Data Quality Profile Creation and Management System

Nikolaos Taflampas

DIPLOMA THESIS

Department of Computer Science and Engineering

School of Engineering

University of Ioannina

February 2024

DEDICATION

To my family, for their unconditional love and support, as well as to those who supported me through hardship.

ACKNOWLEDGEMENTS

Many thanks and acknowledgements, first and foremost, to my thesis supervisor, Professor Panagiotis (Panos) Vassiliadis, who greatly assisted me throughout the entire process by always being available for any inquires and issues that came up, and without whose help this thesis would not be complete.

Also, I would like to thank my colleagues and friends, who were always available to discuss any thesis related troubles.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

Nikolaos Taflampas, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, February 2024.
Data Quality Profile Creation and Management System.
Advisor: Panagiotis (Panos) Vassiliadis, Professor.

With the rise in technology and artificial intelligence, data analysis has never been more in the spotlight. And with data analysis being of such importance, more and more analysts have taken interest in data cleaning. The goal of this thesis is the development of a flexible and versatile tool that a data analyst can use to enforce a number of rules on a dataset, regardless of its size, in order to find, isolate and even remove unclean entries.

Our thesis aims to advance the field of data analysis by proposing a structure of requirements, and designing an extensible data cleaning tool. This tool is capable of incorporating tailored data quality checks and seamlessly handling raw datasets. Furthermore, we present a functional data cleaning tool, implemented using Java Spark, that is able to operate over data hosted in a Spark-based environment. Our tool effectively addresses the inherent complexity of data cleaning tasks, and through rigorous experimentation, we evaluate the efficiency of our tool across varying data sizes and levels of pollution, with our findings confirming the anticipated linear performance complexity of the tool, thereby validating its effectiveness.

ΠΕΡΙΛΗΨΗ

Νικόλαος Ταφλαμπάς, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2024.
Σύστημα διαχείρισης και κατασκευής προφίλ ποιότητας δεδομένων.
Επιβλέπων: Παναγιώτης Βασιλειάδης, Καθηγητής.

Με την τεχνολογία και την τεχνητή νοημοσύνη να εξελίσσεται καθημερινώς, η ανάλυση δεδομένων έχει γίνει ένα θέμα μεγάλης σημασίας. Με την ανάλυση δεδομένων να είναι τέτοιας σημαντικότητας, όλο και περισσότεροι αναλυτές ασχολούνται με τον καθαρισμό δεδομένων. Σκοπός της διπλωματικής εργασίας, είναι η ανάπτυξη μιας ευέλικτης και προσαρμόσιμης εφαρμογής, που προσφέρει στον αναλυτή την δυνατότητα να εφαρμόσει λογικούς κανόνες σε δεδομένα ανεξάρτητου μεγέθους, με σκοπό την εύρεση, απομόνωσης και πιθανής διαγραφής ''βρόμικων'' εγγραφών.

Η διπλωματική μας συνεισφέρει στον τομέα της ανάλυσης δεδομένων προτείνοντας μια δομή απαιτήσεων, και σχεδιάζοντας μια επεκτάσιμη εφαρμογή κάθαρσής δεδομένων. Το εργαλείο αυτό είναι ικανό να ενσωματώνει προσαρμοσμένους ελέγχους ποιότητας δεδομένων και απρόσκοπτα να χειρίζεται δεδομένα κάθε μορφής. Περεταίρω, παρουσιάζουμε ένα λειτουργικό εργαλείο, κάνοντας χρήση της βιβλιοθήκης Spark σε Java, πλήρως ικανό να λειτουργήσει με δεδομένα υλοποιημένα σε περιβάλλοντα Spark. Το εργαλείο μας προσφωνεί την εκ φύσεως πολυπλοκότητα της ποιότητας δεδομένων, και μέσω πολλαπλών πειραμάτων προσδιορίζουμε την αποδοτικότητα του εργαλείου μας σε ποικίλων μεγεθών δεδομένων και πολλαπλών επιπέδων μόλυνσης. Μέσω των πειραμάτων μας, επιβεβαιώνουμε την γραμμική πολυπλοκότητα της εφαρμογής μας, και συνεπώς επιβεβαιώνουμε την αποδοτικότητα.

CHAPTER 1

Introduction

1.1 Objective of Thesis

When using data to extract some form of information or conclusion, many will agree that this conclusion will be as good as the quality of the data being used. Faulty, or unclean data produces faulty results, and this is where data cleaning plays its part. Data Cleaning is the process of finding and fixing or removing entries that are deemed unclean in order to maintain a level of cleanliness and order. In this thesis, we will construct a data quality profiling and management system, that attempts to help the data analyst to enforce cleanliness. The complexity in the objective of ensuring data quality, is the abstract nature of data cleaning, wherein no single tool can provide all conceivable quality enforcement options available to a data analyst. In response to this challenge, our application provides a versatile array of data cleaning functionalities, tailored to accommodate diverse analytical requirements. Finally, our tool facilitates effective communication of data cleaning results to the analyst, and prioritizes flexibility to support future improvements in development, as well as advancements in data cleaning methodologies.

The contributions of this Thesis are as follows:

- We structure the requirements and the design of an extensible data cleaning tool, with the possibility of adding tailored data quality checks, and the ability to work with raw data sets in situ.
- We provide a working data cleaning tool, confirming to the aforementioned requirements, based on Java Spark that can operate over data hosted in a Spark-supporting environment
- We experimentally assess the efficiency of the tool with respect to the data volume and pollution and verify the algorithmically expected linearity in its performance through extensive experiments.

1.2 Thesis Structure

Here we will define a brief summary on what we will present within this thesis. In Section 2.1 and 2.2, we confront the problem of data cleaning, clearly define what kind of data we consider as ”unclean”, inspect tools that tackle this problem and analyse research conducted on the subject. In 2.3, we break down the singular functions that our tool will provide, and in chapter 3 form an overall package and class structure that can facilitate future changes, improvements or even additions to our code. Finally, in chapter 4, we put our application to the test, by putting it through several experiments that examine its ability to handle different types of data, such as the number of entries, the size of each entry as well as the uncleanliness of the entire dataset as a whole. Through our experiments, we analyse the overall behaviour of our application on the aforementioned conditions, and conclude on whether these results are ideal and how they can possible be improved upon.

CHAPTER 2

Related Work

In this section, we analyze the problem surrounding data quality, and how we plan to solve it. We examine several tools that try to deal with it, as well as reference related work and research that has been conducted for said problem. Finally, we talk about the specific goals we are trying to achieve, as well as the functionalities we deem necessary for the completion of this thesis.

2.1 Goals and Similar Solutions

On any work that requires handling an amount of data, data quality plays an important role. Whether that be training a machine learning model on a vast amount of information, or even trying to extract some statistical result from some business related data. Chaotic data sets with information that does not follow a specific format, does not adhere to specific rules or even contains false information can have substantial economic and social impacts. With this thesis, we are trying to study the specific problems data quality faces, and produce a software that handles efficiently said problems, and offers quality solutions for resolving them.

With poor data quality being such an important issue in industry, several people have created tools that tackle several issues that data can have. Some examples of tools that ensure the quality of data, and are commonly used in a professional level of data analysis are:

- **SAS Data Quality**
  SAS Institute is a renowned analytics and data management company that developed the SAS Data Quality tool[9]. This tool has several key features that help to improve the quality of any given data. The tool handles the cleaning of data by detecting outliers, duplicate entries and generally non-standard informations. The tool validates data to standard formats (such as emails, dates, addresses etc) or even predefined rules set by the data analyst. Finally, the tool offers powerful data integration techniques with several ETL (Extral Transform Load) and ELT (Extract Load Transform) activities.

- **Spectrum Quality**
  Spectrum Quality[7], similarly to SAS Data Quality, is a data cleaning tool that ensures the quality of any given data. Unlike SAS Data Quality however, Spectrum Quality focuses more on the natural language aspect of its data. The tool utilizes complex machine learning algorithms for pattern recognition and validation, finding outliers in non-numeric data, thus ensuring the validity and completeness of them. While it lacks the vast amount of features provided by its competitors, its specialization lies on breaking down natural language and extracting information from something so complex, making Spectrum Quality a great example of a data quality tool.

- **Ataccama**
  Ataccama[4] provides features of both Spectrum Quality and SAS Data Quality. It utilizes standard or data analyst-set rules to validate data, as well as automatic AI checks for rectifying data errors, inconsistencies and inaccuracies. Furthermore, Ataccama is designed to work with large amounts of data and cloud environments, thus making it an easy to integrate tool.

2.2 Related Work

A lot of research has been conducted on Data Quality. From best practises and standardisation to finding the ideal methods of handling poor quality data. In this section, we look in some of the related works others have produced on the subject.

In [8], the authors talk about the requirements of data quality for industrial or academical usage, as well as the cost of handling poor quality data. They emphasize on the importance of data quality by pointing out the possibility of Machine Learning (ML) systems to reproduce past errors or discrimination against under-represented groups (such as minorities or groups of little data), or even the disturbance of operational efficiency in a business level of usage, which can create an estimate of 10% to 30% of revenue loss in an attempt to resolve any data quality issues.Furthermore, the authors mention how each data analyst that is meant to handle the final data set, has different requirements that are meant to bring all the information to an ideal state. Even when talking about a more industrial/business oriented data quality requirement, while it should certainly follow a specific ISO format, the specifics can differ from person to person when facing a different problem. For example, a ML System that tries to predict the weather, should have a strict requirement in Accuracy, while another system that focuses on word association, may find more merit in Relevance, Timeliness and Appropriateness, especially when it is meant for normal consumers. The authors reference [2], whose authors manage to conclude on 60 different dimensions (or quality requirements) that data can need. We can conclude from this research, that data quality is far more abstract than we expected, making no specific tool the perfect automatic solution towards all problems. A tool that ensures data quality should be flexible, and provide enough utilities to help the data analyst cover all the required dimensions of data quality that they have dictated to be needed for the specific problem at hand.

In [14], the authors examine the uses of metadata for data cleaning. The authors created a mapping between metadata and data quality issues by generalizing them as a set of EBNF rules. The authors presented several heuristics for spotting outliers in both numeric and non-numeric data, and even suggested how metadata can be utilised to detect errors in the quality, such as incorrect patterns, functional dependency violations or even incorrect data that threaten the accuracy. A heavy focus on the Z-Value function has been placed, as well as its usage on the length of

non-numeric values, something which could prove useful for our goals.

Analysis on the use of metadata, and its benefits on managing data, is provided by the authors of [1]. While Data Profiling may not be the same as Data Cleaning/Data Quality, they tend to be heavily influenced by one another. Data Quality and Data Profiling are the two sides of the same coin, making their processes almost too similar in certain matters. The authors reinforce this by examining several data profiling tasks that can be easily used to reinforce the cleanliness of data. Specifically, they break down some classic Data Profiling tasks into three key categories (Single Column, Multi-Column and Dependency). The authors talk about each sub task of each category, and dissect it to find the most efficient algorithms for the problem each task faces. The authors emphasize that all algorithms have their unique benefits, depending on the type of data we are facing, and even talk about some more specific types of data quality that might be less common, such as Partial or Approximate Dependencies. Overall, this research gives us a lot to work with, by guiding us to research further on certain algorithms to produce results more efficiently, and even gives us some ideas for features for our tool to enhance the data analyst's experience.

After reading the documentations of aforementioned tools, we can deduct that a data quality software should be able to:

- Integrate data from multiple formats and storage locations.
- Clean data by exposing duplicates, outliers and threats to data completeness, accuracy and validity.
- Validate data using standards or data analyst defined rules and constraints.
- Recognise common patterns and validate its data against them.

Having defined the key qualities of our software that follow the state of practise, we will rely on **Apache Spark Library** [3] for our system's implementation. This library provides great tools and utilities on managing big data, as well as the ability to transform said data using SQL expressions. It also provides flexibility on the data we can modify, from CSV and TXT files to actual database tables. Furthermore, Spark provides efficient memory usage on all transformations via the use of Resilient Distribution Datasets (RDDs), as well as a complex optimizer to find the correct strategy and minimize execution times. In order to understand how Spark works with data, we need first to understand how RDDs work.

Resilient Distribution DataSets or RDD [12] is a fundamental structure for Spark, which is an immutable collection of data objects spread across several partitions. RDDs provide several features;

- Resiliance: RDDs are fault tolerant, and with the use of a lineage graph, they can recompute missing or damaged partitions due to node failures.
- Distributed: RDDs' data reside on multiple nodes, and each partition can belong to a different server, making them able to be computed on different nodes of the same cluster.
- Cache-able: RDDs provide the ability to be cached, for further improvement in the efficiency of retrieving data.
- Lazy Evaluation: RDDs do not compute the result after a transformation. Instead they remember the transformation plan, until the data analyst calls for the results.
- Immutable: RDDs do not allow changes, as they are a read-only object. Instead, after a transformation, a new RDD is created with the results.

Overall, RDDs are efficient data structures that can handle vast amount of data from different sources, and transform it with ease. They do however come with several disadvantages;

- Unstructured: RDDs do not support any form of data structure or profile and hold no schema.
- Unoptimized: RDDs do not use any optimization technique to transform their data.
- Garbage Limitation: RDDs heavily relly on Java's automatic garbage collector, which can affect the performance on large amounts of data.
- Memory Reliant: While RDDs can handle large data due to the partitions, RDDs heavily rely their efficiency on RAM, making large data as expected less than ideal in terms of speed.

RDDs leave much space for improvement, but set the grounds for several new object that we can use for our data cleaning. Since Spark 1.3, we are introduced to DataFrames, and soon after in version 1.6 of Spark we see DataSets. **DataFrames**[11] are the successor of RDDs, bringing all the benefits of RDDs and much more to the table;

- RDD Including: Data Frames provide Resiliance, Lazy Evalution, Immutability, Cache-ability and even partition distribution, all features that RDDs hold. In fact, Spark provides the ability to create a DataFrame using an RDD.
- Structured Data: DataFrames provide support for both structured and unstructured data by labeling all columns by name.
- Format Integration: DataFrames can be loaded with common files such as CSV or TXT, but also allow interaction and data loading with storage systems (MySQL, Hive etc).
- Optimization: DataFrames use Catalyst, the Spark's SQL optimizer, to transform any data in a more efficient manner, boosting our processing speed and efficient memory usage by a lot.

DataFrames constitute a great advancement over RDDs, albeit not without problems. DataFrames do not have provision for compile time type safety, meaning that if the data structure is unknown, we cannot manipulate the data. At this point, the overall efficiency we can reach with this library has reached its limit, but we can further assure the safety of our data using one final structure: **DataSets**[13].

DataSets are once again an extension of DataFrames and as expected a great evolution of RDDs. They provide all the benefits of DataFrames, but with the added compile-time type safety which can help catch errors at compile time, instead of runtime. This is safety improvement, leaving little room for runtime errors, and overall increasing the chances of a smooth experience. Furthermore, they support transformations from DataSet to both DataFrame and RDD, leaving flexibility to the data analyst. When it comes down to efficiency however, one should consider DataFrames as a superior to DataSets. Even in low-level transformations, RDDs can outperform both of its successors. Therefore, it comes down to what we value most; efficiency or safety. We chose to use DataSets, since the efficiency lost is minimal, and catching errors as soon as possible should be our priority for a more data analyst friendly experience.

	RDD	DataFrame	DataSet
Release Version	1.0	1.3	1.6
Data Format	Unstructured	Structures and Semi-Structured	Structured and Unstructured
Immutability	Fully	Transformations lose original RDD	Original RDD regenerates
Compile-Time-Safety	No	No	Yes
Query Optimization	No	Yes	Yes
Performance	Only low-level	Optimized high-level	Optimized high-level
Data Sources	Files	Files and Storage Systems	Files and Storage Systems
Memory Management	Full Control	Self Optimized	Self Optimized
Interface	Complex, low level	Simple, high level	Rich, OOP friendly, high level

Table 2.1: Data structures' comparison

2.3 Requirement Analysis

In this section, we look into the specific requirements that we need our application to fulfil, for the thesis to be considered complete. Having done our research, and viewed similar tools, we can form some data analyst stories, and some use cases afterwards, to simply dictate the exact services our tool will provide.

2.3.1 Data analyst Stories

We start with our data analyst stories. These stories will help us better define the utilities of our tool, and thus create more structurally accurate use cases.

Data analyst Story	As a..	I want..	So that..
DL1	Data analyst	To register a file of data	It can be later used for cleaning
DL2	Data analyst	To register data from a database table	It can be later used for cleaning
PK1	Data analyst	To test any number of columns in a dataset for uniqueness	I can maintain uniqueness.
FK	Data analyst	To test if any number of columns of one dataset appear in another dataset's column(s)	I can find possible candidate foreign keys.
DC1	Data analyst	To test if a column has only one data type	I can maintain validity
DC2	Data analyst	To know the minimum/maximum of a numeric column	I can get basic information for domain constraints

FC	Data analyst	To test if a column obeys a specific format	I can maintain validity.
NC	Data analyst	To test if a column contains any NULL values	I can maintain completeness
RC	Data analyst	To set group or holistic rules that my data must abide	I can maintain cleanliness.
FR1	Data analyst	To get a detailed report of all rule violations, data quality threats and general info	I know what might be wrong with my data.
FR2	Data analyst	To separate dirty data entries from clean ones	I have a clean batch of data ready for use or take a closer look to troubling data.

Table 2.2: Data analyst Stories

2.3.2 Use Cases

With our data analyst stories complete, we move on to the use cases. Here, we dictate the flow of our application, and how it completes the requirements of each data analyst story, all in a simple generalized manner.

Use Case Identifier: LOADFILE
Flow: 1.The use case start when the Data analyst chooses to register a file, either by selecting a specific button or a specific option. 2. The application prompts the data analyst to select a file from any local directory. 2.1 If the data analyst selects an invalid file format, the application returns the data analyst to the main screen with an error prompt, ending the Use Case. 3. The application creates a DataSet Profile from the file's data and registers it for future processing.
Post Conditions: The Data analyst is back on the main screen, and has one (or more) profile options.

Use Case Identifier: LOADTABLE
Preconditions: The Data analyst has set the database's information (name, password etc) in the config file.
Flow: 1. The use case starts when the Data analyst chooses to register a database table by selecting a specific button or specific option. 2. The application connects to the database. 2.1 If the details written in the config are wrong, the data analyst is alerted and returns to the main screen. The Use Case ends. 3. The application prompts the Data analyst to give the table's name. 3.1 If the Data analyst gives an invalid name, the application notifies them and allows them to try again. 4. The application creates a DataSet Profile with the table's data and registers it for future processing.
Post Conditions: The Data analyst is back on the main screen, and has one (or more) file options.

Use Case Identifier: RUNCHECKS
Preconditions: The Data analyst has registered at least one DataSet.
Flow: 1. The use case starts when the Data analyst chooses a DataSet Profile to process. 2. The Data analyst specifies which checks will be run on the chosen profile, as well as the Violating Row Policy for when a report is generated. 3. The application runs the chosen checks, marking all data entries that do not pass a test, as well as which test they did not pass. 4. The application keeps the results in memory for the creation of a report.
Post Conditions: The Application has created a in-memory result for the profile.

Use Case Identifier: GENERATEREPORT
Preconditions: The Application has at least one registered profile.
Flow: 1. The use case starts when the Data analyst chooses a DataSet Profile to generate a report from. 1.1 If the profile was not used to run some checks, the data analyst is notified and the use case ends. 2. The data analyst is prompted to choose the type of report to be generated from a list of available types. 3. The data analyst is also asked for a directory to place the produced reports. 4. The application generates a report file for each order that was run on this profile, as well as any other files required by the violating row policy.
Post Conditions: The Application has created one or more report files in the chosen directory.

CHAPTER 3

Implementation & Design

In this chapter we present the application's inner workings and design in detail. We firstly analyse the problems we need to solve, we further examine the specifics in the architecture, how we tested their validity, and lastly what more can be done to further expand on this tool.

3.1 Problem Overview

To begin with, we focus on three major problems that the design of this application has to address.

- Obscure Inner Workings: Our tools should not present needless information to the data analyst. To someone that wishes to only use and not upgrade our code, the application should be easy to use and only present necessary information to the data analyst.
- Versatility and Scalability: Our tool must be able to be further expanded upon with relative ease and be usable in a multitude of scenarios, regardless of data complexity. Additionally, parts of its components should be interchangeable for future improvements.
- Spark Limitations: The core of our application is the Spark Library. Spark provides us with the necessary tools to process large amounts of data from several sources. For this to be done correctly, we must follow Spark's rules such as Serializing all objects involved with spark transformations or making use of the driver-worker system.

3.1.1 Obscure Inner Workings

Our tool must be broken down into components. Each component should have a specific use that it offers to our application. Preferably, communication between components should be as little as possible. For this to work, we rely on the Facade Gang of Four (GoF) pattern. By using the interface `IDataCleanerFacade`, we impose a restriction to two clients operate with the back-end of the system: specifically, clients can communicate only with the facade; any communication with the internal components is not possible, and is attained only indirectly with the components via the facade. To help us implement this strategy, we also utilise the `ClientRequest` class, which allows the data analyst to send a request to our facade to be further processed. The use of a facade interface, ensures that anyone using the interface understands what its main functionalities are without needing to understand the facade's inner workings.The specifics of each class will be further expanded upon on chapter 3.2

3.1.2 Versatility and Scalability

To attain the extensibility of the code, as well as the support for alternative implementations of the same functionality, the use of interfaces is crucial. By abstracting our components, or even the facade itself, via interfaces we allow the developer to understand what each component should generally do. Additionally, by using the Factory GoF Pattern, we also allow each component to be easily changed and the application to be effortlessly expanded upon. Those two tactics solve the versatility and scalability concern.

3.1.3 Spark Limitations

Spark sets a few rules for working with datasets:

- Forced Serialization: All components and classes that interact in any way with a dataset must be serialized. Since Spark works with nodes and clusters to store data, the process of serialization plays an important part on the efficient use of datasets in highly distributed environments. This feature, however, becomes especially problematic whenever we need to use libraries and components that may not be Serializable.

- Transformations on Workers: Spark applications run as independent sets of processes on a cluster, coordinated by the main application called a Driver. Nodes in the cluster that are not the driver are called workers. A worker is used when a dataset is transformed or iterated, which is ordered by the driver. Any required data is sent to the worker(s) and executed accordingly as a form of multi threaded optimization. This, however, presents the issue that when iterating or transforming a dataset, we can not transform or iterate the same dataset or any other dataset at the same time.

- Worker Variables: Another issue with worker nodes is that when they are used, they create a new view of the variable available to the driver. This view is not in sync with the variables in the driver, meaning that we can not change variables in the Driver from a worker. This issue arises when we use lambda functions within for-each loops of a dataset. Since our application requires a form of feedback in certain situations when we iterate or transform a dataset, we need to find a way to obtain a returned value.

In the sequel, we detail each of the aforementioned issues.

Forced Serialization

Besides making most classes serializable by implementing the Serializable interface, the issue can be resolved using one of two methods in the instances that we can not make the class Serializable. Firstly, making a variable Static will make them by definition Non-Serializable, forcing Spark to simply use the class' static value. While the simplest method, this is preferably avoided due to the possibility of corrupting the variable's value by using its class in a non-serial manner (such as using threads).The second method involves using the transient keyword when declaring the variable. This defines a variable alone as "non-serializable", forcing Spark to ignore it.
Both tactics are used in our application. In the report generation, a static `FileWriter` is used on the report generation, to avoid both the Forced Serialization and the Worker Variables issue. The `BPlusWrapper` class also uses the transient tactic, in order to use the Database Class. Both tactics are further explained in section 3.2.

Transformations on Workers

This issue has no specific solution, besides avoiding it. There are several workarounds that can be used to extract information outside of a Spark Transformation/Iteration, to allow us to perform a second transformation after the first has been completed. The `ServerRequest` class has this issue, needing to use a Map function when attempting to check each row individually, and returning a String Dataset as a form of reply. This forces us to further process the reply before creating our result. Further details are mentioned in 3.2.1.

Worker Variables

Similar to the Transformation on Workers problem, we try to avoid setting Driver variables within a Worker environment. One way to solve this problem is by using disk-based data structures, such as BPlusTrees. While effective, several disk-related interactions can be inefficient. To avoid continuous file saves, we can instead use the static keyword. Using static variables, we force all nodes to point to the same variable. As previously stated however, using static variables can prove to be problematic in non-serial execution.

3.2 Architecture

In this section, we detail all of our application's components, as well as present the design that we decided to follow. In our explanation for each component, we will follow a bottom-up approach, starting with our basic components and data structures before moving on to our services. Before anything else, however, we summarize our packages and how their place in our package hierarchy plays an important role in distinguishing each class' purpose.

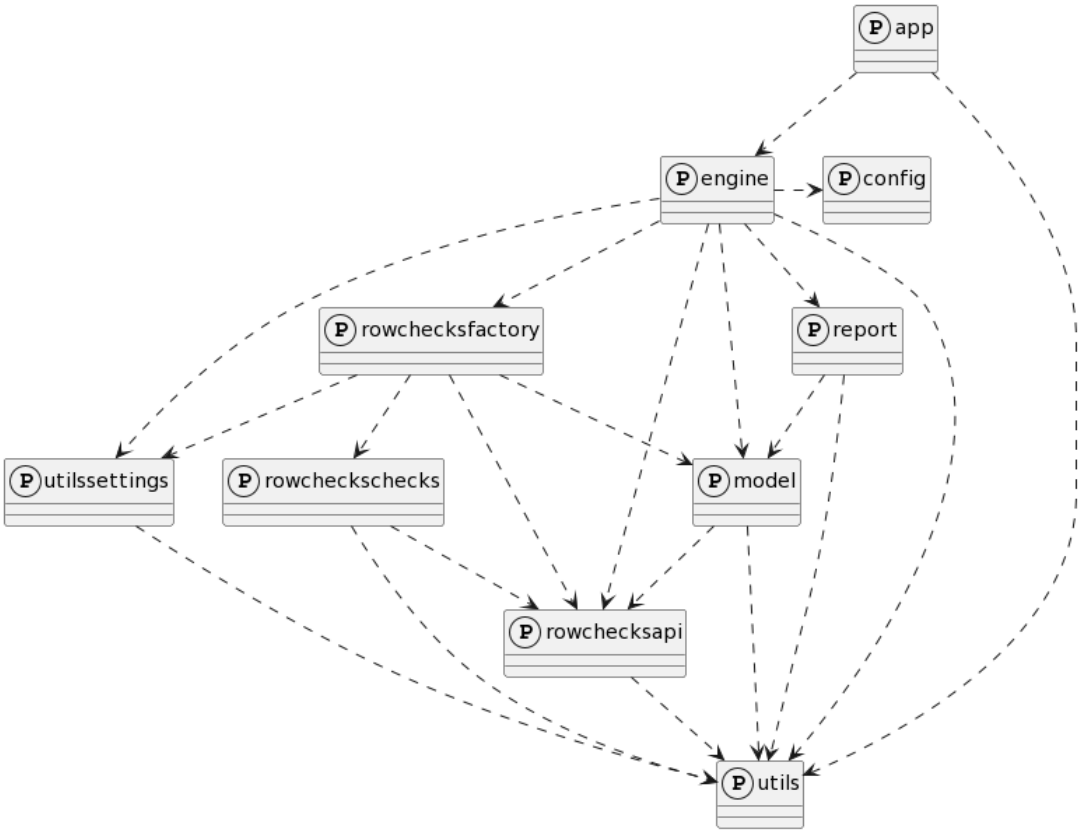

Figure 3.1: Package Hierarchy Diagram

The App package is where front-end client classes reside, also including the main() function or any Front-End/Graphics user interface. This package has a single way of communicating with the rest of our application's components, and that is through the facade, which resides in our engine package.

The Engine package holds our core components and services. Our services are placed here, as the primary components for request processing, excluding the report generation. This package also holds anything that directly communicates with client classes, such as the client request and responses. A detailed UML is displayed at 3.3

The Model package keeps all data structures that our tool uses. Those classes mostly hold and organise information in compact objects that are used to process any request the Client may have, without directly interacting with said client. A detailed UML is displayed at 3.2

The Report package encompasses all classes related to report generation. This package is called by our Facade, and uses several model classes in order to produce reports. A detailed UML is displayed at 3.4

The RowChecks package contains three sub-packages that are all used to implement the rules that our application enforces on data. The rowchecks.api holds the IRowCheck interface that all row checks implement. The rowchecks.factory holds the factory that generates all row checks and provides the scalability our tools requires. Finally, the rowchecks.checks package contains the actual individual checks that we use to enforce data cleanliness. A detailed UML is displayed at 3.5

The Config package holds a single class that is responsible for quickly initialising some basic objects required by Spark.

Finally, the Utils package holds a wide range of utility classes that are used by almost all classes and packages through out our program. This includes Enumerators for clear value distinction and purpose, helper classes for data management and data structure wrappers to help us avoid several Spark Limitations. The utils.settings sub-package contains the Settings classes that are used in our client request processing for better data handling.

3.2.1 Core Models and Data Structures

Starting with our simplest components, in this section we describe all models used throughout our application's processes.

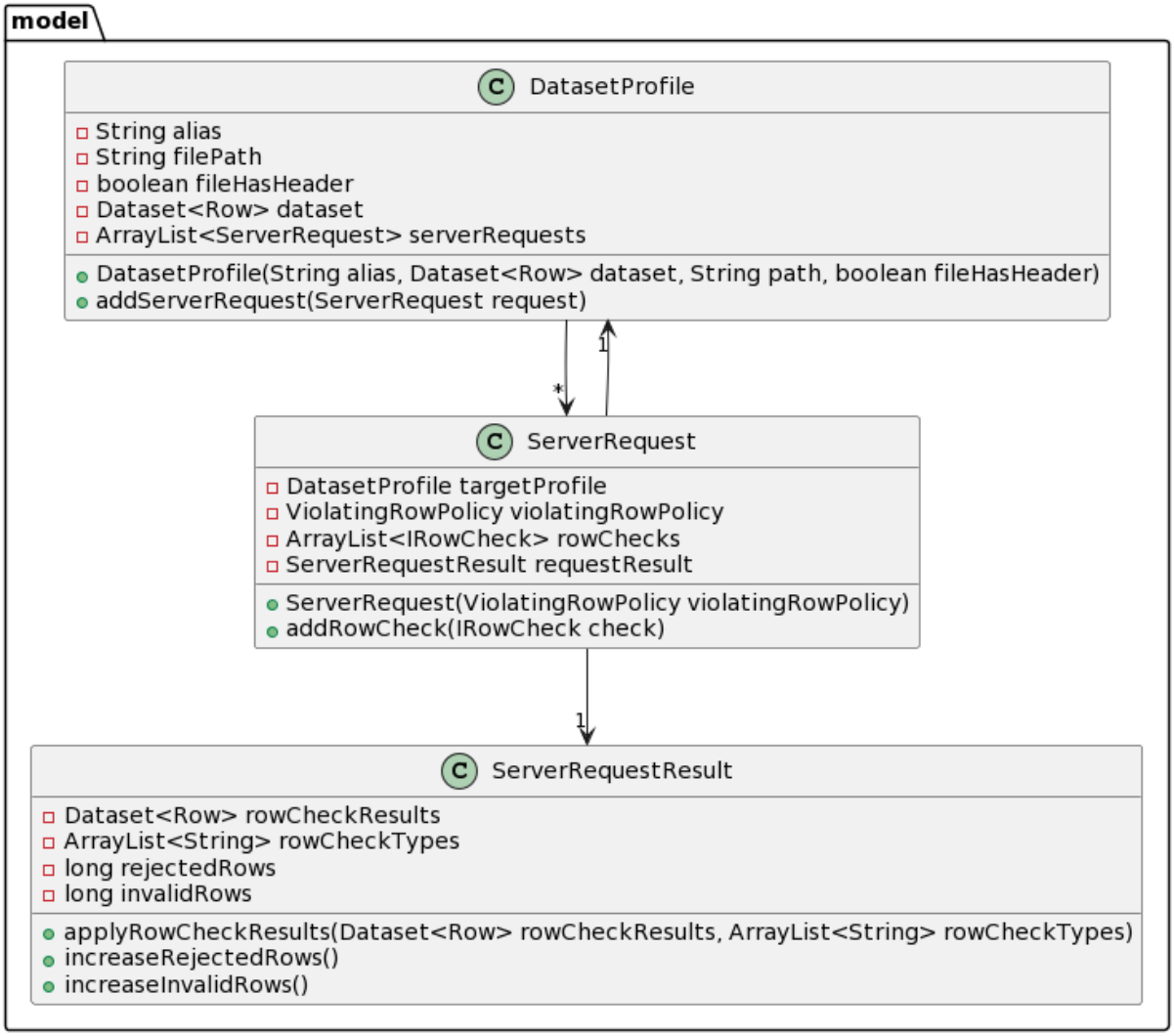

Figure 3.2: Model Package

DatasetProfile

The DatasetProfile model is a class that is responsible for keeping a reference to a dataset that the data analyst has registered, as well as information about its origins and some useful information on its dataset structure. Additionally, a DatasetProfile is responsible to keep a record of all previous orders the data analyst has executed, in order for them to be used during the report generation. The variables kept in this class are:

- String alias: A unique name given by the Data analyst to distinguish this profile from others.
- String filePath: A string containing the path of the file that this dataset was created from. It is replaced with ”DATABASE” if the dataset was pulled from a database.
- boolean fileHasHeader: A boolean value that determines if the datast's file had a header line. False by default if the dataset was pulled from a database.
- Dataset dataset: A Spark Dataset that contains our registered data.
- List<ServerRequest>: A list of all the ServerRequests (or orders) that have been executed on this profile's dataset.

Besides that, the class contains several Accessors and Mutators for its variables.

DatasetRegistrationController

The Dataset RegistrationController is a class of our engine package, responsible for registering Datasets and creating DatasetProfiles on the data analyst's command. The RegistrationController is also responsible for keeping all registered DatasetProfiles and sharing them with the Facade, which in turn allows them to be processed by all the other components of our application.

Settings Classes

To assist with the creation of checks, we use a variety of classes called ”Settings Classes”. Found in the utils package, these Settings classes are simple data holders that are consumed by their respective factory in order to create checks from a ClientRequest. While very basic on their design, they help us organise complex information in some cases, and keep our code clean.

ViolatingRowPolicy

The ViolatingRowPolicy is an enumerator kept within our utils package, that helps us define the three distinct available types of handling our rejected rows. A row is rejected when it fails one or more checks, and is handled according the ViolatingRowPolicy during the report generation. In the current version of our tool, the three supported types are:

- WARN: The default setting for all requests. Simply generates a report in the desired format and does not produce any new datasets.
- ISOLATE: Produces a report exactly as in WARN, but also creates two new TSV files in the designated directory. One file contains all entries that do not violate any check, and the other contains all entries that violate at least one check.
- PURGE: Produces a report exactly as in WARN, but also creates a new TSV file in the designated directory. The file contains all entries that do not violate any check, effectively purging all violating rows.

ClientRequest

The ClientRequest model is the first step for creating an order. Located in the engine.clientRequest package, the ClientRequest represents a set of rules given by the client on a specific Dataset Profile that are to be tested, as well as the tactic that is to be used when faced with a violating entry. The ClientRequest class makes use of Joshua Bloch's Effective Java Builder [6], a useful pattern that allows us to create a complex object in a step-by-step process while also allowing us to omit some information should the object not require them. This is done with the use of a subclass called a Builder, that allows us to dynamically apply small changes on the builder's variables which as finally compiled into a ClientRequest object. The ClientRequest object contains:

- String targetDataset: The alias of the DatasetProfile this order is focusing on. This must always be defined.
- ViolatingRowPolicy violationPolicy: The aforementioned enumeration used to define what must be done should we meet a data entry that breaks one (or more) of our set rules. This is used by our report generation and is set by default to WARN.
- List<Settings>: Several list variables are used, able to contain several Settings objects. These are later used to compile their respective check for the construction of our ServerRequest.

Another advantage of Bloch's pattern is the scalability it provides to our application. If a developer wishes to expand upon the list of checks available, they can easily add another Settings variable by following the defined architecture.

ClientToServerRequestTranslator

The ClientToServerRequestTranslator is one of our engine classes, responsible for translating a ClientRequest into a ServerRequest. A singular instance is kept within our facade and directly used in the execution process of a ClientRequest, transforming the ClientRequest into several checks that are stored within the ServerRequest.

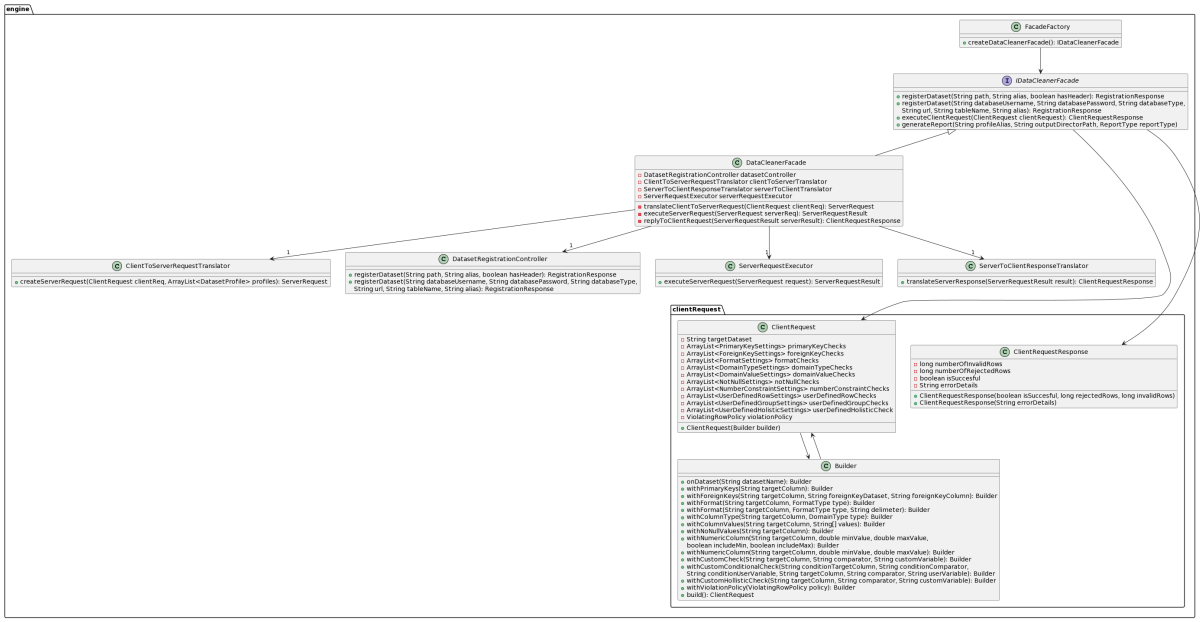

Figure 3.3: Engine Package

ServerRequest

The ServerRequest class is one of our core models. The ServerRequest acts as representative of a ClientRequest to the system's back-end, ready to be executed and produce an actual result. A profile can have multiple ServerRequests within it, and the application automatically runs all checks to produce a result, as soon as one ServerRequest is created. A ServerRequest holds:

- DatasetProfile targetProfile: A backwards reference to the profile that it is stored in.
- ViolatingRowPolicy violatingRowPolicy: The violating row policy that was defined in the ClientRequest.
- ServerRequestResult requestResult: The ServerRequestResult of this ServerRequest. Effectively the finished result of the request's checks.
- List<IRowCheck> rowChecks: A list containing all row checks that this request applies. Generated by the ClientToServerRequestTranslator using the ClientRequest model during the request's creation.

ServerRequestExecutor

With the creation of a ServerRequest, the request is passed to the ServerRequestExecutor, a class of our engine package that is responsible for producing a ServerRequest result. ServerRequestExecutor extracts all necessary checks stored in our ServerRequest, executes them on its respective profile's dataset, and produces a result that is stored in the ServerRequest.

CheckResult

Before moving to the next step of our request execution process, we must first define our CheckResult enumerator, as well as the spark's row. Part of our utils package, the CheckResult enumerator holds the available result values a row can have after a check. A Row serves as the fundamental unit within our application, encapsulating a singular dataset's item. Each Row encompasses values corresponding to the columns present in its dataset. For future reference, we establish an entry, synonymous with a record, as a distinct instance of a Row. The currently available CheckResult values are:

- PASSED: Used when an entry has successfully passed the check. This is considered a Successful result.
- REJECTED: Used when an entry has failed the check. This is considered an Unsuccessful result.
- MISSING_VALUE: Used when the check was forced to deal with a missing value (or null). This is considered an invalid result.
- ILLEGAL_FIELD: Used when a check tries to access an invalid or missing column. This is considered an invalid result.
- INTERNAL_ERROR: Used when none of the previously stated results is used. It dictates that an internal error prevented the check from completing. This is considered an invalid result.

These values are defined as Successful, Unsuccessful and Invalid results, as mentioned above. Successful and Unsuccessful results are the expected results in a good scenario, while Invalid results reveal an error that occurred during the process.

ServerRequestResult

The ServerRequestResult is one of our model classes, placed within the model package, that is automatically created when a ServerRequest is generated. ServerRequestResult represents the result of a ServerRequest, holding information on the profile's data entries on whether they have passed the checks or they were rejected and contains the following variables:

- long rejectedRows: The number of entries that were rejected.
- long invalidRows: The number of entries that failed to produce a result. We consider an entry as "Invalid" when during the check it produced an unexpected error, such as trying to read a non-existent column, or not being able to access a on-disk data structure.
- List<String> rowCheckTypes: A list of strings that each define a row check's type. A row check's type is a string-formatted description that explains what the check is trying to determine, and is being used during the report generation process.
- Dataset<Row> rowCheckResults: A dataset holding the results on all row checks. This dataset holds a column that contains a unique ID for each entry of our dataset, and a number of columns equal to the number of row checks, each holding the entry's CheckResult on the respective row check. For example, if we send a request that has 3 distinct row checks, an example of the rowCheckResults Dataset would be:

_id	c0	c1	c2
0	PASSED	PASSED	PASSED
1	PASSED	REJECTED	REJECTED
2	REJECTED	PASSED	PASSED
3	MISSING_VALUE	MISSING_VALUE	MISSING_VALUE
4	INTERNAL_ERROR	PASSED	REJECTED

Table 3.1: RowCheckResults Dataset example

ServerToClientResponseTranslator

The ServerToClientResponseTranslator is another class of our engine package, responsible for the translation of a ServerRequestResult into a ClientResponse. Similar to ClientToServerRequestTranslator, a single instance of ServerToClientResponseTranslator is kept inside our facade and is used when a ServerRequestResult is finalized in order to return to the client a summary of the results.

ClientRequestResponse

The ClientRequestResponse class, part of the models package, is what finally is returned to the client at the end of the processing and execution of a ClientRequest. The ClientRequestResponse holds a number of helpful variables that provide the data analyst with a quick summary of the request's result:

- long numberOfRejectedRows: The number of entries that had at least one unsuccesful CheckResult value across all checks.
- long numberOfInvalidRows: The number of entries that had at least one invalid CheckResult value across all checks.
- boolean isSuccessful: Is true if the numberOfInvalidRows is equal to zero, otherwise false.
- String errorDetails: A string that defines an error in case of unsuccessful result. Is null if isSuccesful == true.

While the data analyst is not able to get a detailed description on which entries passed or failed our checks, the data analyst is able to correct any obvious mistake they might have made during the request, before moving on to create a report.

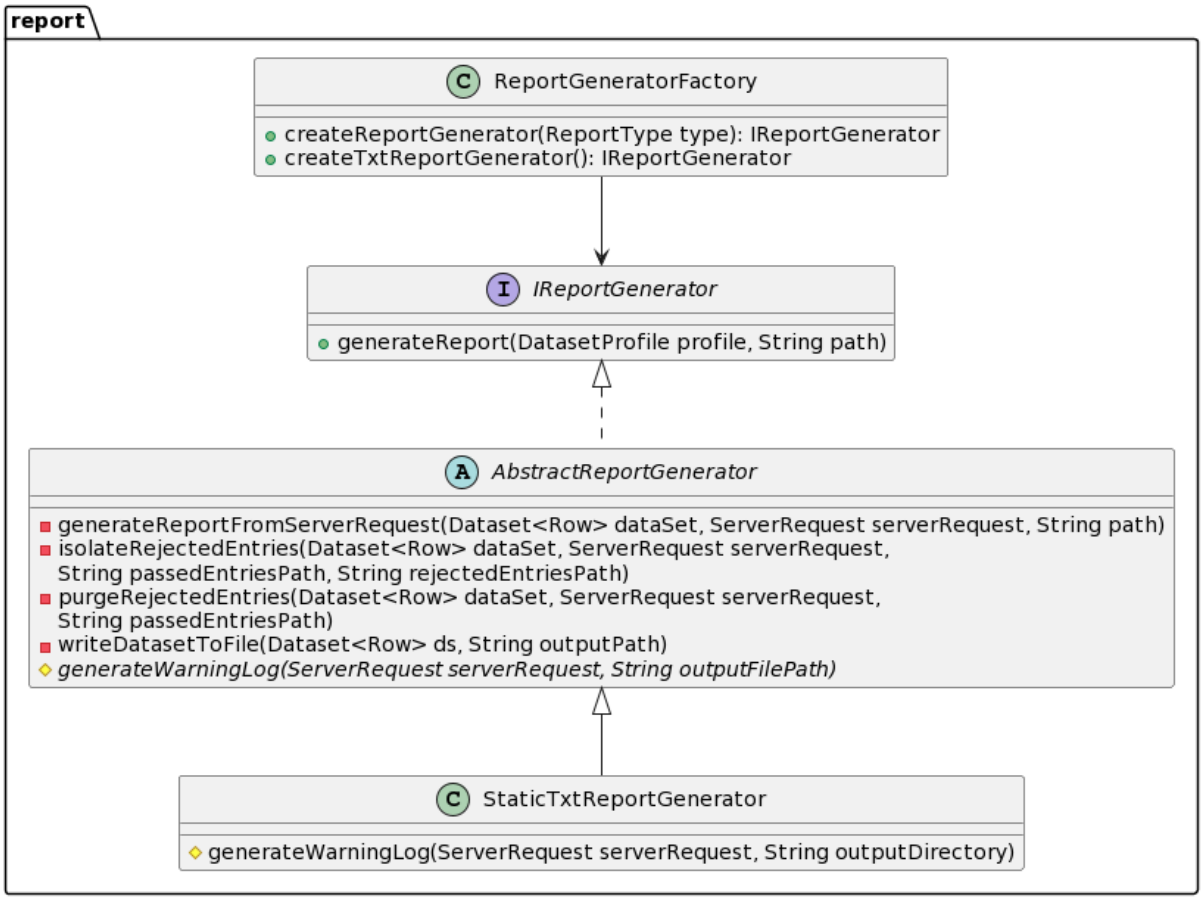

Figure 3.4: Report Package

IReportGenerator

Having completed the discussion of the ServerRequestResult generation, we move on to the creation of a report. IReportGenerator is an interface, part of the report package, that is used by our facade to generate our report files given a DatasetProfile and a directory path to place the report files. The purpose of the IReportGenerator interface, as well as the respective factory (ReportGeneratorFactory), is to improve our application's scalability and versatility, by allowing us to freely switch and expand upon our available Report file options.

In addition to this interface, we make use of the AbstractReportGenerator, an abstract class that implements the IReportGenerator interface, which contains some useful functions that are used across all report generators.

BPlusTreeWrapper

This class is part of our utils package and plays a crucial role in the execution of Primary and Foreign Key Checks. The main purpose of this class is to create a reference to a disc-based BPlusTree data structure. More specifically, we make use of the Berkeley Database [5]. Berkeley DB is an embedded database library developed by Oracle, that uses B+ Trees to store and organise key-value pairs, while also keeping them stored on our disk instead of our memory. This is important in order to be able to process any amount of data regardless of size, in expense of some processing speed. The BPlusTree class itself implements a Database object from Berkeley DB, as well as several helper functions for the database's initialization on the disk and deletion of the database when it's not longer of use. This class also helps us with the Spark Limitation of Forced Serialization, by making the Berkeley Database transient, while being Serializable.

3.2.2 Row Checks

In this section, we take a closer look to our Row Check classes. We define a Check as a rule that has or is about to be applied on a dataset. This rule can be anything relating to the content of a specific column, the format or even the data type of it. Regardless, each entry of a dataset must pass through this rule when ordered by our application, on a Data analyst's request. A Row Check is defined by the IRowCheck Interface, within the rowchecks package, and defines a check that requires a single row (or entry) at a time to produce a result. During the initialisation of a Row Check, we are allowed to perform grouping and holistic actions, as long as we do not perform any further non-row-level actions during execution. A RowCheck must implement the following methods:

- CheckResult check(Row row): A method that when given a Spark's Row object, will run any needed check on it and produce a CheckResult according to the outcome.
- String getCheckType(): A method that returns a small string that explains the check type. This string is used in the report generation process and is used to explain to the data analyst what this check is responsible for.

All row checks are generated with the help of a factory pattern (RowCheckFactory) that provides the versatility and expand-ability that our application requires, allowing future developers to further expand on the available row checks.

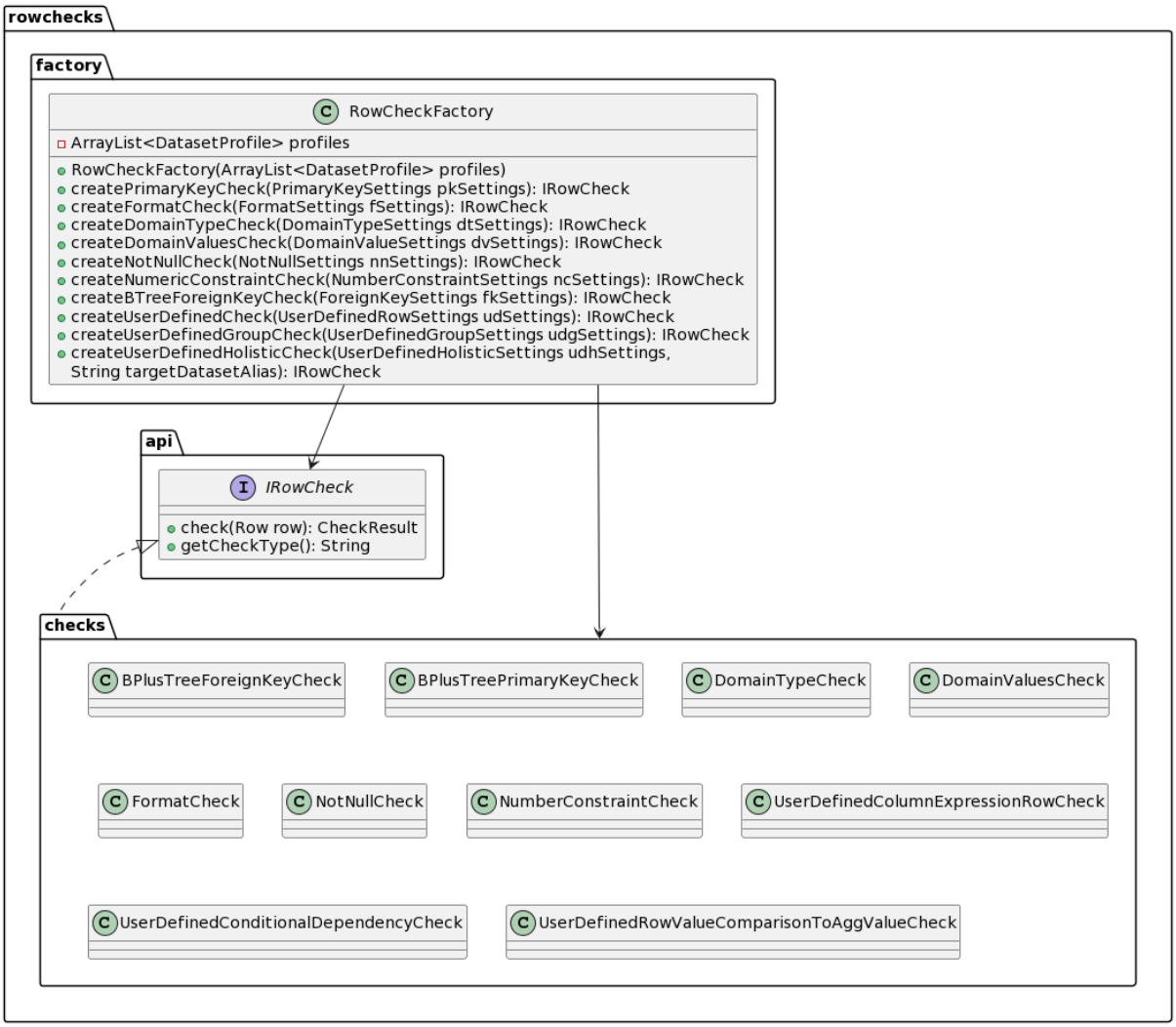

Figure 3.5: Row Checks Package

DomainTypeCheck

This check is responsible of checking whether a column of a dataset contains values of a specific type. The data analyst can select a type using the DomainType enumerator, located within the utils package, The currently available types are:

- INTEGER: Defines numeric elements that represent whole (or rounded) numbers.
- NUMERIC: Defines numeric elements, accepting both integers and decimal numbers.
- BOOLEAN: Defines binary elements. Values that we consider as ”binary” are: 1, 0, true, false, no, yes.

- ALPHA: Defines alphanumeric elements. Practically any string that contains at least one non-numeric character. Note that while a numeric element can be considered as a String, we choose to not accept it in this scenario since the NUMERIC or INTEGER DomainType can instead accept it.

The check, when initialized, keeps the DomainType assigned to it, as well as the column it needs to check. When a row is passed to its check function, the column's value is extracted and tested on a respective regular expression:

- INTEGER: "^-?\d+$"
  The element can begin with a - symbol (if the number is negative) and then must be followed by one ore more numbers from 0 to 9 until the end of the element.

- NUMERIC: "^-?(\d+|\d+.\d+)$"
  The element can begin with a - symbol (if the number is negative) and then can follow one of two scenarios:

  – It can be an INTEGER, therefore following one or more numbers from 0 to 9 until the end of the element.

  – Otherwise, it can be a decimal number, following one or more numbers from 0 to 9, a dot and then again one or more numbers from 0 to 9 until the end of the element.

- BOOLEAN: "^(1|0|true|false|no|yes)$"
  The element can be any of the following elements, case insensitive: "0", "1", "true", "false", "no", "yes". A boolean can be defined in more binary ways, such as using two specific words (ex. Black and White, Passed and Failed etc). For this one should better consider using DomainValues check.

- ALPHA: ".*[a-zA-Z].*"
  The element can contain any character, both numeric and not, however is must contain at least one non-arithmetic character at any position.

Rows that have values in their respective columns that pass the regular expression match are given the PASSED CheckResult, otherwise the REJECTED. In the scenario that the given column does not exist or an internal error occurs, the remaining CheckResult values can be returned accordingly.

FormatCheck

This check is responsible of checking whether a value follows a specific format. We make use of the FormatType enumerator, part of the utils package, to enumerate some of the most common format types. For the purposes of this thesis, we focused on all possible combinations of Date Formats. Future versions can be expanded by adding more types of formats both to the check and the FormatType accordingly. In the current version, we have developed the following types: DD_MM_YYYY, MM_DD_YYYY, YYYY_MM_DD, DD_MM, MM_DD, YYYY_MM, MM_YYYY. All aforementioned format types are full or partial combinations of three values: DD which defines a day from 01 to 31, MM which defines a month from 01 to 12 and YYYY which defines a year which can be any positive integer. Similar to DomainType Checks, when called to check a row, the row's value is extracted and matched against the respective format's regular expression. All regular expressions are combinations of smaller regular expressions that represent either Day, Month or Year:

- Day: (0[1-9]|[12][0-9]|3[01]|[1-9])
  A day can be an integer from 1 to 31. This can be broken down to the following scenarios:
  - A number from 1 to 9.
  - A zero and then a number from 1 to 9 (01, 04, 09 etc)
  - A 1 or a 2 and then a number from 1 to 9
  - A 3 and then either a 0 or a 1.
- Month: (0[1-9]|1[0,1,2]|[1-9])
  A month can be an integer from 1 to 12. This can be broken down to the following cases:
  - A number from 1 to 9.
  - A zero and then a number from 1 to 9.
  - A 1 and then either a 0, a 1 or a 2.
- Year: \d+
  A year can be any integer. So we just check the appearance of one or more numbers from 0 to 9.

A note to be made is that there is no check on the validity of the date, only the format. For example, the date can have the month of February having 31 as a day. Additionally to these regular expressions, the data analyst can provide a delimiter to the format for the separation of each element. The delimiter is by default the forward slash symbol and is replaced inside the final regular expression. With the final regular expression defined, the extracted value attempts to match it, returning a PASSED CheckResult on success and a REJECTED on failure, with internal errors returns the remaining CheckResults.

DomainValuesCheck

This check is responsible of checking whether a value belongs to a domain, i.e., a certain group of predefined values, similar to a foreign key relationship. The data analyst can select the targeting column, as well as a list of string type values (Domain Values) to be used for the check. When a row is passed, the check gets the value from the target column and checks if its value, in a string format, is within the provided Domain Values. If it is, a PASSED CheckResult is returned, otherwise (should a problem not occur) the REJECTED CheckResult is returned.

NotNullCheck

This check is responsible of checking whether a certain column contains any Null values. The data analyst defines the column of interest, and the check marks any entry that contains a NULL value in that column as REJECTED, otherwise as PASSED. Since we are dealing with NULL values, the only other CheckResult that can be returned is the ILLEGAL_FIELD result in the case that the provided column does not exist.

NumericConstraintCheck

This check is responsible of checking whether a specific column that contains numeric elements has only values within a specific range. The data analyst provides the column of interest, as well as a minimum value (left bound) and a maximum value (right bound). Optionally, the data analyst can also provide whether the left and right bounds should include their values, which both are true by default. So for example, if a data analyst sets a left bound of 2 and a right bound of 10 with left bound being inclusive and the right bound not being inclusive, all values of the selected column must belong in [2,10). Entries that belong to the defined interval are given the PASSED CheckResult, while those that do not or simply do not contain a valid numeric element are given the REJECTED CheckResult. Any internal error will also return the respective CheckResult.

BPlusPrimaryKeyCheck

BPlusPrimaryKeyCheck, or simply PrimaryKeyCheck, is the check responsible for checking the uniqueness of a column. The data analyst provides a column of interest, to be checked for uniqueness, and the check creates a BPlusTreeWrapper to register values from that column. Entries are given one by one to this check, following the code seen in algorithm 3.1. Essentially, values already seen are considered duplicates and therefore are given a REJECTED result, otherwise they are unique and considered PASSED. This means that if exactly two entries contain the same value in our targeted column, only one will be rejected instead of both.

Algorithm 3.1 Primary Key Row Check

Require: Targeted Column: $targetColumn$

Input: Dataset's row

1: Initialize $database$ if not already initialized.
2: $targetValue \leftarrow row[targetColumn]$
3: **if** $targetValue$ exists in $database$ **then**
4: **return** REJECTED
5: **else**
6: $database$.insert($targetValue$)
7: **return** PASSED
8: **end if**

BPlusForeignKeyCheck

BPlusForeignKeyCheck, or simply Foreign Key Check, is the check responsible for finding foreign key relations between datasets' columns. It follows a similar process as DomainValuesCheck but instead of comparing entry values and user-given values, it instead checks whether an entry's value exists in another entry's value in a specified column and dataset. The data analyst must define the column of interest, as well as a second DatasetProfile (FK Dataset) and a column from that second DatasetProfile's Dataset. The Foreign Key Check then stores all values belonging to the FK Dataset's column in a BPlusTreeWrapper and awaits execution. When an entry is provided to be checked, we take the value in the column of interest and check whether it exists in our BPlusTreeWrapper's database. If it does, then the Foreign Key relation is not violated, and we consider it passed, otherwise we reject it.

3.2.3 User-Defined Checks

In this section we define our User-Defined Checks. Much like the previous checks, User-Defined Checks are sets of rules that our data must follow in order to not be rejected during execution. In contrast to our previously mentioned checks, User Defined Checks are flexible rules that are freely defined by our data analyst without any constraints. Data analysts may define any mathematical expression to compare with values of a specific column or even use grouping and aggregation tactics to produce more complex rules.

exp4j Library

To assist us with evaluating complex mathematical expressions, we make use of the exp4j library. Exp4j is a small Java Library that utilises Dijkstra's Shunting Yard algorithm in order to break down mathematical expression and evaluate their result with the help of a stack data structure. It is quick and effective at producing numeric results, and even accepts values for any variable within the expression.

Comparator

The Comparator class, part of our utils package, provides a data analyst friendly way of defining a comparison symbol. Instead of actually providing a string formatted comparison symbol (ex. ”>”, ”<=”) the data analyst may use this class’ static variables to avoid misspelling any symbols (ex. Comparator.GREATER, Comparator.LESS_EQUAL). In addition to that, the class has a single function that allows us to perform a comparison using one of the aforementioned comparators. This is mostly used in order to avoid repeating code during mathematical expression evaluation and comparison.

User-Defined Column Expression Row Check

The simplest of the User-Defined Checks, the UserDefinedColumnExpressionRowCheck is responsible for performing row-level checks on a dataset. The data analyst may define a single column (or number), a user-defined variable and a comparator between them. A User-Defined variable is a mathematical expression that can contain all possible mathematical symbols that are supported in Java, as well as any columns in the same dataset for variables. Once all defined, the check extracts all values corresponding to either the targeted column or any variable column, and replaces them to create a mathematical expression. This is then executed using a Exp4j expression to produce a numeric result which is compared and translated into a passed or rejected result. For example, in a dataset that has the numeric columns ”salary” and ”age”, by defining the target column as ”salary”, the comparator as ”>” and the user variable as ”3*age”, the UserDefinedColumnExpressionRowCheck will look for entries that have a value in ”salary” that is greater than three times the value in ”age”. Entries that do not meet this criteria are rejected. A detailed pseudo code of this algorithm is seen in 3.2

Algorithm 3.2 User Defined Row Check Execution

Require:
- $targetColumn$ Row check's Target column (or number)
- $comparator$ Row check's Comparator
- $userVariable$ A mathematical expression

Input: Dataset's row

Initialize $exp4jExpression$
$targetValue \leftarrow row[targetColumn]$ or $targetColumn$ if numeric
for $variable$ in $userVariable$ **do**
    $exp4jExpression$.addVariable($variable$, row[$variable$])
end for
$equationResult \leftarrow exp4jExpression$.evaluate($userVariable$)
if $targetValue$ $comparator$ $equationResult$ **then**
    return PASSED
else
    return REJECTED
end if

User-Defined Conditional Dependency Check

User-Defined Conditional Dependency Check, or Conditional Checks, can be best described as two row checks that one depends on the other. Essentially, this check is used to determine if a specific group of entries follows a row-level rule. The data analyst defines first the Condition Check and then the Row Check, both of which are similarly defined as a UserDefinedColumnExpressionRowCheck. When a row is sent to be evaluated by the Conditional Check, it first determines if the Condition applies. If it does, the Row Check is also evaluated and the result is returned as the final CheckResult. If however the Condition fails, we return a passed CheckResult, since this entry does not fall into the group we are trying to check. For example, if we define a condition check of "age $>$ 18" and a row check of "salary == 0", the User-Defined Conditional Dependency Check will accept all entries that either have "age" equal to or less than 18, or entries that have "age" greater than eighteen and a salary of 0. All other entries are rejected. The algorithm behind this check is shown in 3.3

Algorithm 3.3 User Defined Group Check Execution

Require:

$conditionColumn$	Condition's target column (or number)
$conditionComparator$	Condition's Comparator
$conditionVariable$	A mathematical expression
$targetColumn$	Row check's Target column (or number)
$comparator$	Row check's Comparator
$userVariable$	A mathematical expression

Input: Dataset's row

Initialize $exp4jConditionalExpression$
Initialize $exp4jRowExpression$
$conditionValue \leftarrow row[conditionColumn]$ or $conditionValue$ if numeric
for $variable$ in $conditionVariable$ **do**
    $exp4jConditionalExpression$.addVariable($variable$, row[$variable$])
end for
$conditionEquationValue \leftarrow exp4jConditionalExpression$.evaluate($conditionVariable$)
if NOT $conditionColumn$ $conditionComparator$ $conditionEquationValue$ **then**
    return PASSED
end if
$targetValue \leftarrow row[targetColumn]$ or $targetColumn$ if numeric
for $variable$ in $userVariable$ **do**
    $exp4jRowExpression$.addVariable($variable$, row[$variable$])
end for
$rowEquationValue \leftarrow exp4jExpression$.evaluate($userVariable$)
if $targetValue$ $comparator$ $rowEquationValue$ **then**
    return PASSED
else
    return REJECTED
end if

User-Defined Row Value Comparison To Agg Value Check

A UserDefinedRowValueComparisonToAggValueCheck is similar to the Row Check equivalent of the User Defined checks. The major difference is that in the userVariable, the data analyst may define as a variable one or more aggregation functions on any column of the targeted dataset. The aggregation functions are defined in the same String format as any mathematical expression and are calculated before any check is evaluated. For example, a data analyst may define a check as ”salary >= AVG(salary)” which in return will accept all entries that have ”salary” equal or greater than the average salary of the dataset. The currently supported aggregation functions are:

- SUM(X): Defines the sum of all numeric values in column X.
- AVG(X): Defines the average numeric value from a numeric column X.
- MIN(X): Defines the minimum value from all values in column X.
- MAX(X): Defines the maximum value from all values in column X.

Before evaluating any entry, the variables that use aggregation functions are extracted and calculated using the targeted dataset. The calculated values are then inserted, with a corresponding variable alias, in our exp4jExpression and therefore replaced in the mathematical expression of the userVariable. This process is seen in the 3.4 algorithm. To assist us with the process, the class AggregationVariable is used, part of the utils package, and is responsible for executing with the assistance of Spark the corresponding aggregation function to generate the value. A note to be made is that aggregation variables within our userVariable are replaced with aliases. This is done because variables in Java, and therefore in our exp4jExpression, can not be defined using special symbols such as parentheses. The initialization process is seen in the 3.4 algorithm. Having correctly initialized, the holistic check can accept entries to be checked. The process is identical to that of a UserDefinedColumnExpression-RowCheck, as seen in 3.2

Algorithm 3.4 User-Defined RowValueComparisonToAggValueCheck Initialization

Require:
- $targetColumn$ — Row check's Target column
- $comparator$ — Row check's Comparator
- $userVariable$ — A mathematical expression that may contain aggregation functions

Input: Dataset's row

Initialize $exp4jExpression$
$uniqueIdCounter \leftarrow 0$
for $aggregationVariable$ in $userVariable$ **do**
    $aggValue \leftarrow$Calculated value of $aggregationVariable$
    $aggKey \leftarrow$ "x"+$uniqueIdCounter$
    Replace $aggregationVariable$ in $userVariable$ with $aggKey$
    $exp4jExpression$.addVariable($aggKey$, $aggValue$)
    $uniqueIdCounter++$
end for

3.2.4 Application Sequence & UML

Having looked into the detailed components of our tool, here we summarize the complete process that takes place for a request to be executed and finally be presented as a report.

First, the client registers a new dataset. The Facade sends all related information to the DatasetRegistrationController, which in turn creates and registers the DatasetProfile with its corresponding Dataset as seen in figure 3.6, with the figure 3.7 focusing on the registration's the class UML.

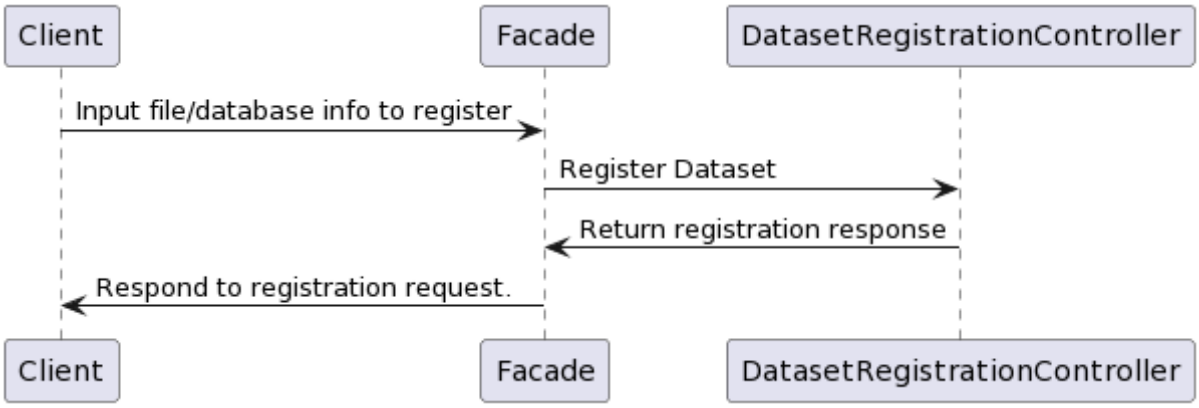

Figure 3.6: Application's Dataset Registration Sequence

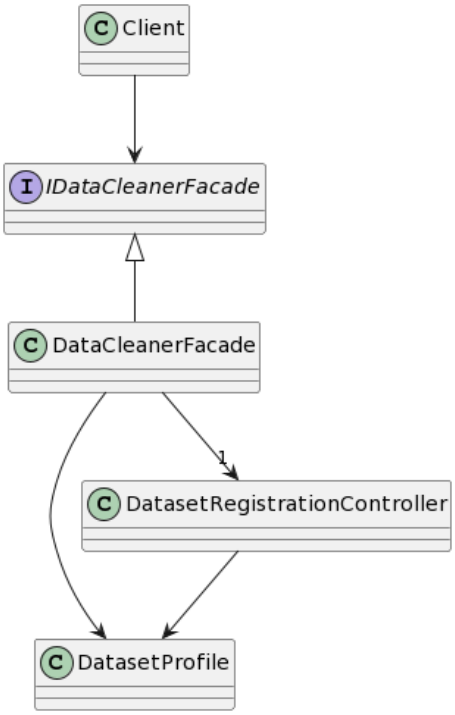

Figure 3.7: Application's Dataset Registration UML

The client now can create an order, sending it to the facade for execution. The facade translates the client's request into a server request using the ClientToServer-RequestTranslator, and stores it within the request's respective DatasetProfile. The ServerRequest is immediatly executed using the ServerRequestExecutor, where a Server-RequestResult is produced and stored within the ServerRequest. Afterwards, the ServerRequestResult is translated into a ClientRequestResponse using the ServerTo-ClientResponseTranslator and returned to the client as seen in figure 3.8. In figure 3.9 we also see the class UML related to the ClientRequest execution process.

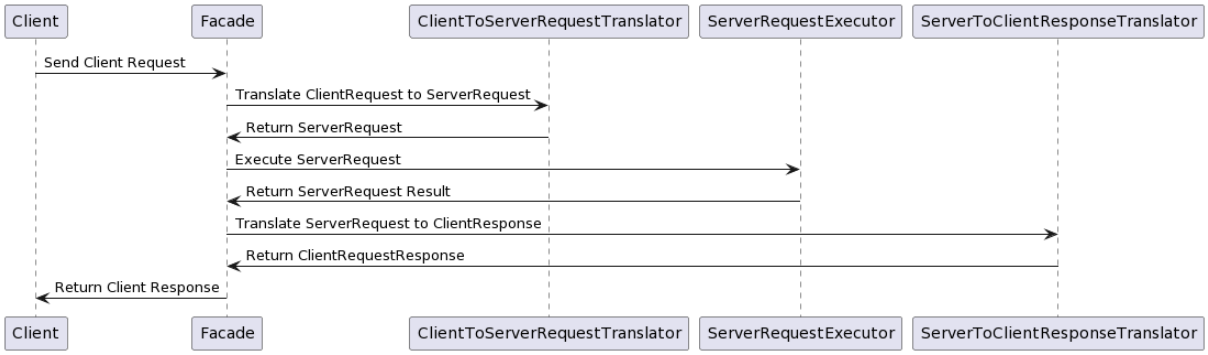

Figure 3.8: Application's ClientRequest Execution Sequence

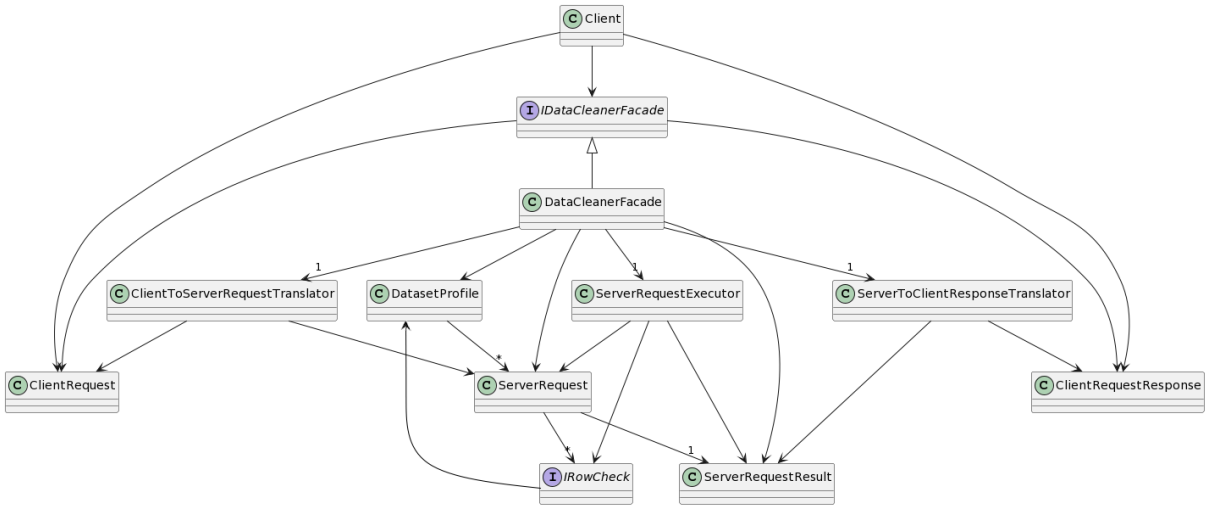

Figure 3.9: Application's ClientRequest Execution UML

Finally, the client can now request for the creation of a report, by giving the DatasetProfile's alias and the directory path where the reports will be generated. The Facade uses the ReportGenerator to generate a report for each ServerRequest that the target DatasetProfile possesses. At the end, in the designated directory, one or more reports have been created. The processes is seen in figure 3.10 and class UML defined in 3.11

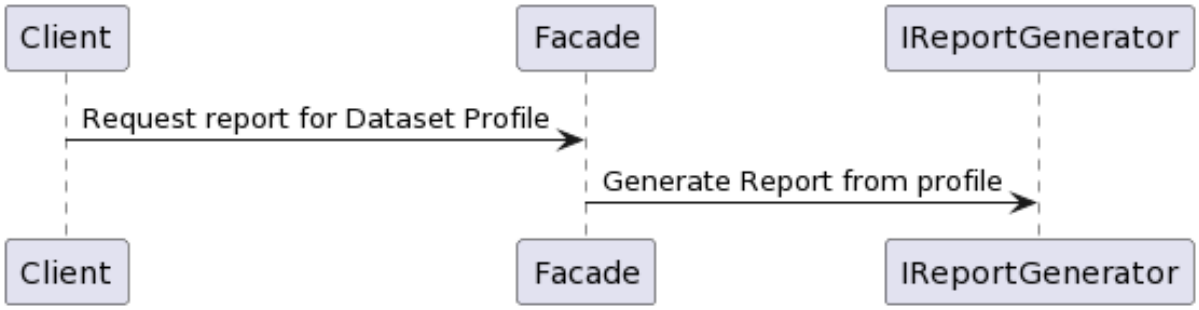

Figure 3.10: Application's Report Generation Sequence

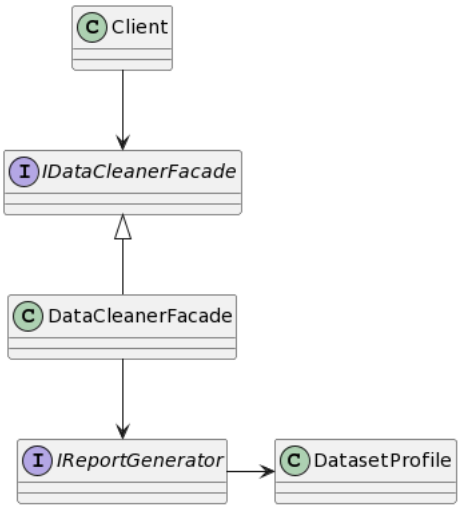

Figure 3.11: Application's Report Generation UML

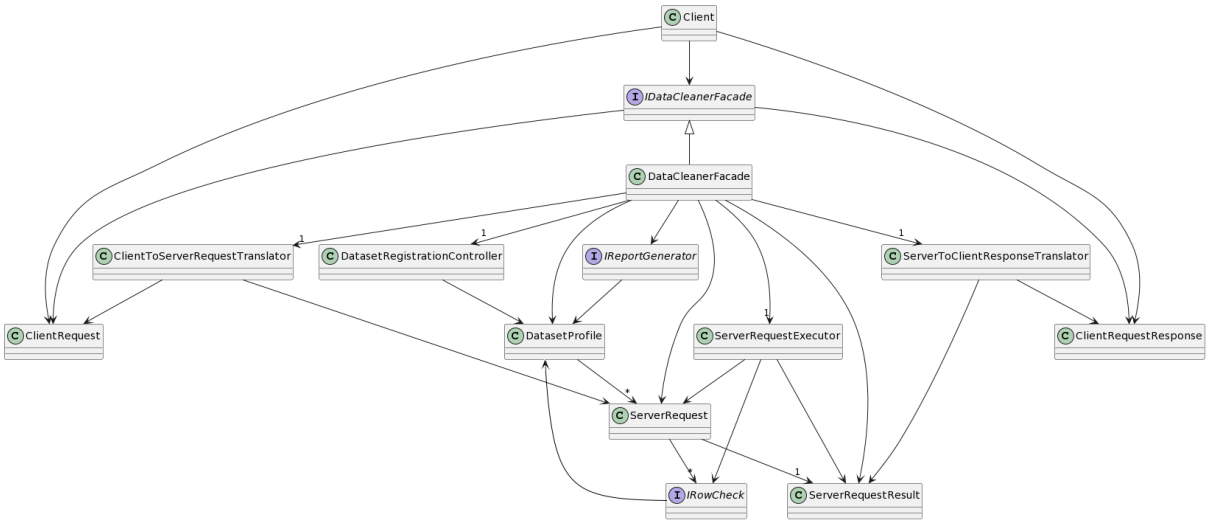

Figure 3.12: Application's Full UML (Simplified)

3.3 Validity Testing

In this section, we report on how we tested the validity and safety of our application, as well as the automated tools that we provide to facilitate the development of this tool with ease in the future. Our application provides different ways of communicating the appearance of errors to the data analyst/developer, by using certain classes, enumerators or even exceptions.

3.3.1 Dataset Registration

When the data analyst attempts to register a dataset, whether that comes from a file or a database, there is always the chance of something going wrong. To help us with this, we make use of the **RegistrationResponse** enumerator, to provide us with a result when a registration concludes. The RegistrationResponse enumerator implements to following:

- SUCCESS: Returned when a registration has concluded succesfully. Indicates that the dataset has been registered within a DatasetProfile.
- FAILURE: Returned when a registration has failed. This is usually due to a Spark internal error or even memory overflow. The FAILURE RegistrationResponse indicates an internal unexpected error.
- ALIAS_EXISTS: Returned when a registration has been attempted with an alias that already belongs to another DatasetProfile in our registered Datasets. A constraint that ensures the uniqueness of Dataset aliases.

Regardless of the error raised, one of these responses will be returned when a registration is attempted (and concluded). This guarantee allows the application to continue functioning without crashing due to unforeseen complications.

3.3.2 Client Request Execution

When the data analyst attempts to execute a ClientRequest, a number of things can go wrong. The most common possible error is the wrong definition of a check. This is resolved with the use of Invalid results. The CheckResult enumerator provides three distinct ”invalid” results (INTERNAL_ERROR, MISSING_VALUE, ILLEGAL_FIELD) that can be returned in case of an unexpected error. As mentioned before, each value has its own purpose and is used when a specific error occurs. Overall, the final ClientRequestResponse and the respective report provide extensive details on the invalidated rows for the data analyst to resolve, and keep the application in a functional state.

Additionally, the Facade provides some minor exception catching by making sure that a ClientRequest has at least one check and a valid registered DatasetProfile. This is also communicated via the ClientRequestResult, which is returned with a special errorDetail description. Even in situations that internal parts of the applications that are more prone to failure (such as the disk-based BPlusTree data structures) fail, we make sure to let the data analyst know via a printed message and invalidation of any related row as done before, without crashing the application.

3.3.3 Automated Tests

Finally, to solidify the correctness and safety of our tool, we have created several tests with the assistance of Java’s JUnit library. The purpose of the tests is to ensure the correctness of our code, and, viewing the future too, to guarantee that any refactoring done to the core components of our application does not alter the results. To assist us with this, we have generated two custom datasets that contain data that has been manually tested for cleanliness. The two datasets are (a) the `Cars_100_tests` dataset, and, (b) the `formatTests` dataset. We detail the two data sets right away.

Cars_100_tests

An all-inclusive dataset about cars that contains one hundred and three entries. The columns that are present provide miscellaneous information that we can use to verify the validity of the test.

- manufacturer: A alphanumeric column with strings about who made the car.
- model: A alphanumeric column with strings about the car's model
- year: An integer column with the year the car was made
- price: An integer column with the price of the car
- transmission: A alphanumeric column that can be "Manual", "Automatic" or "Semi-Automatic"
- mileage: An integer column with the miles traveled with the car.
- fuelType: A alphanumeric column that contains the type of fuel consumed by the car
- tax: A numeric column with the tax to be added to the price
- mpg: A numeric column with the miles per galon consumed
- engineSize: A numeric column with the size of the car's engine
- dateOfSale: A column that contains the date the car was sold in DD/MM/YYYY format.
- uniqueId: An integer column with a unique increasing number from 0 to 102
- chaos: A column that contains either a number, a number of characters or a null value.
- taxedPrice: A numeric column that contains the taxed price, equal to price + tax/150
- isOldModel: A boolean column that defines if the car was made before 2016
- mileagePlusPrice: An integer column that is equal to the car's mileage plus the price.

formatTests

This dataset is made specifically to validate the FormatCheck. It contains seven different column that each represents a date format. The name of each column defines the format, being a combination of the words ”day”, ”month” and ”year”. For example, the column monthyear contains dates with the format MM/YYYY, while the column monthdayyear contains dates with the format MM/DD/YYYY. One hundred entries are contained within this dataset, all randomly but correctly generated.

The two datasets are used within our tests in the tests folder of the project. All tests are broken down into packages that each contain automated tests related to a specific category:

- The **engine** test package contains the FacadeTests, which validates the full work flow of a client request, as well as the proper registration of a dataset.
- The **model** test package contains the ServerRequestTests which validates the execution process of a ServerRequest.
- The **report** test package which contains all tests validating the individual report formats.
- The **rowchecks** test package, which contains a test class for each row check. Each test class attempts to validate the check both in a good day scenario, as well as a scenario where everything goes wrong.

3.4 Installation Process

This project is hosted on the respective GitHub page and is ready to be launched without requiring any further data analyst interference. In terms of hardware requirements, any data analyst that attempts to test for PrimaryKey or ForeignKey relationships using our checks should have enough disk space to support the disk-based BPlusTree generated for each check, especially in large datasets. As of now, the back-end is invoked via a client class and an example of how it should be done is demonstrated on the Client.java file within the **app** package.

3.5 System Extensibility

Having concluded with the architecture of the current version of our tool, we can now elaborate on the future development of this application and what improvements can be applied. In its current state, we have defined components in such manner that they can easily be replaced and improved. Even during development, several components were marked as Deprecated to showcase previous implementations that were scrapped due to inefficiency. Research done on Data Quality Management has brought up several ideas on further improving our tool, that were ultimately not implemented due to limited resources.

- We provide a vast array of row checks. Yet, with the needs of data cleaning always evolving, the number of checks can always be expanded upon. In our research, we came across [10] where an algorithm for finding keys and non-keys of any dataset is explained in detail. This is a good example of a further expansion of our applications that if turned into a disk-based model could provide a powerful tool to assist with data exploration and validation. Another more abstract idea is using genetic algorithms or trained classifiers to extract a minimal regular expression for format checks, instead of providing a specific format. This can help finding format outliers. While both suggestions approach data analysis more closely than data cleaning, they are important to keep in mind since both fields are closely intertwined.

- Efficiency is always an important part of any tool or application. We mostly focused on maintaining a level or reliability when it comes to large datasets. The use of smart data structures to further speed up some disk-based processes (e.g., Primary/Foreign Key Checks) was crucial to achieve acceptable execution times. This however was not with perfect optimization in mind. The individual checks do not communicate with each other, and thus, we can perform needless checks and unnecessarily increase our execution time. Also, a more efficient conversion of `ClientRequest` to optimized `ServerRequests` could greatly decrease the execution time.

- Several enumerators were used throughout the application to allow the data analyst to select what inner workings will be used by our application (e.g., concerning the type of Report that will be generated, the way the application

handles rejected rows, the types of format, and many others). The purpose of using enumerators is to allow any developer to further expand upon those features with ease. More format types, report types or even more detailed registration responses can prove useful for future implementations and would require maintenance of the respective enumerations.

- Amongst the row checks, the FormatCheck is the one that has the greater potential for improvement. An improved version could check the formatted date for validity, by making sure that the given date can exist from a logical perspective (ex. 31/2/2000). Furthermore, a smarter format checking mechanism that automatically detects the format and the delimiter that separates the format's fields could prove very beneficial to the data analyst.

CHAPTER 4

Experimental Evaluation

In this chapter, we utilise experimentation techniques to measure the performance of our system with respect to the factors that affect its efficiency, as well as to stress-test the limits of our system.

4.1 Experimentation Methodology

Before starting with our experiments, we must first define two things. Firstly, we need to select which row checks will be used to measure the efficiency of our tool. To produce generalized results and observe the full scope of how a data analyst might interact with this application, we will separate our row checks into two categories:

- *Direct Checks*, which we define as checks that require no pre-processing in order to be executed: each row is tested over the check independently from the rest of the data set, and on the basis of its own attribute values only. The following checks belong to this category: DomainTypeCheck, DomainValuesCheck, FormatCheck, NotNullCheck and NumericConstraintCheck
- *Composite Checks*, which we define as checks that require pre-processing before validating any rows: each row is tested for its conformance to a test which

requires the prior initialization of certain objects, or requires some form of holistic information that depends on the iteration of the entire dataset. The following checks belong to this category: BPlusTreeForeignKeyCheck, BPlusTreePrimaryKeyCheck, UserDefinedColumnExpressionRowCheck, UserDefinedConditionalDependencyCheck and UserDefinedRowValueComparisonToAggValueCheck.

The reason behind this taxonomy is to separate the checks that influence the initialization time before a request heavily, from checks that immediately operate on each row locally.

Workloads. To extract realistic results for each of our variables, we define three types of requests that will be executed to measure their respective time:

- *Direct-Only Request*, which is a ClientRequest that is composed of Direct Checks only. In our experiments, the checks chosen to represent this type of requests are:
  - A Column Type check with a Domain Type of Integer
  - A NotNull values check.
  - A Numeric Constraint check.
- *Mixed Request*, which is a ClientRequest that is composed of both Direct and Composite Checks. In our experiments, the checks chosen to represent this type of requests are:
  - A BPlusTree Primary Key Check.
  - A User Defined Aggregation Check, comparing a column's value to the average value of the same column.
  - A BPlusTree Foreign Key Check between a column of the targeted dataset, and a column of a different dataset.
  - A Numeric Constraint check.
  - A Column Type check with a Domain Type of Integer.
  - A NotNull values check.

- *Composite-Only Request*, which is a ClientRequest that is composed of only Composite Checks. The checks chosen to represent this request are:
  - A BPlusTree Primary Key Check.
  - A BPlusTree Foreign Key Check between a column of the targeted dataset, and a column of a different dataset.
  - A User Defined Conditional Check.
  - A User Defined Row Check.
  - A User Defined Aggregation Check, comparing a column's value to the average value of the same column.

Essentially, Direct-Only and Composite-Only requests demonstrate our two extremes, with the data analyst defining either a potentially fast or slow request, respectively. The Mixed request is our average scenario, which in theory should provide a more clear picture on how our application is affected by a variable in normal conditions. Additionally, each check's parameters have been selected according to the dataset being targeted.

Variables involved in the experiments. Our application's performance will be tested with three variables in mind; the number of entries (rows) in a dataset, the number of columns in a dataset and the pollution factor of our data (what percentage of the data set violates a check). All experiments were executed with allocated memory of 2GB using only a driver node for execution to simulate a scenario where Spark does not use parallelism to increase our speed.

4.2 Experiment Results

In this section, we present our results from a number of experiments to assess the efficiency of our system.

4.2.1 Spark Initialization

Before presenting our results, we must first discuss about an external factor that lowers our efficiency, and that is Spark. Spark allows us to process file that might not fit fully within our available memory, and thus is the perfect framework for our application. To facilitate its operation, however, Spark has an initialization time (or warm-up time) that will run once when our application starts. This time is on average 3.61 seconds with a 0.6 seconds standard deviation.

4.2.2 Dataset Registration

Another thing to measure is the time required to register a dataset before executing any requests. For our experiments, we make use of nine datasets, randomly generated with certain features for each dataset, aimed to assist on a specific experiment. The recorded variables of each dataset that we chose are the file's size, the number of rows and the number of columns.

Dataset	File Size	Rows	Columns	Average Registration Time (seconds)	Standard Deviation (seconds)
dataset10k	0.3Mb	10k	5	<0.1	<0.1
dataset100k	3Mb	100k	5	0.212	0.05
dataset1mrows	31Mb	1000k	5	0.22	0.05
dataset10mrows	312Mb	10M	5	0.23	0.05
dataset100mrows	3.2Gb	100M	5	0.32	0.07
dataset10col	5.9Mb	100k	10	0.21	0.07
dataset100col	58Mb	100k	100	0.17	0.03
dataset1000col	581Mb	100k	1000	0.26	0.08
dataset1percent	1.5Mb	100k	4	0.2	0.07
dataset5percent	1.5Mb	100k	4	0.15	0.03
dataset10percent	1.5Mb	100k	4	0.16	0.04

Table 4.1: Registration per Dataset Time Analysis

It is notable that the registration times, on average, are similar between the datasets, regardless of size or schema. This is expected due to Spark's lazy evaluation, which

essentially does not store the dataset in memory, and instead keeps a reference to it for future transformations.

4.2.3 Spark Overhead

In our initial experiments with small sizes for both the number of rows and columns, we observed a non-linear growth in execution times. This growth however seems to somewhat turn into a linear one as the workload increases. To explain this anomaly in our data, we made use of Spark's embedded profiling application. This revealed, that Spark has an overhead processing time for each query executed. This overhead is relatively small, thus we do not account for it during our experiments, since it only affects the execution time of requests that handle small workloads.

Dataset	Direct-Only Overhead (s)	Complex Overhead (s)	Composite-Only Overhead
dataset10k	0.17	0.22	0.413
dataset100k	0.12	0.23	0.42
dataset1mrows	0.1	0.27	0.31
dataset3col	0.3	0.45	0.63
dataset10col	0.21	0.37	0.65
dataset100col	0.31	0.4	0.33
dataset1percent	0.1	0.2	0.33
dataset5percent	0.11	0.21	0.43
dataset10percent	0.1	0.23	0.31

Table 4.2: Dataset Overhead Times

Overall, we observe an overhead that consumes less than a second of our time calculation, with Direct-Only requests having a 0.16 mean with a 0.08 standard deviation, Complex having a 0.28 mean and a 0.08 standard deviation, and finally Composite-Only with a 0.42 mean and a 0.12 standard deviation. In conclusion, the overhead values are similar amongst the requests, only seemingly being slightly affected by the type of checks being executed.

4.2.4 Efficiency Assessment with respect to the Number of Rows of a dataset

To begin with the three core experiments, we first need to see how the number of entries (or rows) of a dataset affects the execution time of our three chosen requests. To assist us with this, we will focus on four datasets; ”dataset10k”, ”dataset100k”, ”dataset1M”, ”dataset10M”. These datasets have the same randomly generated schema of five columns, all containing either numeric or non-numeric values with no null values. The main difference of these datasets, as their names suggest, is the number of entries each dataset has.

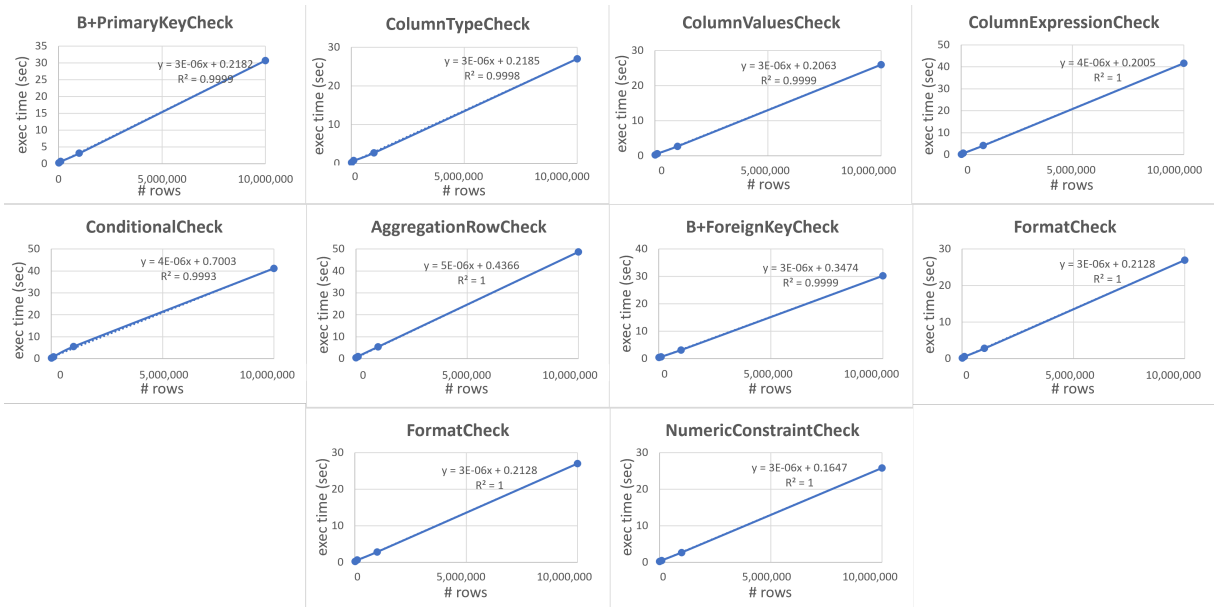

Figure 4.1: Individual Check’s Execution Time as a function of row size

Our application focuses on handling large datasets, so the number of entries is of especial interest to our experiments. We begin with testing each individual check alone in order to establish the overall complexity. Our experiments confirm the linear relationship between the number of rows and execution time, as seen in Figure 4.1. Something worth mentioning is the sudden increase in execution time between the 10k and 100k datasets, which, however, is most probably due to the overhead applied to all checks from Spark (an alternative explanation could be the intense data serialization required).

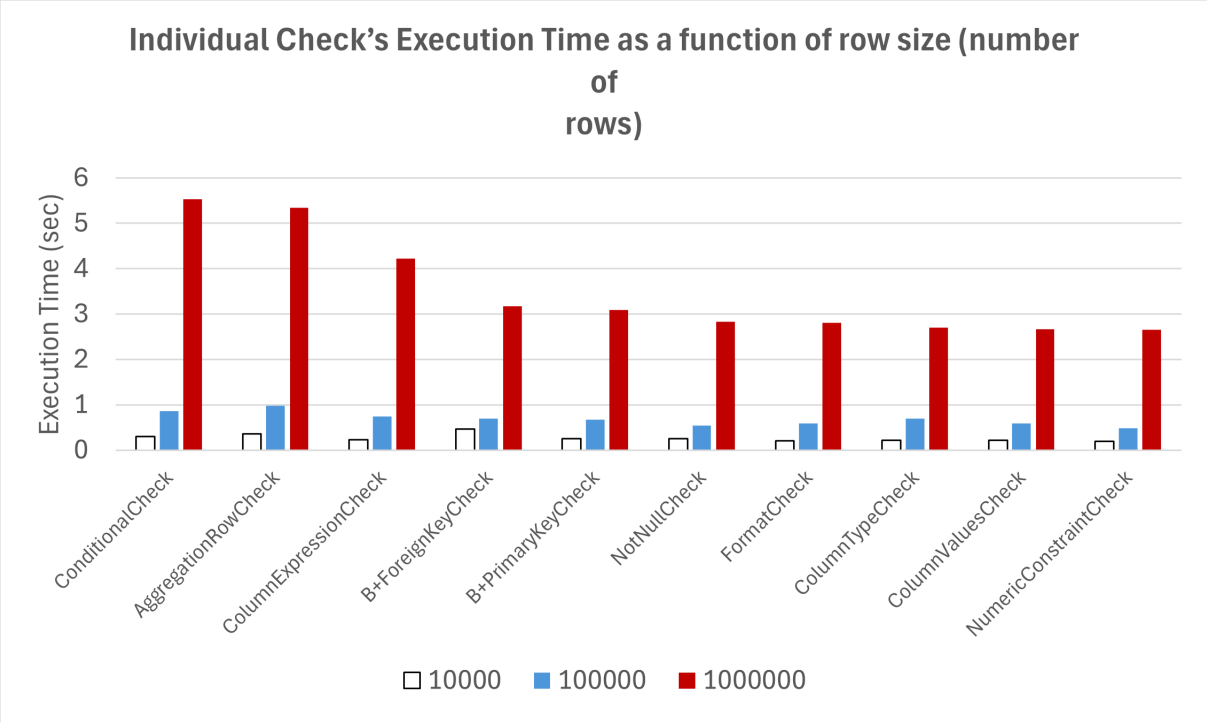

Figure 4.2: Individual Check's Execution Time as a function of row size (number of rows).

In order to complete our experiments on the number of rows, we further analyze the results presented on the bar plots of Figures 4.2 and 4.3. Both individually for each check, and as complete requests, we observe an increase in execution time the more entries are present in a dataset. This behaviour is expected, as the application and each check requires more time to iterate over all the rows. Execution time is also affected by the allocated memory and the dataset's size; with datasets that do not fully fit in our memory requiring more time for Spark to transfer our data between memory and disk. When testing with an 100 Million rows dataset (3GB in size), we observed an expected delay in execution time, slightly deviating from the expected linear result.

In regards to our three chosen requests, the Direct-Only request produces the fastest mean execution time, followed by the Mixed and finally the Composite-Only. Once more, this behaviour is expected, since the processing time for our Direct checks is faster when compared to the Composite ones.

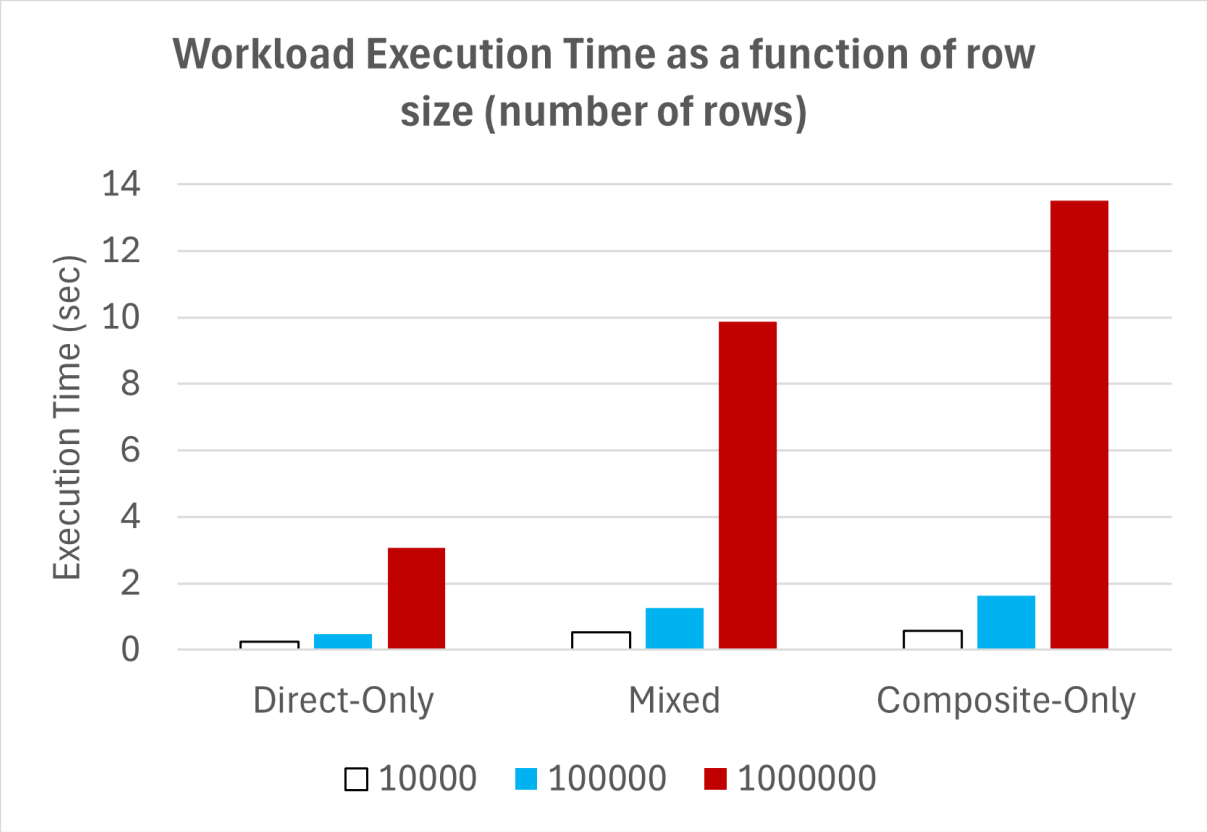

Figure 4.3: Workload Execution Time as a function of row size (number of rows).

Regardless of time, a point of interest is the ability of our application to handle big datasets given enough time. During the trials, we checked how our application would handle datasets with 10 and 100 million rows. In order to compare it, we switched disk-based components, such as the BPlusTree checks, with their respective components that fully rely on our memory. As expected, the BPlusTree components managed to eventually produce results, while the on-memory components abruptly ended with a memory overflow exception.

4.2.5 Efficiency Assessment with respect to the Number of Columns of a dataset

Our second variable, and therefore experiment, is the number of columns of a dataset and how it affects our execution time. To assist us with this, we make use of three datasets; the ”dataset10col”, ”dataset100col” and ”dataset1000col”, each having 10, 100 and 1000 columns respectively. Besides the number of columns, all datasets have 100k entries and randomly not-null generated alphanumeric data.

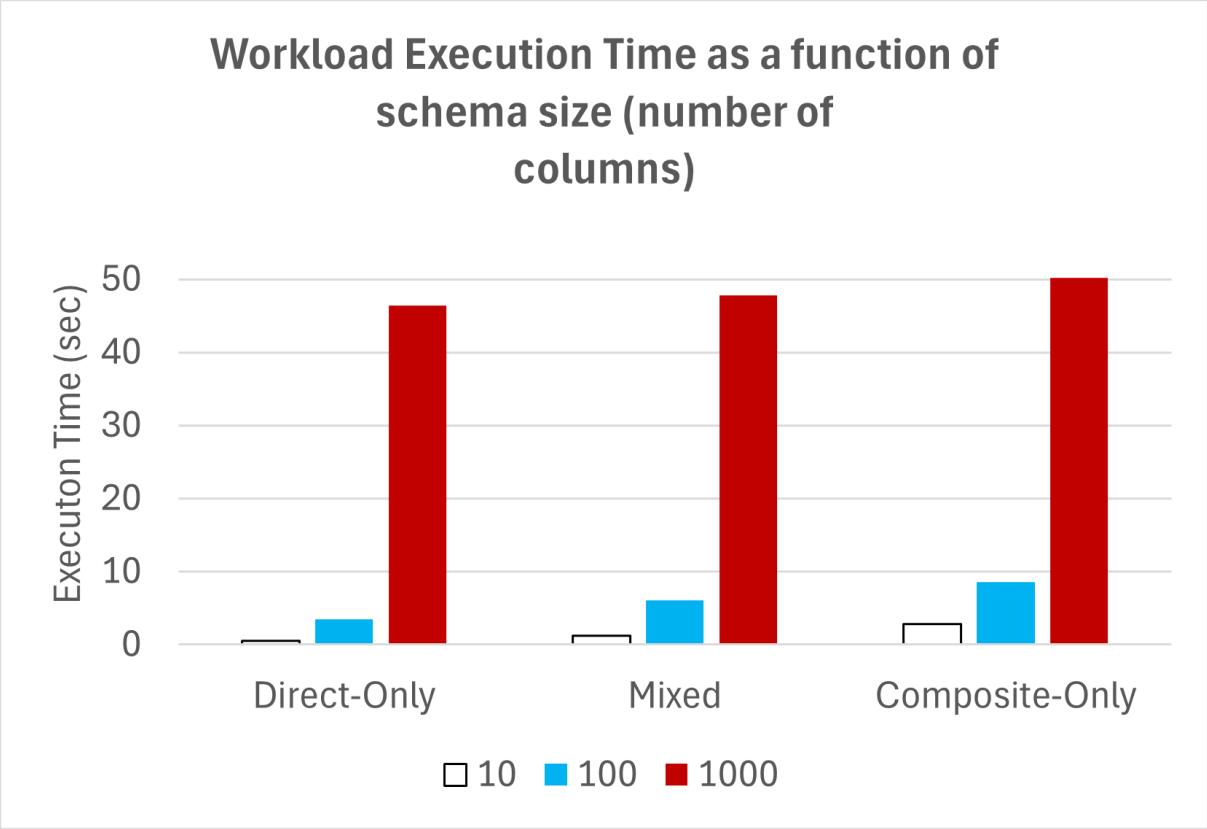

Figure 4.4: Workload Execution Time as a function of schema size (number of columns).

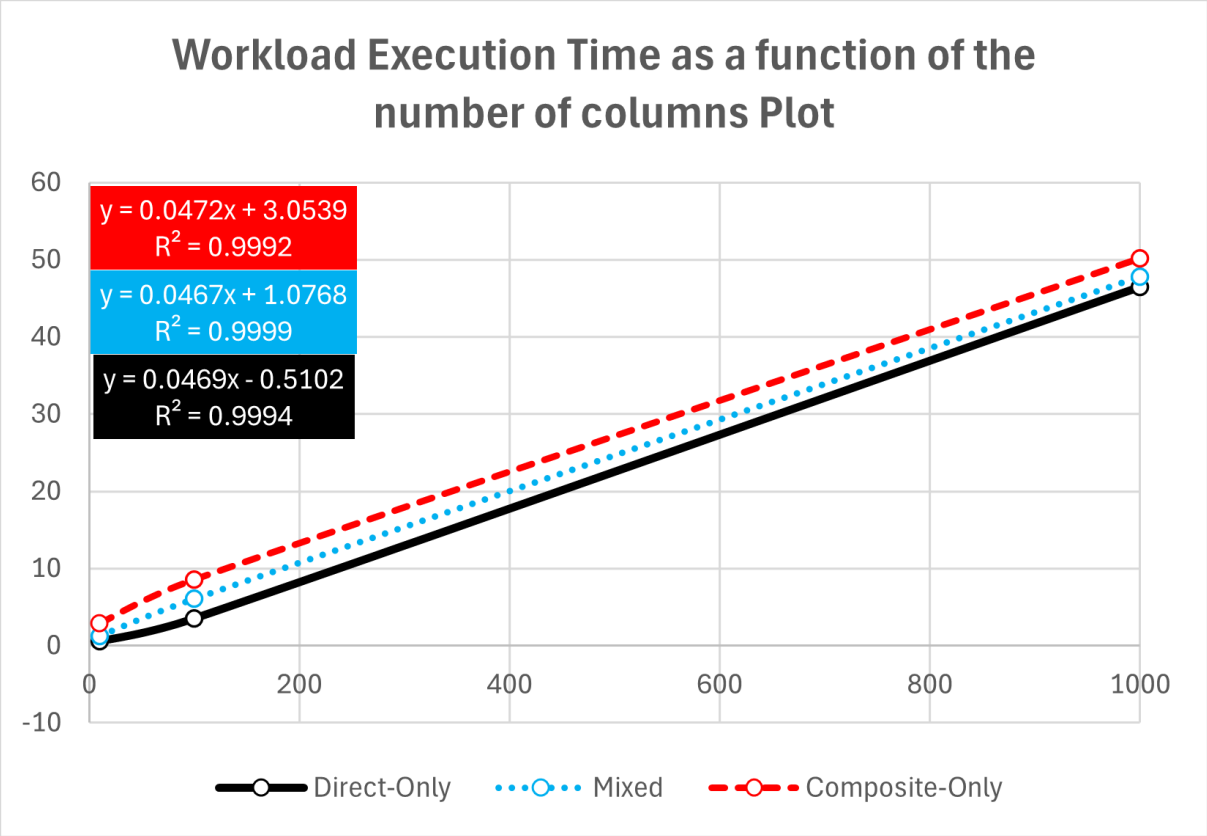

Figure 4.5: Workload Execution Time as a function of schema size (number of columns) Plot.

By examining the presented results in figure 4.4 and 4.5, we observe a linear increase in execution time as the number of columns increases. Interestingly, while the number of entries iterated remain the same, we can conclude the the time required for Spark to fetch a single row for processing increases as our columns increase. Between the requests, the average time difference is relatively small, with the largest difference being between Direct-Only and Composite-Only at 4 seconds during the 1000 columns experiment. We can conclude that the number of columns increases our overall execution time, with almost no correlation to the number or type of checks we execute.

4.2.6 Efficiency Assessment with respect to the Pollution of a dataset

Finally, for our third variable, we will experiment on the level of uncleanliness or pollution of our data. Pollution is an abstract term, since all checks define a value as ”unclean” in their own unique way.

We define a dataset as p-polluted with respect to a workload if each test of the workload isolates p% cells as violating the test. A row might be violating multiple checks under this testing.

For this experiment, we will utilize 3 data sets, with 1%, 5%, and 10% pollution. The datasets used to test pollution have four columns and 100k entries. To give a concrete example, for a dataset with p% pollution we have the following columns, named after the tests applied to them:

- uniqueId: A column that signifies the unique row number of the entry. There is a p% chance of this number being a duplicate.
- lowerThan10: A column that have numbers between 0 and 9, with a p% chance of being above (or equal to) 10.
- notNull: A column that contains a random not-null value, with a p% chance of being null.
- number: A column that contains a random number, with a p% chance of being a non-numeric string.

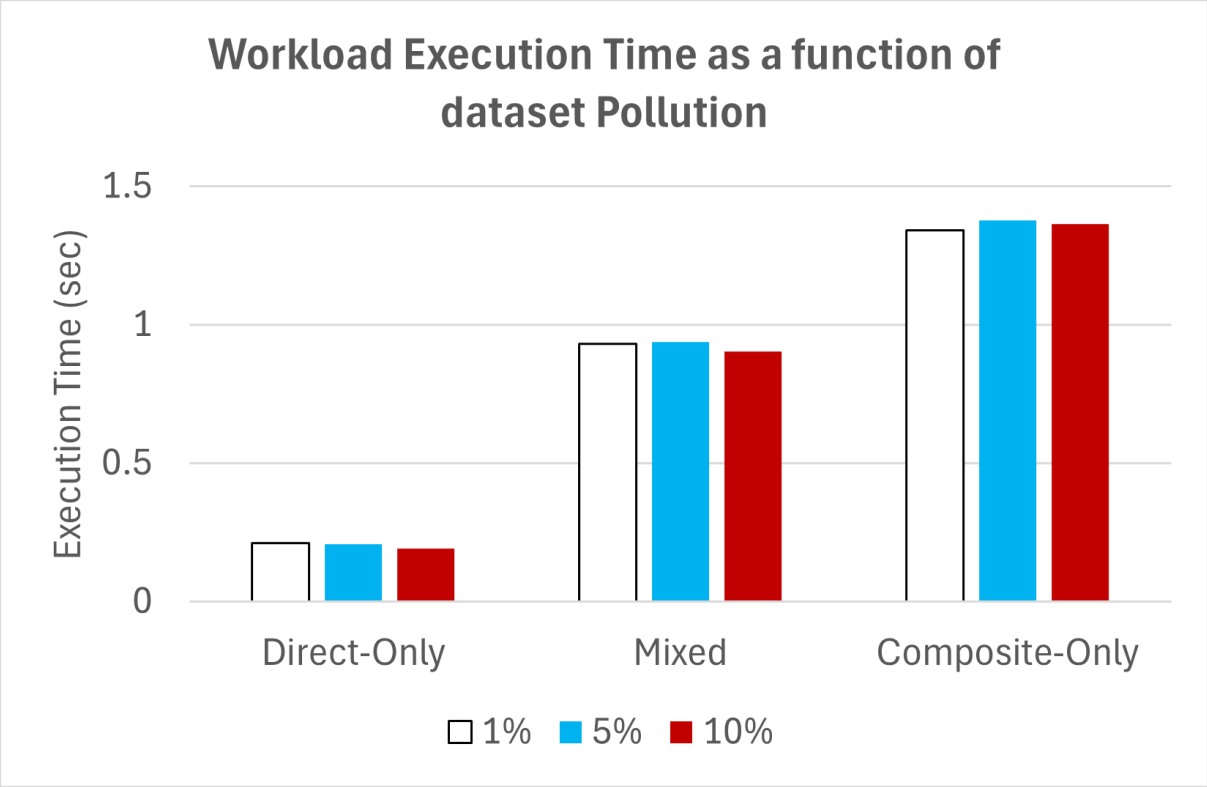

Figure 4.6: Workload Execution Time as a function of dataset Pollution

The results are presented in 4.6, and reveal no significant change across pollution levels. As expected, different percentages of rejected entries do not affect our execution times, proving that our application is ideal for handling both highly clean and unclean datasets.

A point of interest is that this experiment evaluates the ability to handle rejected entries, but not invalid. Invalid entries appear either due to a system error, an unexpected null value or simply due to human error such as misspelling a column during the creation of checks.

CHAPTER 5

Epilogue

5.1 Conclusions

In this thesis, we studied the concept of data analysis and through the meticulous inspection of studies concerning the quality of data and tools commonly used by data analyst to enforce cleanliness, we isolated the key features a tool responsible for quality enforcement should have. We embarked on the implementation phase of our application's design, with our foremost priorities centered on creating a versatile tool capable of addressing diverse data cleaning needs, while also ensuring its adaptability to future developments in both software engineering and the ever-evolving landscape of data cleaning methodologies. Finally, we experimented on our application, testing its limits and reactions to a variety of changing conditions, and figured out its strengths and weaknesses.

With our thesis, we advanced the field of data analysis by proposing a structure of requirements, and designing an extensible data cleaning tool. This tool is capable of incorporating tailed data quality checks and seamlessly handling raw datasets. Furthermore, we presented a functional data cleaning tool, implemented using Java Spark

that is able to operate over data hosted in a Spark-based environment. Our tool effectively addressed the inherent complexity of data cleaning tasks, and through rigorous experimentation, we evaluated the efficiency of our tool across varying data sizes and levels of pollution, with our findings confirming the anticipated linear performance complexity of the tool, thereby validating its effectiveness.

5.2 Future Work

With such an open-ended subject that our application tackles, there is much to add, and with our focus being versatility and flexibility, we can certainly agree that there is much improvement to be facilitated. Some future work that can be conducted on this tool is:

- From an architectural perspective, in Section 3.5, we have already mentioned improvements and additions that can enhance the experience of a data analyst.
- From our experiments, we have concluded that high levels of pollution do not affect our efficiency. But due to the way Java handles exceptions, a large amount of invalid entries can increase greatly our execution times. Catching those exceptions in a more efficient manner can improve our performance in such cases where the number of invalid entries is great.
- There is room for improvements in terms of efficiency. The checks themselves, as well as the individual components of the entire application can be enhanced to improve the overall execution time. For example, performing some preprocessing such as sorting before performing a Numeric Constraint Check in order to avoid checking all entries, could prove beneficial.
- Finally, to enhance the friendliness towards our user, a graphical user interface could prove beneficial. Our application is mainly meant to be used by data analysts, who have some knowledge of Java, but regardless a form of interface could assist them on a human level of efficiency.

BIBLIOGRAPHY

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015.

[2] Peter van Nederpelt Andrew Black. Dimensions of data quality (ddq). Technical report, Stichting Dama NL, De Kleefse Kamp 60, 6674 DT Herveld, September 2020.

[3] Apache spark. `https://spark.apache.org/`. Accessed 23 August, 2023.

[4] Ataccama. Ataccama quality. https://www.ataccama.com/platform/data-quality, 2022.

[5] Oracle berkeley db. `https://www.oracle.com/au/database/technologies/related/berkeleydb.html`. Accessed 20 January, 2024.

[6] Frank Kiwy. Joshua bloch's effective java builder. *Java Magazine*, 2021. Accessed 12 January, 2024.

[7] Precicly. Spectrum quality. https://www.precisely.com/product/precisely-spectrum-quality/spectrum-quality, 2021.

[8] Maria Priestley, Fionntán O'Donnell, and Elena Simperl. A survey of data quality requirements that matter in ML development pipelines. *ACM J. Data Inf. Qual.*, 15(2):11:1–11:39, 2023.

[9] SAS. Sas data quality. https://www.sas.com/en_us/software/data-quality.html, 2021.

[10] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L.

Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 691–702. ACM, 2006.

[11] Dataflair on dataframes. https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/. Accessed 23 August, 2023.

[12] Dataflair on rdds. https://data-flair.training/blogs/spark-rdd-tutorial/. Accessed 23 August, 2023.

[13] Dataflair on datasets. https://data-flair.training/blogs/apache-spark-dataset-tutorial/. Accessed 23 August, 2023.

[14] Larysa Visengeriyeva and Ziawasch Abedjan. Anatomy of metadata for data curation. *ACM J. Data Inf. Qual.*, 12(3):16:1–16:30, 2020.