Big Data Validation

Raya Rizk

Subject: Information Systems
Corresponds to: 30 hp
Presented: VT 2018
Supervisor: Steve McKeever
Examiner: Andreas Hamfelt

Department of Informatics and Media
Abstract

With the explosion in usage of big data, stakes are high for companies to develop workflows that translate the data into business value. Those data transformations are continuously updated and refined in order to meet the evolving business needs, and it is imperative to ensure that a new version of a workflow still produces the correct output. This study focuses on the validation of big data in a real-world scenario, and implements a validation tool that compares two databases that hold the results produced by different versions of a workflow in order to detect and prevent potential unwanted alterations, with row-based and column-based statistics being used to validate the two versions. The tool was shown to provide accurate results in test scenarios, providing leverage to companies that need to validate the outputs of the workflows. In addition, by automating this process, the risk of human error is eliminated, and it has the added benefit of improved speed compared to the more labour-intensive manual alternative. All this allows for a more agile way of performing updates on the data transformation workflows by improving on the turnaround time of the validation process.

Keywords

big data, data testing, data validation, data quality, big data validation process, big data validation tool
Acknowledgements

I would like to thank my supervisor Steve McKeever for giving me the time and guidance during this study. I am also very thankful to my teachers at Uppsala University for their encouragement and continuous support throughout my studies.

This thesis was conducted with the Swedish company Klarna. I am very grateful for all the support that the company provided and in particular for the help of my supervisor Johan Petrini and for his advice and assistance during the work. I would also like to thank everyone at Klarna’s Uppsala office who shared their knowledge and experiences on the issues covered in this thesis, especially Jörgen Falk, who always had answers to my questions from the first day.
# Table of Contents

1 Introduction .......................................................... 1  
1.1 Background ....................................................... 1  
1.2 Problem statement .............................................. 3  
1.3 Research questions .............................................. 4  
1.4 Related work .................................................... 4  
  1.4.1 Research in big data validation .......................... 4  
  1.4.2 Big data validation tools ................................ 5  
1.5 Delimitations ..................................................... 6  
1.6 Outline .......................................................... 6  

2 Theoretical Framework ............................................... 8  
2.1 Big data and relational databases ............................... 8  
2.2 Big data testing and validation ................................ 9  
  2.2.1 Testing big data applications ............................ 9  
  2.2.2 Big data processing and validation stages .......... 10  
2.3 Development frameworks ...................................... 12  
  2.3.1 Hadoop .................................................... 12  
  2.3.2 Hive ....................................................... 13  

3 Methodology .......................................................... 17  
3.1 Research approach .............................................. 17  
3.2 System specifications ......................................... 18  

4 Implementation ...................................................... 19  
4.1 Validation process .............................................. 19  
  4.1.1 Deduplication ............................................ 20  
  4.1.2 Row validation .......................................... 25
### 4.1.3 Column validation

### 4.2 Difftong validation tool

### 5 Test Cases and Analysis

5.1 First test case

5.2 Second test case

5.2.1 Row statistics

5.2.2 Column statistics

5.3 Performance measurements

5.4 Data correctness and accuracy

### 6 Discussion and Conclusions

6.1 Summary of the findings

6.2 Limitations and future work

### References

### A Appendix: Generated HiveQL Script for a Data Validation Example

A.1 Deduplication

A.2 Row validation

A.3 Column validation

### B Appendix: Helper Functions

B.1 Hash function

B.2 Delta function

B.3 Percentile function

### C Appendix: Test Data Set

C.1 TPC-H
List of Tables

4.1 Original databases schemas ........................................... 19
4.2 Data in “DB1” ......................................................... 20
4.3 Data in “DB2” ......................................................... 20
4.4 Deduplicated databases schema ....................................... 22
4.5 Data in “DDB1” ......................................................... 22
4.6 Data in “DDB2” ......................................................... 22
4.7 Row validation statistics .............................................. 26
4.8 Validation status for row delta table ................................ 28
4.9 Validation database schema - row validation tables ............... 29
4.10 Data in “ValidationDB” - row delta table ......................... 30
4.11 Column validation statistics .......................................... 34
4.12 Validation database schema - column validation tables .......... 35
4.13 Data in “ValidationDB” - column delta table ..................... 36

5.1 Row validation statistics for two identical databases ............ 41
5.2 Row validation statistics for two non-identical databases ........ 42
5.3 Column validation statistics for the “LineItem” table .............. 45
5.4 Column validation statistics for the “Orders” table ............... 45
5.5 Run-time for two test cases measured in milliseconds .......... 48
5.6 CPU-time for two test cases measured in milliseconds .......... 48

B.1 Description of the “hash” function in Hive ....................... 64
B.2 Description of the “multi_hash” function .......................... 65
B.3 Description of the “calc_diff” function ............................ 66
B.4 Description of the “percentile” function in Hive ................... 67
List of Figures

2.1 Process flowchart of big data framework ....................................... 11
2.2 Big data validation stages .......................................................... 12

4.1 Count of records in “DB1” and “DB2” tables ................................. 21
4.2 Row differences between two tables in “DDB1” and “DDB2” .............. 26
4.3 Median and quartile values ........................................................... 32
4.4 Column differences between two tables in “DDB1” and “DDB2” ...... 34

5.1 Row validation chart for the difference count ................................. 44
5.2 Row validation chart for the difference percentage ......................... 44
5.3 Column validation chart for the difference count ........................... 46
5.4 Column statistics chart for the “LineItem” table ............................ 46
5.5 Column statistics chart for the “Orders” table ................................ 47

C.1 The TPC-H schema ................................................................. 69
# List of Scripts

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Template for creating a database</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Template for obtaining database tables</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Template for obtaining table columns</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Template for creating a table in the deduplicated database</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Setting Hive configuration to optimize statistics collection</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Template for analyzing a table</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>Template for obtaining the count of rows in a table</td>
<td>27</td>
</tr>
<tr>
<td>4.8</td>
<td>Template for obtaining the absolute value of the difference between two</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>numbers</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Template for creating a row delta table in the validation database</td>
<td>30</td>
</tr>
<tr>
<td>4.10</td>
<td>Template for obtaining the percentage of differences</td>
<td>31</td>
</tr>
<tr>
<td>4.11</td>
<td>Template for creating a column delta table in the validation database</td>
<td>36</td>
</tr>
<tr>
<td>4.12</td>
<td>Template for analyzing a table for columns</td>
<td>37</td>
</tr>
<tr>
<td>4.13</td>
<td>Template for obtaining column statistics</td>
<td>37</td>
</tr>
<tr>
<td>4.14</td>
<td>Template for obtaining first, second, and third quartiles</td>
<td>38</td>
</tr>
</tbody>
</table>
1 Introduction

This chapter discusses the background and the purpose of the study. It defines the problem area along with the related research questions. The delimitations of the thesis are also be presented and an outline for the rest of the paper is listed at the end.

1.1 Background

Due to the rapid development of big data and the continuous increase of the amount of various data that is being generated at an unprecedented speed, there has been an exponential growth in the number of programs that use big data techniques [19]. Applications and workflows that process data in diverse domains and transform them into business values have been developed. In order for those transformations to provide solutions for business problems in organizations, it is necessary to perform data analysis based on valid inputs and then ensure the correctness and accuracy of the outputs produced at later stages [4]. In other words, it is imperative to employ continuous validation during all phases of data processing, starting from data collection, to data analysis and transformations, and ending with reporting the results [9].

Therefore, the demand for implementing effective and efficient testing methods for big data applications has increased in order to ensure data quality at all processing phases [32]. Data quality can be defined as the degree to which the state of data serves its purpose in a given context [9], where aspects of data quality include completeness, accuracy, correctness, and consistency. According to Experian Data Quality [27], 75% of businesses are wasting an average of 14% of their revenue due to poor data
quality. In addition, Gartner research indicates that the average annual impact of such low data quality on organizations is as high as $9.7 million [22]. This can be also linked to IBM’s estimation in 2016 [16], where they stated that the yearly cost of poor quality data in the US alone was $3.1 trillion. This is likely to worsen with the increased amount and complexity of data. Consequently, and with such high costs, it is a critical issue to maintain the quality as it is essential to derive value from data and it makes significant impacts on decision-making in companies [35].

However, the large volume and fast velocity in which heterogeneous data is being generated and processed makes measuring data quality difficult [4]. Most big data sets lack clear structures since data are extracted from a diversity of data sources. Therefore, it can be unclear what to test exactly and it is difficult to define how to perform the test process. This poses challenges on big data testing processes [10]. Furthermore, manual data validation is difficult, inefficient, and time-consuming as it was mentioned in Harvard Business Review [28] that about 50% of knowledge workers’ time is wasted trying to identify and correct errors. Thus, testing automation is required in order to detect the effect of every data transformation that has occurred.

Although big data and its applications are an active academic research topic and there has been numerous papers published in this area, further research is still needed as current work seldom addresses the validation process of big data and how to assure data quality during different stages of an application [9]. In addition, and even though numerous data validation tools were developed and used by organizations to manage their data in a more efficient manner, there is still clearly a need for more solutions as reflected in the statistics that are mentioned earlier in this section. Most of the current tools only provide the common basic data validation functions, such as checking null values and data types, ranges, and constraints [9]. Thus, a more detailed data validation that examines all values in a data set and detects potential errors is still lacking.
1.2 Problem statement

As big data is evolving rapidly and the number of related applications is increasing continuously, organizations must seek solutions to ensure the validity of their data and preserve its quality. However, and as indicated in the previous section, testing big data applications is not an easy process and can be considered as one of the biggest challenges that organizations face as it is sometimes hard to define what to test, how to test, and how to automate the testing process [13].

Companies develop workflows (data transformations) to process data continuously. Those workflows transform data using a series of steps to extract value from them, then store the new results in databases to be used for planning and decision-making or as inputs to other transformations.

Driven by evolving business needs and due to the rapid increase in the volume and variety of data, those workflows need to be updated and improved constantly. For instance, the updates can be related to handling new data models or meeting changing business requirements, while improvements of workflows can be achieved through implementing more efficient and optimized transformations. Preserving data quality after developing a new version of a workflow is critical and manual data validation can be tedious, time-consuming, and error-prone. Thus, it might seem like a good idea to either skip any meaningful validation and hope for the best, or abort the update. Both alternatives have negative consequences as they will lead to either producing data with poor quality, or restricting any attempts to improve organizations methods.

Therefore, it is necessary to implement a data validation tool that automatically compares the outputs of a pre-update workflow and its post-update counterpart by taking the same inputs and indicating whether the results of the new workflow match the previous one, or whether it produces incorrect results or even more correct and accurate outputs. The correctness of the data is linked to the context of the workflow where new requirements can be introduced in the new version. In addition, applying
more efficient solutions as a part of the updated transformation can lead to a more accurate output.

Starting from the needs of a Swedish company to validate the results of their workflows, this study aims to find a general solution for comparing any two databases that have the same schemas, then highlighting the differences and similarities between them in order to meet various business needs in organizations. In broad terms, the study aim to provide a new solution that attempts to tackle some of the big data validation problems by ensuring the quality of big data applications in general.

1.3 Research questions

The following research questions are posed in order to serve the purpose of this study:

- What steps are needed to validate big data in terms of comparing two different databases with identical schemas?
- How can the validation process be automated?
- Can the validation process provide a generic mechanism for validating different types of databases?

1.4 Related work

1.4.1 Research in big data validation

As stated in the previous sections of this chapter, validation techniques have struggled to cope with the exploding size of data suggesting a need of continuous research in the area to find new solutions for validating those types of data. Current research in
big data quality considers the issue in broad terms, but there is little focus on how to effectively validate big data applications [9].

In their paper, [9] discuss big data validation and quality assurance by giving a definition of big data quality, listing its dimensions and related issues, and summarizing big data quality validation processes which includes data collection, cleaning, transformation, loading, analysis, and report. Moreover, the study presents a comparison of eight existing data validation tools concluding that all of them provide the basic validation criteria that are set in the paper such as checking data types, formats, ranges, logic, and null values, even if this comes with some limitations.

More detailed analysis was done by the same authors a year later [35] where they conducted a case study and displayed the differences between the results that various data validation tools generate when validating the same set of data. It worth noting that there is a lack of tools that generate descriptive statistics, which help to gain a deeper insight in the data. Moreover, the study defined a quality checklist for big data including other data validation types than the basic ones, such as checking for duplication in data, inconsistency, and incompleteness.

The study reflected on the current solutions of data validation and discussed primary challenges and needs. In general, the authors stressed that there is a gap in the available research in relation to big data validation and data quality issues are still open and have not been solved yet. Hence, it follows that further research and studies in this field should be conducted in the future as only few published papers addresses big data validation methods.

1.4.2 Big data validation tools

QuerySurge [29] is a tool that compares data that reside in two data stores (source and target). The input of this tool is two queries (QueryPairs) to be run against both source and target data stores, and the output is reflected in showing the difference in
row numbers along with listing the values that differ in both stores. The tool comes with some limitations regarding a maximum row size of a result set and a maximum number of QueryPairs to compare in one run. *QuerySurge* is a closed source software that can be distributed under a licensing agreement.

Another example of tools that attempt to calculate the delta between two values of large data sets is *BigDiffy* [31], an open source library for pairwise field-level statistical differences of data sets developed by Spotify using Scala. Delta is defined by Spotify as “a change of any changeable quantity” and the tool provides a record-oriented comparison that is undertaken on a columnar level based on unique keys. *BigDiffy* generates statistics and corresponds to the column validation step in *Diftong* - the tool that is implemented in this study - and which is a part of the whole validation process that is proposed in this dissertation. Column validation step is described in details in the *Implementation* chapter in this paper.

### 1.5 Delimitations

The study is about validating the data obtained from big data application processes. The project will only focus on parts of the “process” validation stage of the overall big data testing process and not on the “data staging” or “output” validation that are explained later in the *Theoretical Framework* chapter. In addition, only data correctness and accuracy - two of the data quality measures that are also defined in the *Theoretical Framework* chapter - will be considered in terms of data quality aspects.

### 1.6 Outline

This dissertation contains a total of six chapters, and it has been organised in the following way:
The first chapter provides a general overview of the study and presents the problem statement with the related research questions and some related work in the field. Chapter two describes the theoretical framework and covers the concept of big data, its testing and validation, in addition to the development frameworks that are used in this project. The research methodology is then displayed in the third chapter while a detailed explanation of the proposed validation process and its implementation is listed in chapter four. This chapter also introduces Diftong, the validation tool that was implemented. Chapter five shows two test cases and analyzes the results. Discussion and conclusions are presented in chapter six.


2 Theoretical Framework

This chapter explains the concept of big data and how it defers from the relational data model. In addition, it discusses how big data applications can be tested and what are the stages for processing the data. Lastly, the Hadoop framework and the Hive query interface are described.

2.1 Big data and relational databases

Relational Database Management Systems (RDBMS) - such as MySQL, Oracle, and Microsoft SQL Server - have been the dominant model for database management for more than 40 years [36]. The relational data model has been used for storing information in an organized way and managing structured data using a query interface based on Structured Query Language (SQL) [21].

RDBMS ensure data integrity and guarantee high transaction reliability as they support ACID properties (Atomicity, Consistency, Isolation, and Durability) [36]. The relational model has a rigid schema with a clear structure where the data are correlated with specific characteristics and adhere to the schema [14]. Information is stored in tables with primary keys that allow to uniquely identify any record. Tables can contain indexes and constraints, and can relate to other tables using foreign keys [20]. Enforcing specific rules and structures makes the management of data easier as it helps maintaining clean data and prevents undesirable redundancy.

Despite the robustness of RDBMS, it has limitations. It cannot handle the increasing size of data or process unstructured information due to the aforementioned rigid schema [14]. However, with the exponential growth of data produced from various sources in a structured, semi-structured, or unstructured format [17], there was a
need to develop scalable database techniques and tools to overcome the disadvantages of relational databases and manage this big data [36].

Big data is more than just size, it is also varied and fast-growing. It is characterized by three main aspects referred to as “V’s” and continues to increase rapidly in all its dimensions [37]:

- **Volume**: The large amount of data generated, measured in terabytes, petabytes, or even exabytes.
- **Velocity**: The high rate of data generation.
- **Variety**: The heterogeneity of data sources.

It follows that from this rapid expanse, big data poses challenges to data management [21]. Starting from the need to address the scalability of databases and to manage the semi-structured and unstructured data, non-relational data models - such as NoSQL (Not Only SQL) and Hadoop - were introduced [36].

Non-relational databases do not use the RDBMS principles and do not store the data in tables [14]. The main characteristic of these databases is having a schema-free [24], i.e. they do not require schema definition before inserting the data, which results in unstructured data that the user has to interpret when retrieving it. The absence of structure, rules, and constraints makes it difficult to maintain clean and correct data without duplications. It also affects the overall quality of information and introduces complexity in the testing and validation processes.

### 2.2 Big data testing and validation

#### 2.2.1 Testing big data applications

Testing software that uses big data techniques is significantly more complex than testing other more traditional data management applications. Simple test cases cannot be used for big data applications as processing the data requires a longer
time and data are being transferred and transformed at different points during each process of the application [19]. Therefore, and in order to test big data applications effectively, continuous validation throughout the stages of the data transformations is highly stressed [11]. The data need to be tested and validated in several areas and transition points of the workflows to ensure that it is being processed correctly without any errors.

There are different types of tests that can be conducted to maintain a good quality of data, such as functional and non-functional testing. Data quality includes various parameters that should be measured like data accuracy, correctness, consistency, usability, completeness, accessibility, scalability, etc. [9].

Functional testing includes the validation of data. Data validation is an important step to improve data quality and it covers many of its factors such as data correctness and accuracy [35]. It is imperative to ensure the correctness of the data in terms of data types and formats [9], and in relation to the context [35]. In addition, data accuracy is usually measured by comparing the data in multiple data sources, as this quality factor refers to how close the results are to the values that are accepted as being true [9].

Non-functional testing, on the other hand, includes performance and failover tests, which play a key role to ensure the scalability of the process [13]. As part of performance testing, job completion time and memory utilization are captured, while failover testing focuses on the recovery process after any failure in the network.

2.2.2 Big data processing and validation stages

The processing of big data, and thus big data validation, can be divided into three different stages [23]:

1. **Data staging**: Loading data from various external sources, such as weblogs, social media, RDBMS, etc. into a big data system like Hadoop Distributed
File System (HDFS). The validation process of this first step includes verifying that the needed data were extracted and retrieved correctly, then uploaded into the system without any corruption.

2. **Processing:** Executing MapReduce operations to obtain the desired output. In this step, it is required to validate the results of MapReduce and other similar big data application processes - explained further in this chapter - while covering the accuracy and correctness of the data.

3. **Output:** Extracting the output results of the processing stage and loading it into a downstream system which may be a data warehouse, business intelligence tools, or a repository for generating big data analytics and reports. The validation of this final step includes checking whether the data have been loaded correctly into the target system for any further processing. It can also include verification of the content of the generated reports.

The Figure 2.1 below shows an example of this process based on the Hadoop framework (explained in the next section), while Figure 2.2 illustrates the related validation stages in more details.

![Figure 2.1: Process flowchart of big data framework](image)

Figure 2.1: Process flowchart of big data framework [23]
2.3 Development frameworks

An overview of the development frameworks and tools (Apache Hadoop and Apache Hive) that are used in this study is presented in this section.

2.3.1 Hadoop

According to [20], Apache Hadoop is “an open source software framework for storage and large scale processing of datasets on clusters of commodity hardware”. In other words, Hadoop supports running applications on big data using parallel processing. It is designed to scale up to thousands of machines that offer local storage and computation.
The Hadoop framework ensures data availability, scalability, and portability. It distributes the work on multiple processors and platforms after dividing the data into fixed-size pieces and replicating them across all nodes [36]. This helps solving problems with large and complex data sets in a reliable manner, as it also provides automatic fault tolerance and recovery.

The Hadoop framework includes two important modules [20]:

- **Hadoop Distributed File System (HDFS)**: A distributed file system that handles the storage of large data sets in low-cost hardware. In other words, it provides scalability, fault-tolerance, and cost-effective storage for big data. HDFS follows a master/slave structure where one master node controls one or more slave devices. The NameNode (master) manages metadata in the file system and a DataNode (slave) stores the actual data. Data are replicated in more than one node in order to provide more reliability and recover any data in case a node failure.

- **Hadoop MapReduce**: A computational paradigm that provides parallel processing of large data sets. It works by dividing the application into many small fragments that are executed on different nodes in the cluster. MapReduce has two distinct tasks as the name depicts. The “Map” phase - the first part of MapReduce parallel processing - takes a set of data and converts it into key/value pairs. While the “Reduce” phase - executed after the “Map” task - takes each key/value pair as its input and produces new output to be stored in the HDFS. MapReduce engine consists of a JobTracker (master node) that divides the submitted job into tasks and passes them to TaskTrackers (slave nodes) that work close to the related data.

### 2.3.2 Hive

Apache Hive is a data warehouse built on top of Hadoop [20]. It was developed by Facebook to reduce the complexity of big data frameworks to gain easier access to
the desired data. Due to the limited query capabilities of Hadoop and the complexity of MapReduce framework, developers were required to write complex programs that might be hard to maintain and reuse even for simple analysis [33]. This can be expensive and time-consuming. Therefore, there was a need to introduce a new framework that was capable of running on more traditional interactive query engines [30].

Hive supports queries expressed in a declarative language similar to SQL - called HiveQL - and has its own Data Definition Language (DDL) and Data Manipulation Language (DML) commands. Those queries are compiled into MapReduce jobs that use the parallel processing in Hadoop [30]. Hive also provides an option for developers to define their own custom functions in Java using User-Defined Functions (UDFs), User-Defined Aggregates (UDAFs), or User-Defined Table Functions (UDTFs) in order to extend its functionalities [33].

In other words, Apache Hive provides a mechanism to project structure onto the large volume of data that reside in distributed storage. Thus, it facilitates the management of these data by executing queries using an SQL-like query language that is simple and easy to learn by anyone who is already familiar with SQL.

**Hive data model**

Since Hadoop is able to store and process any kind of data (unstructured, semi-structured, or structured), Hive supports multiple schemas and has a “schema-on-read” feature - as opposed to “schema-on-write” in RDBMS - which defers the application of a schema until accessing the data [30]. This results in faster data loading but relatively slower queries.

Hive provides a data model that is similar to conventional relational databases. It makes the data in Hadoop look like it is stored in tables that consist of a number of rows, and each row has a specified number of columns [33]. In addition, each column has an associated type that can be either primitive or complex. The
primitive types that are currently supported in Hive are: numeric, character/string, date/time, boolean, and binary. Hive also supports complex types like arrays, lists, and structs. The data can then be accessible via different interfaces - such as JDBC (Java Database Connectivity) - from applications that support SQL languages [33].

**Hive metastore**

In order to make sense of the unstructured nature of big data, a certain level of metadata is to be expected. In Hive, this is commonly achieved through a metastore service; it is a system catalog, created in a RDBMS, that contains within itself information and details about the objects definitions (schemas, tables, partitions, columns, etc.) [30]. By separating parsing instructions from the actual data, the metastore acts as a form of index for otherwise impenetrable data files. In addition, metastore deals with supplementary statistics concerning the data, providing utility in data exploration, query optimization, and query compilation [33].

**Apache Tez**

Although MapReduce is the engine used to run Hive jobs in Hadoop, an improved paradigm for Hive execution - called Tez - is now commonly used to process big data. Tez provides a number of advantages over MapReduce, such as faster execution plans and optimal management of resources [30]. It improves the speed and performance of data processing, where one single Tez job can replace multiple MapReduce jobs.

**Hive configuration properties**

A number of configuration variables in Hive is available to change the behavior of the default installation settings. For instance, in order to get the number of rows in a table faster, Hive provides configurations to store the results of some queries -
such as the “count” statistics - in the metastore [3]. This will optimize the statistics collection as the values will be read from the metastore directly instead of being calculated each time. Those variables can be configured using the “set” command that works on a session level, where the new values will apply to all subsequent statements of the “set” command [1].

As an example of some of the configuration variables that are used in this study, “hive.execution.engine” is set to “tez” in order to use Apache Tez engine, that is described in the previous section, instead of the default option MapReduce. In addition, setting the “hive.stats.autogather” configuration to “true” enables automatic gathering and updating of statistics during Hive DML operations. “hive.compute.query.using.stats”, on the other hand, answers queries like “min”, “max”, and “count” solely using statistics stored in the metastore.

Based on the information provided in this chapter, a validation process for big data is proposed and the implementation of this process is explained in details in the next chapter.
3 Methodology

This chapter is rather concise and is devoted to explaining the approach taken in developing a tool that can correctly perform data validation. In addition, it covers specifications concerning the IT artifact being built.

3.1 Research approach

In order to answer the research questions and to investigate the purpose of this study, design science research (DSR) that results in an IT artifact was conducted; the presence of an IT artifact meeting the specifications is considered sufficient to answer the research questions.

Given the fact that big data and its validation is an active research topic, it is interesting to define the knowledge contribution of this study according to the DSR knowledge contribution framework [12]. This framework consists of four different categories (Invention, Improvement, Adaptation, and Routine Design) and is used to understand and position the contributions of a project. Since this thesis addresses an existing problem related to big data validation with a new solution, the knowledge contribution can be considered as an “Improvement”, which refers to developing new knowledge or solutions for known problems.

DSR includes multiple stages [25] starting from problem identification where the problem and motivation of the research is defined and the objectives of a solution is set. The following step is the solution design where the system is analyzed, designed, and developed. Finally, a demonstration of the artifact accompanied by an evaluation is undertaken to confirm the solution and then summarize the results.

The project was conducted with Klarna - a Swedish bank that offers online financial
services and provides payment solutions to customers [18] - in an iterative process where additional functionalities were added continuously to the data validation tool that was being developed. The project started with studying Klarna's current tools and solutions for big data validation and other related works. Then, new solutions, changes, and improvements were implemented and the results were validated after each iteration.

3.2 System specifications

This study focuses on big data validation taking into consideration the special features and needs of big data applications. Moreover, a new validation tool for detecting differences and similarities between two databases with identical schemas was implemented. The tool was undertaken using Java and Hive and evaluated in an Apache Hadoop environment. Multithreading was used within the program whenever possible in order to reduce execution time. In addition, Hive jobs in Hadoop were run using the Tez execution engine instead of MapReduce as it provides better performance. By using Hive as the gateway to access the data in Hadoop, a solution has been implemented based on an SQL-like query language (HiveQL) that provides data query, summarization, and analysis.

The tool was developed to create an efficient solution for companies to validate their data by providing them with a holistic view of the changes that occurred based on any modifications in their workflows. This was achieved by generating an overall statistical summary on both row and column levels for all tables, thus allowing for accurate and more agile data transformation updates. As a consequence, the results should guarantee both data correctness and accuracy, which are the data quality measures that are considered in the evaluation of this project.
4 Implementation

In this chapter, the implementation of the big data validation tool Diftong is presented alongside the different steps that are necessary to perform during the validation process. This process is explained with the help of a recurrent example that is used throughout this chapter. The results of this implementation are also displayed.

4.1 Validation process

The proposed validation process consists of three main steps: deduplication, row validation, and column validation. A description of each step together with an example is listed below.

In this example, two databases “DB1” and “DB2” with identical schemas will be validated; they contain a table named “Users” with four columns (id, name, salary, and birthday) as displayed in Table 4.1.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>salary</td>
<td>int</td>
</tr>
<tr>
<td>birthday</td>
<td>timestamp</td>
</tr>
</tbody>
</table>

Table 4.1: Original databases schemas - “Users” table

Tables 4.2 and 4.3 show the data in the “Users” table in databases “DB1” and “DB2”
respectively.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>25000</td>
<td>1990-08-07 08:00:00</td>
</tr>
</tbody>
</table>

Table 4.2: Data in “Users” table - DB1

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>3</td>
<td>sofia</td>
<td>25500</td>
<td>1990-08-07 08:30:00</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
</tr>
</tbody>
</table>

Table 4.3: Data in “Users” table - DB2

The validation steps are described below. A reference for all HiveQL scripts that are listed in the following sections can be found in the Apache Hive SQL language manual [2].

4.1.1 Deduplication

Unlike RDBMS, the non-relational model is not dependent on primary keys [37]. Thus, rows are not allocated with a unique number and there is a possibility to have duplicated data. A duplication in data is where all values in one row match exactly all the values in another row. In order to perform a correct comparison between two databases, any duplication in the data should be taken into account. Otherwise, having a different number of rows for the same record in the two databases will not be detected and will produce incorrect results. Therefore, each row in both databases
is counted and the results are stored for use in subsequent stages of the validation process.

Based on the data in the “Users” table that is shown in both Tables 4.2 and 4.3, all records appear once in each table except for the record with the \( id = 2 \) in “DB2”. This record has two identical rows where the values of all columns are equal as shown in Figure 4.1. Hence, the “count” for all the rows is 1 except for the record with the \( id = 2 \) in “DB2”, which is 2 instead.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>25000</td>
<td>1990-08-07 08:00:00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
</tr>
<tr>
<td>3</td>
<td>sofia</td>
<td>25500</td>
<td>1990-08-07 08:30:00</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
</tr>
</tbody>
</table>

Figure 4.1: Count of records in “Users” table in “DB1” and “DB2”

After the deduplication step, two new databases are created. Each database contains the same tables and data as the original databases before deduplication, but with one additional column “row_count” that shows the number of times each row appears in a table. Table 4.4 shows the schema of the databases “DDB1” and “DDB2” - deduplicated copies of “DB1” and “DB2” respectively - while Tables 4.5 and 4.6 show their data.
Table “Users”

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>salary</td>
<td>int</td>
</tr>
<tr>
<td>birthday</td>
<td>timestamp</td>
</tr>
<tr>
<td>row_count</td>
<td>int</td>
</tr>
</tbody>
</table>

Table 4.4: Deduplicated databases schema - “Users” table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>25000</td>
<td>1990-08-07 08:00:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.5: Data in “Users” table - DDB1

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>sofia</td>
<td>25500</td>
<td>1990-08-07 08:30:00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.6: Data in “Users” table - DDB2

The results show that the different number of records for the user with $id = 2$ is detected. Without deduplication, the row validation algorithm that is executed as the next step in the validation process would conclude that the data related to that user in “DB1” and “DB2” are identical, which would be wrong, as “DB2” contains duplicate entries of that user, which need to be accounted for. To avoid such false positives, it is important that databases are deduplicated prior to row validation.
The implementation steps for the deduplication stage are displayed in more details in the following sections.

**Creating the deduplicated databases**

First, a deduplicated copy of each database is created. The general template of the HiveQL script that is used to create an empty database is shown in Script 4.1:

```hive
CREATE DATABASE IF NOT EXISTS {database_name}
COMMENT {database_comment}
LOCATION {database_location};
```

*Script 4.1: Template for creating a database*

Then, in order to create all related tables in the newly created databases, there is a need to extract all table- and column names from the original databases. This can be done using Scripts 4.2 and 4.3 where the “show tables” script lists all table names in the specified database and the “show columns” script shows the list of column names for the given table.

```hive
SHOW TABLES IN {database_name};
```

*Script 4.2: Template for obtaining database tables*

```hive
SHOW COLUMNS IN {database_name}.{table_name};
```

*Script 4.3: Template for obtaining table columns*

Using the extracted table and column values, a dynamic HiveQL script is generated for each table taking into consideration the counting of all rows and saving the calculated value in a new column called “row_count” as displayed in Script 4.4. By using “count” and “group by” in this script, all rows that contain similar values will be represented in the new deduplicated database using only one row with the
extra column “row_count”. Thus, it follows that all duplications are eliminated.

```
CREATE TABLE IF NOT EXISTS {deduplicated_database_name}.{table_name} AS
SELECT {columns_names}, COUNT(*) AS row_count
FROM {original_database_name}.{table_name}
GROUP BY {columns_names};
```

Script 4.4: Template for creating a table in the deduplicated database

**Counting rows optimization in Hive**

As previously stated in the *Theoretical Framework* chapter, the number of rows “count(*)” in each table can be obtained faster if stored in the Hive metastore. To achieve that, both “hive.stats.autogather” and “hive.compute.query.using.stats” configurations have to be set to “true” as shown in Script 4.5. In addition, the “analyze table” query that is described in Script 4.6 is run on each table for more advanced statistics collection.

```
SET hive.stats.autogather=true;            -- Default value: true
SET hive.compute.query.using.stats=true;  -- Default value: false
```

Script 4.5: Setting Hive configuration to optimize statistics collection

```
ANALYZE TABLE {database_name}.{table_name} COMPUTE STATISTICS;
```

Script 4.6: Template for analyzing a table

**Example script**

For further details, the script for creating the deduplicated database “DDB1” of “DB1” is listed in Appendix A.1. Note that the code is similar for “DDB2”, the deduplicated copy of “DB2”.

24
4.1.2 Row validation

In order to get an overview of the total number of changes in each table alongside the difference percentage, the differences between the rows of all tables in the two deduplicated databases are calculated. Consider the case of two databases “DB1” and “DB2”, deduplicated respectively into “DDB1” and “DDB2”. The following statistics are generated in this step:

- Number of rows in DB1 (4.1)
- Number of rows in DB2 (4.2)
- Absolute difference of DB1 and DB2 (4.3)

- Number of rows in DDB1 (4.4)
- Number of rows in DDB2 (4.5)
- Absolute difference of DDB1 and DDB2 (4.6)

- Number of differences in row values between DDB1 and DDB2 (4.7)
- Percentage of differences in row values between DDB1 and DDB2 (4.8)

To calculate the number of differences in the data values of each row (4.7) with the related percentages (4.8), there is a need to detect which records in each table from “DDB1” have been changed in “DDB2” and what those changes are.

Figure 4.2 illustrates the differences in the values of all rows and columns in the “Users” table. It can be seen that five rows out of seven are different between “DDB1” and “DDB2”, thus the difference percentage is 71.4%.
Figure 4.2: Row differences between “Users” table in “DDB1” and “DDB2”

As a result, Table 4.7 displays the calculated row statistics for “Users” table and depicts the number of changes on a row-based level with the difference percentage.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>36000</td>
<td>1982-12-16 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>27000</td>
<td>1990-08-07 08:00:00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.7: Row validation statistics - “Users” table

A detailed description of how those row statistics “from (4.1) to (4.8)” are calculated is provided in the following section.
**Row statistics: (4.1), (4.2), (4.4), and (4.5)**

The HiveQL template shown in Script 4.7 counts the number of rows in a table. It is used to calculate the total number of rows in all tables of the databases “DB1”, “DB2”, “DDB1”, and “DDB2”.

```
SELECT COUNT(*) FROM {database_name}.{table_name};
```

Script 4.7: Template for obtaining the count of rows in a table

**Row statistics: (4.3) and (4.6)**

Statistics (4.3) and (4.6) reflect the absolute difference of the number of rows between (“DB1”, “DB2”) and (“DDB1”, “DDB2”) respectively. The absolute value is calculated using $abs$ function in Hive as per Script 4.8.

```
SELECT ABS({table1_rows_count} - {table2_rows_count});
```

Script 4.8: Template for obtaining the absolute value of the difference between two numbers

**Row statistics: (4.7)**

To calculate the total number of differences in the data of each table in “DDB1” and “DDB2”, there is a need to compare the values in each row and detect any potential changes. Those differences and changes are referred to as the “delta” values. The differences are stored in row delta tables that help in generating the related statistics at a later step in this validation stage. The chosen method to store the delta tables is to create a new database that is used for both row and column validation stages. Script 4.1 can be used to create the validation database.
One way to reflect changes in data is to represent each change as a record in the delta table. Each record contains the affected values with a validation status. The validation status can be either “insert” or “delete” where:

- **Insert**: is when there is a new record in “DDB2” that does not exist in “DDB1”.
- **Delete**: is when a record exists in “DDB1” but not in “DDB2”.

Note that the “update” status is actually a “delete” operation followed by an “insert” of the new values.

Table 4.8 displays an example of a possible representation of the differences that are shown in Figure 4.2 for “Users” tables in both deduplicated databases, along with the related delta validation status.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
<th>birthday</th>
<th>row_count</th>
<th>validation_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>1</td>
<td>DELETE</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>2</td>
<td>INSERT</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>25000</td>
<td>1990-08-07 08:00:00</td>
<td>1</td>
<td>DELETE</td>
</tr>
<tr>
<td>3</td>
<td>sofia</td>
<td>25500</td>
<td>1990-08-07 08:30:00</td>
<td>1</td>
<td>INSERT</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
<td>1</td>
<td>INSERT</td>
</tr>
</tbody>
</table>

Table 4.8: Validation status for row delta table - “Users” table

In other words, and based on the validation status of each record of the row delta calculation, applying the changes in sequence on the related table in the first deduplicated database will result the data of the same table in the second one.

In order to implement the previous representation using Hive, delta values are calculated and stored in the newly created delta tables based on the Script 4.9. The row delta table contains all columns from both tables that are being compared along with the validation status that indicate what kind of change (“insert” or “delete”) has occurred on the data in those tables. Table 4.9 shows the schema of
the newly created validation database “ValidationDB” with the “Users_row_delta” table.

In the Script 4.9, a full join between the tables in “DDB1” and “DDB2” is applied based on a unique value that identify each record, and the related validation status is then calculated. Lastly, the result is filtered to get the rows that are only represented in either table.

Note that since there are no primary keys to identify the rows in big data, a new hash function (UDF) - called “multi_hash” - was implemented to generate a unique identifier for each row based on the values in all its columns. The hash value that was obtained from this UDF is then used to compare and join similar records in those tables. A detailed description of the implemented “multi_hash” function can be found in Appendix B.1.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>int</td>
</tr>
<tr>
<td>name1</td>
<td>string</td>
</tr>
<tr>
<td>salary1</td>
<td>int</td>
</tr>
<tr>
<td>birthday1</td>
<td>timestamp</td>
</tr>
<tr>
<td>row_count1</td>
<td>int</td>
</tr>
<tr>
<td>id2</td>
<td>int</td>
</tr>
<tr>
<td>name2</td>
<td>string</td>
</tr>
<tr>
<td>salary2</td>
<td>int</td>
</tr>
<tr>
<td>birthday2</td>
<td>timestamp</td>
</tr>
<tr>
<td>row_count2</td>
<td>int</td>
</tr>
<tr>
<td>validation_status</td>
<td>string</td>
</tr>
</tbody>
</table>

Table 4.9: Validation database schema - “Users_row_delta” table
CREATE TABLE IF NOT EXISTS {validation_database_name}.{row_delta_table_name} AS
SELECT ddb1.\{columns_names\},
       ddb2.\{columns_names\},
       CASE WHEN ddb1.hash IS NULL THEN 'INSERT' ELSE 'DELETE' END AS validation_status
FROM
  (SELECT multi_hash(ARRAY(\{columns_names\})) AS hash, *
   FROM \{deduplicated_database1\_name\}.{table_name}\) ddb1
FULL JOIN
  (SELECT multi_hash(ARRAY(\{columns_names\})) AS hash, *
   FROM \{deduplicated_database2\_name\}.{table_name}\) ddb2
ON (ddb1.hash = ddb2.hash)
WHERE -- Look for all the rows that are only represented in either table
       ddb2.hash IS NULL OR ddb1.hash IS NULL;

Script 4.9: Template for creating a row delta table in the validation database

As a result, Table 4.10 displays the data that have been changed with the related validation status column as the result of applying Script 4.9 on the “Users” table. A better representation of the data in “Users_row_delta” table is shown earlier in this section using Table 4.8.

<table>
<thead>
<tr>
<th>id1</th>
<th>name1</th>
<th>salary1</th>
<th>birthday1</th>
<th>count1</th>
<th>id2</th>
<th>name2</th>
<th>salary2</th>
<th>birthday2</th>
<th>count2</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>DELETE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td>Mark</td>
<td>31000</td>
<td>1987-06-23 02:00:00</td>
<td>2</td>
<td>INSERT</td>
</tr>
<tr>
<td>3</td>
<td>Sofie</td>
<td>25000</td>
<td>1990-08-07 08:00:00</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>DELETE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>3</td>
<td>sofia</td>
<td>25500</td>
<td>1990-08-07 08:30:00</td>
<td>1</td>
<td>INSERT</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>4</td>
<td>Anna</td>
<td>29000</td>
<td>1992-09-22 10:15:00</td>
<td>1</td>
<td>INSERT</td>
</tr>
</tbody>
</table>

Table 4.10: Data in “Users_row_delta” table - “ValidationDB”

After getting the data of row validation delta tables, the total number of differences in values of rows between “DDB1” and “DDB2” can be calculated based on counting the number of rows in those row delta tables in the validation database using Script 4.7.
Row statistics: (4.8)

The percentage of the differences in each table equals the result of dividing the total number of differences that are calculated in (4.7) by the sum of the number of rows in both deduplicated databases tables, then multiply the result by 100 and rounding it to one decimal place as per Script 4.10.

```
SELECT ROUND({delta_rows_count} / ({table1_rows_count} + {table2_rows_count}) * 100, 1);
```

Script 4.10: Template for obtaining the percentage of differences

Counting rows optimization in Hive

Similar to what was stated in the deduplication step, running the “analyze table” script that is described in Script 4.6 on all tables in “DDB1” and “DDB2” and on all row delta tables in the validation database helps in optimizing the calculation of row statistics [3]. Note that tables in “DB1” and “DB2” were already analyzed during the deduplication phase.

Example script

For further details, all the scripts needed for performing the row validation step using the previous example are listed in Appendix A.2.

4.1.3 Column validation

Starting from the results of the row validation step, column based statistics are calculated for the tables that contain differences in order to get a deeper insight of the changes that have occurred. It is important to first know how many changes have occurred and in what range do those changes fit. This can be achieved by calculating
the total number of differences along with the minimum and maximum difference in the data. Calculating the average was also considered at the beginning but it did not give much added value to the results, so it was excluded from the generated statistics at a later stage.

Moreover, it is significant to measure the spread of the differences and get a deeper understanding of the data distribution. This can be obtained using the quartile statistics. By dividing the differences into four equal parts, the estimated quartiles and the median of the distribution can be calculated using 25th, 50th and 75th percentiles of the data. Those percentiles reflect the first quartile, the median, and the third quartile respectively. As shown in the Figure 4.3, the lower quartile (Q1) is the middle number between the smallest number and the median. While the middle quartile (Q2) is the median of the data. Finally, the upper quartile (Q3) is the middle value between the median and the highest number in the data set [38].

The inter-quartile range provides a better measure of the overall data spread as it ignores the values that are outside of the expected range, i.e. it is not affected by outliers. This is also one of the reasons why the quartile statistics were included in the results while the standard deviation was excluded, as the latter only measures the spread of the data around the mean value.
Consequently, the following statistics are generated for each column in the tables that are being validated:

- **Total number of differences in data** (4.9)
- **Maximum difference in data** (4.10)
- **Minimum difference in data** (4.11)
- **First quartile (25th percentile)** (4.12)
- **Second quartile (50th percentile) − median** (4.13)
- **Third quartile (75th percentile)** (4.14)

Consider the case of two databases “DDB1” and “DDB2”, deduplicated copies of “DB1” and “DB2”, in addition to a validation database “ValidationDB” that contains the row delta tables from the previous step. In order to generate the previous statistics, there is a need to compare and calculate the delta between each value inside the tables of “DDB1” and the related one in “DDB2”.

Figure 4.4 illustrates the differences between values of the columns in the “Users” table between “DDB1” and “DDB2”. It can be seen that there is one difference in each of the columns:

- **Name**: In the record with \(id = 3\), “Sofie” and “sofia” differ in the first and last letters “S/s” and “e/a”, thus the delta is equal to 2.

- **Salary**: The delta between “25000” and “25500” in the record with \(id = 3\) is equal to 500.

- **Birthday**: There is “1,800,000” millisecond difference between “1990-08-07 08:00:00” and “1990-08-07 08:30:00” in the record with \(id = 3\).

- **Row count**: There is one more row of the record with the \(id = 2\) in “DDB2”.

33
Figure 4.4: Column differences between “Users” table in “DDB1” and “DDB2”

Note that any identical rows in both databases (such as the record with $id = 1$), or rows that only exist in one of the databases (like the record with $id = 4$) will not be a part of this validation stage.

As a result, Table 4.11 displays the calculated column statistics for “Users” table. It shows the total number of differences in the data along with the related maximum and minimum delta values. In addition, it displays the three quartile values of each column. Notice that the numbers in all statistics match in this example due to having a single difference in each column.

Table “Users”

<table>
<thead>
<tr>
<th>Column</th>
<th>Diff. count</th>
<th>Max</th>
<th>Min</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Salary</td>
<td>1</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Birthday</td>
<td>1</td>
<td>1,800,000</td>
<td>1,800,000</td>
<td>1,800,000</td>
<td>1,800,000</td>
<td>1,800,000</td>
</tr>
<tr>
<td>Row count</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.11: Column validation statistics - “Users” table
The implementation of the column validation step starts with comparing the values in each column and calculating the delta between them. The differences are stored in new delta tables that help in generating the related statistics at a later step in this validation stage. Similar to the row validation stage, the column delta tables are stored in “ValidationDB”.

For each record, the differences between all columns are calculated and stored in column delta tables that have the schema shown in Table 4.12.

<table>
<thead>
<tr>
<th>Table “Users_col_delta”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column name</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>name_diff</td>
</tr>
<tr>
<td>salary_diff</td>
</tr>
<tr>
<td>birthday_diff</td>
</tr>
<tr>
<td>count_diff</td>
</tr>
</tbody>
</table>

Table 4.12: Validation database schema - “Users_col_delta” table

Based on the data of row delta tables in “ValidationDB” (Table 4.10 is given as an example), column delta tables are generated using Script 4.11. First, a self join is applied on the row delta tables based on the “id” column of both “DDB1” and “DDB2” tables. Note that the “id” column should be provided by the user due to the lack of primary keys when working with big data. Then, and for each column, the difference between all values of the tables in both deduplicated databases is calculated with the help of a UDF called “calc_diff”. This function takes as parameters two values of the same type (integer, double, string, date, timestamp, or boolean), then calculates and returns the delta between them. A detailed description of the implemented “calc_diff” function can be found in Appendix B.2.
CREATE TABLE IF NOT EXISTS `{validation_database_name}.col_delta_table_name` AS

```sql
SELECT delta1.{column_id},
        calc_diff (delta1.{columns_names}, delta2.{columns_names}) AS {columns_names}.diff
FROM `{validation_database_name}.row_delta_table_name` delta1
JOIN `{validation_database_name}.row_delta_table_name` delta2
ON (delta1.{column_id} = delta2.{column_id});
```

Script 4.11: Template for creating a column delta table in the validation database

As a result, Table 4.13 displays the result of running Script 4.11 on “Users_row_delta” table. Note that NULL is stored if both values that are being compared are identical - i.e. the delta is equal to zero -, or if both inputs are NULL.

<table>
<thead>
<tr>
<th>id</th>
<th>name_diff</th>
<th>salary_diff</th>
<th>birthday_diff</th>
<th>count_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>500</td>
<td>1,800,000</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table 4.13: Data in “Users_col_delta” table - “ValidationDB”

Starting from the values in the column delta table, column statistics “from (4.9) to (4.14)” are generated. A description of how they were generated is provided in the following section.

**Column statistics: (4.9), (4.10), and (4.11)**

Calculating the count of changes along with the minimum and maximum differences for all columns can be costly. Thus, in order to optimize generating those statistics and similar to the approach that is used in the previous validation stages, Hive configurations are set to obtain the values of “min”, “max”, and “count” using the statistics that are stored in the metastore instead of calculating them each time.

Consequently, and in addition to analyzing the tables using Script 4.6 that is described earlier in this chapter, another script - Script 4.12 - is executed to
analyze column delta tables in “ValidationDB” and compute column-level statistics for all existing columns.

```sql
ANALYZE TABLE `{database_name}.{table_name}` COMPUTE STATISTICS FOR COLUMNS;
```

Script 4.12: Template for analyzing a table for columns

As for the next step, executing Script 4.13 on all the columns that hold information regarding the existing differences will generate the needed numbers to be used in the column statistics. The minimum and maximum values of differences are obtained along with the number of null values in each column, which reflects that no changes have been made to the data in the related fields of that column. As for the total number of changes, it is calculated by the subtraction of the number of null values from the total number of rows in the delta table. The latter is generated from executing Script 4.7.

```sql
USE `{database_name}`;
DESCRIBE FORMATTED `{table_name}.{column_name}`;
```

Script 4.13: Template for obtaining column statistics

Column statistics: (4.12), (4.13), and (4.14)

Hive provides a function called “percentile” that can be used to calculate the lowest and highest quartile of a data set along with the median value. This function calculates the specified percentiles for a data set, which is the list of differences for each column in this case. The provided percentile values are (0.25, 0.5, 0.75) as 25%, 50%, and 75% reflect the values of Q1, Q2, and Q3.

Script 4.14 shows the template for obtaining the first, second, and third quartile. Note that the values in the columns should be converted to an “integer” type as
the percentile function does not support floating-point types. A detailed
description of the “percentile” function can be found in Appendix B.3.

| SELECT percentile(CAST(column_name AS BIGINT), ARRAY(0.25, 0.5, 0.75)) 
FROM {database_name}.{table_name}; |

Script 4.14: Template for obtaining first, second, and third quartiles

Example script

For further details, all the scripts needed for performing the column validation step
using the previous example are listed in Appendix A.3.

4.2 Diftong validation tool

_Diftong_ is a big data validation tool implemented based on the validation process
described in the section above. The tool has many features that create a solution
for the validation of big data. It helps organizations to ensure the correctness of
their transformations by detecting any changes in the generated data of different
versions of the same workflow. In a more general context, the tool compares any two
databases that have the same schemas and highlights the differences and similarities
between them.

As previously stated, the tool was implemented using Java and Hive version 1.2
and evaluated using the Hadoop framework. Nevertheless, and even though the
tool was developed for the validation in a big data environment, the proposed
validation process is generic and can be easily tailored to other table-based
database management systems.

The advantage of using HiveQL to implement the core functionalities of the system
makes the generated scripts reusable in any framework that supports SQL. For
instance, the validation process can be applied in RDBMS by ignoring any Hive specific configurations and taking into consideration the constraints that are posed in the relational data models. Since RDBMS ensures the uniqueness of the records in the tables, there is no need for deduplicating the data prior to the calculation of the statistics, or using a hash function to generate identifiers for rows, as primary keys can be simply used in this case. Consequently, small alterations to the scripts will allow the data validation tool to be configured to more traditional hierarchical databases, thereby demonstrating the elegance and flexibility of the HiveQL platform.

Diftong is easy-to-use and provides the ability to customize the validation process. This is achieved by giving the user the option to choose which steps to execute and which tables and/or columns to include or exclude from the validation. In addition, the generated row- and column based statistics are exported to three different formats: text files, comma-separated values (CSV) files, and Elasticsearch analytics engine [6] where Kibana - an analytics and visualization platform - is used to visualize the results [7] with the help of tables and graphs.

To conclude this chapter, row-based and column-based statistics are generated by applying the aforementioned validation process. Row validation gives an overview of the total number of differences in each table with the difference percentage. Column validation, on the other hand, generates more detailed statistics such as the number of differences and the minimum and maximum difference in the data in each column. It also calculates the first quartile, the median, and the third quartile of the data. The analysis of those statistics is discussed in the next chapter.
5 Test Cases and Analysis

In order to make sense of the comparison results of two different versions of a transformation, a fourth step should be added to the three validation stages (deduplication, row validation, and column validation) that are discussed in the Implementation chapter. This step includes the analysis of those results, which is needed to help understanding the generated statistics and examining the correctness and accuracy of the data.

For the purpose of the analysis, two test cases were conducted on a test database with the size of 1TB. TPC-H is a decision support benchmark [34] that was used in the validation of the results of this study. A Hive testbench [15] that provides experiments with Apache Hive at any data scale was used to generate and populate a test database based on the TPC-H benchmark with a scale factor equal to 1000 (1TB). The TPC-H database consists of eight tables (Customer, LineItem, Nation, Orders, Part, PartSupp, Region, and Supplier) that contain several billions of records. A detailed description of the schema of this database and the generated test data can be found in Appendix C.1.

5.1 First test case

The Diftong tool was used to compare the previously stated test database - named “TPCH” - with an identical copy of it named “TPCH_copy”. The data in both databases were first deduplicated, then the row validation stage was executed. As expected, no differences were found in any of the tables as displayed in Table 5.1.
Table 5.1: Row validation statistics for two identical databases

<table>
<thead>
<tr>
<th>Table</th>
<th>TPCH</th>
<th>TPCH_copy</th>
<th>Δ</th>
<th>Deduplicated TPCH</th>
<th>Deduplicated TPCH_copy</th>
<th>Δ</th>
<th>Diff. count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>150,000,000</td>
<td>150,000,000</td>
<td>0</td>
<td>150,000,000</td>
<td>150,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LineItem</td>
<td>5,999,989,709</td>
<td>5,999,989,709</td>
<td>0</td>
<td>5,999,989,709</td>
<td>5,999,989,709</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nation</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Orders</td>
<td>1,500,000,000</td>
<td>1,500,000,000</td>
<td>0</td>
<td>1,500,000,000</td>
<td>1,500,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Part</td>
<td>200,000,000</td>
<td>200,000,000</td>
<td>0</td>
<td>200,000,000</td>
<td>200,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PartSupp</td>
<td>800,000,000</td>
<td>800,000,000</td>
<td>0</td>
<td>800,000,000</td>
<td>800,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Region</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Supplier</td>
<td>10,000,000</td>
<td>10,000,000</td>
<td>0</td>
<td>10,000,000</td>
<td>10,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

As a result, there is no need to run the column validation stage in this case. However, and in order to fulfill the testing purposes of this test case, the column validation step was executed anyway based on the results of the previous stages. Once again the tool behaved as expected by detecting that no differences were found and aborting the execution after displaying an information message to the user stating the reason behind not continuing with the execution of this stage.

5.2 Second test case

The following modifications were made to the data in the “TPCH_copy” database before executing the tool again:

- **Table “LineItem”**:  
  - **New rows**: One new row was added.
  - **Duplication**: Four rows were duplicated, the first was duplicated once, the second twice, the third three times, and the fourth ten times.
  - **Column “L_returnflag”**: According to the specification of the TPC-H database [34], this column can contain one of the following values: “R”,

41
“A”, or “N”. All “R” values were uncapitalized and replaced by “r”. The number of rows that contain this value is 1,480,675,200.

- **Table “Orders”:**
  - **Column “O_orderdate”:** All dates in this column were shifted one day forwards.

## 5.2.1 Row statistics

Table 5.2 displays the new generated row statistics. Notice that there is a 17-rows difference (1 new row + (1 + 2 + 3 + 10) duplicated rows) in the “LineItem” table between the two databases before the deduplication step is applied. This number decreases to 1 (only the new row) when it reflects the delta between row numbers in both deduplicated databases.

<table>
<thead>
<tr>
<th>Table</th>
<th>TPCH</th>
<th>TPCH_copy</th>
<th>Δ</th>
<th>Deduplicated TPCH</th>
<th>Deduplicated TPCH_copy</th>
<th>Δ</th>
<th>Diff. count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>150,000,000</td>
<td>150,000,000</td>
<td>0</td>
<td>150,000,000</td>
<td>150,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LineItem</td>
<td>5,999,989,709</td>
<td>5,999,989,726</td>
<td>17</td>
<td>5,999,989,709</td>
<td>5,999,989,710</td>
<td>1</td>
<td>2,961,350,409</td>
<td>24.7</td>
</tr>
<tr>
<td>Nation</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Orders</td>
<td>1,500,000,000</td>
<td>1,500,000,000</td>
<td>0</td>
<td>1,500,000,000</td>
<td>1,500,000,000</td>
<td>0</td>
<td>3,000,000,000</td>
<td>100</td>
</tr>
<tr>
<td>Part</td>
<td>200,000,000</td>
<td>200,000,000</td>
<td>0</td>
<td>200,000,000</td>
<td>200,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PartSupp</td>
<td>800,000,000</td>
<td>800,000,000</td>
<td>0</td>
<td>800,000,000</td>
<td>800,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Region</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Supplier</td>
<td>10,000,000</td>
<td>10,000,000</td>
<td>0</td>
<td>10,000,000</td>
<td>10,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Row validation statistics for two non-identical databases

As was explained in the previous chapter, the total number of differences can be calculated based on the number of changes that have occurred. In addition to the new row and the four duplications in rows, 1,480,675,200 values from the “TPCH” database were changed in “TPCH_copy” because of the modification that was undertaken in the column “L_returnflag”. Thus, the total number of rows that are
different in both databases is equal to:

\[
(1480675200 \text{ changed values} + 4 \text{ rows without deduplication}) \text{ in TPCH} + \\
(1480675200 \text{ changed values} + 4 \text{ deduplicated rows} + 1 \text{ new row}) \text{ in TPCH_copy} \\
= 2961350409
\]

Based on this result and on the number of rows in both deduplicated databases, the difference percentage can be calculated as follows:

\[
\frac{2961350409}{(5999989709 + 5999989710)} * 100 = 24.7\%
\]

Similarly, it can be seen that there is 100% difference regarding the table “Orders” while no new rows were added and no rows were duplicated. That is, all existing rows in this table in the “TPCH” database are different from the related ones in the “TPCH_copy” database, and this is due to the change that was made on all dates in the “O_orderdate” column.

Figures 5.1 and 5.2 display the results of the row validation step in graphical interfaces as a part of the outputs of the Diftong tool. As the results indicate, there is clearly a need to execute the column validation step in both tables “LineItem” and “Orders” in order to get more detailed insights regarding the changes that have occurred.
5.2.2 Column statistics

The outcome of the row validation stage of the tool detects which columns should be analyzed and investigated by the company to make any further required corrections. In Table 5.3, the total number of differences that were caused by uncapitalizing a
letter in the “LineItem” table is reflected, with a minimum and maximum of 1, which means that only one letter has changed. In addition, the four duplication operations are also listed with a range from 1 to 10. The quartile values here give a good measure of the distribution of data, as it shows that smaller number of row duplications have occurred in general even if the maximum value is 10, which is correct considering how data was spread in this example (1, 2, 3, and 10).

<table>
<thead>
<tr>
<th>Column</th>
<th>Diff. count</th>
<th>Max</th>
<th>Min</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.returnflag</td>
<td>1,480,675,200</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>row_count</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>1.75</td>
<td>2.5</td>
<td>4.75</td>
</tr>
</tbody>
</table>

Table 5.3: Column validation statistics - “LineItem” table

As for the “O.orderdate” column in the “Orders” table, it can be noticed from Table 5.4 that even though all columns were affected by a change, this change was stable and evenly distributed. The number of milliseconds reflects that the dates were shifted one day forwards in the “TPCH_copy” database. Those kind of changes in the data that are related to timestamps might be linked to a certain event in time that will vary depending on different circumstances, which is also similar to the case of having a column with random numbers. If this is the case, considering the differences in such columns might not give any added value, and thus they can be excluded from the validation process. Excluding the “O.orderdate” column in this example will make the difference percentage disappear.

<table>
<thead>
<tr>
<th>Column</th>
<th>Diff. count</th>
<th>Max</th>
<th>Min</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.orderdate</td>
<td>1,500,000,000</td>
<td>90,000,000</td>
<td>82,800,000</td>
<td>86,400,000</td>
<td>86,400,000</td>
<td>86,400,000</td>
</tr>
</tbody>
</table>

Table 5.4: Column validation statistics - “Orders” table

Figures 5.3, 5.4, and 5.5 show the results of the column validation stage of the
Diftong tool for both tables “LineItem” and “Orders”.

Figure 5.3: Column validation chart for the difference count

- X-axis: Difference count
- Y-axis: Column name

Figure 5.4: Column statistics chart for the “LineItem” table

- X-axis: Statistics
- Y-axis: Column name

- Max difference
- Min difference
- Q1
- Q2
- Q3

46
5.3 Performance measurements

The performance of the validation process was evaluated by measuring both run-time and CPU-time. The times were measured by summing the duration of executing all HiveQL queries that are described in the previous chapter of this document. Tables 5.5 and 5.6 display the times for both aforementioned test cases - that validate two databases with the size of 1TB each - in milliseconds, noting that those numbers are approximate and they might slightly defer each time a task is executed as the execution depends on the availability of the compute resources and the cluster workload.
By converting the total numbers displayed in those tables to hours, it can be noted that the first test case took about \textit{1.6 hours} and \textit{324 CPU-hours} comparing to \textit{2 hours} and \textit{409 CPU-hours} in the second test case. Due to the fact that the CPU-time is different than the run-time, and in order to get a better estimation of the meaning of those numbers, there is a need to relate them to the duration of running a simple query in the same environment. Therefore, a query to count the rows in all tables in both databases (total of 16 tables) was executed using Tez execution engine and it took \textit{12,643,883 CPU-milliseconds} to finish (about \textit{3.5 CPU-hours}).

It is notable from both tables that the measurement of the deduplication stage is almost similar in both test cases as the same database structures were used and only few more rows were added to the data in the second case. However, the row validation step took a longer time comparing to the first case since differences were detected.
and row statistics were calculated and generated. As for the column validation step, there was no need to execute it in the first test case since no differences were found, while statistics were generated for both “LineItem” and “Orders” tables in the second case resulting in more time being added to the total number of milliseconds.

Moreover, by analyzing and comparing the numbers in the two columns “First test case” and “Second test case”, it can be noted that up to 80% of the time is devoted to the deduplication stage, plus creating and populating row delta tables in the row validation stage in order to detect whether any differences have occurred or not. The cost of those two operations come from the fact that deduplicating a table means that all rows in the table are grouped on all columns using the Hive group by operator, while row validating two tables (A and B) means selecting all rows in A that are not in B and all rows in B that are not in A; row validation was implemented using the Hive full outer join operator. Investigating and evaluating more efficient execution strategies for the deduplication and row validation stages would be interesting but was outside the scope of this study. Due to the optimizations that were made while calculating both row- and column statistics, the total time for generating those statistics was relatively short with only 20% (approximate of 10% each) of the whole execution time.

5.4 Data correctness and accuracy

This chapter has shown that analyzing the generated statistics is required to get a clear idea on what changes have occurred. The two test cases have demonstrated how the tool was able to detect every change that has happened, providing results that help ensuring both data correctness and accuracy. Correctness is achieved by giving the users the ability to prevent any unwanted alternatives, accept other changes based on updated business models, or ignore some of the differences depending on the applied context. Data accuracy, on the other hand, is measured by comparing the data in one database to another source that holds values that are accepted as
being true, pointing out the differences and changes that have occurred.
6 Discussion and Conclusions

This chapter is devoted to summarize the findings of this study and answering the research questions posed at the start of the paper. Furthermore, it includes reflections and some discussions on the limitations of the project and proposes possible improvements.

6.1 Summary of the findings

Overall, this dissertation aims to provide a new solution for validating an extraordinary amount of data. The first two research questions stated in the Introduction chapter of this paper were:

1. What steps are needed to validate big data in terms of comparing two different databases with identical schemas?

2. How can the validation process be automated?

The main result of this study is a blueprint for a validation process to help maintain the quality of big data. This is achieved by first comparing and detecting differences in two databases that hold the outputs of various versions of workflows, followed by a manual analysis of the results to decide on the correctness and accuracy of the new version of the transformations.

The first research question is answered by three validation stages (deduplication, row validation, and column validation) that generate row- and column based statistics. Row statistics give an overview of the data in all rows and identify the total number of differences in each table with the difference percentage. Detecting the differences and similarities on a row-based level gives the user a clear idea if any changes have
occurred. If this is the case, the user can ask for a deeper level of statistics based on each column in that table. Column statistics provide a more detailed view of the changes in data. The number of differences is calculated for each column along with the minimum and maximum difference in the data, which gives an insight about the range of differences that have occurred. In addition, the quartile values (lower, middle, and upper) give more understanding of the distribution of data by measuring how differences are spread.

Moreover, and based on the need to automate as much of the validation process as possible, a new validation tool that provides overall statistics of the differences between two databases that share the same schemas was implemented. The automation of the process eliminates the time-consuming manual labour and the risk of human error, which enables a more agile way of updating and improving big data transformations in general. Hence, given that the tool operates with minimal user inputs, it follows that the second research question is answered; an automated validation process can be done by abiding by the three aforementioned validation stages.

The findings show that identifying the differences and similarities between two databases helps organizations manage their large amount of data in a faster and more accurate way, with the help of automated validation tools that are capable of handling the exploding size of data and that become necessary for going forward.

The third research question was:

3. Can the validation process provide a generic mechanism for validating different types of databases?

The results show that the proposed process is not only limited to big data and a generic mechanism can be made for this validation process as it can be applied to any type of databases that supports SQL, such as the traditional relational data model. Using HiveQL, that provides standard SQL functionality, as the main development tool in this project delivers backwards compatibility and allows for the
interoperability between various database management systems, taking into consideration that some of the implementation steps might not always be needed depending on the characteristics and needs of each system.

In other words, the validation process and the related implementation provide a general approach that can be re-used and run in the same way in different environments that support the structure of tables, rows, and columns. The importance of this type of generalization comes from the fact that SQL is an abstract declarative language that deals with the data as logical objects regardless of their physical storage. SQL is still considered as the most effective way to manage data and still the preferred solution to various types of applications, even with the emergence of many other alternatives that store the data in different formats such as NoSQL databases.

Even though NoSQL databases were found to overcome some of the disadvantages of RDBMS, they do not extinguish the need for SQL databases and will not replace them. In fact, many NoSQL databases work with a database built on SQL such as OrientDB, or come with the support for SQL, like MongoDB and Hadoop, by implementing services that allow users to access the data using SQL syntax. As a result, the solution provided in this study can be generalized to a majority of database systems that still depend on SQL as a core function for the management of the data.

6.2 Limitations and future work

Apart from the current results of the implemented tool, there is still room for improvements. The performance measurements showed that a significant amount of time is devoted to performing the deduplication stage; as it is considered a crucial step before statistics generation can proceed, it follows that all run times will be correspondingly lengthy. Indeed, if the implementation of a workflow consumes a matter of hours, but the validation stage, given a large enough data set, may take
days, one would have to omit the validation stage entirely so that the implementation stays relevant. One option would be to exclude deduplication from the validation process, but as explained in the implementation of the tool, it would hardly be a valid one due to the chance of producing inaccurate results. It is unclear, at this stage, whether or not it would be possible to optimize deduplication in such a way that execution times can improve while maintaining correctness in the data.

Furthermore, it might be possible to add additional measures that can give the users a broader view of the changes in their data. For instance, one of the reasons to include the difference count, minimum, and maximum values is the ease of access to those statistics that are already calculated and stored in the metastore. Including values such as the mean and standard deviation, for example, is not an action available through the metastore, and would negatively impact execution time further, a problem already prevalent due to the high cost of deduplication. However, adding the average could provide more context to the meaning behind the quartile values included in the column validation stage. It stands to reason then, that future work in this area could be expanded to encompass additional descriptive statistics.

In addition, future investigations might be conducted to see to what extent the proposed solution in this study can be used in other types of databases that do not have a table-based structure. One could make the argument that row- and column validation are de facto steps concerning tables and cannot by definition be applied to databases that are dependent on neither columns nor rows. As such, future research needs to account for this when implementing these validation processes and statistics on those types of databases.

Moreover, further research should be undertaken to investigate how the tool can be used in regards to maintaining the data quality, by conducting more tests and measuring to what degree it preserves each of the data quality aspects, including but not limited to data consistency, completeness, accessiblity, and scalability.
References


A Appendix: Generated HiveQL Script for a Data Validation Example

Reference to the example listed in the Implementation chapter, the generated SQL script based on the two databases “DB1” and “DB2” that are being compared and validated is displayed in this appendix.

A.1 Deduplication

The generated scripts for creating deduplicated database “DDB1” of “DB1” are as follows:

--- Create deduplicated database DDB1
CREATE DATABASE IF NOT EXISTS DDB1
COMMENT 'Diftong deduplication of DB1'
LOCATION '/warehouse/DDB1';

--- Get the names of all tables in DB1
SHOW TABLES IN DB1;

--- Get the names of all columns in Users table in DB1
SHOW COLUMNS IN DB1.users;

--- Analyze Users table in DB1
ANALYZE TABLE DB1.users COMPUTE STATISTICS;

--- Create Users table in DDB1
CREATE TABLE IF NOT EXISTS DDB1.users AS
SELECT id, name, salary, birthday, COUNT(*) AS row_count
FROM DB1.users
GROUP BY id, name, salary, birthday;

A.2 Row validation

The generated scripts for performing the row validation step in this example are as follows:

-- Create validation database ValidationDB
CREATE DATABASE IF NOT EXISTS ValidationDB
COMMENT 'Diftong validation database'
LOCATION '/warehouse/ValidationDB';

-- Create Users_row_delta table in ValidationDB
CREATE TABLE IF NOT EXISTS ValidationDB.users_row_delta AS
SELECT ddb1.id AS id1, ddb1.name AS name1, ddb1.salary AS salary1,
       ddb1.birthday AS birthday1, ddb1.row_count AS row_count1,
       ddb2.id AS id2, ddb2.name AS name2, ddb2.salary AS salary2,
       ddb2.birthday AS birthday2, ddb2.row_count AS row_count2,
       CASE WHEN ddb1.mhash IS NULL THEN 'INSERT' ELSE 'DELETE' END AS validation_status
FROM
    (SELECT multi_hash(ARRAY(CAST(id AS STRING), CAST(name AS STRING),
                              CAST(salary AS STRING), CAST(birthday AS STRING),
                              CAST(row_count AS STRING))) AS mhash, * FROM DDB1.users) ddb1
FULL JOIN
    (SELECT multi_hash(ARRAY(CAST(id AS STRING), CAST(name AS STRING),
                              CAST(salary AS STRING), CAST(birthday AS STRING),
                              CAST(row_count AS STRING))) AS mhash, * FROM DDB2.users) ddb2
ON (ddb1.mhash = ddb2.mhash)
WHERE ddb2.mhash IS NULL OR ddb1.mhash IS NULL;

-- Calculate row statistics for Users table:
A.3 Column validation

The generated scripts for performing the column validation step in this example are as follows:

```
-- Create Users column delta table in ValidationDB
CREATE TABLE IF NOT EXISTS ValidationDB.users_col_delta AS
SELECT delta1.id_1 AS id,
       calc_diff (delta1.name_1, delta2.name_2) AS name_diff,
```


```
calc_diff (delta1.salary_1, delta2.salary_2) AS salary_diff,
calc_diff (delta1.birthday_1, delta2.birthday_2) AS birthday_diff,
calc_diff (delta1.row_count_1, delta2.row_count_2) AS row_count_diff
FROM ValidationDB.users_row_delta delta1
JOIN ValidationDB.users_row_delta delta2
ON (delta1.id_1 = delta2.id_2);

-- Calculate column statistics for Users table:

-- Analyze Users_col_delta table
ANALYZE TABLE ValidationDB.users_col_delta COMPUTE STATISTICS;
ANALYZE TABLE ValidationDB.users_col_delta COMPUTE STATISTICS FOR COLUMNS;

-- Get number of rows in Users_col_delta
SELECT COUNT(*) AS col_delta_count FROM ValidationDB.users_col_delta;

-- Get column statistics for each column in Users_col_delta table
USE ValidationDB;

DESCRIBE FORMATTED users_col_delta.name_diff;
DESCRIBE FORMATTED users_col_delta.salary_diff;
DESCRIBE FORMATTED users_col_delta.birthday_diff;
DESCRIBE FORMATTED users_col_delta.row_count_diff;

SELECT percentile(CAST(name_diff AS BIGINT), ARRAY(0.25, 0.5, 0.75)),
       percentile(CAST(salary_diff AS BIGINT), ARRAY(0.25, 0.5, 0.75)),
       percentile(CAST(birthday_diff AS BIGINT), ARRAY(0.25, 0.5, 0.75)),
       percentile(CAST(row_count_diff AS BIGINT), ARRAY(0.25, 0.5, 0.75))
FROM users_col_delta;
```
B Appendix: Helper Functions

In this appendix, an explanation for the implemented helper functions that are used in the validation stages of big data is provided.

B.1 Hash function

Hive provides a “hash” function that takes many arguments and generates a related integer hash value where the same input always results the same output. Table B.1 represents this function as described in the Hive language manual [2):

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>hash(a1[, a2...])</td>
<td>Returns a hash value of the arguments</td>
</tr>
</tbody>
</table>

Table B.1: Description of the “hash” function in Hive

However, the hash function works with ranges and performs some arithmetic operations on the input values to generate integer output. Therefore, there is a chance of producing the same output value for two different inputs, meaning that there is a probability of a hash collision [26].

An example of a hash collision in Hive hash function is shown below, where applying this function on two different inputs produces the same output ‘-339500666’:

SELECT hash (‘9 SH305EJ5’); => -339500666

SELECT hash (‘9 SH305EIT’); => -339500666

Therefore, the hash values of this function cannot be used as unique identifiers for rows in the Script 4.9 as there are chances of getting duplicates. Since it is important
to avoid collisions in this case and always getting unique results from different values, a new hash function “multi_hash” that tends to generate unique values when given different inputs is implemented and used instead.

“Multi_hash” (described in Table B.2) is a UDF that takes one input argument as an array of strings and returns the SHA-512 hash for the concatenation of all string values in the input array. The package DigestUtils [5] in Java is used for the hashing.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>multi_hash([s1, s2...])</td>
<td>Returns a hash value of the concatenation of all strings in the input array</td>
</tr>
</tbody>
</table>

Table B.2: Description of the “multi_hash” function

First, all values in the array are concatenated taking into account the length of each string. Then, SHA-512 digest is calculated for the concatenated string and the value as a hex string is returned.

Concatenating all strings with the length of each string before applying the hash function is done to prevent any collision that might appear from moving text from one string to the next. Thus, moving any text will not produce collisions since the lengths would change. Note that in case one of the inputs strings is NULL, the value “null0” is added to the concatenated string instead.

As an example of moving texts, applying “multi_hash” function on the array of strings ['hello', 'world!'] results a hex string that is different than the generated hash from the same function for the array ['hellow', 'orld!'], as the length of each word is taken into account.

The results from applying “multi_hash” function on both ['hello', 'world!'] and ['hellow', 'orld!'] are listed as follows:

```
SELECT multi_hash(ARRAY('hello','world!'));
```

65
The concatenation of ['hello', 'world!'] is: 'hello5world!6'

The hash is: a1876e83d340a9b184f9e93980ac03161fb0be42d885ea27b3941eb35c61171405e8340e99c992ed1f15043aebccef34a915c7b26076941b40d6bd042403ab

```
SELECT multi_hash(ARRAY('hello', 'world!'));
```

The concatenation of ['hellow', 'orld!'] is: 'hellow6orld!5'

The hash is: f4b64222fc69dd14c61c26324a3601308fc6c97a7a2b6a9b386e657ae760d308746e3918fcfb4186b0eeef509aa545077910060e6d7a0381f0a85984fd6043bf

## B.2 Delta function

The delta function “calc_diff” is an implemented UDF that takes two input arguments of the same type (integer, double, string, date, timestamp, or boolean) and returns the difference between the two values.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>calc_diff(value1, value2)</td>
<td>Returns the delta between the 2 values</td>
</tr>
</tbody>
</table>

Table B.3: Description of the “calc_diff” function

The following are examples of the result of executing “calc_diff” UDF on two values from the same type. Note that the function returns NULL if both inputs are identical \((\text{difference} = 0)\) or both inputs are NULL.

**Integer**: returns the absolute difference of the two numbers

\[
\text{calc_diff}(16, 10) = 6
\]

**Double**: returns the absolute difference of the two numbers

\[
\text{calc_diff}(5.9, 10.2) = 4.3
\]
**String**: returns the number of characters that does not match in two strings
\[
\text{calc}_\text{diff}(\text{“Christin”, “christian”}) = 3
\]

**Date**: returns the absolute difference of the two dates in millisecond
\[
\text{calc}_\text{diff}(2010/10/16, 2010/10/15) = 86400000
\]

**Timestamp**: returns the absolute difference of the two timestamps in millisecond
\[
\text{calc}_\text{diff}(2010/10/16 12:50:00, 2010/10/16 00:00:00) = 46200000
\]

**Boolean**: returns 1 if there is a difference
\[
\text{calc}_\text{diff}(\text{true, false}) = 1
\]

### B.3 Percentile function

Hive provides a “percentile” UDAF (User-Defined Aggregation Function) that helps in calculating one or more percentiles of a data set. The function takes two arguments as inputs; a column that contains the data set and an array of the percentiles to calculate. It then returns an array of floating-point numbers which represents the result of applying this function on the column values for each of the input percentile respectively. Table B.4 represents the percentile function as described in the Hive language manual [2].

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array of double</td>
<td>percentile(BIGINT col, array(p1 [, p2]...))</td>
<td>Returns the exact percentiles p1, p2,... of a column in the group</td>
</tr>
</tbody>
</table>

Table B.4: Description of the “percentile” function in Hive

Note that the percentile function can only be computed for integer values and does not work with floating-point types. In addition, there are several definitions of percentile; a description of the method that is used in Hive can be found in the Apache Software Foundation (ASF) project [8].
This appendix provides a description of the test data that are used in the Test Cases and Analysis chapter. The results that are generated from running the Diftong tool on these data will help in validating the proposed process of big data validation in this study.

C.1 TPC-H

The Transaction Processing Performance Council (TPC) is an organization that defines transaction processing and database benchmarks [34]. TPC-H is a decision support benchmark that is used in the validation process in this study. This benchmark and the data populating the database illustrates decision support systems that have broad industry-wide relevance, and that helps in examining large volumes of data.

In order to create and populate a test database at a desired data scale, Hive testbench [15] can be used as it provides data generation with a set of queries for experimenting Apache Hive at scale. It allows experiencing Hive performance and configuration on large test data sets.

The components of TPC-H consist of eight tables. The definitions of those tables along with the relationships between the columns are illustrated in the ER diagram that is shown in the Figure C.1. The numbers that are displayed below each table name in this diagram represent the number of rows in that table. Some numbers are factored by the Scale Factor (SF) that reflects the chosen database size as TPC-H supports various data set sizes which can be defined when populating the test database. In this study, the $SF = 1000$ is used when generating the test data, and
which consists of the base row size * 1000 (several billion elements).

More detailed information regarding this test database is available in the TPC Benchmark™ H (TPC-H) specification [34], and an explanation on how to generate and populate databases using Hive testbench can be found in the project documentation [15].

Figure C.1: The TPC-H schema [34]