values are enclosed within two quotation marks or other characters; If it is absent, the default one is used.
- escape: indicates the character to be used as escape; If it is absent, the default one is used.
- schema: field used to define the schema that we want to use to handle this file. In case both the schema and the header are defined, an illegal argument exception is thrown.

[4]Variables that may or may not have a value are identified in Scala with the Option type.

- keyfields: This field is only for the final part of the pipeline: here we indicate which columns of the file (and relative values) we want to keep when we go to save this source thru the postprocessing and target phases.

The flow begins with the analysis of the configuration file, from which the parameters that define the sources are retrieved and a map is returned with the key to the id of the source and as a value the class that defines the source, such as HdfsFile. Subsequently, this map is iterated and for each source, the dataframe is loaded and the relative load checks are performed. As for the virtual sources, the fundamental parameters are retrieved from the configuration file and are created starting from the basic sources.

3.1.2 Metric

Metrics are the main working unit of the Data Quality process. They specify the actual calculation to be made in order to get Key Performance Indicators and an analytical overview over input sources data. This is done by creating the proper Typesafe config like listing 3.3. The metrics embed the information about the sources we are analyzing, reporting meaningful results that will be used in the check phase. Thru these results, we can extrapolate useful informations about the raw state of the data. There are two macro metric types, **COLUMN** and **FILE**: the former are applied on single or multiple columns of an input source or virtual source. The latter consists only in the **Row_count**, which reports the number of rows of a given source, often used as a comparison value with other metrics.

```
1  Metrics: [
2    {
3      id          = "customer_row_count"
4      name        = "ROW_COUNT"
5      type        = "FILE"
6      description = "rowcount"
7      config = {
8        file = "customer"
9      }
10   },
11   {
12     id          = "null_values"
13     name        = "NULL_VALUES"
14     type        = "COLUMN"
15     description = "null␣values␣in␣column␣attacker_size"
16     config = {
17       file = "GOT_B",
18       columns = ["attacker_size"],
19       positions = [1]
20     }
```

```
21    },
22  ...
```

Listing 3.3: Example of a Typesafe configuration for a file metric and a column metric.

In the listing 3.3 we can observe an example of a definition of a column metric and a file metric: The first metric, of type file, will count the number of rows in the "customer" source; the second metric will count the number of null_values in the column "attacker_size" of the "GOT_B" source.

The following is a list of column metrics provided by the application. All metrics take as input the values of the column defined in the configuration file, in the Metrics section:

- **Distinct_values**: calculates count of distinct values in processed elements. If we have a big diversity of elements and do not need an exact result, we should use approximate distinct values.
- **Approximate_distinct_values**: as above, is an optimized version of distinct values.
- **Null_values**: calculates amount of null values in processed elements.
- **Empty_values**: calculates amount of empty strings in processed elements.
- **Min_number**: calculates minimal value for provided elements.
- **Max_number**: calculates maximal value for provided elements.
- **Sum_number**: calculates sum of provided elements.
- **Avg_number**: calculates mean value for provided elements.
- **Std_number**: calculates standard deviation for provided elements.
- **Min_string**: calculates minimal length of processed elements.
- **Max_string**: calculates maximal length of processed elements.
- **Avg_string**: calculates average length of processed elements.
- **Formatted_date**: calculates amount of strings in provided date format.
- **Formatted_number**: calculates amount of elements that fit provided format.
- **Formatted_string**: calculates amount of strings with specific requested length.

- **Casted_number**: calculates amount of string that can be casted to numerical, in double format.
- **Number_in_domain**: calculates amount of elements in provided domain set.
- **Number_out_domain**: calculates amount of elements out provided domain set.
- **String_in_domain**: calculates amount of string from provided domain.
- **String_out_domain**: calculates amount of string out of provided domain.
- **String_values**: calculates count of appearance of requested string in processed elements.
- **Regex_values**: calculates amount of rows that fits the provided regular expression.
- **Numbers_values**: calculates count of requested values' appearance in processed elements.
- **Median_values**: calculates these metrics by using TDigest[5] library. We have also **First_quantile**, **Third_quantile**, **Get_quantile** and **Get_percentile**.
- **Top_N**: calculates top N element for processed elements, with N user-specified.
- **Column_eq**: calculates number of equal rows.
- **Day_distance**: calculates the number of the rows for which the day difference between the two columns given as input is less than the threshold.
- **Levenshtein_distance**: calculates amount of rows where Levenshtein distance between two columns is lesser than threshold. Informally, the Levenshtein distance is defined as the distance between two words, which is the minimum number of single-character required to change one word into the other. This is a string metric for measuring the difference between two sequences.
- **Co-moment**: calculates covariance between values of two columns.
- **Covariance**: In probability theory and statistics, covariance is a measure of the joint variability of two random variables.
- **Covariance_Bessel**: In statistics, Bessel's correction is the use of $n - 1$ instead of n in the formula for the sample variance and sample standard deviation, where n is the number of observations in a sample.

[5] https://github.com/isarn/isarn-sketches

Through the definition of dedicated config files and the use of basic arithmetic formulas, we can compose different metrics by creating new ones. As reported by listing 3.4, using the id of the metrics preceded by the $ symbol in the formula field, we can create new custom metrics, which implement rules or business logic. We can use either a base metric or another composed metric in the formula. In the example reported by listing 3.4 we have a composed metric that will simply add two to the result of the metric with id "row_count", mentioned above.

```
1 ComposedMetrics: [
2   {
3     id          = "SE"
4     name        = "qwe"
5     description = "qwe"
6     formula     = "$row_count+2"
7   },
8 ]
```

Listing 3.4: Example of a Typesafe configuration for a composed metric.

After having grouped all the sources within a Scala Sequence, a map is applied and the following steps are performed for each element:

- creation of classes that contain information about file metrics, if any. These are always obtained from the configuration file.
- creation of classes that contain information about column metrics, if any.
- call to the processAllMetrics function. This function will be detailed within the subsection 3.3.3.

In figure 3.2 we can observe the results of column metrics obtained after an execution of the application. The column metric with id "null_values" reported in listing 3.3 has its result in the row number 5 of the figure 3.2. The file metric with id "customer_row_count", also reported in listing 3.3, has its result in row 2 of figure 3.3. The result of the composed metric with id "SE" reported in listing 3.4 is reported in figure 3.4, being the only composed metric used in the application configuration.

3.1.3 Check

Checks are the main part of the Data Quality job. They define the crucial parts of our analysis pipeline. Applied mostly over metric results or SQL queries, they allow to really understand the input data quality. Basically, through the checks we are going to define the rules to be applied to the metrics deriving from the sources examined. Their task is to report whether the applied rule was successful or not, allowing us a deep control on the state of the analyzed data. A typically check

consists in a comparison between a metric and a threshold value defined by the user: the check reports if the comparison holds true or not, saving the result on the supporting database, if present.There are four base types of checks:

Load

These are the only checks that are applied to the sources rather than the metrics. They are executed during the loading phase of the sources, in order to minimize calculation time and their task is to check file formats and serialization (CSV or Avro), check if the file exists in the given path and if the number of columns are equal or greater than a given parameter. Listing 3.5 reports an example of a load check: this one will check if the "customer" source can be loaded with the "UTF8" encoding.

```
1 LoadChecks: [
2   {
3     id     = "customer_encoding_check"
4     type   = "ENCODING"
5     source = "customer"
6     option = "UTF-8"
7   },
8 ]
```

Listing 3.5: Example of a Typesafe configuration for a load check.

Snapshot

Compare a calculated metric over a snapshot of the input data versus a given threshold. This is the most common check used by the framework's users. We have four subtypes:

- **Differ_by_Lt**: calculates the relative error between two given metrics.
- **Equal_to**: checks if a threshold/metric result is equal to a given value.
- **Greater_than**: checks if a given value is greater than a certain threshold/-metric result.
- **Less_than**: checks if a given value is less than a certain threshold/metric result.

Listing 3.6 reports a trend check where we are checking if the metric "row_count" is greater than the user-defined threshold 10: in case the inequlity holds true, a succes message will be reported.

```
Checks: [
  {
    id          = "teracheck"
    type        = "snapshot"
    subtype     = "GREATER_THAN"
    description = "check␣for␣number␣rows␣limit␣with␣
       threshold␣on␣table␣A"
    name        = "row_check"
    config = {
      metrics = ["row_count"]
      params = {threshold: "10"}
    }
  },
]
```

Listing 3.6: Example of a Typesafe configuration for a snapshot check.

Trend

Analyze previous results of the metric in order to evaluate the behaviour of the data. Previous results are intended to be found in a configured historical metrics database. They are based on the following formula, which is used in whole or only one of the two parts.

$$(1 - T) * A <= C <= (1 + T) * A \tag{3.1}$$

In 3.1 T is the user-defined threshold, A is the average result and C is the current result of the metric. When we perform this kind of check, we retrieve from the historical metrics database the average results of the interested metric, calculated across n last executions and we check if the inequality holds true or not. This type of check is identified by the subtype **Average_bound_full_check**, while using **Average_bound_upper_check** will use the latter part of the above formula and **Average_bound_lower_check** the former. As additional fields, the trend check accepts two types of rules when comparing the metrics: the first is called "record" and compares the current metric result with previous n records; the second is called "date" and compares the current metric result with results made in last n days. As for parameters, we can specify a threshold between $[0, 1]$ that represents the allowed difference level between results to pass the check and a time-window which represents the amount of days/records we are considering. There is a subtle difference between records and dates: when we run the batch job, we specify a reference date among the various parameters. When we want to consider only the last n records, the results with the same date as that used during this execution are taken. Instead, when we want to consider the time window, the results obtained

in the last n previous days of execution are considered. Listing 3.7 brings us an example of how to write a trend check: we are checking if the metric "row_count" satisfies equation 3.1, with T equal to 0.5 and time window equal to 2. This mean that we will considerate the last "two days" of execution.

```
Checks: [
{
description = "some␣basic␣trend",
id          = "trend_check",
subtype     = "AVERAGE_BOUND_FULL_CHECK",
type        = "trend"
config = {
  metrics = ["row_count"],
  params = {threshold: "0.5", timewindow: "2"},
  rule = "record"
},
},
]
```

Listing 3.7: Example of a Typesafe configuration for a trend check.

SQL

These have the task to run a query on a source which is a remote database and check if the result is zero or not.

- **Count_eq_zero**: returns success if the result of the query is zero, else failure.
- **Count_not_eq_zero**: returns success if the result of the query is not zero, else failure.

The result of the load check defined in listing 3.5 is at row 1 of figure 3.5. The result of the snapshot check define thru listing 3.6 can be found at row 2 of figure 3.6 and the trend check of listing 3.7 can be found in the first row of figure 3.6.

3.1.4 Target

Targets are the components to be leveraged to obtain outcomes from the previous steps, like saving the metrics and check results and/or generate notifications, alerts and e-mails. Targets are grouped by two types:

- **Regular**: saves the result as files of predefined format. All those files will have a predefined schema.
- **System**: used to define a list of checks and alerting, if some of them failed. E-mail servers configuration should be placed in the application config file in order to get these notifications and alerts.

```
Targets: [
  {
    type   = "FILE-METRICS"
    config = {
      fileFormat = "csv"
      path = "./tmp/results"
      delimiter = ","
    }
  },
]
```

Listing 3.8: Example of a Typesafe configuration for a target.

Listing 3.8 reports an example of a target configuration: we are telling the application to save all the file metrics in a CSV format in the "path", using as delimiter ",".

3.1.5 Postprocess

Postprocess is the final block of the Data Quality job workflow. Its purpose is to enrich and transform particular DataFrames (built from sources and virtual sources) in order to obtain the desired form, which may be used in future applications, i.e. for reporting. Currently, there are four types of postprocessing:

- **Enrich**: attaching values or constants to already defined DataFrame or new one. Is used to connect source, metrics, checks and all additional together to create "body" of the future report.
- **Transpose by key**: transposing the input DataFrame, but keeping key columns untouched. It's creates extra rows.
- **Transpose by column**: on the other hand, transposing each column individually (detaching header and putting it as a column). In particular, in takes key/all columns in the order as they are present, adding column with header to the left and adding/trimming extra columns to fit required structure.
- **Arrange**: rearrange columns in the DataFrame and cast them to a specific type.

Result of each postprocessor is a DataFrame stored as file (csv, avro), which can be used as an input to next pipeline. All post processing are executed in the order they were defined. This phase was strongly influenced by the use-case in which the framework was born, therefore it brings with it limitations, such as the particular format of the output tables and the schema with which they are created.

```
Postprocessing: [
  {
    mode   = "enrich"
    config = {
      source = "BTL_FILTERED"
      metrics = ["y_avg","1011"]
      checks = ["teracheck"]
      extra = {
        pasta = "test"
        person = "Rocco"
        ingridient = "Cream"
        test = "Test"
      }
      saveTo = {
        fileName = "tera_enriched"
        fileFormat = "csv"
        path = "./tmp/postproc"
        delimiter = ","
      }
    }
  },
]
```

Listing 3.9: Example of a Typesafe configuration for a postprocess.

Listing 3.9 indicates that the source "BTL_FILTERED", which has already the keyfields "name", "year", "battle_number", "attacker_king" and "defender_king" to be saved as a CSV file with the name "tera_enriched" with the addition of three new columns: "y_avg", "1011" (which are both metrics calculated during the current run) and "teracheck" which is a check result. Moreover, we are requiring also the addition of four more columns (found in the extra field) with the relative values.

3.2 Evolving the framework to a new Scenario

The current framework is able to satisfy the main requests necessary to perform a data quality analysis for standard data, i.e. all those forms of data that are finite in nature. Very often, this type of data is available even before carrying out the actual analysis, but it is still necessary to be able to subject them to a preventive analysis before going to make business decisions on them.

Naturally, with the evolution of the main big data scenarios, the need arose to be able to carry this execution logic also in streaming environments. In this type of big data analysis, the data to be analyzed are never known a priori, only their structure is. This places more emphasis on the need to have tools similar to this framework to perform the analysis in near real time. Being able to analyze the

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	column_names text[]	params text	result text	additional_result text
1	col_eq_kings_lev	2021-02-09	LEVENSHTEIN_DISTANCE	GOT_B	{attacker_king,year}	{"threshold" : 0.1}	0.0	
2	y_std	2021-02-09	STD_NUMBER	GOT_B	{year}		0.6801498938568182	
3	y_avg	2021-02-09	AVG_NUMBER	GOT_B	{year}		299.10526315789474	
4	1011	2021-02-09	MEDIAN_VALUE	GOT_B	{year}		299.11764705882354	
5	null_values	2021-02-09	NULL_VALUES	GOT_B	{attacker_size}		0.0	
6	null_values_col	2021-02-09	NULL_VALUES	GOT_B	{attacker_size}		0.0	
7	average	2021-02-09	AVG_NUMBER	GOT_B	{attacker_size}		299.10526315789474	
8	col_eq_king	2021-02-09	COLUMN_EQ	GOT_B	{attacker_king,defender_king}		1.0	
9	col_eq_kings	2021-02-09	COLUMN_EQ	GOT_B	{attacker_king,defender_king}		1.0	
10	col_eq_kings_lev99	2021-02-09	LEVENSHTEIN_DISTANCE	GOT_B	{attacker_king,defender_king}	{"threshold" : 0.99}	35.0	
11	topn_3_3	2021-02-09	TOP_N_3	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.19444444444444445	Balon/Euron Greyjoy
12	topn_6_5	2021-02-09	TOP_N_5	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.0	
13	topn_3_1	2021-02-09	TOP_N_1	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.3888888888888889	Joffrey/Tommen Baratheon
14	topn_6_2	2021-02-09	TOP_N_2	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.2777777777777778	Robb Stark
15	topn_3s_1	2021-02-09	TOP_N_1	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.3888888888888889	Joffrey/Tommen Baratheon
16	topn_3s_3	2021-02-09	TOP_N_3	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.19444444444444445	Balon/Euron Greyjoy
17	topn_3s_2	2021-02-09	TOP_N_2	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.2777777777777778	Robb Stark
18	topn_3_2	2021-02-09	TOP_N_2	GOT_B	{attacker_king}	{"targetNumber" : 3}	0.2777777777777778	Robb Stark
19	topn_6_3	2021-02-09	TOP_N_3	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.19444444444444445	Balon/Euron Greyjoy
20	topn_6_4	2021-02-09	TOP_N_4	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.1388888888888889	Stannis Baratheon
21	topn_6_6	2021-02-09	TOP_N_6	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.0	not_present
22	topn_6_1	2021-02-09	TOP_N_1	GOT_B	{attacker_king}	{"targetNumber" : 6}	0.3888888888888889	Joffrey/Tommen Baratheon

Figure 3.2: A report of the columnar metric results

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	result text	additional_result text
1	row_count	2021-02-09	ROW_COUNT	GOT_B	38.0	
2	customer_row_count	2021-02-09	ROW_COUNT	customer	11.0	

Figure 3.3: A report of the file metric results

data and, at the same time, perform checks on their goodness is not a fact to be underestimated.

In the current case, that is the one already solved by the current framework, we can consider the various data analyzed as simple tables that have a certain structure and to which we can apply calculations to obtain results or simply apply SQL queries to obtain the aggregates that interest us. To accomplish this, Spark provides us with many abstractions, which allow us to simplify the logic we need to develop. In the case of streaming data, the original table is to be understood as if it was a table to which new results are continuously appended, bringing with it a series of very different problems from those faced in the current version.

The difficulties to be able to face this type of conversion are certainly not few,

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	formula text	result text	additional_result text
1	SE	2021-02-09	qwe		$row_count+2	40.0	

Figure 3.4: A report of the composed metric results

Data Output | Explain | Messages | Notifications

	id text	src text	tipo text	expected text	date text	status text	message text
1	customer_encoding_check	customer	ENCODING	UTF-8	2021-02-09	Success	
2	customer_file_type	customer	FILE_TYPE	avro	2021-02-09	Failure	Source can't be loaded as avro
3	customer_file_existence	customer	EXIST	true	2021-02-09	Success	Source is present in the file system
4	customer_exact_column	customer	EXACT_COLUMN_NUM	1	2021-02-09	Failure	Source #columns {2} is not equal to 1
5	customer_min_column	customer	MIN_COLUMN_NUM	2	2021-02-09	Success	2 >= 2

Figure 3.5: A report of the load check results

above all because there are many abstractions at play that do not facilitate the task. The basic concept is already very different, since in the first case we already have a finite set of data available for which we want to obtain some metrics while, in the second, we must consider that the data is potentially infinite and, consequently, also the logic to be applied are profoundly different. We will see in the following chapter that the problems are numerous and simply adding new sources (of the streaming type) is not a solution.

3.3 Methodology

In this section we will report what changes have been applied to be able to modify the framework in order to obtain support for streaming datasets. The structure will try to follow the one presented in the previous sections, in order to have a concise structure divided by key concepts of the application and will focus on why certain choices were made.

3.3.1 Streaming container

Before getting to grips with the core of the program and its various components, we need to prepare and set up the streaming application container. The first element to create and configure is the SparkSession. The SparkSession is the entry point of a Spark application, starting from version 2.0 (previously, the entry point was

Data Output | Explain | Messages | Notifications

	check_id text	check_name text	description text	checked_file text	base_metric text	compared_metric text	compared_threshold text	status text	message text	exec_date text
1	trend_check	AVERAGE_BOUND_FULL_CHECK	some basic trend	GOT_B	row_count	Some(row_count)	0.5	Failure	Check trend_check for metric RO...	2021-02-09
2	teracheck	GREATER_THAN	check for number rows limit with threshold on table A	GOT_B	row_count	None	10.0	Success	Check teracheck for metric ROW...	2021-02-09

Figure 3.6: A report of the check results

the SparkContext) and allows us to create DataFrame, Dataset, access Spark SQL services, run SQL queries, load tables and access the DataStreamReader interface, which allows us to load Datasets starting from streaming sources. The SparkSession also allows us to enable Apache Hive support. Spark does not place any constraints on the number of sessions that can be opened and, once it is no longer needed, they can be stopped with the SparkSession.stop method. Among the various methods made available by SparkSession, the first we have to deal with is the builder method: this allows us to retrieve a spark session if it exists, otherwise it builds another. The builder allows us to set other parameters concerning the session, such as the name of the application (which will appear in the webUI[6] available on localhost:4040), specify the master (if spark will run locally or on a cluster, in the case of local run, we can also specify the number of cores which translates into the number of executors) and any configuration files. Another very important method that a Spark session makes available to us is the ability to import the implicits. These contain the encoders for the basic types of Scala, which are essential to be able to serialize and deserialize the data that we are going to insert in the datasets.

3.3.2 Source

The first step is the creation and definition of the streaming sources that the application will have to support. As for the input sinks, the focus was on file folders and Kafka.

Regarding the first, we defined a new case class which contained the following parameters: id, path, fileType, header, date, delimiter, quote, escape, schema and keyfields. Basically we find the same parameters used to define a static source. This is because the difference, from this point of view, is very slight: in this case, path indicates the path to the folder that will contain the files and not the absolute path for the single file. The conceptual difference between static files (such as a CSV file) and dynamic sources is very small: in the first case, we are going to specify the absolute path to Spark (on an HDFS for example) to load the file we want to analyze; in the second case, we specify to Spark which is the absolute path of the folder where we will find the file that we want to analyze. Otherwise, the

[6] https://spark.apache.org/docs/2.4.7/monitoring.html

various extra options and parameters are very similar since the difference is between analyzing a single file or keeping track of a folder where files are continuously added.

As regards the file formats, we have decided to replicate the support already provided by the basic version of Spark Structured Streaming, so support for CSV, JSON, ORC and Parquet files is offered. For the actual data loading part, we had to create a Scala object named StreamingFolderReader, which incorporates all the functions to load the various supported file formats of a streaming nature. For all the above-mentioned formats, we have chosen to make it mandatory for the user to insert a schema during the definition of the source itself. This restriction allows us to use a consistent schema, even in the event of query failure. Spark also offers the possibility of schema inference of a dataset in loading phase by setting spark.sql.streaming.schemaInference to true, but we preferred the first approach. Within this object we have condensed the different functions to load the various types of formats mentioned above, paying particular attention to what are the individual needs and precautions for each format.

All loading functions return a sequence of dataframes: this is because the original dataframe is repartitioned according to the value present in sparkContext.defaultParallelism. This parameter is useful to leverage all the Spark's optimizations and is the key to obtain a good-performing application.

As for the kafka sources, the parameters that characterize the case class are:

- id: identifier used within the application.
- server: a comma separated list to identify the kafka server (s) and its ports, i.e. "host1: port1, host2: port2".
- date: this parameter can be optional, as if absent it is taken from the application configuration file.
- subscribe: a comma separated list of topics to subscribe[7].
- subscribePattern: a Java regex string representing the pattern used to subscribe to topic(s)[8].
- assign: a json string {"topicA":[0,1],"topicB":[2,4]} that represents specific TopicPartitions to consume[9].
- startingOffset: allowed values are "earliest", "latest" (streaming only), or json string """ {"topicA":{"0":23,"1":-1},"topicB":{"0":-2}} """. The start point when

[7]Only one of "assign", "subscribe" or "subscribePattern" options can be specified for Kafka source.

[8]see 7

[9]see 7

a query is started, either "earliest" which is from the earliest offsets, "latest" which is just from the latest offsets, or a json string specifying a starting offset for each TopicPartition.

- dataStored: here the type of data that contains the topic is highlighted and the allowed values are UTF8, JSON or Avro.
- schema: field used to define the schema that we want to use to handle this file.
- keyfields: This field is only for the final part of the pipeline: here we indicate which columns of the file (and relative values) we want to keep when we go to save this source.

Here too we have created a new Scala object, called KafkaReader: for the general structure, we have decided to follow the one reported for StreamingFolderReader, so we have a main function that contains all the methods to load the various formats that we have decided to support.

The constraint that only one from "assign", "subscribe" or "subscribePattern" can be defined is strengthened via code, through a check on which of the three variants has been set by the user and, if there are more than one, an IllegalParameterException is thrown, ending the execution of the program.

The dataStored and schema parameters are used for parsing the binary values present within the raw Kafka message.

When we read a Kafka source to create a dataframe, what we get is a dataframe with the following schema (3.10). These are fields of a Kafka record and the metadata associated with it. In our case, what interests us is the message itself, which we find inside value in binary format, so we need a way of parsing the field. With the dataStored parameter we are asking the user to tell us in what format the data was entered into the Kafka message: we currently support JSON, Avro and standard strings.

For the first case, we consider the message in JSON format, so when we go to read from the topic(s), we perform a conversion and apply the contents of the schema parameter to the result, so as to obtain the message with the same structure with which it was put into Kafka. What we get is a dataframe with the same schema present inside the JSON file.

As for Avro, in the schema parameter we expect to find the path to retrieve the .avsc file, which contains the message's schema. If it is not present, the application throws an IllegalParameterException, ending the execution. Also in this case, we perform the conversion of the binary value in Avro format, applying the above mentioned schema to obtain a correctly structured message. What we get is a dataframe with the same schema present inside the .avsc file.

The latter case is the most trivial, as we simply convert the binary values of key and value into UTF8 encoded strings, obtaining a dataframe with structure [key: String, value: String].

In conjunction with the loading of the streaming sources, the application performs the calculation of the load checks 3.1.3. In detail, the post load checks are calculated, i.e. checks on the exact number of columns or if the latter exceeds a certain user defined threshold or not. Once the sources have been loaded and the load checks performed, we continue by uploading the result of the load checks on the external database.

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Listing 3.10: Schema of a DataFrame created from a Kafka Sink without further transformations

3.3.3 Metric

The concept adopted by the original program to calculate metrics is that there is first a conversion of the static dataframe to rdd and then a three-point process that is intended to actually calculate the metrics. There is a basic optimization as regards the calculation of metrics, with the aim of reducing the overhead introduced by the use of a large number of classes and instances. This optimization consists in creating abstract calculators, which have the task of performing the actual calculation of the metrics required by the user. The creation does not respect a one-to-one relationship with user-defined metrics, but a single calculator instance is created for each type of metric characterized by the same parameters, even if it has different columns as the subject of the analysis, even from different datasets. Furthermore, the metrics are calculated for each partition of the dataset, in order to take advantage of the parallelism offered by Spark.

In order to calculate the metrics, the framework uses a process divided into three steps: first it iterates over the RDD and passes the values of the columns under examination to the respective calculators. Then the partition calculators are updated and finally they are reduced in order to merge the results. Throughout this process, column and file metrics are stored separately. The raw calculation is performed with the help of a function made available to Spark, the treeAggregate.

These steps are performed by the MetricProcessor object, which takes for each defined source as input, a dataFrame, the related column and file metrics that the user wants to calculate and the key fields of the dataframe. The function returns two maps: the first contains information regarding the column metrics and the second that of files.

The first map contains, as a key, a sequence of names of the metrics obtained and as a value a further map, which in turn contains as a key the class that represents a metric of columns and as a result a tuple: the first value, a Double , contains the result of the calculated metric while the second value contains any extra parameters.

The second map contains, in a similar way to the first, a key represented by the class that abstracts the concept of file metrics within the program and as a value a tuple, identical to the previous one.

The treeAggregate (Figure3.7) method is a specialized implementation of aggregate that iteratively applies the combine function to a subset of partitions. This is done in order to prevent returning all partial results to the driver where a single-pass reduce would take place as the classic aggregate does. For all practical purposes, treeAggregate follows the same principle as aggregate with the exception that it takes an extra parameter to indicate the depth of the partial aggregation level. Aggregate lets us transform and combine the values of the RDD at will. It uses two functions:

The first operation transforms and adds the elements of the original collection of type [T] in a local aggregate of type [U] summarized as: (U,T) => U. This operation is applied locally to each partition in parallel.

The second operation takes two values of the result type of the previous operation [U] and combines it in to one value. This operation will reduce the partial results of each partition and returns the actual total.

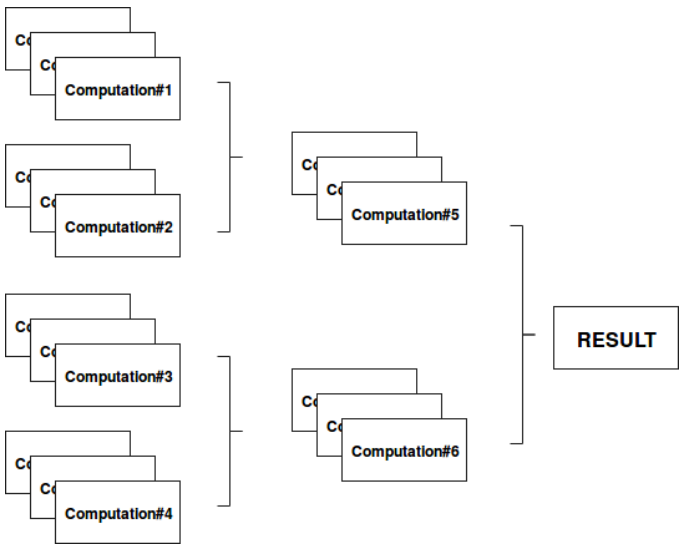


Figure 3.7: A visual representation on how treeAggregate works [30].

A deep connoisseur of Spark will surely have noticed that this involves a big problem for the purpose of our thesis, as the treeAggregate (and, of course, the aggregate) is not a function supported by dataFrame of a streaming nature. The

problem that arose was to find a way to apply user-defined functions to a finite group of rows of the dataFrame, in order to extrapolate the metrics requested by the user.

In order to perform the required calculation without having to upset the nature of the application and the abstractions put in place by the original developers to calculate the metrics, we decided to apply a series of functions to convert the external logic of how this calculation is performed.

First of all, we created a KeyValueGroupedDataset from the original dataframe with the help of the groupByKey function. This function has been applied to the dataframe with the addition of three columns and a watermark:

the first column we added was that of the timestamp, obtained through the current_timestamp function, which returns the current timestamp as a timestamp column. We need this to be able to operate with the abstraction provided by the windowing functions.

This choice is motivated by the fact that, when we talk about streaming dataframes, we cannot absolutely calculate any type of value or aggregate. For example, think about the requirements for calculating the average value of a set of data: we need all the data that make up that set. This, unfortunately, is not obtainable when we are dealing with streaming source datasets, as the original data can be part of a presumably infinite collection. For this reason, we have decided to insert the timestamp in the original dataset in order to have values for certain time frames. In this way, we have a finite time interval where we can calculate any required metric or aggregate, as we consider the data that fall within this last interval. With this abstraction that we wanted to introduce, we can summarize the whole core of the thesis, as our analysis moves from having absolute results to having results relative to time intervals.

The second column we inserted in the starting dataFrame is a window: a window is made up of two columns, which represent the beginning of a window and its end and is characterized by three fundamental parameters: the first indicates which column of the dataset that contains the time event on which to apply the window, the second indicates the duration of the window in terms of time, known by the name of window duration and the third and last parameter is the sliding window, i.e. every time a new window is generated. With this last column we have enclosed the concept expressed previously: move the logic of the calculation for streaming sources. By wrapping the data that comes in continuously in time intervals, we can apply the abstractions and optimizations already present for static datasets.

Finally, we have applied a watermark so that the intermediate stage accumulated during the analysis does not become too onerous to manage. By applying the watermark to the timestamp column, we are telling Spark to discard any data that reaches the application in an instant longer than that indicated by the watermark, in order to "close" the analysis of that batch of data.

The grouping by key was performed on the two columns produced by the window application, i.e. window start and window end. Subsequently, the mapGroups function was applied to be able to calculate the metrics using the code already working for static datasets. mapGroups applies the given function to each group of data. For each unique group, the function will receive the group key and an iterator that contains all of the elements in the group. The function can return an element of arbitrary type which will be returned as a new Dataset. This function needs an implicit Encoder in order to convert from Spark SQL internal representation to that of the JVM and vice versa. As an Encoder, we have decided to use kryo encoders. These encoders save each line of the dataset as simple flat binary objects.

Applying this logic we were able to obtain the required results; the metric calculation returns a sequence of dataframes, one for each source of which we want to calculate metrics, characterized by the following parameters:

- Tuple containing the window's starting and ending timestamps.
- Map where we have a sequence of metric names as a key and another map as a value; the latter contains the column metric as a key and a double, which represents the result, as a value. We also have optional additional parameters (used for the TOP_N metric).
- Map where we have as a key a file metric and as a value a double for the result, with optional extra parameters

Through a series of basic transformations made available by Spark, such as map and flatMap, we were able to correctly extrapolate and structure the metrics of both the column and the files obtained from the datasets, in order to obtain a clear and understandable representation by the user.

The next step was to create a common representation of these two types of metrics, since it is necessary for the calculation of composite metrics, which require all available metrics as input, in order to create new ones through the application of user-defined formulas. This purpose was achieved with the help of a new case class that incorporates the common characteristics of the two metrics, such as the name of the metric, the final result and the subject of the analysis (the column to which it was applied for the case of column metrics; the source to which it was applied for the case of file metrics). The latter was used to feed the function that takes care of calculating compound metrics: as input it takes all the metrics and takes the results to use them within user-defined formulas.

For the actual calculation, we apply the same logic already seen for the calculation of the metrics: we group by key the dataset composed of the two types of metrics and apply the mapGroups. In this way we are able to apply a function to a finite group of metrics, managing to calculate the compound metrics.

Since everything written inside the Spark lambdas is executed in parallel by the various executors of the cluster, we must make sure that every single object

is serializable. For this reason we had to modify the methods that performed the computation of composed metrics, since non-serializable objects were passed.

The concept of serializable is fundamental in Spark: everything we write to the driver and that must be executed by executors must be serializable. To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object after it has traveled across the network that links the Spark cluster. When we write lambda functions for example, these will be transmitted over the network to each executor along with the part of the data where it will have to be applied, and then be rebuilt on the other side of the connection. As reported by[10], serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. Often, this will be the first thing we should tune to optimize a Spark application. Spark aims to strike a balance between convenience (allowing us to work with any Java type in your operations) and performance. It provides two serialization libraries:

- Java serialization: Spark serializes objects using Java's ObjectOutputStream framework, and can work with any class that implements java.io.Serializable. Java serialization is flexible but often quite slow.
- Kryo serialization: Spark can also use the Kryo library to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types.

3.3.4 Check

As regards the calculation of the checks, the logic we applied is the same as that used for the previous cases. Since, basically, the logic that is used to perform the calculation in the case of static data is the same used in the calculation of compound metrics: as input, all the previously calculated metrics are taken, i.e. those of file, column and composed ones, and the checks are calculated on those metrics specified by the configuration file.

In the streaming case, we went to group all the metrics for the start and end of the window and applied the mapGroups function. In the lambda we used the functions already implemented for the calculation of the checks, making the necessary changes so that we could take advantage of serialization and deserialization, taking advantage of the parallelization offered by Spark. This function gives us a dataset consisting of the main information concerning the checks, such as the type, the result, the name, the description, the metric object of the check and finally the status. The latter contains the result of the check itself, i.e. whether it was passed

[10] https://spark.apache.org/docs/2.4.7/tuning.html

or not. When we create a check, we usually compare the value of a certain metric with a threshold. If our statement is true (for example we want to know if the number of columns is greater than a certain value) then the check status will be successful, otherwise it will fail. The status has the purpose of quickly reporting the result of the check itself, communicating the failure or not, which often weighs in the decision and analysis on the quality of a given source.

3.3.5 Target and Postprocess

Regarding this section, it was not necessary to add support for the streaming environment, as it is not common to have to perform post-processing operations on the data just analyzed. The basic concept is to run the analysis of streaming data on an incoming stream, reducing unnecessary interactions as much as possible, to make the analysis as fast as possible. Since in most cases, this data will go as input to new pipelines for future analysis, it was not considered necessary to add support for particular change or save operations. For the same reason, support for joins between streaming datasets has been overlooked, as it is not a common operation in this area. The only type of target currently implemented is the saving on the external database, which has the task of containing all the results coming from the various checks and metrics.

Chapter 4

Evaluation

In this chapter we will present a comparison between the work of the basic framework and the one with streaming support. The analysis will start from a common dataset, which presents possible data quality problems (such as the lack of values), followed by a brief description of the data quality dimensions that we will analyze and will end with the comparison of the results obtained in both methods.

4.1 The analysis

Due to the very nature of the design, it is very difficult to make a direct comparison in terms of performance or overhead introduced between the two applications. Speaking only in terms of execution speed, it is obvious to state that the basic framework is faster than the two, we can even say that it represents, again in terms of execution times, a lower limit to the framework presented by this thesis. Since a streaming application can also be designed to ideally run for an infinite amount of time, we absolutely cannot evaluate the result of this search in terms of time. The overhead introduced by the second version of the application, of course, is substantial, but even this measurement is quite complicated to perform, since it deals with abstractions contained within the JVMs that run under the Spark process. Being able to perform an analysis of this kind would require the skill and time in dissecting the structure of the JVM itself and unfortunately, it was not the purpose of this thesis. For the reasons set out above, it was decided to perform a direct comparison between the two applications, analyzing in both cases the same dataset that presented some artificial data quality problems.

4.2 The dataset

Since it is very difficult to find datasets that are "dirty", i.e. that have problems related to the quality of the data they contain, such as missing values, we decided to

take an existing and complete dataset and to add some impurities. The protagonist of our analysis is a dataset provided by USGS[1], which is a scientific agency of the United States government. In particular, the database chosen contains all the records of the earthquakes that occurred during the year of 2011: among the various recorded values, which appear as columns of the analyzed CSV file, we find the following (we limited ourselves to reporting only those of interest, but for the record the CSV file contains 21 column fields):

- Date and Time: represent the date and time when the phenomenon was recorded.
- Latitude;
- Longitude;
- Depth: the depth at which the phenomenon occurred, in kilometers.
- Magnitude;
- Root Mean Square: as reported by the documentation, "The root-mean-square (RMS) travel time residual, in seconds, using all weights. This parameter provides a measure of the fit of the observed arrival times to the predicted arrival times for this location".
- ID: unique identifier for each single record, used as the registration key.

The database consists of a total of 710 entries, and was first analyzed by the framework without the modifications for the streaming support and, subsequently, by the framework resulting from this thesis. As for the impurities introduced, we have changed about 25% of the rows of the dataset: we report precisely the errors introduced:

- addition of null rows: we have deleted a series of records, leaving only the date and time fields, to simulate incomplete registrations of entries within the dataset.
- sabotage of latitude and longitude values: excessive values have been introduced for these two parameters. Since, by definition, latitude can be between -90.0 and 90.0, we have modified rows to contain values that exceed that range, both negative and positive. Similarly we proceeded with regard to longitude, which we know to be defined only in the interval -180.0 and 180.0.
- Unscrupulous increase in depth values: here too we wanted to simulate errors by exceeding the maximum acceptable values for this measurement, by entering values such as 999.999 km, where the maximum value present inside the database before the modification was 612.10 km.

[1] https://www.usgs.gov/

- values of magnitude equal to zero: again to simulate errors in the insertion of records, we have decided to modify a random number of rows by placing the magnitude value equal to zero. What is the point of a database that keeps all the earthquakes that have occurred if the magnitude of these is equal to zero?

- finally, we have also decided to put an rms value of 20 in some rows, knowing that the value of 2.0 has never been exceeded in the database.

All these changes were made to simulate a dataset in poor condition from the point of view of data quality. We remind you that the database is easily available from the USGS website, since they are real data. In order to have a more uniform distribution of these impurities, it was decided to apply them to every fifth row of the dataset. This choice was motivated by how the analysis of the streaming framework is carried out and will be further explored subsequently.

4.3 Results

The analysis focused on checking the range of values that the columns of the dataset could assume. First of all, we loaded the dataset into the framework: the data of our interest are found in a csv file consisting of 710 rows and 21 columns, as previously mentioned. As for the so-called "static" part, we omitted the manual scheme and we preferred to have it deduced by Spark. For the metrics of our interest, we opted to mainly check the columns considered most important to us, namely the id, latitude, longitude, depth, magnitude and rms. For the first column, we calculated metrics for the number of distinct values, nulls, and blank values. The first metric is motivated by the fact that we know that a unique id is present for each measurement recorded within the dataset, so from this first metric we can deduce if ids are missing, since the number of rows should correspond to the number of unique ids. For the other two metrics, the motivation was driven by the desire to know, if the first metric does not correspond with the number of rows in the dataset, how many of these are null or simply not present. For latitude and longitude, in addition to counting the number of null values present, we have chosen to return the maximum and minimum value of each one to understand if there was the possibility that some entries in the dataset have values that exceed the intervals of these two dimensions. The same reasoning was applied to the remaining columns mentioned above. Recall that these metrics are only the first step to perform an in-depth analysis of the data quality of this dataset. As composed metrics, we have decided to calculate simple percentages of null values, to get a relative idea of the incompleteness of this dataset: the general formula applied for each metric that calculates the number of values is the following: we multiply the metric of interest by hundred and divide by the row_number metric, which gives us the total number of rows in the file. In this way, we can have the percentage, for example, of the null values of the longitude

with respect to the entire dataset. The checks are simply the transposition of what we have already discussed in the previous section, that is, we check if the longitude, latitude, depth, magnitude and rms values are in the logical intervals to which they must belong.

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	column_names text[]	params text	result text	additional_result text
1	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		604.0	
2	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		106.0	
3	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0.0	
4	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		999.999	
5	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-210.123	
6	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		89.0	
7	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		102.0	
8	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		999.999	
9	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		0.02	
10	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		122.122	
11	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-200.123	
12	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		81.0	
13	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		20.0	
14	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.47	
15	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		92.0	
16	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		9.1	
17	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		-9999.0	
18	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		97.0	

Figure 4.1: The results of the columnar metrics of the static case.

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	formula text	result text	additional_result text
1	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	85.07042253521126	
2	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	14.929577464788732	
3	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	11.408450704225352	
4	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	12.535211267605634	
5	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	14.366197183098592	
6	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	13.661971830985916	

Figure 4.2: The results of the composed metrics of the static case.

In figure 4.1 we can see reported the various results of the aforementioned metrics. The table shows various information, such as the name of the metric, the date the analysis was performed, the source and the type of metric. We can observe that we find a total of 604 distinct values regarding the ID keys. This results is given by the fact that we have 106 rows where there is no unique id value. In fact, in the second row we can see that we have exactly 106 null values for the id column. As for the longitude and latitude, we can immediately observe that for both columns we have two maximum values well beyond the threshold of their range, in fact we recorded 999.999 as the maximum value. As for the minimum values, we find

Data Output | Explain | Messages | Notifications

	metric_id text	source_date text	name text	source_id text	result text	additional_result text
1	row_count	2021-02-25	ROW_COUNT	USGS	710.0	

Figure 4.3: The results of the file metrics of the static case.

Data Output | Explain | Messages | Notifications

	check_id text	check_name text	description text	checked_file text	base_metric text	compared_metric text	compared_threshold text	status text	message text	exec_date text
1	depthGreaterThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	None	700.0	Failure	Check depthGreaterThan for m...	2021-02-25
2	depthLessThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	None	0.0	Success	Check depthLessThan for metr...	2021-02-25
3	latLessThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	None	-90.0	Failure	Check latLessThan for metric ...	2021-02-25
4	lonLessThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	None	-180.0	Failure	Check lonLessThan for metric ...	2021-02-25
5	lonGreaterThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	None	180.0	Failure	Check lonGreaterThan for metr...	2021-02-25
6	magGreaterThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	None	10.0	Success	Check magGreaterThan for me...	2021-02-25
7	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	None	10.0	Failure	Check rmsLessThan for metric...	2021-02-25
8	latGreaterThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	None	90.0	Failure	Check latGreaterThan for metri...	2021-02-25
9	magLessThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	None	0.0	Failure	Check magLessThan for metri...	2021-02-25

Figure 4.4: The results of the file metrics of the static case.

-210.123 for both. Finally, we have 89 null values for longitude and 81 for latitude. We also find 92 missing values for the rms and 97 for the magnitude. Already from this first analysis we can confidently affirm that our dataset is not free from problems. In figure 4.2 we can observe the results of the compound metrics. In the first line we see the percentage of distinct values, which is 85.07: this translates into the fact that we have 85% of values other than null as regards the ids, which is a fairly acceptable value but far from a dataset free of quality problems. We have 14.92% null values for the id, 11.40% null values for latitude and 12.53% null values for longitude: this confirms the results obtained from the basic metrics. For depth we have 14.36 percent null values and 13.66 percent null values for magnitude. In figure 4.3 we have the result of the only file metric calculated or the row metric: the result matches the number of rows of the file itself. In figure 4.4 we have the summary of the controls applied. Starting from the first line, we can see that the check on the upper end of the depth range has failed, as the framework has found a maximum depth value greater than the threshold value set by us, that is 700. On the other hand, for as far as the lower bound is concerned, the check reports success, since there is no value less than the zero threshold value: it would not make sense to manage negative depths in an earthquake-focused dataset. Following the report, we observe that the checks on the maximum and minimum, both of the longitude and of the latitude have all had negative results, as the program has detected maximum and minimum values higher than the threshold set by us. The check on the maximum value of the rms column also failed as there are values greater than ten. The last line also reports a failure as the inequality is not strict, so the zero value found within the dataset is equal to the threshold value we have defined. As expected, the base version of this framework does its job, managing to

identify and reporting all these data quality problems.

The streaming analysis started from the definition of the main parameters of the application, that is the duration of the window, the sliding interval and the duration of the trigger. The first parameter defines the width of the window we want to use to analyze the incoming data and the second defines how often we generate a new window. For these first two parameters, we opted for five minutes, in order to have well-defined windows with no overlapping values. The last parameter defines the interval that must pass between one saving and the other of the results: here too we have set a value equal to five minutes. Subsequently, we proceeded to upload the dataset: in this case, we simply defined a folder where we will find our csv files, specifying the header since it is a requirement of the streaming application. To simulate a streaming environment, we created a script in Python. This program takes as input a csv file and splits it into as many files as required, which are inserted in the destination folder with a frequency equal to the one passed as parameter. As parameters, we have set the generation of a row every two seconds. So, since we have a five minute window, we should find 150 values in each window.

In the figures from 4.5a to 4.5e we have the set of all the results of the metrics. We can immediately see that we have many more entries, as the analysis lasted longer, about 24 minutes: this is the time it takes for the Python script to produce all 710 lines of the original dataset, taking into account that it produced one every two seconds. The repetition of the metrics in all the figures is due to the fact that the framework perfoms these calculation for each single window, so having 24 minutes of data input, the framework produced and analyzed about fives windows. In the first two columns we can immediately notice the execution windows: the first window that is created is the one that goes from 12:25:00 to 12:30:00, therefore a five-minute window. The application, therefore, produces the previously mentioned metrics every five minutes, reporting the values for the set of rows it is analyzing. The rest of the table is identical to the static version, in fact we find the same columns. The difference lies precisely in the results, since the latter are calculated for windows lasting five minutes. For example, we can see that the DISTINCT_VALUES metric increases with each window, this is because the number of keys increases every five minutes. We can already say that this is the desired behavior, that is to have the ability to calculate metrics on a continuous data stream. As always, remember that these metrics are intended to trigger the checks, which are the most important part of the entire analysis pipeline.

In the figures from 4.6a to 4.6b we have the set of all the results of the composed metrics. Here too we can see the analysis window, which always starts from 12:25:00 to 12:30:00. The results obtained in this part are more interesting because percentages are calculated taking the previous metrics as input. In fact, we can observe how in the window 12:30:00 - 12:35:00, the percentage of columns with zero latitude (precisely in row nine of figure 4.6a) was equal to 44.7%, that means more than one third of the records in that window had a null value regarding latitude,a

worrying value from the point of view of the quality of the stream.

Among the figures 4.7a and 4.7b we find the results of the checks carried out in

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	columnNames text[]	params text	result text	additionalResult text
1	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		4	
2	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		2	
3	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
4	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		171.631	
5	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-73.3259999999999934	
6	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		1	
7	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		1	
8	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		576.799999999999955	
9	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		10	
10	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		-4.45800000000000018	
11	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-59.7909999999999968	
12	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		1	
13	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		1.10000000000000009	
14	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.849999999999999978	
15	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		1	
16	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.20000000000000018	
17	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		5.59999999999999964	
18	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		1	
19	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		52	
20	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		33	
21	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
22	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		175.996000000000009	
23	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-177.808999999999997	
24	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		26	
25	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		18	
26	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		525	
27	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		5.90000000000000036	
28	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		55.9179999999999993	
29	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-59.232999999999997	
30	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		38	

(a) The results part one.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	columnNames text[]	params text	result text	additionalResult text
31	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		1.78000000000000003	
32	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.680000000000000049	
33	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		22	
34	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.20000000000000018	
35	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		5.5	
36	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		18	
37	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		104	
38	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		45	
39	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
40	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		149.776999999999987	
41	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-173.205000000000013	
42	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		29	
43	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		29	
44	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		510.600000000000023	
45	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		0.699999999999999956	
46	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		40.482999999999997	
47	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-53.2100000000000009	
48	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		29	
49	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		1.43999999999999995	
50	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.469999999999999973	
51	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		29	
52	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		9.09999999999999964	
53	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		5.5	
54	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		29	
55	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		136	
56	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		13	
57	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
58	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		179.873999999999995	
59	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-210.12299999999999	
60	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		13	

(b) The results part two.

Figure 4.5: The results of the columnar metrics of the streaming case

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	columnNames text[]	params text	result text	additionalResult text
61	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		13	
62	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		583.600000000000023	
63	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		5	
64	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		122.122	
65	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-59.4380000000000024	
66	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		13	
67	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		1.51000000000000001	
68	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.680000000000000049	
69	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		18	
70	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.09999999999999964	
71	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		5.5	
72	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		13	
73	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		148	
74	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		0	
75	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
76	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		999.999000000000024	
77	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-210.12299999999999	
78	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		11	
79	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		0	
80	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		999.999000000000024	
81	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		3	
82	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		122.122	
83	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-200.12299999999999	
84	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		0	
85	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		1.47999999999999998	
86	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.670000000000000004	
87	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		9	
88	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.59999999999999964	
89	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		0	
90	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		36	

(c) The results part three.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	columnNames text[]	params text	result text	additionalResult text
91	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		135	
92	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		14	
93	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
94	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		179.989000000000004	
95	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-179.531000000000006	
96	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		10	
97	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		37	
98	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		999.999000000000024	
99	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		5	
100	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		57.8900000000000006	
101	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-63.2449999999999974	
102	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		1	
103	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		20	
104	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.619999999999999996	
105	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		12	
106	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.40000000000000036	
107	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		-9999	
108	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		1	
109	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	distinct_values_id	2021-02-25	DISTINCT_VALUES	USGS	{ID}		24	
110	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_id	2021-02-25	NULL_VALUES	USGS	{ID}		0	
111	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	empty_values_id	2021-02-25	EMPTY_VALUES	USGS	{ID}		0	
112	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	max_number_lon	2021-02-25	MAX_NUMBER	USGS	{Longitude}		178.554000000000002	
113	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	min_number_lon	2021-02-25	MIN_NUMBER	USGS	{Longitude}		-179.09899999999999	
114	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_lon	2021-02-25	NULL_VALUES	USGS	{Longitude}		0	
115	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_depth	2021-02-25	NULL_VALUES	USGS	{Depth}		5	
116	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	max_number_depth	2021-02-25	MAX_NUMBER	USGS	{Depth}		555.5	
117	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	min_number_depth	2021-02-25	MIN_NUMBER	USGS	{Depth}		6.90000000000000036	
118	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	max_number_lat	2021-02-25	MAX_NUMBER	USGS	{Latitude}		47.0090000000000003	
119	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	min_number_lat	2021-02-25	MIN_NUMBER	USGS	{Latitude}		-56.5330000000000013	
120	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_lat	2021-02-25	NULL_VALUES	USGS	{Latitude}		0	

(d) The results part four.

Figure 4.5: The results of the columnar metrics of the streaming case (cont.)

streaming environments. In particular, we can observe that in 4.7a almost all the checks are passed: this is because the first part of the dataset is less "dirty" than the

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	columnNames text[]	params text	result text	additionalResult text
121	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	max_number_rms	2021-02-25	MAX_NUMBER	USGS	{"Root Mean Square"}		20	
122	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	min_number_rms	2021-02-25	MIN_NUMBER	USGS	{"Root Mean Square"}		0.719999999999999973	
123	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_rms	2021-02-25	NULL_VALUES	USGS	{"Root Mean Square"}		2	
124	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	max_number_mag	2021-02-25	MAX_NUMBER	USGS	{Magnitude}		7.09999999999999964	
125	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	min_number_mag	2021-02-25	MIN_NUMBER	USGS	{Magnitude}		5.5	
126	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_values_mag	2021-02-25	NULL_VALUES	USGS	{Magnitude}		0	

(e) The results part five.

Figure 4.5: The results of the columnar metrics of the streaming case (cont.)

final part. These checks guarantee us that, at least for the columns analyzed, up to the window between 12:40:00 and 12:45:00 the data are within the regular intervals and therefore are correct. By focusing on 4.7b we can already see how the framework begins to detect anomalies, given that the number of unsuccessful checks increases.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	formula text	result text	additionalResult text
1	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	66.6666666666666714	
2	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	33.3333333333333357	
3	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	16.6666666666666679	
4	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	16.6666666666666679	
5	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	16.6666666666666679	
6	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	16.6666666666666679	
7	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	61.176470588235297	
8	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	38.823529411764703	
9	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	44.705882352941174	
10	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	30.5882352941176485	
11	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	21.1764705882352935	
12	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	21.1764705882352935	
13	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	69.7986577181208077	
14	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	30.2013422818791959	
15	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	19.4630872483221466	
16	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	19.4630872483221466	
17	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	19.4630872483221466	
18	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	19.4630872483221466	
19	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	91.275167785234899	
20	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	8.72483221476510096	
21	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	8.72483221476510096	

(a) The results part one.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	metricId text	sourceDate text	name text	sourceId text	formula text	result text	additionalResult text
22	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	8.72483221476510096	
23	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	8.72483221476510096	
24	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	8.72483221476510096	
25	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	100	
26	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	0	
27	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	0	
28	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	7.43243243243243246	
29	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	0	
30	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	24.3243243243243228	
31	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	90.604026845637577	
32	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	9.39597315436241587	
33	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	0.671140939597315467	
34	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	6.71140939597315445	
35	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	24.8322147651006695	
36	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	0.671140939597315467	
37	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	total_keys_percentage	2021-02-25	totalKeysPercentage		($distinct_values_id * 100) / $row_count	100	
38	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_id_percentage	2021-02-25	nullIdPercentage		($null_values_id * 100) / $row_count	0	
39	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_lat_percentage	2021-02-25	nullLatPercentage		($null_values_lat * 100) / $row_count	0	
40	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_lon_percentage	2021-02-25	nullLonPercentage		($null_values_lon * 100) / $row_count	0	
41	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_depth_percentage	2021-02-25	nullDepthPercentage		($null_values_depth * 100) / $row_count	20.8333333333333321	
42	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	null_mag_percentage	2021-02-25	nullMagPercentage		($null_values_mag * 100) / $row_count	0	

(b) The results part two.

Figure 4.6: The results of the composed metrics of the streaming case

We can therefore confidently affirm that the streaming analysis occurs correctly, as the same checks fail in both the cases. Of course, the controls do not have the same timing, but anomalous data are detected in the same way, triggering the same alarms as in the static case. In these last lines of text lies the main concept about this new version of the application: we migrated the calculation logic, starting from analyzing finite datasets to analyze streams of data over constant periods of time. Connecting this framework to a streaming data produce, like Kafka for example, assures us that we can have reports, at intervals defined by us, on the quality of the data that is arriving, regardless of the input stream.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	checkId text	checkName text	description text	checkedFile text	baseMetric text	comparedMetric text	comparedThreshold text	status text	message text	execDate text
1	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
2	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
3	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Success	Check depthLessThan for me...	2021-02-25
4	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
5	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Success	Check latLessThan for metric ...	2021-02-25
6	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Success	Check lonGreaterThan for me...	2021-02-25
7	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
8	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Success	Check rmsLessThan for metri...	2021-02-25
9	2021-02-26 12:25:00+01	2021-02-26 12:30:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Success	Check magGreaterThan for m...	2021-02-25
10	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
11	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
12	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Success	Check depthLessThan for me...	2021-02-25
13	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
14	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Success	Check latLessThan for metric ...	2021-02-25
15	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Success	Check lonGreaterThan for me...	2021-02-25
16	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
17	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Success	Check rmsLessThan for metri...	2021-02-25
18	2021-02-26 12:30:00+01	2021-02-26 12:35:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Success	Check magGreaterThan for m...	2021-02-25
19	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
20	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
21	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Success	Check depthLessThan for me...	2021-02-25
22	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
23	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Success	Check latLessThan for metric ...	2021-02-25
24	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Success	Check lonGreaterThan for me...	2021-02-25
25	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
26	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Success	Check rmsLessThan for metri...	2021-02-25
27	2021-02-26 12:35:00+01	2021-02-26 12:40:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Success	Check magGreaterThan for m...	2021-02-25
28	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
29	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
30	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Success	Check depthLessThan for me...	2021-02-25
31	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25

(a) The results part one.

Data Output | Explain | Messages | Notifications

	time_window_start text	time_window_end text	checkId text	checkName text	description text	checkedFile text	baseMetric text	comparedMetric text	comparedThreshold text	status text	message text	execDate text
32	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Failure	Check latLessThan for metric ...	2021-02-25
33	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Failure	Check lonGreaterThan for me...	2021-02-25
34	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
35	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Success	Check rmsLessThan for metri...	2021-02-25
36	2021-02-26 12:40:00+01	2021-02-26 12:45:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Success	Check magGreaterThan for m...	2021-02-25
37	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Failure	Check latGreaterThan for met...	2021-02-25
38	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
39	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Failure	Check depthLessThan for me...	2021-02-25
40	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
41	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Failure	Check latLessThan for metric ...	2021-02-25
42	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Failure	Check lonGreaterThan for me...	2021-02-25
43	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Failure	Check lonLessThan for metric...	2021-02-25
44	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Success	Check rmsLessThan for metri...	2021-02-25
45	2021-02-26 12:45:00+01	2021-02-26 12:50:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Failure	Check magGreaterThan for m...	2021-02-25
46	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
47	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
48	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Failure	Check depthLessThan for me...	2021-02-25
49	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
50	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Success	Check latLessThan for metric ...	2021-02-25
51	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Success	Check lonGreaterThan for me...	2021-02-25
52	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
53	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Failure	Check rmsLessThan for metri...	2021-02-25
54	2021-02-26 12:50:00+01	2021-02-26 12:55:00+01	magGreaterThan	GREATER_THAN	check for magnitude greater than 0	USGS	min_number_mag	[null]	0	Failure	Check magGreaterThan for m...	2021-02-25
55	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	latGreaterThan	GREATER_THAN	check for latitude greater than -90	USGS	min_number_lat	[null]	-90	Success	Check latGreaterThan for met...	2021-02-25
56	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	magLessThan	LESS_THAN	check for magnitude less than 10	USGS	max_number_mag	[null]	10	Success	Check magLessThan for metr...	2021-02-25
57	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	depthLessThan	LESS_THAN	check for depth less than 700	USGS	max_number_depth	[null]	700	Success	Check depthLessThan for me...	2021-02-25
58	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	depthGreaterThan	GREATER_THAN	check for depth greater than 0	USGS	min_number_depth	[null]	0	Success	Check depthGreaterThan for ...	2021-02-25
59	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	latLessThan	LESS_THAN	check for latitude less than 90	USGS	max_number_lat	[null]	90	Success	Check latLessThan for metric ...	2021-02-25
60	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	lonGreaterThan	GREATER_THAN	check for longitude greater than -180	USGS	min_number_lon	[null]	-180	Success	Check lonGreaterThan for me...	2021-02-25
61	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	lonLessThan	LESS_THAN	check for longitude less than 180	USGS	max_number_lon	[null]	180	Success	Check lonLessThan for metric...	2021-02-25
62	2021-02-26 12:55:00+01	2021-02-26 13:00:00+01	rmsLessThan	LESS_THAN	check for root mean square less than 10	USGS	max_number_rms	[null]	10	Failure	Check rmsLessThan for metri...	2021-02-25

(b) The results part two.

Figure 4.7: The results of the checks of the streaming case

Chapter 5

Conclusions

In this final chapter we will recapitulate the results obtained from the work of this thesis and make proposals for possible future improvements.

As we have seen for the course of this thesis, the support for streaming applications facilitated by the platform offered by Apache Spark has been correctly implemented. The confirmation of the achievement of the objectives set at the beginning of this path was also provided by the previous chapter, where we highlighted that the same alarms are triggered in both use cases, i.e. both the static and the dynamic subject of this thesis. The ability to analyze and control streaming flows allow us to apply quality concepts and business strategies, which certainly offer an advantage on the big data market, before we actually analyze that input flow of data.

As far as the possibility of optimization and improvement of the solution proposed in this thesis is concerned, the most important is certainly the one regarding trend checks (3.1.3). It would be interesting to evaluate a possible translation of the streaming window concepts applied in the context of trend checks. the basic idea is to consider the results of the previous analyzes as if they were a new stream of data, applying them a window in which we perform the check calculations while keeping a minimum of consideration of the previous data, so as to be able to intervene promptly when we notice net changes in the results. A further improvement to be made is certainly that which concerns the increase in the input sinks that the application is capable of managing. This can be facilitated by the flexibility offered by the Apache Spark platform, in particular by structured streaming. This flexibility is aided by the abstractions offered by the *foreach* and *foreachbatch* functions, and by these means can be easily implemented.

Bibliography

[1] Amir Gandomi and Murtaza Haider. "Beyond the hype: Big data concepts, methods, and analytics". In: *International Journal of Information Management* 35.2 (2015), pp. 137–144. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2014.10.007. URL: http://www.sciencedirect.com/science/article/pii/S0268401214001066.

[2] Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. "Business Intelligence and Analytics: From Big Data to Big Impact". In: *MIS Quarterly* 36.4 (2012), pp. 1165–1188. ISSN: 02767783. URL: http://www.jstor.org/stable/41703503.

[3] Nicole Martin. *How Much Data Is Collected Every Minute Of The Day*. 2019. URL: https://www.forbes.com/sites/nicolemartin1/2019/08/07/how-much-data-is-collected-every-minute-of-the-day/?sh=4f0945243d66 (visited on 01/21/2021).

[4] L. Cai and Y. Zhu. "The Challenges of Data Quality and Data Quality Assessment in the Big Data Era". In: *Data Science Journal* 14 (2015). DOI: http://doi.org/10.5334/dsj-2015-002.

[5] Damilola Ojo, Patrick Taleng. *Data Quality Dimensions*. URL: https://www.deltapartnersgroup.com/managing-data-quality-optimize-value-extraction.

[6] C. Cichy and S. Rass. "An Overview of Data Quality Frameworks". In: *IEEE Access* 7 (2019), pp. 24634–24648. DOI: 10.1109/ACCESS.2019.2899751.

[7] Yang W. Lee et al. "AIMQ: a methodology for information quality assessment". In: *Information & Management* 40.2 (2002), pp. 133–146. ISSN: 0378-7206. DOI: https://doi.org/10.1016/S0378-7206(02)00043-5. URL: http://www.sciencedirect.com/science/article/pii/S0378720602000435.

[8] Leo L Pipino, Yang W Lee, and Richard Y Wang. "Data quality assessment". In: *Communications of the ACM* 45.4 (2002), pp. 211–218.

[9] Batini Carlo et al. "A data quality methodology for heterogeneous data". In: *International Journal of Database Management Systems* 3.1 (2011).

[10] Reza Va, Mehran Mohsenzadeh, and Jafar Habibi. "TBDQ: A Pragmatic Task-Based Method to Data Quality Assessment and Improvement". In: *PLOS ONE* 11 (May 2016), e0154508. DOI: 10.1371/journal.pone.0154508.

[11] Thomas L Saaty. "Decision making with the analytic hierarchy process". In: *International journal of services sciences* 1.1 (2008), pp. 83–98.

[12] Robert S Kaplan and Steven R Anderson. "Time-driven activity-based costing". In: *Available at SSRN 485443* (2003).

[13] Arie Van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: An annotated bibliography". In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.

[14] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems* 26.2 (June 2008). ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: https://doi.org/10.1145/1365815.1365816.

[15] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[16] Matthias J. Sax. "Apache Kafka". In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–8. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_196-1. URL: https://doi.org/10.1007/978-3-319-63962-8_196-1.

[17] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28.

[18] Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[19] Apache Spark. *Stream as a Table*. URL: https://spark.apache.org/docs/2.4.7/structured-streaming-programming-guide.html.

[20] Apache Spark. *window*. URL: https://spark.apache.org/docs/2.4.7/structured-streaming-programming-guide.html.

[21] Ashish Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (2010), pp. 996–1005.

[22] Ashish Thusoo et al. "Hive: a warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[23] Hiren Patel. "HBase: A NoSQL Database". In: (May 2017). DOI: 10.13140/RG.2.2.22974.28480.

[24] V. Bhupathiraju and R. P. Ravuri. "The dawn of Big Data - Hbase". In: *2014 Conference on IT in Business, Industry and Government (CSIBIG)*. 2014, pp. 1–4. DOI: 10.1109/CSIBIG.2014.7056952.

[25] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[26] Michael Stonebraker and Lawrence A Rowe. "The design of Postgres". In: *ACM Sigmod Record* 15.2 (1986), pp. 340–355.

[27] Lawrence A Rowe and Michael R Stonebraker. *The POSTGRES data model*. Tech. rep. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCE, 1987.

[28] Michael Stonebraker, Eric Hanson, and Chin-Heng Hong. "The design of the POSTGRES rules system". In: *1987 IEEE Third International Conference on Data Engineering*. IEEE. 1987, pp. 365–374.

[29] Michael Stonebraker. *The design of the Postgres storage system*. Tech. rep. CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB, 1987.

[30] Bartosz Konieczny. *tree aggregation*. URL: https://www.waitingforcode.com/apache-spark/tree-aggregations-spark/read.