De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Query guided anomaly detection in event data

Pieter Olaerts

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Frank NEVEN

2019
2020

UHASSELT

KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Query guided anomaly detection in event data

Pieter Olaerts

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Frank NEVEN

Acknowledgements

First of all, I would like to thank my promoter Prof. Dr. Frank Neven for the good supervision throughout the thesis. In addition, I would also like to acknowledge Kris Peeters and Data-Minded for providing the problems they experience in practice and the necessary data sets to perform tests on. As a result, they not only provided the incentive to conduct research in this very interesting topic, but also made it possible to do this with the necessary resources.

As this thesis marks the end of a 6 year education at the University of Hasselt, I would like to thank my parents for always supporting me and being ready when needed. In addition, a special thanks to my fellow students Erik, Niels and Mathias, who I now can call friends and have ensured that I can look back on this period with a big smile.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research questions . . . 2

2 Data platforms 5
- 2.1 Data lake . . . 5
- 2.2 Data mesh . . . 6
  - 2.2.1 Data and distributed domain . . . 7
  - 2.2.2 Data and product thinking convergence . . . 8
  - 2.2.3 Data and self-serve platform design convergence . . . 9
- 2.3 Conclusion . . . 9

3 Data cleaning in theory 11
- 3.1 Outlier detection . . . 11
  - 3.1.1 Taxonomy of outlier detection methods . . . 11
  - 3.1.2 Statistics-based outlier detection . . . 13
- 3.2 Clustering . . . 15
  - 3.2.1 Clustering techniques . . . 15
  - 3.2.2 K-means . . . 15
- 3.3 Data profiling . . . 17
  - 3.3.1 Single-Column analysis . . . 17
  - 3.3.2 Dependency Discovery . . . 18
  - 3.3.3 Metanome . . . 21
- 3.4 Machine learning approach . . . 21
  - 3.4.1 Introduction . . . 21
  - 3.4.2 Machine learning for data deduplication . . . 22
  - 3.4.3 Conclusion . . . 22
- 3.5 Limitations of data error detection . . . 23
  - 3.5.1 Current state . . . 23
  - 3.5.2 Setup . . . 24
  - 3.5.3 Evaluation . . . 25
  - 3.5.4 Conclusion . . . 25

4 Data cleaning in practice 27
- 4.1 Trifacta . . . 27
- 4.2 Tableau . . . 28
- 4.3 OpenRefine . . . 29
- 4.4 Tamr . . . 29
- 4.5 Deequ . . . 30
- 4.6 Great Expectations . . . 31
- 4.7 Conclusion . . . 31

5 Anomaly detection in event data 33
5.1 Data . . . 33
5.1.1 Event data . . . 33
5.1.2 Data structure: NMBS . . . 33
5.2 Aggregation . . . 34
5.3 Challenges . . . 35
5.3.1 Queries . . . 35

6 A query ranking algorithm 37
6.1 Based on statistics . . . 37
6.1.1 Approach . . . 37
6.1.2 Implementation . . . 39
6.1.3 Results . . . 40
6.2 Based on clustering . . . 40
6.2.1 Approach . . . 40
6.2.2 Implementation . . . 41
6.2.3 Results . . . 43
6.3 Based on consistency . . . 43
6.3.1 Approach . . . 43
6.3.2 Implementation . . . 44
6.3.3 Results . . . 45

7 A monitoring system 47
7.1 Initialisation . . . 47
7.1.1 Cleaning and filtering the data set . . . 48
7.1.2 Searching for clusters . . . 48
7.1.3 Selecting the best queries . . . 48
7.2 Monitoring . . . 49
7.2.1 Adding a new day . . . 49
7.2.2 Summarize the results . . . 49
7.2.3 Recalculations . . . 50
7.3 Extensions . . . 50

8 Experiments 53
8.1 NMBS . . . 53
8.2 DeLijn . . . 54
8.2.1 Data . . . 55
8.2.2 Implementation . . . 56
8.2.3 Results . . . 56
8.3 Conclusion . . . 58

9 Conclusion 63

10 Future work 65

Appendices 67

A System specifications 69
A.1 Setup . . . 69
A.1.1 System overview . . . 69
A.1.2 Installation . . . 69
A.1.3 Metanome . . . 70
A.2 Structure . . . 70

A.2.1 Algorithms . . . 70
A.2.2 Projects . . . 70
A.3 API . . . 71
A.4 Screenshots . . . 80

B Nederlandse Samenvatting 93
B.1 Inleiding . . . 93
B.2 Aanpak . . . 94
B.2.1 Event data . . . 94
B.2.2 Aggregaties . . . 95
B.2.3 Uitdagingen . . . 95
B.2.4 Query . . . 96
B.3 Algoritmen . . . 96
B.3.1 Filteren . . . 96
B.3.2 Clusteren . . . 97
B.3.3 Consistentie . . . 97
B.4 Conclusie . . . 98
B.5 Vervolg . . . 99

Chapter 1

Introduction

1.1 Motivation

Today, every company produces data in all kinds of structures, coming from the different departments (sales, marketing,..) of the company. Nowadays in the so-called modern companies there is also *internet data*, such as *clickstreams*, which contain the behavior of visitors on the company web page. All this data ends up in *data warehouses* or *data lakes*, both can be seen as one large database. This data can then serve for analysis or for use in machine learning (ML), but before this can happen, it must first be made usable. Making it usable mainly consists of cleaning the data, this is very important in both cases. The data used for analysis may indirectly influence future business decisions. When used in ML, however, the data can be used in multiple phases. In all phases, non-clean data has a bad influence on the accuracy of the model [14]. For example, the functions found when training the model depend on the used training data. And in a later phase, it is necessary that the data fed to the model is similar to the training data. To clean the data, there are dozens of tools that can be divided into several categories, one may focuses more on a specific aspect of data cleaning, while another may offer more general functionalities. An example of this is Trifacta [22], a very popular tool with a wide range of options:

- Detect outliers in a numeric column.
- Detect length deviations in an alphanumeric column.
- Detect data type deviations in a column (e.g. a string in a numeric column).
- Recognize a common pattern and detect deviations from this pattern.
- ...

This tool is therefore used daily by many companies for data cleaning. Part of data cleaning is thus the detection of outliers, or so-called anomalies. The data type or the number of (unique) values is then checked for each column, among other things. Furthermore, dependencies can be imposed between columns that must apply throughout the data set. However, these *data cleaning* tools lack the ability to define rules regarding the actual content of the data set. For example: *the train from Genk to Blankenberge runs 10 times every day* or *every month an average of 100 euros is spent on Jef's bank account in the 'Sports & Culture' category.* These rules can be partly expressed on the basis of an aggregation in the form of a query. For example, one can find out, from the data, how many times the train from Genk to Blankenberge has traveled with the following query:

```
SELECT COUNT(DISTINCT train number)
FROM data
GROUP BY route
WHERE route = "Genk to Blankenberge"
```

Then, one need a predetermined timetable or historical data to check whether the result of this query can be considered 'normal'. This approach causes some problems:

- In most cases there is a need for a *domain expert* to select the interesting queries.
- Data sets nowadays quickly take on a considerable size, making it not feasible to manually search and define all interesting queries.

1.2 Research questions

The aim of this thesis is to investigate whether it is possible to find an algorithm that can (semi-) automatically detect interesting queries. So that outliers can then be detected and monitored based on these queries. We will focus on a particular type of data, namely *event data*. This gives us the following research questions:

- Can we filter out queries without executing them?
- Can we cluster the data (of several days) based on queries?
- Can we detect interesting queries?
  - How relevant are recommended queries? Are these meaningful?
  - How unique are these queries? Are there any queries with the same meaning?
- Can we monitor these queries?

This work consists of two main parts: the first part provides the necessary background to understand the full story (Chapters 2,3 and 4), the second part provides a solution to the problem (Chapters 5,6,7 and 8).

In Chapter 2 we first see that *Data warehouses* were previously used for data storage and how this has evolved into *Data lakes*. The latter appears to show the same defects as its predecessors. The second part of this chapter provides an answer in the form of a solution to the current pain points.

Chapter 3 then provides a summary of all kinds of *data cleaning* techniques. For example, various outlier detection techniques are discussed, such as statistics-based outlier detection. In addition, we also look at *data profiling*, which is closely related to *data cleaning*. Subsequently, it is briefly discussed how ML can be of help. Finally, we see what limitations there are currently regarding data error detection.

Next, in Chapter 4 some well-known and less well-known *data cleaning* tools are discussed. For example, we go over the functionalities of Trifacta and Deequ. The conclusion we could draw from this, brings us to the problems that we are trying to solve in this work.

In Chapter 5, we first discuss what *event data* actually entails and which data we used. Then it is explained, in more detail, how we want to solve the problems and what challenges there are.

Then, in Chapter 6, the process that tries to derive interesting queries using various techniques is discussed. For example, one of these techniques attempt to exclude queries early on the basis of column statistics. For each technique, the general approach is first explained, then it is looked at how it was implemented and finally the results that could be achieved with it are discussed.

Thereafter, in Chapter 7 we see how all techniques form previous chapter are brought together in one system. The elaborated system consists of two phases: initialization and monitoring. In the first phase, a new project is created and the user selects some queries from the list of queries proposed by the system, among other things. The second phase consists of daily monitoring

of these selecting queries and detecting any outliers. At the end of the chapter, some possible extensions of the system are suggested.

The system was continuously tested with the data set that was available. In Chapter 8, we discuss the final results of one of the experiments. So here an answer is provided to the question of which queries are now being proposed as interesting by the system. In addition, we also test another data set, to see which adjustments are needed and how well the system still works. This is to get a better idea of how the system should be adapted in the future to be able to process other similar data.

Finally, in Chapters 9 and 10, we make a final conclusion and we look ahead in which areas the problem can be further investigated in the future.

Chapter 2

Data platforms

This current chapter is an adaption of [4]. Many companies today are trying to become a data-driven organization. This means that they use data and analytics to improve business processes and business decisions. But despite the investments in building data and intelligence platforms, the organisations find the results mediocre. In this chapter we will first look at the history of data platform architectures, then we will gain a better insight in the failures of the previous and current generation. In the last section we introduce a (possible) new enterprise data architecture, called data mesh.

2.1 Data lake

In a first attempt to get more value from the different data sources of a company, a system was created where all the generated data was first brought together, called *data warehouses*. Then infer *business intelligence* from this collected data, which may help the company improve their processes and business decisions. One of the disadvantages of a data warehouse is that it works on structured (with a data scheme) and processed data. Due to the increasing demand to also store and analyze unstructured (raw) data, a new concept, the *data lake* was introduced. A data lake is capable of handling large amounts of unstructured data at low costs. Furthermore, data lakes provided a more flexible environment, every data analyst who uses a certain data file can clean and transform it to his needs. This generation is now evolved into a system to meet modern requirements, such as analyzing real-time data, cloud-based storage services, ML platforms and many others.

But despite the improvements of the current generation of data platforms, they have the same characteristics that led to the failure of previous generations. The consequences of this are that, among other things, the data platforms are difficult to maintain and need specialized data engineers. These characteristics are:

- Centralized and monolithic
- Coupled pipeline decomposition
- Siloed and hyper-specialized ownership

Centralized and monolithic

Although domain-driven design and bounded contexts are successfully applied in operational systems nowadays, these domain concepts are ignored in the data platforms. One data platform where all data, in all kinds of formats, from different departments of the company comes together. That subsequently must be processed (clean, enrich, transform) in order to meet the needs of the various consumers. So instead of domain oriented data ownership, data platforms have

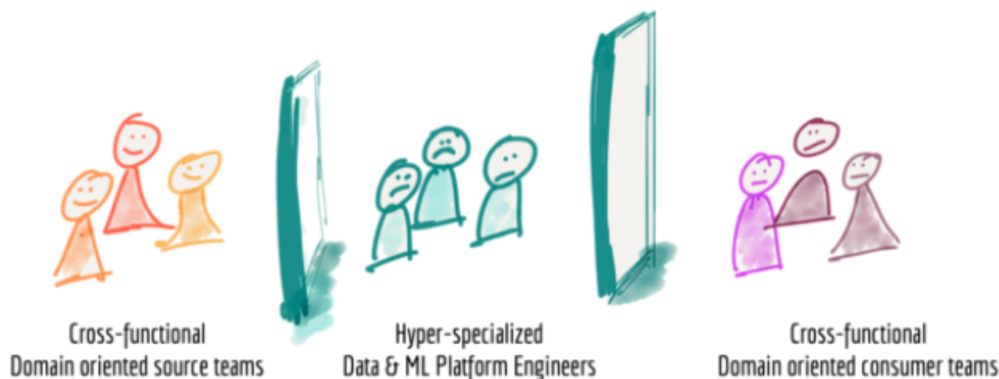


Figure 2.1: Siloed hyper-specialized data platforms team

centralized domain agnostic data ownership. This may work for organizations that have a smaller and simpler domain, but it will fail for companies that have a rich domain with a large number of (data) sources and a diverse set of consumers. In other words, the current solution does not scale organizationally.

Coupled pipeline decomposition

To create a scalable data platform that can withstand the constant growth of new resources and new data platform users, architects split the data platform into a pipeline of data processing stages. A pipeline that, at a high level, provides functional cohesion around the technical implementation of processing data. This model provides a certain level of scalability by assigning teams to each stage of the pipeline. But this results in high coupling between the stages of the pipeline to deliver value, the teams are dependent on each other. Whenever something needs to be changed in the data platform, it has to be synchronized with all the teams. This limits the ability to achieve higher speed and scale in response to new consumers and data sources.

Siloed and hyper-specialized ownership

The platform is built and is maintained by a group of hyper-specialized data engineers (as shown in Figure 2.1). Data engineers who know how to deal with big data, but have little or no knowledge of the business and the domain. Nevertheless, they must process the data and convert it into meaningful and correct data that must serve different purposes. If we look into practice we find disconnected source teams, frustrated users and an overloaded data platform team.

2.2 Data mesh

To solve the problems mentioned in the previous section, there is a need for a paradigm shift in the next generation of data platforms. A data platform architecture is proposed that is a convergence of the following techniques: *Distributed domain driven architecture*, *self-serve platform design* and *product thinking with data*. Techniques that each had a positive impact on modernizing the technical foundations of the operational systems.

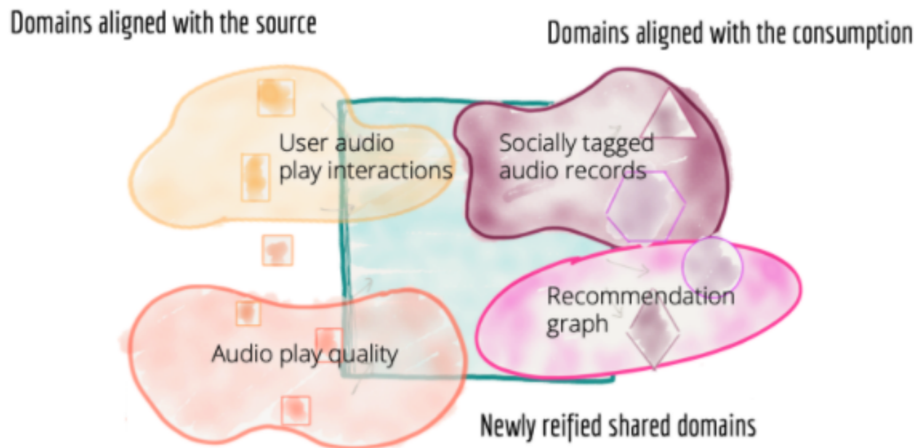


Figure 2.2: Decomposing the architecture and teams owning the data based on domains - source, consumer, and newly created shared domains

2.2.1 Data and distributed domain

Domain oriented data decomposition and ownership

To move away from the monolithic data platform, one must stop centralizing all data from each domain in one place. Domains (as shown in Figure 2.2) need to host and serve their data sets themselves in an easily consumable way. So a *push and ingest* model gives way to a *serving and pull* model across all domains.

Source oriented domain data

Source domain data sets are the most foundational data sets, when created they are close to the raw data and are not yet suitable for usage by a consumer. Because business facts do not change that frequently and the source domain data sets are based upon it, they change less often. These data sets are expected to be always available and always up-to-date. This way, when developing a data-driven and intelligence services, the organization can always fall back on the business facts and create new aggregations or projections.

Consumer oriented and shared domain data

Consumer domain data sets and the teams responsible for these sets aim to satisfy a number of related use cases. Compared to the source domain data sets, the consumer domain data sets have to undergo some structural changes. A team in the domain-oriented data platform should be able to easily generate such a data set given the source.

Distributed pipelines as domain internal implementation

Now that the domains themselves own their data, the data pipeline becomes an internal complexity and implementation of each domain (as shown in in Figure 2.3). A source domain must provide the data pipeline stages such as cleansing and deduplicating itself, a consumer domain will implement aggregations itself. So each domain is responsible for the implementation of their own data pipeline and the quality of the data that they ultimately deliver.

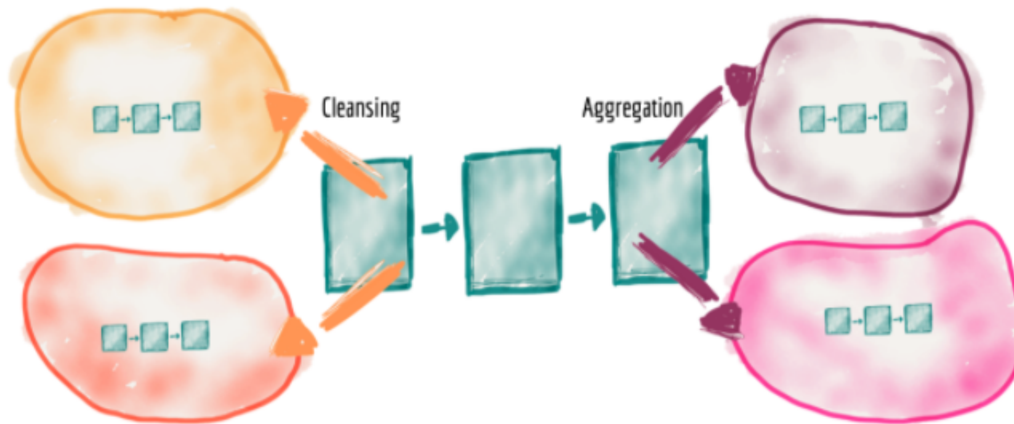


Figure 2.3: Moving the pipeline and the internal implementation of the domain to the domains themselves

2.2.2 Data and product thinking convergence

Now the business domains have control over the data ownership and the data pipeline implementation, applying product thinking is one of the things that is important to continue to provide accessibility, usability and harmonization of the distributed data sets. In the form of APIs, accessible to the rest of the organization, operational domain teams have already applied product thinking in the last decade. Each team strives to create the best developer experience for their domain APIs. The APIs are the building blocks for creating and providing higher order value and functionality. The domain data teams must also apply this way of thinking for the data sets they provide to form a successful data platform. Their data sets are the products for the rest of the organization that need to be delivered with the best possible service. In order to reach this kind of service, the domain data products need to have the following characteristics:

Discoverable

Data consumers, engineers and scientist in an organisation must be able to find a dataset of their interest easily. This can be done by a data catalogue of all available data products with their meta information such as their source, owners, samples, etc.

Addressable

It is self-serving for any organization to clearly define standards for addressability of data sets in order to remove the friction that arises in a multiform environment, when finding and accessing information. Consequently providing each data product with a unique address according to a global convention should make it straightforward to access a data product, once it is discovered.

Trustworthy and truthful

It is important that consumers have confidence in the data products. To improve this, every data product must be provided with acceptable service level objectives. This concerns the reliability of the data, how closely the data represents the truth or the probability that the data was generated. Applying techniques such as cleansing, testing data integrity and making useful metadata available can help achieve this goal. The service level objectives of course depend on the product.

Self-describing semantics and syntax

Quality products can be independently discovered, understood and consumed. Ideally, each data set should have well-described semantics and syntax, along with some data samples as an example. This is the starting point to deliver self-serve data assets.

Inter-operable and governed by global standards

To effectively correlating data across domains and merge them together, in a distributed domain data architecture, raises the need for certain standards and harmonization rules. These rules need to be determined on global scale to enable interoperability between polyglot domains. Interoperability and standardization of communications, governed globally, is one of the foundational pillars for building distributed systems.

Domain data cross-functional teams

To enable each domain to offer their data as a product, each domain must be occupied by a cross-functional team consisting of data product owners and data engineers. A data product owner makes all decisions regarding the evolution of the data product. Continuously verifying and improving data quality and keeping consumers satisfied are some of a data product owner's tasks. Data engineers are necessary in order to build and operate the internal data pipelines of the domains.

2.2.3 Data and self-serve platform design convergence

Since each domain now owns its data, each domain must also maintain its own data pipeline technology stack. In order to avoid unnecessary duplication, there is a need for a common data infrastructure platform, which provides the necessary technology for the domains to capture, process, store and serve their data product. It is then necessary that the data infrastructure platform does not contain domain specific concepts or business logic, hides all underlying complexity and serves the data infrastructure components in the form of self-service. The self-service ensures that new data products can be implemented quickly.

2.3 Conclusion

The datamesh platform is therefore a distributed data architecture where management is centralized and there is inter-operability between the different polymorphic entities through standardization. All this is possible through the shared and harmonized self-service data infrastructure. The data lakes and data warehouses that we all know can actually be used as the nodes of this system. The data lake may no longer be the center of the entire system, but some of its principles will still be applied. An example of such a principle is the provision of immutable data for analysis to the source oriented domain data products. It is now up to the engineers and business leaders of the organizations to realize that the existing paradigm will again stumble over the same problems as previous generations of data platforms.

Chapter 3

Data cleaning in theory

The first part of this chapter is a summary of all kinds of techniques that are used in data cleaning, among other things. Section 3.1 provides an overview of outlier techniques, 3.2 briefly discusses clustering and 3.3 is more about data profiling, which is closely related to data cleaning. This knowledge is required to fully understand the applications in the second part of the thesis used to create the tool. The second part of the chapter briefly discusses how machine learning can be used for data cleaning, Section 3.4, and what limitations are encountered in the search for data errors in the last section, 3.5.

3.1 Outlier detection

An outlier is an observation that deviates from the other data point in consideration and therefore exhibits abnormal behavior. As a result of this, outliers can sometimes strongly influence the results and cause that the wrong analyzes are ultimately made. This is the reason why it is important that outliers can be identified and possibly be excluded in a later phase. There are different methods for detecting outliers, each with its pros and cons. A suitable method should be selected depending on the application domain. The following section will give an overview of the different methods of outlier detection. Thereafter, in Section 3.1.2, the statics-based detection method will be discussed in more detail, as it is the most appropriate method for the created tool. More details can be found in [9], of which this section is a summary.

3.1.1 Taxonomy of outlier detection methods

Outlier detection methods mainly differ in how they define normal behavior. They can be split into 3 categories (Figure 3.1):

- Statistics-based outlier detection
- Distance-based outlier detection
- Model based outlier detection

Statistics-based outlier detection

Statistics-based outlier detection considers a stochastic model and assumes that normal data points occur in the high probability regions, while outliers would appear in the low probability regions of the stochastic model. This detection technique can be further divided into two categories, *hypothesis testing* and *fitting distribution*. Hypothesis testing methods first calculate a test statistic based on the observed data points. This test statistic is then used to determine if the null hypothesis (there are no outliers) should be rejected. The Grubbs Test and the Tietjen-Moore test are examples of the hypothesis testing method.

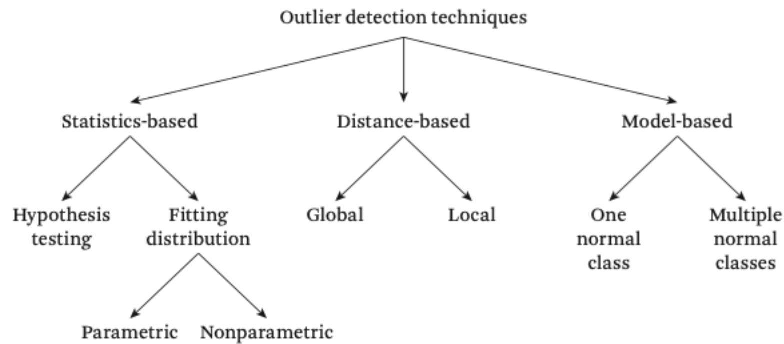


Figure 3.1: A taxonomy of outlier detection methods

The other category tries to derive a probability density function (pdf) based on the observed data. Data points that have a low probability according to the pdf are considered outliers. This *fitting distribution* technique can be further divided into a parametric and non-parametric approach. The parametric approach assumes that the observed data is generated according to a certain underlying distribution and attempt to retrieve the parameters of this distribution. In the non-parametric case, attempts are made to find out the distribution of the data without making any assumptions in advance.

Pros:

- Statistical techniques usually provide a confidence score, the score can be used while making a decision for a data point as additional information.
- Unsupervised: without any need for labeling data etc..

Cons:

- Relies on the assumption that the data is generated from an underlying distribution. This assumption does not always hold (high-dimensional real datasets).
- Choosing the best test statistic is often not a straightforward task.

Distance-based outlier detection

In distance-based outlier detection, a distance measure between data points is defined, which is used to describe normal behavior. A data point that behaves normally is close to many other points. Data points are considered abnormal, and therefore as outliers, if they do not exhibit this behavior. Depending on which population is used to determine whether a point is an outlier, the method is classified as local or global distance-based outlier detection. The global method uses the entire data set, while the local method uses only nearby points (neighbors) to determine if a point is an outlier.

Pros:

- Data driven (no assumptions).
- Easy to adapt to different data types, need only to define an appropriate distance measure.

Cons:

- Will fail when normal data points have not enough close data points or when abnormal data points have many close data points.

- High computational complexity, distance between every pair must be computed in the testing phase.

Model-based outlier detection

Model-based outlier detection technique uses a classification model. This model is trained with a set of labeled points and then applied to check whether a data point is an outlier. The technique is classified as one-class or multi-class model based, depending on the labels used to train the classification model. One-class model-based techniques assume that the labeled points from the training data set belong to one normal class, while multi-class techniques assume multiple normal classes.

Pros:

- Uses powerful algorithms to distinguish instances belonging to different classes.
- Fast testing phase.

Cons:

- Relies on the accuracy of the labels for the classes.

3.1.2 Statistics-based outlier detection

Hypothesis testing for outlier detection

Grubbs test Grubbs test is an example of hypothesis testing and is used to detect a single outlier in a univariate dataset with an approximately normal distribution.

Algorithm:

1. Define a null hypothesis and an alternative hypothesis
2. Calculate the mean and the standard deviation of the dataset
3. Calculate the Grubbs test statistic G

   The Grubbs test statistic is the largest absolute deviation from the sample mean $\overline{Y}$ in units of the sample standard deviation s.

$$\frac{max_{i=1,..,N}|Y_i - \overline{Y}|}{s}$$

4. Calculate G_{critical} (with the significance level)

   If $G > G_{\text{critical}}$: reject null hypothesis; else: accept

Tietjen-Moore Test is a generalization of the Grubbs test for more than one outlier (number of outliers must be specified).

Fitting distribution: parametric approach

Univariate Univariate outlier detection calculates the z-score for every element in the data. The z-score is described as the number of standard deviations away an element is from the mean. So, to calculate the z-score we first need to calculate the mean and the standard deviation, assuming the data follows a normal distribution (= fitting to a normal distribution). Data elements that have a z-score greater than a predefined threshold are declared to be outliers.

$$z - score(x_i) = \frac{|x_i - \mu|}{\sigma}$$

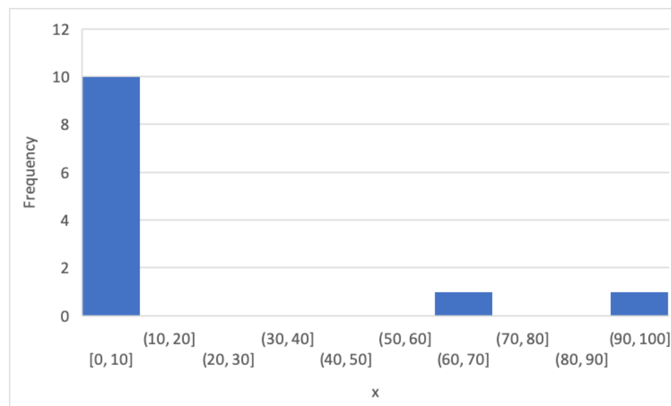


Figure 3.2: Equi-width histogram for outlier detection

Robust Univariate Statistics Outliers sometimes have a major impact on the mean. As a result, certain outliers are missed when the mean is used to detect the outliers. A more *robust* approach is the use of the median absolute deviation (MAD). The MAD is defined as the median or the absolute deviations from the data's median. In contrast to the standard deviation, which is influenced by large deviations (caused by outliers), the deviations of some outliers are of no great importance because the MAD uses the median of the absolute deviations.

$$\text{MAD} = median_i(|x_i - median_j(x_j)|)$$

Multivariate Some outliers are not detectable if we only look at one dimension. Outliers that are only visible if we consider multiple dimensions are called multivariate outliers. The mahalanobis distance is the multidimensional generalization of measuring how many standard deviations away a point is from the mean of the population. The mahalanobis distance of a point x to the mean vector μ is defined as:

$$Mahalanobis(x) = \sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

Robust Multivariate Statistics As with Univariate, outliers can strongly influence the multivariate mean and covariance matrix since they are not *robust*. The Minimum Covariance Determintant (MCD) is the most popular method for the robustification of multivariate mean and covariance matrix.

Fitting distribution: non-parametric approach

Histograms A histograms (Figure 3.2) represent the graphical distribution of the numerical data values. To create a histogram, the range of values must first be split into a series of intervals (consecutive and non-overlapping). The number of values per interval is then counted. If all intervals are of equal size, a bin is created with a height corresponding to the frequency (the number of values in the interval). These types of histograms are called equi-width histograms. If the intervals are chosen so that all intervals have the same frequency, the histogram is called an equi-depth histogram.
Equi-width histograms can be used to detect outliers. Bins that have a very low frequency contain data points that are propably outliers. The main challenge with this detection method is to choose the right size of intervals. If the intervals are too narrow, most bins will have a low frequency and normal points will possibly be marked as outliers. Otherwise if the intervals are too wide, most bins will have a high frequency and outliers will be absorbed in bins with normal points.

3.2 Clustering

The purpose of applying clustering to a collection of data points is to group these data points so that similar points are in the same group. The equivalence between the different data points is determined by a well-chosen distance measure. For example, if we want to group coordinates, we can literally calculate the distance between the different points and thus group the different points. Similar points, i.e. coordinates with a small distance between each other, then end up in the same group, also known as cluster. In this example it is not difficult to define a good distance measure for the given data points, unfortunately this is not always the case. The following section, based on [16], will first give an overview of the different clustering techniques. Then a point assignment algorithm, named K-means, will be discussed in more detail.

3.2.1 Clustering techniques

Clustering algorithms can be divided into two groups that have fundamentally different strategies: *hierarchical* and *point assignment* clustering. But they can also be divided based on what space they assume: *Euclidean* or *non-Euclidean* space.

Hierarchical algorithms

In hierarchical algorithms, also called agglomerative algorithms, each point is first placed in a cluster separately. The clusters closest to each other are then combined based on the selected distance measure. This joining process is repeated for a predefined reason. Examples of such reasons: a predetermined number of clusters is reached or a predetermined form of compactness is exceeded when clusters are joined.

```
while it is not time to stop do
    pick the best two clusters to merge;
    combine those two clusters into one clusters;
end
```

Algorithm 1: Pseudo code of a hierarchical algorithm

Point Assignment

The other class of algorithm use point assignment. After the initial clusters have been determined, all points are assigned to the nearest cluster based on the chosen distance measure. This process can vary, sometimes clusters may still be split or some points may not be assigned because they are outliers.

(Non-)Euclidean space

In the Euclidean case, it is possible to summarize the set of points within a cluster as the average of the points, called the centroid. In non-Euclidean space it is not possible to summarize the points into a centroid. So, the cluster must be summarized in another way. This can be done by calculating a clustroid, the point of the cluster closest to the rest of the points in the cluster, which then can be used to represent the cluster.

3.2.2 K-means

K-means is one of the most famous clustering algorithms in the class of the point-assignment algorithms. In order to be able to apply K-means, it is assumed that we work in an Euclidean space and that the number of clusters is known in advance. There are some techniques to derive

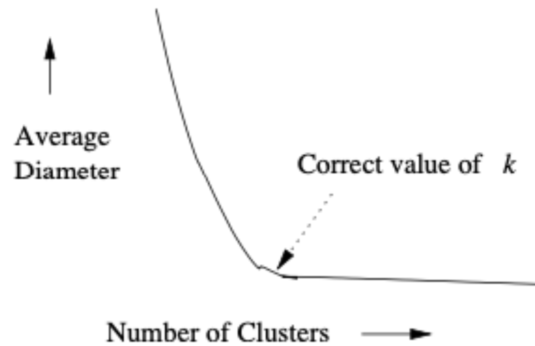


Figure 3.3: If there are to few clusters the average diameter or another measure of quality will rises quickly

the number of clusters for a given data set. One of these techniques, called the elbow-technique, will be discussed later in this section.

The algorithm

In the first step of the algorithm, k points are chosen to represent the clusters. These points are best chosen in such a way that there is a good chance that they will eventually end up in different clusters. One way to find these points is to run the clustering algorithm on a small subset of the data and select one point from each found cluster. After we have chosen k points that represent the centroids of the clusters, we arrive at the most important part of the algorithm. In this part, each point is considered and assigned to the nearest cluster, that is, the cluster whose centroid is closest to the given point. An optional next step is to recalculate the centroids for each cluster and reconsider all points and assign them to the nearest cluster. Often most points end up in the same cluster again. The latter strongly depends on how well those initial k points are chosen.

Initially choose k points that are likely to be in different clusters;
Make these points the centroids of the clusters;
for *each point p* **do**
    Find the centroid to which p is the closest;
    Add p to the cluster of the nearest centroid;
end

Algorithm 2: Pseudo code of the basic k-means algorithm

Determining the right value of K

It is not always obvious to know in advance how many clusters there will be in a given data set. But by running the K-means algorithm several times, for a different number of clusters, we can more or less determine how many clusters are present in the given data set. It is possible to calculate the quality of the clusters using a certain parameter. Examples of such parameters are the average radius or diameter of the clusters, this value usually grow steadily as long as we have more clusters than the actual number. But once there are too few clusters, this value will increase exponentially. When these results are plotted in graph, a clear *kink* is visible (Figure 3.3), the so-called elbow. No further explanation is needed as to why this is called the elbow technique.

3.3 Data profiling

3.3.1 Single-Column analysis

Single column analysis is one of the most simple profiling tasks in which individual columns are analyzed independently. In this part, which is based on [2, 3], the most common single-column profiling tasks will be discussed briefly.

Cardinalities

Cardinalities refer to the numbers that summarize simple metadata.

- *Number of rows*: gives important information about the number of entities which are present in the table, and is collected by most database management systems (DBMS) to estimate the query cost or to assign storage space.
- *Number of null values (within a column)*: gives more information about the data completeness and therefore data quality
- *Number of distinct values (or entities)*: together with the total number of rows, the column uniqueness can be calculated, which indicates if the column is a candidate key.

Almost all the above statistics can be computed in a single pass, besides the counting of distinct values, which can be done using sorting or hashing.

Value distribution

Profiling tasks that summarize the distribution of values within a column.

- *Histogram*: A common way to represent the distribution of values within a column is by using histograms. As already seen in Section 3.1.2, they are also used to detect outliers. More, histograms are very useful to users as they summarize the distribution's mass and shape which makes them easy to interpret. DBMS use histograms for query optimization where they provide a more accurate estimate for queries or operators than other naive methods (e.g. assuming a uniform distribution).
- *Extremes*: In a numeric column the extremes are described by its maximum and minimum values. These values support the identification of outliers.
- *Constancy*: The frequency of the most frequent value divided by the total number of values is the constancy of a column. A high constancy may indicate that the most frequent value is a (pre-defined) default value.
- *Entropy*: The entropy of a column indicates the average *level* of information. In other words, the entropy is the amount of storage space (e.g. the number of bits) it takes to store all values. This can be intuitively seen as the amount of information in that column. There are several formulas to calculate entropy (each applicable for a particular domain). A frequently used formula in this domain is Shannon's formula, with P_i the probability that value at position i in the list of size M occurs.

  $$H = -\sum_{i=1}^{M} P_i \, log_2 \, P_i$$

  In Figure 3.4, the entropy of a coin toss is visualized. In this case there are always 2 possibilities: heads or tails. The entropy (y-axis) depends on the chance (x-axis) that the coin will land on one of the two sides.

By use of the correct methods (sorting or hashing) each of these statistics can be computed in a single pass, with the exception of equi-depth histograms.

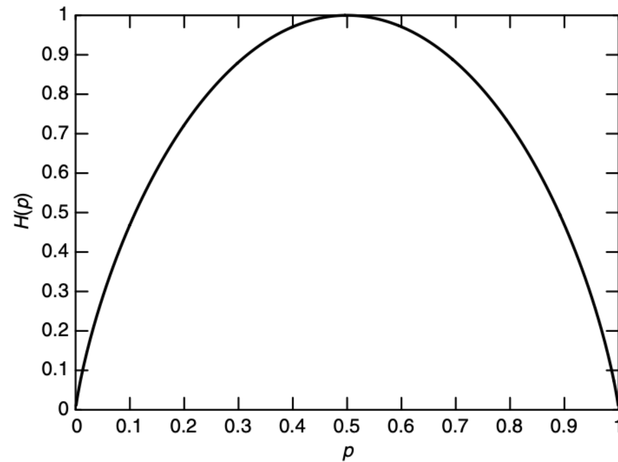


Figure 3.4: Visualization of the entropy of a coin toss (2 possible values). If the coin were to land always on the same side, this would result in an entropy of 0.

Data types, patterns and domains

Profiling tasks that focus on data types, patterns and domains.

- *Basic data type*: The basic data types: numeric, alphabetic and alphanumeric, can be verified by the presence or absence of numeric or non-numeric characters. The other two types, date and time, can be detected by checking for numbers within certain ranges and for groups of numbers separated by special characters, for example a time can be presented as: 2 digits - a colon - 2 digits (multiple possible formats).

- *Data management system data type*: These systems allow more specific data types, which can be checked in a similar way as the basic data types. Here it is important that the most specific data type is selected, e.g. a Boolean column can be evaluated as an integer, but the Boolean type is preferred. To distinguish data types as decimal, float and double, one must first determine maximum size and the number of decimals. This is also the case for alphanumeric column, but then with various lengths.

- *Patterns*: If the data type of a column is know, one can look for frequently occurring patterns of values that can indicate that a column e.g. consists of telephone numbers (pattern: +(dd) ddd dd dd dd). It is very likely that the data set contains some errors and outliers, thus it is enough that most values within a column satisfy the frequent pattern.

- *Domains*: Identifying the semantic domain involves figuring out the meaning or interpretation of the data rather than its semantic properties and is therefore the most difficult task. The presence of certain regular expressions may suggest a semantic domain telephone number, but it may also correspond to fax numbers. This task is not, and usually can not be fully, automated.

3.3.2 Dependency Discovery

Traditionally, constraints such as keys, foreign keys and functional dependencies, have been considered predetermined constraints of a schema, acknowledged when designing the data schema. However, today there are many data sets for which these constraints are unknown. Consequently there is a need for algorithms that can derive these constraints from data sets. Especially, because these constraints are also very useful for detecting logical errors in data sets. Errors that

cannot be traced by other data cleaning tasks, for example impossible ZIP code and place name combinations in an address. This section, which is based on [8, 9], gives an overview of the different kind of dependencies.

Functional dependencies

A functional dependency (FD) is defined as A → B, with A and B attributes from a relational schema R. The FD is valid in R if for any two tuples X and Y: if X[A] = Y[A], then X[B] = Y[B]. In order words, A → B asserts that all tuples with the same value in attribute A must also have same values in attribute B. Thus, values in B functionally depend on the values of A.

Violations against the FD may indicate that there are some errors in the data set. For example, given the FD: *ZIP Code → Place name.* If for a certain tuple the zip code *3600* translates to *GENK* and some other tuple this same zip code translates to *HASSELT*. Then, one of these tuples must be incorrect and is cleaning needed.

There are two types of approaches for detecting FDs: schema-driven and instance-driven. The schema-driven approach usually has a schematic way of listing all candidate FDs and an efficient way of checking whether an FD is valid or not based on the enumeration. TANE is an example of a schema-driven algorithm, it uses a level-wise strategy for candidate generation and pruning, and checks the validity of the FDs using a linear algorithm. The data-driven approach starts with creating a summary of the data sets and then searches for valid FDs based on that summary. An example of this approach is FASTFD, which first calculates the difference sets of the data and then applies a heuristic driven depth-first search algorithm to find all covers of difference sets. Both algorithms have their advantages and disadvantages. For example, TANE is sensitive to the size of the schema, while FASTFD is sensitive to the size of the instance.

Conditional Functional dependencies

Sometimes it is not necessary that a FD holds for all the data, but just from some part(s). This can not be expressed with a FD, but can with a *conditional* functional dependency which is an extension of FDs. A conditional functional dependency (CFD) can be defined as (A → B, T_p), with A → B a FD from a relational schema R and T_p a pattern tableau.

A pattern tableau can be seen as some kind of table, where for every attribute of the FD is specified for which values the FD is valid. These values can be constants (a value in the domain of the given attribute) or wild cards (-).

For example, suppose we have a data set of sales, where each record consists of the attributes: name, type, country, price and tax. The following FD was drawn up for this data set: $name, type, country \rightarrow price, tax$. The FD does not apply to the entire data set, but it does in these 3 cases:

- The type is *clothing*
- The country is *France* and the type *book*
- The country is *United Kingdom*

This can then be expressed by the CDF (name, type, country → price, tax, T_p), with T_p, as shown in table 3.1.

Similar techniques are used to detect CDF as for FD discovery. Examples of such algorithms are CDFMiner, CTANE and FASTCDF, each with their advantages and disadvantages. Two major challenges emerge during the detection. The first, which also applies to FD discovery, is that the number of possible FDs is exponential with the number of attributes present in the

Name	Type	Country	Price	Tax
-	clothing	-	-	-
-	book	France	-	0
-	-	UK	-	-

Table 3.1: Pattern tableau T_p of the CDF: $name, type, country \rightarrow price, tax$

schema. The second challenge is that the number of possibilities (of constants) in the pattern tableau are enormous.

If an FD is given, the CDFs problem equals generating the best possible pattern tableau. The quality of the pattern tableau then depends on the support and confidence. The support of a pattern tableau is determined by the fraction of tuples in the data set corresponding to the LHS of the pattern tuples in the tableau. The confidence of a pattern tableau is defined as the maximum fraction of tuples in the data set corresponding to the tableau.

Denial constraints

With FDs and CDFs many rules can be expressed, but they are still not capable of expressing many real-life data quality rules. For example, *If 2 people are affiliated with the same telephone company, the person with fewer calling minutes has to pay less* or *The net salary is less than the gross salary.* Denial constraints (DC), a universally quantified first order logic formalism, are able to express this kind of rules. It is even possible to write FDs and CFDs as DCs.

For example, the rule that states that two persons with the same zip code live in the same town can be expressed as:

$$c : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$$

This rule says that there should not be 2 tuples (t_α, t_β) in the data set (R) where the postcodes are the same and the towns are different.

In the case of DC discovery, the schema-driven approach, as seen with FD discovery, is less useful. This is mainly due to the fact that DCs are more complex: FDs only contain *equality*, while DCs can also contain predicates such as *greater than* and *less than.* Consequently, it is difficult to systematically check the validity of the space of all DCs. The currently most used algorithms for DCs discovery are: FASTDC (instance-driven, similar to FASTFD for FD discovery) and Hydra (Hybrid algorithm, similar to HYFD for FD discovery).

Others

There are of course many other types of constraints, each designed for a different purpose:

- *Inclusion dependencies*: used for detecting inconsistencies or information incompleteness and for schema matching.
- *Matching dependencies*: generalize the equality condition used in FDs by use of similarity measures, to support record linkage across two tables.
- *Metric functional dependencies*: capture small data variations (specialization of matching dependencies).
- *Numeric functional dependencies*: capture interesting constraints involving numeric attributes.

- *Editing rules*: detect and correct errors, the latter by use of a master table.
- *Fixing rules*: detect and correct errors, the latter if there is enough evidence present indicating how to correct the error.
- *Sherlock rules*: mark the correct and erroneous attributes, and precisely tell how to fix the errors by referencing master tables.

3.3.3 Metanome

The metanome project [7, 13] is a collaboration between the Hasso-Plattner-Institut (HPI) and the Qatar Computing Research Institute (QCRI). It is a framework that offers a range of efficient algorithms related to data profiling on one common platform. It is possible as a user to expand the system with self-developed data profiling algorithms. The algorithms can be applied to both existing (internal) data sets and external data sets indicated by the user. All this makes the tool ideal for both algorithm engineers and data scientists.

Input sources

- Databases: DB2, MySQL, Oracle,...
- Files: txt, tsv, csv, xml,...

Profiling algorithms

- Unique column combinations: HyUCC, DUCC
- Inclusion depencency: BINDER, SPIDER, MANY, FAIDA
- Functional dependency: HyFD, DFD, Tane, Fun, DepMinder,...
- Matching dependencies: HYMD
- Denial constraints: Hydra, DCFinder
- Basic statistics: SCDP

In practice

Metanome has often been used for this work, among other things to calculate statistics of the columns and dependencies of the various data sets. A very handy tool, just a pity that it also has some flaws. For example, Metanome uses outdated software (Java JDK 1.8, Maven 3.1.0) and does not work on newer versions of this software. In itself this is not really an insurmountable problem, but it does become very difficult because it is also not easy to get this outdated software (Maven 3.1.0). In addition, the internal algorithms are all very memory intensive, which means that the program quickly gets into trouble with the larger data sets and it takes either a very long time to generate the result or the execution stops / crashes completely. So, if you want to quickly find out more information about a relatively small data set and you can install Metanoma, it is an excellent tool.

3.4 Machine learning approach

3.4.1 Introduction

Due to the increase in popularity and availability of resources to build large-scale machine learning (ML) solutions, the use of ML techniques for data cleaning has increased. In a ML

environment, data cleaning is viewed as a probabilistic database problem, where static and probabilistic interpretation of data errors can lead to more general solutions for detection and repair. This approach has been used for years, but only for numerical outlier detection techniques. The biggest advantage of applying a probabilistic view is that it is possible to handle different types of data errors in one platform. This is ideal, because in reality data errors in databases are not limited to one specific type. Studies have also shown that the application of multiple tools, each for one type of error, on a *dirty* database has a bad influence on the results.

Many ML concepts and techniques are used today to build data cleaning solutions. For example, *factor graphs* are used to record the correlation between the different features and signals, and to predict the most likely value for a faulty cell in the database. Active learning is used to involve experts in labeling data used to train ML models.

This section reviews briefly the findings made in [9], where they focused on the cleaning task *data deduplication*.

3.4.2 Machine learning for data deduplication

ML solutions can be used to detect duplicate records in the data, with similarities scores as features, which is by definition a binary classification problem. When there is a shortage of sufficiently labeled data with examples of duplicate and non-duplicate records, a good classifier cannot be taught. In this case, active learning can be a fully-fledged alternative. Also deep learning (DL) can be useful in some data deduplication scenarios.

Active learning in data deduplication

The general idea is that user feedback is requested for unlabeled record pairs. When labeled by the user, they are most useful in the training process. A well-known solution to this is ALIAS [17]. The main problem is that the accuracy of the classification model for data duplication is highly dependent on the quality of the training data. After all, the training data should have a balanced number of both cases (duplicates and non-duplicates) and ideally also examples of cases that are likely to be misinterpreted.

Deep learning in data deduplication

In a recent study [11], several DL solutions were compared for matching records. This study showed that the DL solutions are certainly not much better than the traditional ML solutions for structured EM (*Entity Matching*) tasks, where the entities have the same scheme and the attribute values are clean. However, DL solutions significantly outperform traditional ML solutions on textual EM tasks (entities only contain text attributes) and dirty EM tasks (attribute value of entities contain errors).

But here too is the lack of high-quality training data a problem. Most models require large amounts of labeled data to learn a particular classification task. The labeling of data can be done by a domain expert, but their time is far too precious to be concerned with this. As a result, models that require less training data are preferred over others. In addition, there is a lack of explanation for the decisions made for the obtained output, this explanation is necessary in some cases.

3.4.3 Conclusion

There are still some stumbling blocks that must be overcome before ML can be offered as the number one solution for data cleaning. Especially the shortage of sufficient qualitative training data is a big problem. But the use of ML solutions for developing better data cleaning solutions is certainly promising.

3.5 Limitations of data error detection

As indicated earlier, data today is used to support decisions in companies and is necessary in ML environments. It is clear that data has become a very important asset, more specifically clean data. The demand for clean data has led to numerous tools. In [1], a study was conducted to answer the following two questions: (1) Are the current tools robust enough to record most errors in real data sets? (2) What is the best strategy to holistically implement multiple tools to optimize the detection effort? The main findings are summarized in this section. First we look at the current state of data cleaning tools and what challenges there are in this research. Then is described which types of errors and which data sets were considered, together with the different tools that were compared. Finally, an overview is given of the most important conclusions from this paper.

3.5.1 Current state

Tools

The available data cleaning tools can be divided into 4 categories:

- Rule-based detection algorithms
  - Rules can range from very simple (e.g. *not null*) to more advanced constraints (e.g. *functional dependencies*) or user-defined functions.
  - Example(s): DC-Clean
- Pattern enforcement and transformation tools
  - Tools that look for certain patterns in the data, these patterns can be both syntactic and semantic in nature. By use of these patterns errors can be detected, in other words cells that do not follow the found patterns.
  - Example(s): Trifacta, OpenRefine, Katara, etc.
- Quantitative error detection algorithms
  - Algorithm that detects outliers in the data
  - Example(s): dBoost
- Record linkage and deduplication algorithms
  - Algorithm that detect duplicates in the data
  - Example(s): Tamr

Challenges

When researching these tools, a number of challenges emerge:

- *Synthetic data and errors*: Data (synthetic or real) with synthetic added errors are often used to evaluate the tools. This gives a good picture of the correct functioning of the algorithm, but not of the effectiveness of the tool on real errors in data. Which makes it difficult to assess the existing tools.
- *Combination of type of errors and tools*: Real data usually contains errors of different types. In addition, the same error can be traced by multiple tools. So when only one type of algorithm is considered, the opportunity to gather evidence from different types of tools is missing.

- *Human involvement*: In almost all tools there is a need for people who draw up rules, verify the detected errors or give feedback to the ML model. It is desirable to keep human involvement as minimal as possible.

3.5.2 Setup

Types of data errors

A value is referred to as an error if it has a deviation from the basic truth value. The following types of errors are considered in this study:

- *Outliers*: are the values that deviate from the distribution of values in a column of a given table.
- *Duplicates*: are multiple different records that refer to the same entity. If not all values of the different attributes match, this could indicate a possible error.
- *Rule violations*: are the values that does not meet a predefined integrity constraint.
- *Pattern violations*: are the values that do not meet predefined syntactic or semantic constraints.

Certain errors may belong to more than one type.

Data sets

For the study, a number of data sets were used, which were obtained from a number of organizations active in different sectors, consisting of *real* data. Of which one dataset consists of artificially manufactured errors.

Data cleaning tools

The data cleaning tools were chosen so that all types of errors discussed earlier could be detected by at least one tool. So for some types of errors, multiple tools were chosen, usually each focusing on a different subcategory (e.g. semantic vs. syntactic). Furthermore, the selection of tools consisted of both commercial and publicly available tools. To ensure that every tool was used as optimally as possible, each tool was first fine-tuned. Below an overview of the tools that were used per type.

- Outlier Detection: dBoost
- Rule-based Error Detection: DC-Clean
- Pattern-based Detection: OpenRefine, Trifacta, Katara, Pentaho and Knime
- Duplicate Detection: Tamr

Multiple tools

In addition to evaluating the effectiveness of each tool on the data sets, the effect of combining several different tools was also investigated. Two simple strategies were used to combine multiple tools: *Union all* and *Min-k*. The first strategy, *Union all*, simply takes the union of all errors detected by all tools. The second strategy, *Min-k*, considers all errors detected at least by k-tools. Therefore, the errors that are not detected by k-tools are disregarded. Furthermore, a third more sophisticated *ordering based* strategy was also tested, in which the users evaluate a sample of detected errors. On the basis of this evaluation, a sequence of tools can then be determined in which they must be applied. More details about this last strategy can be found in the paper.

3.5.3 Evaluation

To determine the effectiveness of each tool, the accuracy of finding potential errors was measured based on precision and recall. Precision is defined as the fraction of cells correctly identified as error and recall as the fraction of true errors. The final *score* of a tool is the *harmonic* mean of the precision and the recall.

Individual effectiveness

It was remarkable that all tools performed very well on the data to which errors had been added in a synthetic way. So in later work, to determine the effectiveness of tools, one should certainly not use a data set to which errors have been added in a synthetic way. It was also noticeable that no tool performed well on all data sets, so this is very dependent on the type of errors that occur. For example, *Tamr* (duplicate detection) was very effective on data that contained many duplicates and not on any other data sets.

Combination effectiveness

When using the *union strategy*, the recall naturally increases and is better than for any of the tools separately, which makes sense since more types of errors can be detected. Unfortunately, this also results in more false positives and therefore a decrease in precision.

The loss in precision can be counteracted by using the *k-min strategy* which increases precision, but decreases recall. Furthermore, it could be determined that there is no k, which is most effective for all data sets. So the effectiveness of the k-min strategy depends on the chosen k, the data set and the tools used.

When using the *ordering based* strategy, the user has to evaluate a significantly lower number of errors detected, with the result that only a few real errors are lost compared to the union strategy. So a fraction of recall is sacrificed for less human involvement.

3.5.4 Conclusion

The main points to be taken from this paper are:

- There is no single dominant tool for all data sets and all different types of errors.
- The correct order of applying tools can improve precision and reduce the necessary human input.
- Domain specific tools can achieve better precision and recall compared to general tools.
- Data enrichment can positively impact rule-based systems and duplicate detection.

Chapter 4

Data cleaning in practice

In this chapter a better insight is gained in the world of data cleaning and profiling tools (as shown in Figure 4.1) and what they can and can not do. Because there are too many, only the most popular/interesting tools will be get a closer look.

4.1 Trifacta

Trifacta [22] is an online platform that provides software to explore, transform and join diverse data for analysis more efficient. This in the first place for organisations, but also for individuals. Data sets can be imported in all kind of different formats (CSV, plain text, parquet, JSON, XML, etc.) and also from several external sources (database, file system, app or cloud storage) in case you own the PRO version. After the data is imported in, flows can be created by merging data and adding *recipes*. Recipes in Trifacta are a list of operations that are executed on the data sets. Examples of such operations are filtering, sorting, renaming, restructuring and many more. Afterwards, if a flow is completed, they can be scheduled and shared. The output is in CSV or a JSON format.

Trifacta has its own data wrangling process that consists 6 steps, each step is characterized by the tasks that are possible.

1. Discovering
    - Suggestions: transformations, counts,...
    - Filtering: hide/remove columns
    - Locating outliers (single-column)
    - Calculate metrics across columns
    - ...

Figure 4.1: An overview of a small part of the different data cleaning (and profiling) tools

2. Validation
    - Find bad data
    - Find missing data
    - Manage null values
3. Structuring
    - Split columns
    - Create aggregations (on groups)
    - Pivot data
4. Cleansing
    - Remove data
    - Deduplicate data
    - Standardize using patterns
    - Applying conditional transformations
    - ...
5. Enrichment
    - Two columns: add one into another, add selective values from one into another, add two columns into a new column,...
    - Adding lookup data
    - Joining and appending data
    - Inserting metadata
6. Publishing

4.2 Tableau

Together with Trifacta, Tableau [19] is one of the more popular tools for data visualisation and cleaning among data scientists. Tableau is mainly know as the data visualisation tool. But in fact, Tableau consists of several products. *Tableau Desktop* is the one that everyone knows as *Tableau*, the visualisation tool. An other useful product is *Tableau Prep* where data can be combined, shaped and cleaned in an visual and direct way. Tableau prep is actually a combination of 2 sub-products: *Tableau Prep Builder* and *Tableau Prep Conductor*.

- Tableau Prep Builder
    - Row level data, profiles of each column and the preparation process are presented in 3 coordinated views.
    - Works on millions of rows of data, each action will be instantly visible on the data.
    - Repetitive tasks, like grouping by pronunciation are turned into one click operations by employing *fuzzy* clustering.
- Tableau Prep Conductor
    - Flows can be run and published in your server environment.
    - The execution of flows can be scheduled when you need them - day or night.

  - Data sources can be shared securely.
  - Data prep processes can be automated, so there is always fresh data that is prepped and ready to be analyzed.

Tableau also has other interesting products to support you with your data management, but these will not discussed here.

4.3 OpenRefine

OpenRefine [21], previously known as *Google Refine*, is an open-source tool that comes in handy when dealing with messy data and cleaning, transforming or extending needs to be done. OpenRefine consist of 4 steps:

1. Importing, format will be guessed based on the file extension
2. Faceting, explore data by applying filters
    - Default text facets: e.g. counts of each element in a column
    - Multiple facets: combine multiple facets (intersection)
    - Custom facets: program a facet
3. Editing
    - Cell editing: by use of text facets, transformations, splitting, clustering or search & replace
    - Column editing: move, add, delete, merge or split by use of string, length or regular expression
    - Row editing: remove
4. Exporting, to TSV, CSV, Excel or HTML table

4.4 Tamr

By use of limitless power of probabilistic human-guided machine learning, Tamr [20] makes it fast and easy to replace labor and time-intensive, rules-based data cleaning and preparation. In Tamr three kind of projects are possible:

- Mastering
  - Solves the tasks of finding records that refer to the same entity within and across input data sets.
  - Other names for this type of task are data mastering, entity resolution or record linkage.
- Golden records
  - Solves the task of creating golden record from a cluster of records that refer to the same entity.
  - Golden records can be created on clusters obtained from a mastering project or can be created for any data set containing a grouping key.
  - This task is also known as entity consolidation.
- Schema matching

  - Allows to map attributes from many input data sets into a set of attributes known as a single unified schema.
  - A unified schema can be a list of attributes or fields associated with an entity, such as a customer, or an organization across multiple datasets.

4.5 Deequ

Deequ [18, 10] is a library that can be used to detect errors early in large datasets. It can work on all kinds of tabular data, for example CSV files, database tables, logs, flattened JSON files, basically anything that can fit into a Spark dataframe. The user can create its own assumptions in the form of a *unit-test* for data. If errors are found in the dataset, these will be quarantined and the user can fix them before the the data is used in an other application. Deequ's functionalities:

- Storing computed metrics
  - Calculate and store metrics.
  - Query stored metrics by use of a tag or timestamp.
- Single column profiling
  - Understanding and cleaning of a raw dataset.
  - Profiling of a column:
    - Completeness
    - Number of distinct values
    - Inferred datatype
    - Descriptive statistics (e.g. minimum, maximum, mean, etc.) for numeric columns
    - Value distribution (only for columns with a low number of distinct values)
  - Scales to large datasets with billions of rows.
- Anomaly detection (Deequ contains a number of built-in strategies)
  - Batch Normal
    - Detects anomalies based on the mean and standard deviation of all available values.
    - Assumes that the data is normally distributed.
  - Online Normal
    - Detects anomalies based on the running mean and standard deviation.
    - Assumes that the data is normally distributed.
  - Rate of change
    - Detects anomalies based on the rate of change of values.
  - Simple threshold
    - Checks if values are in a specified range.
- Automatic suggestion of constraints
  - Profiles first the data and applies then a set of heuristic rules to suggest constraints.
  - The constraints that can be suggested for a column are: datatype, completeness and the number of distinct values.

4.6 Great Expectations

Great expectations [5] is an open-source tool that executes automated testing on data sets. The user formulates its own expectations to create a testing pipeline. This pipeline shall then monitor data and guard against upstream data changes. The key features are:

- *Expectations*: provide a flexible, declarative language for describing expected behavior
- *Data profiling (automated)*: explore data faster, and capture knowledge for future documentation and testing
- *DataContexts and DataSources*: provide convenience libraries to introspect most common data stores
- *Tooling for validation*: store validation results to a shared bucket, summarize results, post notifications to slack, handle differences between warnings and errors, etc.

4.7 Conclusion

As mentioned earlier, at the beginning of this chapter, there are many other tools (e.g.: Tibco clarity, Data ladder, DataCleaner, WinPure, Drake, Cloudingo, Reifier) for data cleaning and profiling that are not discussed here. These tools contain either similar functionality or focus on a specific aspect of data cleaning. The aim of this search was mainly to get a better picture of what functionalities these tools have to offer today. It is quite clear that there are plenty of tools and that they offer a wide range of functionalities that can help users clean their data.

However, based on what we have seen in this chapter and Chapter 3, we can conclude that all of these tools lack the ability to write/discover rules based on aggregations. On the basis of aggregations you can write more substantive rules, which must apply in the data set, such as:

- *Train A* stops every day at 10 stations.
- Every month maximum 100 euros is spent on *sports & culture* on Jef's bank account.
- An average of 10000 customers order an *action movie* every week.

We assume that these tools do not support the discovery of such rules because there are too many options and it would be too difficult to choose, if they were generated. In the following chapters, we will describe an approach to find such rules in event data and how they can be monitored.

Chapter 5

Anomaly detection in event data

This chapter discusses how we try to apply anomaly detection to event data. First, the concept of *event data* is defined and the data that was used in the first phase of the development process is described. Then we explain how we try to detect anomalies in this data. Finally, we look at the challenges involved.

5.1 Data

5.1.1 Event data

In this work, the focus was on detecting anomalies in event data. The event data that was considered consists of events that are logged during a certain period of time, usually this is done per day. These events are repeated (several times) every day. A good example of this is the data of trains or buses. Every day (mostly) the same trains and buses run and stop at the same stations and stops. These events are all recorded and maintained today and form the event data.

5.1.2 Data structure: NMBS

In the first phase of the development process, *NMBS* train data was used as a representative example of event data. This data was used to test ideas and work out the final system. At a later stage, other data was also used to fine-tune the system for general use. The *NMBS* event data is kept per day and has the following structure:

- The *NMBS* event data (of 1 day) is kept in a CSV file.
- Each file usually consists of 70,000 rows on average, and thus 70,000 events.
- Each event represents a passage of a train at a stop.
- 18 attributes are maintained for each event (= 18 columns).

The attributes of each train event are:

- Datum van vertrek (*DATE*): The date of departure
- Treinnummer (*SMALLINT*): The number of the train (id of the train)
- Relatie (*VARCHAR[16]*): The abbreviated value describing the route the train is following
- Spoorwegoperatoren (*VARCHAR[16]*): The organizer/owner of the train
- Spoorlijn van vertrek (*VARCHAR[16]*): The railway on which the train departs

- Uur van reele aankomst (*TIME*): The hour the train arrived
- Uur van reele vertrek (*TIME*): The hour the train departed
- Uur van geplande aankomst (*TIME*): The hour the train should arrive
- Uur van geplande vertrek (*TIME*): The hour the train should depart
- Vertraging bij aankomst (*REAL*): The delay that the train has on arrival at the stop
- Vertraging bij vertrek (*REAL*): The delay that the train has on departure at the stop
- Richting van de relatie (*VARCHAR[48]*): The full value that describes the route the train is following
- Naam van de halte (*VARCHAR[32]*): The name of the stop
- Spoorlijn van aankomst (*VARCHAR[16]*): The railway on which the train arrives
- Datum van geplande aankomst (*DATE*): The date the train should arrive
- Datum van geplande vertrek (*DATE*): The date the train should depart
- Datum van reele aankomst (*DATE*): The date the train arrived
- Datum van reele vertrek (*DATE*): The date the train departed

The following functional dependencies can be derived:

- Treinnummer → Richting van de relatie
- Treinnummer → Relatie
- Treinnummer → Spoorwegoperatoren

5.2 Aggregation

For the rest of this work, we assume that events are tracked per day in a specific data structure to which aggregations can be applied.

As mentioned earlier in this work, the detection of anomalies (Section 3.1) is the detection of abnormal behavior. Examples of possible deviations in the train data we work with are:

- Normally 10 trains run from *Genk* to *Bruges* on Tuesday, today only 8 trains did.
- Train 25 normally rides 5 times on Monday. However, it only drove 4 times today.
- An average of 5,000 trains run on a weekday and 3,000 trains on a weekend. At the end of the day only 2,000 trains ran.

We can extract these examples and other results from the event data by using aggregation. An aggregation operation can be applied to one or more columns (e.g. the count operation, to calculate the number of occurrences of each entity in a specified column). Those aggregated values can then be used to subsequently check whether there are any anomalies. This is achieved by comparing the result of today's aggregation with the historical aggregated data. The historical aggregation consists out of the aggregation applied daily to a predetermined number of previous days. The comparison uses an appropriate outlier detection technique to determine whether or not the last (aggregated) value deviates from historical aggregated data.

To apply the aggregations to the data, SQL queries are used. For example, to check if there are some routes with a deviating value in the number of *distinct* trains that have run for a given day, we have to write a query (see below) that aggregates all routes summing the number of *distinct* trains, by use of the COUNT-operator.

```
SELECT relatie, COUNT( DISTINCT treinnummer)
FROM event_data_tuesday
GROUP BY relatie
```

5.3 Challenges

In order to be able to perform anomaly detection day by day, it is first necessary to determine which things must be monitored. Checking every possible aggregation every day is neither feasible (1) nor interesting (2). The first, because there are a huge number of possible aggregations: each aggregation consists of an operator that is applied to one or more columns. The number of possible aggregations therefore quickly increases as the number of columns in the data set increases. So, there is a need for a way to exclude uninteresting aggregations in advance without calculating results. The second, because it is possible that aggregations either never or always give outliers. So we have to look for a way to follow-up only the interesting aggregations. Since interest is something subjective, there is no general definition of an interesting aggregation. In this work, an interesting aggregation is defined as follows: *An aggregation is considered interesting if it almost always gives the same result, but sometimes not.* In other words, every aggregation must be checked if it is *consistent*. To do this, there is a need for historical data (which later also serves to detect any outliers) and a way to determine the consistency of an aggregation. Supposing we have a way of calculating the consistency of an aggregation, there is an additional problem, we cannot simply consider every day equally. If we look at the data of the trains, we see that not the same number of trains are scheduled every day. For example, fewer trains run on weekends than during the week. Therefore, if we were to regard all days as equal, we would possibly unfairly exclude potentially interesting aggregations. So there is a need for:

- Filtering of uninteresting aggregations
- Selecting interesting aggregations, i.e. calculating the consistency of the results of an aggregation
- Grouping of days based on event data

5.3.1 Queries

To somewhat reduce the complexity of this work, we focused on a certain type of aggregation (from now on called query). For this, we looked at what kind of information in the train event data is interesting from a data scientist's point of view and what the queries for this information look like. Things that may be interesting in the train event data are:

- How many trains run in a day? (and which)
- How many routes are served by trains? (and which)
- How many trains run a given route?
- How many times does a train stop?
- The average (or minimum, maximum) delay at departure/arrival per train

- The average (or minimum, maximum) delay on departure/arrival per route
- ...

This showed that a certain type of query is often needed:

```
SELECT columnA, COUNT(DISTINCT columnB)
FROM event_data
GROUP BY columnA
```

For the rest of this work, this type of query will be referred to as the *GBC-query* ,GroupBy-Count query. At a later stage, the types of queries were expanded to queries of a similar format:

```
SELECT columnA, AVG(columnB)
FROM event_data
GROUP BY columnA
```

```
SELECT columnA, COUNT(DISTINCT columnB)
FROM event_data
```

Other similar queries that can easily be added:

```
SELECT columnA, MAX(columnB)
FROM event_data
GROUP BY columnA
```

```
SELECT columnA, MIN(columnB)
FROM event_data
GROUP BY columnA
```

Chapter 6

A query ranking algorithm

This chapter covers the process of trying to find out the interesting queries. This process uses different techniques discussed in each of the following sections. Each section discusses first which approach (the idea behind the technique) is used, then how it is implemented and finally the results that were achieved with it.

6.1 Based on statistics

6.1.1 Approach

The first thing we want to do in the search for interesting queries is to pre-reduce the number of possible queries that need to be evaluated, because as mentioned earlier, there are quickly too many. An initial filtering, that is applied, is based on column statistics. It analyses each column separately and gives it a score for every possible position in the query. So for the *GBC-query*, it might not be interesting to perform a *GROUPBY* operation on a particular column, because the column only contains distinct values and therefore the operation has no effect. However this column may be interesting to apply the *COUNT* operation. As one can see, in this phase no exact science is applied and we will rather intuitively exclude queries. Whether or not a query is excluded is based on the following elements:

- **Data Type**: The data type plays an important role. Grouping or counting unique column cells with type *TEXT* or *TIMESTAMP* is clearly not an interesting case. Since these are column types that almost always contain only unique values and have little meaning after one of the operations has been applied to them. Other types such as *SMALINT* or *VARCHAR[16]* are more likely to be interesting for the *GROUPBY* or *COUNT* operation.
- **Entropy**: The entropy provides a good indication of the distribution of the data in a column. The smaller the entropy value the better. This is mainly the case for the *GROUPBY* operation, because you would rather have a limited amount of values after applying the *GROUPBY*.
- **Number of Distinct Values**: In addition to entropy, we also look at the exact number of different values. As indicated with entropy, you prefer to keep a limited number of values. But since entropy is expressed in percentages, it can sometimes give a wrong impression. Therefore it is useful to also look at the number of different values specifically.
- **Number of Null Values**: If a column has many empty cells or zero cells, this may already indicate that it is considered less important.
- **Functional Dependencies**: FDs tell more about the relationship between selected columns. This relationship can be used to identify and exclude less interesting queries. We demonstrate this with the following example. Take the following FD: *Treinnummer* $\rightarrow$ *Relatie*.

```sql
SELECT treinnummer, COUNT( DISTINCT relatie)
FROM "NMBS1"
GROUP BY treinnummer
LIMIT 5
```

	treinnummer	count
1	10	1
2	11	1
3	12	1
4	13	1
5	14	1

Figure 6.1: GBC-query that translates to "the number of routes a train runs" and its result, which is always 1 because of the FD.

```sql
SELECT relatie, COUNT( DISTINCT treinnummer)
FROM "NMBS1"
GROUP BY relatie
LIMIT 5
```

	relatie	count
1	EURST	21
2	EXTRA	38
3	IC 01	39
4	IC 02	35
5	IC 03	40

Figure 6.2: GBC-query which translates to "the number of trains serving a route" with the result of it. This could be a potentially interesting query.

It tells us that if you know the *Treinnummer*, you also know the *Relatie*, the route this train serves. If we fill in these two columns in the *GBC-query*, with *GROUPBY* on *Treinnummer* and *COUNT* on *Relatie*, we see that it results in a *COUNT* of 1 for each train number (Figure 6.1). This is a logical consequence of the FD. This query can not be consider interesting due the fact that the results never change as long as the FD is valid. The reverse query, with *COUNT* on *Treinnummer* and *GROUPBY* on *Relatie*, in contrast may be interesting (Figure 6.2). So in general, if $A \rightarrow B$ is a functional dependency, then a *GROUPBY* on A and a *COUNT* on B is not desired.

It was also noteworthy that columns that appear as a dependent in an FD are more often suitable for applying the GROUPBY, while columns that appear as a determinant in an FD are more suitable for applying the COUNT. It has not been investigated whether there is actually a direct connection (outside the scope of this work), therefore this element has only a small influence.

Those characteristics, together with the position of the column in the query, determine the score of the column. After each column has been scored, the score of the query can be calculated by multiplying the scores of the columns. Multiplication is applied so that a low score of a column has a large influence on the final score of the query. Based on a pre-defined threshold, we can filter the queries whose score falls below this threshold. In this phase, the main aim is to exclude queries that are surely not interesting, so we may still include a few queries that prove to be not interesting later on.

6.1.2 Implementation

In the current implementation, each column is given a score of 8 for each position. All the elements discussed therefore do not all weigh equally on the final score. The final score of the entire query is then recalculated to 100 (instead of 64).

A column for the *GROUPBY* position is scored as follows:

- Data type (3):
  - 3 points, in case of data types *SMALLINT*, *INT* and *VARCHAR[16]*.
  - 1 point, in case of data type *VARCHAR[32]*.
  - 0 point, all data types not mentioned.
- Entropy (2):
  - 2 points, in case the entropy is less than 6, but greater than 0.
  - 1 point, in case the entropy is less than 12, but greater than 6.
  - 0 point, in all other cases.
- Number of distinct values (1):
  - 1 point, if the number of distinct values is not 1.
  - 0 point, otherwise.
- Number of null values (1):
  - 1 point, in case there are no zero values.
  - 0 point, otherwise.
- Functional dependency (1):
  - 1 point, in case the column is a *dependent* in a valid FD.
  - 0 point, otherwise.

A column for the *COUNT* position is scored as follows:

- Data type (3): Same as *GROUPBY*.
- Entropy (2):
  - 2 points, in case the entropy is less than 12, but greater than 6.
  - 1 point, in case the entropy is less than 6, but greater than 0.
  - 1 point, in case the entropy is less than 15, but greater than 12.
  - 0 point, in all other cases.
- Number of distinct values (1): Same as *GROUPBY*.
- Number of null values (1): Same as *GROUPBY*.
- Functional dependency (1):
  - 1 point, in case the column is a *determinant* in a valid FD.
  - 0 point, otherwise.

For example, the total score of the query in Figure 6.2 can be calculated by multiplying the scores of the columns *Relatie* and *Treinnumer*, each for their position in the query. So if we assume that the score of *Treinnummer* for its position in the query is 8 and that of *Relatie* 6, it gives a total score of 48/64 or after recalculation 75/100.

REMARK This rating system is the result of many iterations, which works properly to exclude uninteresting queries in the context of event data used in this work. It is not guaranteed that this system will produce equally good results for other domains. In that case the query filtering can be changed to a more applicable filter procedure.

6.1.3 Results

If we apply this in practice, we see that approximately ±85% of the queries can be excluded. Everything, of course, depends on the threshold that is set in advance that the potentially interesting queries must exceed. Depending on how strict the selection must be in this first phase, the threshold can be increased or decreased.

It is very difficult to determine how well this algorithm performs. One can look at the queries that are excluded and see if there are any interesting ones. Or one can look at the percentage of queries that are ultimately considered interesting in the last step. But in the end it turns out that this is all subjective and can be very different from person to person.

6.2 Based on clustering

6.2.1 Approach

After sufficient queries are excluded in the previous step (pre-reduction), it is now feasible to execute the remaining queries on the data. Based on the results of those queries, it can then be determined whether a query is consistent or not. Due to the fact that consistency checks need to know which types of days to compare (as we will see in Section 6.3), the data files need to be divided into clusters. A cluster in this case represents one type of day. If we look at the train event data, there are some types of days that may have different schedules:

- Ordinary days in a normal week;
- Saturday and Sunday (Weekend);
- Days during a holiday period;
- Days during a strike period;
- Days during a *lockdown* period due to a virus.

In order to divide the different days into clusters, it must first be determined based on which *subdata* this happens. As established before, it is not feasible to compare the complete data sets with each other. It was, subsequently, decided to use the results of the queries themselves as *subdata*. Thus, each query is executed on the set of daily event files (representing the different days) and based on the query results, the clusters are formed. This gives a good idea on which data the cluster best can be calculated. In this phase, the clusters of each query are calculated to:

- Determine the clusters of the data set
- To (possibly) exclude more queries
  - Case 1: If no clusters can be found, this means that the results are always different.
  - Case 2: If only one cluster is found, this means that the results are always the same.

  Neither case is what we are looking for in the search for interesting queries

6.2.2 Implementation

Algorithm

To find the clusters, a self-implemented *iterative* K-means algorithm (see also Section 3.2) is used. Since it is not known in advance how many clusters there are, the standard K-means algorithm can not be applied, because it expects the number of clusters as a parameter. As a solution, the K-means algorithm is run multiple times. In the first run the algorithm tries to create one cluster. In the second run it will try to form two clusters. This way the number of clusters is gradually increased until the correct number of clusters is found. The elbow technique is used to determine the correct number of clusters. As mentioned earlier, it uses a well-chosen parameter (e.g. cluster radius or diameter) to measure the quality of the clusters found. By starting with one cluster and increasing the cluster amount, the selected parameter will drop exponentially until the correct number of clusters is reached, after which the parameter remains more or less constant. When calculating how much the value has dropped in a given step, the reference point is always the value of the first run. Specifically the calculation is the following:

$$drop = ((lastDiameter - currentDiameter)/firstDiameter) * 100.$$

Tests

Tests were conducted to determine the percentage above which it can be assumed that the value of the selected parameter will no longer drop exponentially, as there is no generally accepted value. For this purpose, data sets with the number of clusters known in advance were used to determine the percentage. More specifically, it determined to what extent the percentage drops after the ideal number of clusters is exceeded in the next run. This was done for two different parameters: diameter and distortion. The diameter of a cluster is the maximum distance between two data points of the cluster, and here the average diameter of all clusters is used. The distortion is the average distance from the data points to the centroid/clustroid of the cluster where they belong to. The results of the tests, using two data sets, are shown in Tables 6.1 and 6.2. The first data set consists of 47 train event files and contains 2 clusters. The second data set consists of 117 train event files of and contains 3 clusters. In both cases, it can be seen that distortion is the most reliable parameter, as it does not drop more than 2% in subsequent runs, after the right number of clusters is reached. This is less the case with the diameter, where this value fluctuates even after the number of clusters is exceeded in subsequent runs. Due this result, the distortion was used as parameter to determine the number of clusters. The threshold was set at 6%, this means as soon as the distortion drops below 6%, the number of clusters in the data set is said to be found.

Clusters	Diameter		Distortion	
	Value	Drop (%)	Value	Drop (%)
1	46596	/	15162	/
2	12746	73	3549	77
3	10798	4	3203	2
4	9159	4	2939	2
5	6387	6	2663	2

Table 6.1: Results for the first data set with 2 clusters and 47 files. The diameter value does stagnate temporarily and finally increases after the right number of cluster found in run 2. The Distortion is more reliable as it stays around 2%.

Clusters	Diameter		Distortion	
	Value	Drop (%)	Value	Drop (%)
1	57813	/	16161	/
2	23654	59	5266	67
3	16699	12	3411	11
4	11605	9	3093	2
5	11032	1	2885	1

Table 6.2: Results of the second data set which contains 3 clusters and 117 files. Both parameter show a slow drop after the right number of clusters is found. But the rate of change is much lower for the distortion.

A	B	C	D
10	8	2	20

A	B	D	F
10	5	20	3

Table 6.3: Examples of two data points consisting of key-value pairs.

In these tests it was noticed that a lot depends on the data set used. It is important that they are large enough, in order to the clusters can be determined correctly. If a data set contains too few files, the correct clusters are not found and often each file ends up in a separate cluster. The reason for this is that the parameter never really stabilizes if there are too few data files. This is because the reassignment of a data file to another cluster has in this case an too great influence.

Data representation

The representation of the data points has an influence on how to determine the distance between two data points. In case the data points consist of a single value, the calculation is fairly straightforward. But in case of the *GBC-query* (and also other possible aggregations), the data used to determine the clusters consists of key-value pairs. So, the difference between two data points cannot simply be calculated by subtracting two values. It was then chosen to see the distance between two data points as the number of changes that have to be made to equalize the 2 points. Adding/removing a key counts as much as increasing/decreasing a certain value. For example, the difference between the two data points shown in Table 6.3, consisting of key-value pairs, can be calculated as follows:

$$abs(10 - 10) + abs(8 - 5) + abs(2 - 0) + abs(20 - 20) + abs(0 - 3) = 8$$

Because of the key-value pairs, it is not self-evident to calculate the centroid of a cluster. One possibility is to work with average values for every key that occurs, but this requires besides extra memory and complexity (representation of the centroids), a lot of calculations every iteration. That is why, it was decided to work with clustroids instead of centroids, so each cluster is represented by the data file closest to all others (in the cluster).

An additional motivation for the above choice is that at the beginning of the algorithm for each file a list is created with the distances to all other files. This is because the files are very often compared and then the distance between these files does not have to be recalculated every time. Since the distances between all files are already available, no new calculations have to be made when determining the clustroid each iteration.

```
read data files;
calculate distances between data files;
numberOfClusters = 1;
while !(distortionDrop < 6%) do
    run K-Means algorithm for numberOfClusters (a number of times) ;
    select best clustering (of all runs);
    calculate distortionDrop;
    numberOfClusters++;
end
```

Algorithm 3: Pseudo code of the iterative k-means algorithm for 1 query

6.2.3 Results

Queries that use different columns of the data set or queries of a different type generate different data to cluster on, which may lead to other clusters being found. The clustering result for each query separately can, therefore, be used to mark any queries as not interesting (as explained in Section 6.2.1). From the clustering results of all queries, one must be chosen, which counts as clustering for the entire data set. For this, the most common clustering, i.e. the clustering found by most queries, will be selected as the clustering of the data set

The most common clustering of the event data of the trains, is found by more than 1/3 of all remaining, possible interesting, queries. If we increase the threshold in the previous filtering step and thus have fewer queries, it is noticeable that the percentage of queries that give the same clustering increases. The following clusters are found:

- Weekdays
- Weekdays during a holiday period
- Weekend days

The algorithm has been tested on various data sets and the results are very promising. It is able to derive the different logical clusters and even identify in some cases files that are outliers, e.g. incomplete data files, by placing them in separate clusters. Furthermore, the algorithm is written in such a way that it can be extended with all kinds of queries that can be used to cluster the data.

But there are, of course, also some limits when using this algorithm. For example, the event data of the trains differs slightly from day to day during the week, and the algorithm is not able to distinguish this. A distinction can only be made once there is a clear difference, as is the case for week and weekend days. In addition, as already mentioned, when there are too few data files in the data set, the algorithm is not always able to recognize the correct clusters. The minimum number of files generally required is difficult to determine as it depends on data set used. Tests has shown that with a minimum of 20 to 30 files, it can be assumed that the algorithm is capable of successfully deriving the clusters. Finally, the algorithm is not parallel. As the number of queries and/or files in the data set increases, the execution time will also increase. There are possibilities to improve the algorithm in terms of speed, e.g. the different K-means algorithm runs, for the same number of clusters, can be run side by side.

6.3 Based on consistency

6.3.1 Approach

Given that it is known which files belong to which cluster and which files need to be compared, the consistency of the query can be calculated. This is not a trivial task, as there are many

possible ways to calculate consistency of a query. At the level of the clusters, it must be determined if consistency of a query is calculated for one or all clusters. In the case of one cluster, this result is taken as consistency for the entire clustering. If all clusters are used, it must be determined if the average, minimum, maximum or some other aggregation of all values is chosen as final consistency result (which applies to the entire clustering of the query). At the level of the results within a cluster, in case of the *GBC-query*, we are dealing with key-value pairs. This entails some choices that need to be made:

- Which measure is used to calculate the consistency of values?
- How is the consistency of a key determined?
- When is a key inconsistent?
- How to calculate the final consistency of the cluster?

Different answers are possible to all these questions and it is difficult to determine which is the best option. It is, nonetheless, important that all factors are taken into account. For example, if you only look at the values of the keys to calculate consistency, the results may be biased. To illustrate this, suppose that keys themselves are not consistent, i.e., they only occur once in 10 times. If only that one is considered, then the key can not be indicated as inconsistent. So, determining when the results of a query can be considered consistent, or not, is not something that can be addressed quickly.

6.3.2 Implementation

In this implementation, the final consistency of a query is chosen to be the average value across all clusters. In the case of key-value pairs in the cluster, the keys are first assessed separately, and are considered consistent if they occur in at least 75% of the results. If a key is considered consistent, and only then, its values are considered. The median and MAD of these values are then determined, which decide if a value is seen as an inconsistency, i.e. an outlier. After each key and its values has been processed, the average number of *normal* values per key is considered as the final consistency of the query within that cluster. In case the cluster simply consists of a series of values, the median and MAD are determined directly on them, which again decides whether a value is considered as an inconsistency. The percentage of non-outliers is then taken as the final consistency of the query.

```
percentage = 0;
for k in keys do
    get all values of key k;
    count values of key k;
    if k is consistent then
        calculate MAD of all values;
        calculate MEAN of all values;
        calculate percentage of values of key k that are considered outliers;
        update percentage;
    else
        update percentage (inconsistent key);
    end
end
calculate average percentage of all keys;
```

Algorithm 4: Pseudo code of the calculation of the consistency of a query.

6.3.3 Results

Now that the consistency is known over several days, it can be determined whether a query should be labeled as interesting or not. If we look at the two extremes that can be achieved, it is noticeable that both are not interesting:

- 0% consistency: The results of the query change daily.
- 100% consistency: The results of the query never change.

In general, it can therefore be assumed that queries are interesting if they achieve a high percentage in terms of consistency, but not 100%.

If we look again at the event data of the trains, there are some queries that are consistent. Some of the queries are even 100% consistent, so they are not interesting. Of the other part of the queries, there are many queries that achieve a percentage between 82-97% and are therefore certainly eligible to be selected.

Chapter 7

A monitoring system

After discussing the algorithms in Chapter 6 to detect interesting queries, we will now see how this all fits together in one system. A system capable of providing these interesting queries, then detecting anomalies and finally displaying the results. The system consists of 2 parts: initializing and monitoring. In the initialization everything happens from uploading the data set to finally selecting the queries to be monitored and is covered in Section 7.1. The monitoring consists of first uploading a new daily file and finally analyzing the found anomalies, and is explained in Section 7.2. At the end of this chapter, in Section 7.3, we discuss what possible extension the system can undergo.

7.1 Initialisation

At the beginning of the initialization of a new project, some necessary things need to be done:

- *Name the new project*
- *Uploading a sample file*: Such that for later calculations there is certainly a file that is known to be a good representative example of the daily event data. If a file is randomly selected from the initial data set, there is a chance that it may not be a good example of the event data (e.g. a file with missing or bad data). This could eventually lead to incorrect results.
- *Uploading the statistics of the sample file*: The statistics of (preferably) the sample file, generated by the Metanome tool. The system here expects the statistics in the (JSON) format that Metanome generates them, otherwise they cannot be parsed. In principle, some of these statistics could be calculated by the system itself using existing libraries and the sample file. The main reason that Metanome is used is that it can recognize the different types of columns better than other tools/libraries.
- *Uploading the functional dependencies of the sample file*: The functional dependencies of (preferably) the sample file, generated by the Metanome tool. The system here expects the FDs in the format that Metanome generates them, otherwise they cannot be parsed. The self-search for FDs in the sample file is not self-evident, but could possibly be later added as an extension of the system.
- *Uploading the initial data set*: The data set required to determine the clusters in the event data and can be used as the first historical data. The files can be delivered in both CSV and PARQUET format. When uploading CSV files, it is important to clearly specify which delimiter must be used to parse the files.

7.1.1 Cleaning and filtering the data set

In this step, after the necessary files have been uploaded, the data set is named and cleaned up if necessary:

- *Name the data set*: This means that a date must be assigned to each file. This is mainly necessary to display the history in a correct and clear manner in the monitoring phase. The system is (for the time being) unable to deduce this itself and sometimes it is not even possible to extract it automatically from the files.
- *Clean up data set*: Before the data set is used to find the interesting queries, it is checked for bad files. By bad files we mainly mean incomplete or corrupt files (e.g. wrong columns). So, first is derived from the data set what the event data files normally look like and based on this, files may be marked as a possible error. It is then up to the user to determine whether this is actually the case. At this point, the system mainly checks the number of columns and rows, which can be expanded if necessary.

Subsequently, an initial ranking can be made of all possible queries, this is done as discussed in Section 6.1. So all columns get a score for every possible position in the queries. The queries whose total product (of the columns used) is above the defined threshold are taken to the next step. The other eliminated queries are no longer considered.

7.1.2 Searching for clusters

The clustering is then calculated, as seen in Section 6.2, for any query left after the filtering. Depending on the number of queries and the size of the intimate data set, this can take some time. The most common clustering, the clusters identified by the majority of the queries, is presented to the user. As mentioned earlier, the system is limited to a certain level of detail to distinguish the types of days. Therefore, the user is given the opportunity to verify the clustering found and make changes if desired.

7.1.3 Selecting the best queries

Once the clustering is known, the consistency for each query can be calculated, as indicated in Section 6.3. Thus, each query must be executed on all data of each cluster and its results must be assessed, in order to obtain a joint score of all clusters which represents the consistency of the query. Ultimately, the user gets an ordered list of the queries that the system finds interesting. Each query in the list has the necessary information attached so that the user can make a good choice:

- *The written query*
- *Example output*: An example of the output the query generates, to help the user understand the query better
- *Consistency score*
- *Data score*: The score of the query that was given to filter in the first step (based on the statistics)
- *Similarity*: The similarity that the clustering of the query itself has with the ultimately selected clustering of the data set. Logically, queries that have similar clustering are more likely to be consistent. The similarity has a small influence on the final score.
- *Final score*: The final score where consistency has the largest share. As mentioned earlier, a query that is 100% consistent is not interesting (since it never has outliers), so such a query won't score high as a result.

	04/06/19	05/06/19	06/06/19	12/06/19	13/06/19	16/06/19	19/06/19	23/06/19	25/06/19
Treinnummer	3790	3795	3794	3799	3833	3814	3828	3763	3775

Figure 7.1: Example of the historical data of a simple *COUNT-query*

After the scores for each query have been determined, the centroid is calculated and saved for each cluster. The calculation is based on the top-rated query, which also has a similar clustering to that selected by the user in the previous step. The centroids serve to identify the nearest cluster of a new event data file more quickly during the monitoring phase. This avoids having to recalculate the entire clustering (unnecessary) every day.

In addition, for each selected query, a sort of summary is already made per cluster, which will serve as the quickly accessible historical data. This should ensure that as soon as the cluster of a new day is known, it is not necessary to query every data file in the cluster every day to collect all the previous results and detect any outliers more quickly. In Figure 7.1, an example is shown of what this data would look like from a simple *COUNT-query*, containing only one value for each day, in this case the number of trains that ran.

7.2 Monitoring

Once everything is set up, daily monitoring of any outliers can begin.

7.2.1 Adding a new day

When a new day is uploaded, the structure of the file will be checked first, as is also the case in the initialization phase. This is to prevent the use of an incomplete or corrupt file. If the structure of the file is correct, the user can go to the next step where the cluster of the new day is determined. This is done on the basis of the stored centroids of the selected clusters. The nearest centroid determines the cluster, which is presented to the user. The user is subsequently given the option to agree to the proposed cluster or select another one. As soon as the cluster of the new day is known, then it can be checked if there are any outliers. This is done for all selected queries in the initialization phase and is based on the generated history (historical data) stored for each of the queries. This history consists of a fixed number of days of that cluster and is updated when a new day is added. This ensures that any *bad* days with many deviations will not affect the expected values forever. In the previous step, the user has also the option to indicate whether or not to add the new day to the existing history. This option allows the user to compare a day with the history of a particular cluster, without influencing the historical data.

7.2.2 Summarize the results

To determine possible outliers of the monitored queries, the mean and standard deviation of the historical data are used. Each outlier is than classified based on how many standard deviations the value deviates from the mean. So that, at a later stage, a distinction can be made in the type of warning that must be given. The user may not be interested in any small deviations that may occur. The output for each query is given in the form of a histogram, so that the trend of the last days in the cluster is quickly visible (each bar represents a value for a specific day). The color of each *bar* in the histogram may vary depending on whether the value is an outlier or not. In the case of an outlier, a further distinction can be made in color, by varying it according to the number of standard deviations that this value is removed from the mean. In Figure 7.2 an example is shown.

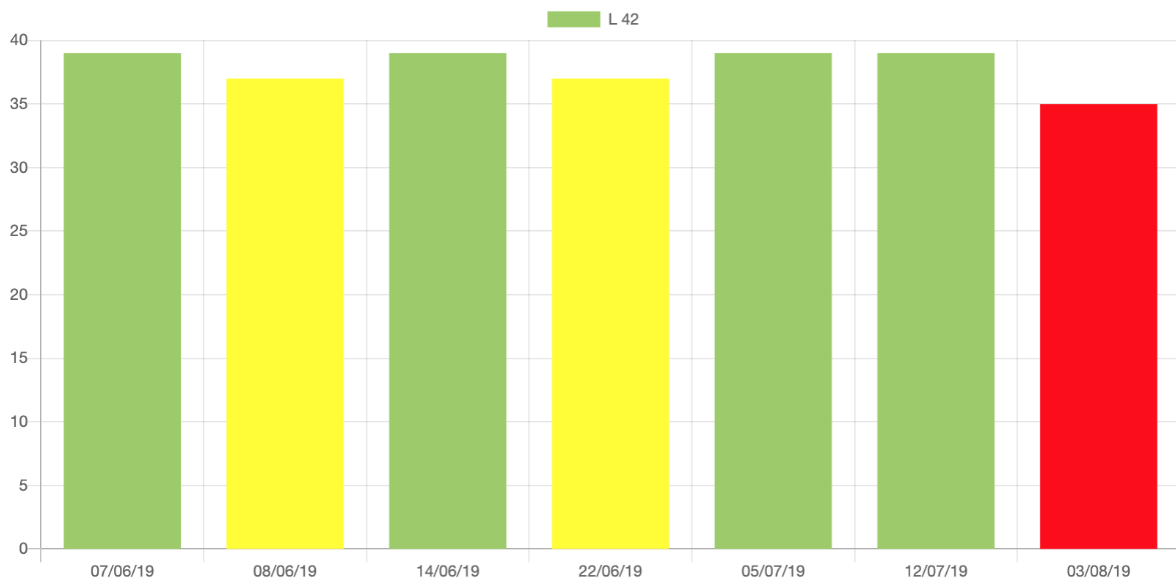


Figure 7.2: The output of a monitored query, where the expected value (colored green) is 39: small deviations are colored yellow, while greater deviations are colored red.

7.2.3 Recalculations

If the correct cluster is not suggested when adding a new day, the centroids of certain clusters may be close together. If this is the case, it is recommended to recalculate the centroids. Based on the new days added to each cluster, a more accurate centroid for each cluster may be calculated. In principle, the recalculations may happen every day, but it is not necessary if the centroids of the clusters are accurate enough. The recalculation of the centroids is currently not yet supported.

7.3 Extensions

The current system can be seen as a kind of prototype or basic tool that can be further expanded. Below is a list of possible extensions in response to certain shortcomings of the current system.

- The system uses metanome to derive the types of the different columns in the event data. A possible extension is that this can be implemented as part of the the system (possibly using an existing library if there exists one for this). This is also the case for deriving FDs.
- At this time, the dates of each file have yet to be specified, although this may be derived automatically in some cases.
- When cleaning the data set, an overview is given of all files present and the columns in these files. A possible addition is to give more detail about the columns, such that the user may be given the option of already designating some columns as not interesting.
- When cleaning the data set, very simple error detection is now done based on the number of columns and rows. A possible extension is to check if the column names in all files are the same, as one would expect. (This can be done with the sample file)
- When selecting the clustering, only the most common clustering is shown. Extend this so that the user can choose from several common clusterings.
- The possibility to indicate that a particular query, which is not presented as interesting by the system, should be monitored daily.

- Once a day has been added to the history of a particular query for a given cluster, it can no longer be deleted.
- For each query per cluster, history is currently maintained for a fixed number of (10) days. Allowing this number to be set dynamically is a possible addition.
- The recalculation of the centroids is currently not yet supported and can be implemented.
- Exporting the results of one or more monitored queries

Chapter 8

Experiments

Chapters 6 and 7 already mentioned which intermediate results were achieved with the event data of the trains in the different steps (filtering and clustering). In Section 8.1, the final results are discussed. So, which queries are considered interesting and can possibly be monitored. Then Section 8.2, the data set of *DeLijn* is tested with the implemented system. In principle, it can be assumed that this data is fairly similar since it is the data of buses instead of trains. These tests can provide more insight into how the system can get a more general setup that also works for data from other domains. As last, in Section 8.3, we summarize our findings about the results of both data sets.

8.1 NMBS

This experiment made use of available event data from the months June 2019 and July 2019. The initial data set consists of 27 files, so clearly a lot of data is missing. However, this data set normally contains all possible types of days that one can expect. In the first filtering step, 2 files are identified as possible outliers. Our own analysis shows that these files can considered as normal, so they are not removed from the data set. In the next step, the clusters are found as described in 6.2.3: weekdays, weekdays during a holiday period and weekend days. After the proposed clustering has been approved, we can deduce the following from the list of suggested queries:

- The next 2 queries achieve the highest scores, both queries are expected to be presented as interesting:

  ```
  (A) SELECT Relatie, COUNT(DISTINCT Treinnummer)
      FROM train_event_data
      GROUP BY Relatie
  ```

  This query indicates how many trains are running each route. The high recommendation is due to the consistency of 83% and maximum scores on the other aspects.

  ```
  (B) SELECT Treinnummer, COUNT(Treinnummer)
      FROM train_event_data
      GROUP BY Treinnummer
  ```

  This query indicates how many times each train stops. The high recommendation is due to the consistency of 84%, 100% similarity of the clusters (clusters of query vs. selected clusters) and a data score of 75%.

- This query ranks high, but this is not within expectations:

```
(C) SELECT Relatie, COUNT(DISTINCT Spoorwegoperatoren)
    FROM train_event_data
    GROUP BY Relatie
```

  The reason for which Query C should not be recommended, is because one should expect a valid FD (*Relatie* → *Spoorwegoperatoren*) to be found for these columns. This query gives the number of railway operators per route. Since each route designation is unique per railway operator, this results in a logical 1-to-1 mapping. As discussed in Section 6.1.1, this is an uninteresting query. Because this query is not filtered out and therefore scores a high consistency of 97%, the query is highly recommended. The reason the FD is not found remains unclear.

- There are queries that give a similar, or even exactly the same result:

```
(D) SELECT Relatie, COUNT(DISTINCT Spoorlijn_van_vertrek)
    FROM train_event_data
    GROUP BY Relatie
```

```
(E) SELECT Relatie, COUNT(DISTINCT Spoorlijn_van_aankomst)
    FROM train_event_data
    GROUP BY Relatie
```

  The above two queries give a very similar result with a slight deviation. After all, it is logical for a train to have the same number of departure tracks as arrival. The scores of these two queries are therefore equal. We assume that the reason for the slight deviation may be due to missing data.

```
(F) SELECT Treinnummer, COUNT(DISTINCT Naam_van_de_halte)
    FROM train_event_data
    GROUP BY Treinnummer
```

  Although this query does not use the same columns, it still gives the exact same result as Query A, which gives the number of stops per train. This is not illogical, since every stop happens at a stop, with the result that the numbers are identical. If there occurs a deviation in the number of stops, outliers will be detected for both queries. The lower recommendation of this query is due to a lower *data score*, the other scores are of course the same as the high recommended query.

8.2 DeLijn

This section first discusses the data set of *DeLijn*. Then is looked at the adjustments that were necessary in the implementation. And finally, the results of the system with this data set are discussed.

8.2.1 Data

Description

Although both data sets contain stops of vehicles, *DeLijn*'s data is less readable compared to that of *NMBS*. The data set contains many columns of IDs that refer to entities described in other data sets. This makes the data very cryptic, with the result that, for example, a data scientist will not be able to derive many useful information based on this data set alone.

Structure

The event data of *DeLijn* is kept, like the event data of *NMBS*, per day and has the following structure:

- The event data (of 1 day) is kept in multiple parquet files (There drive a lot more buses than trains).
- Each file consists of ±250000 rows, usually there 4 files a day, so on average more than 1000000 events take place per day.
- Each event represents the passage of a bus at a stop.
- 24 attributes are stored for each event (= 24 columns).

The attributes of each bus event are:

- Partition (*SMALLINT*): public number of the route the bus is driving
- PartitionKey (*DATE*): date of the event
- RowKey (*VARCHAR[32]*): composition of the PartitionKey and the Partition/internalLineNumber
- blockId (*VARCHAR[32]*): block id (no more info available)
- blockidentifier (*DATE*): date of the block
- delay (*REAL*): measured delay of the bus at the stop
- destination (*TEXT*): final destination of the bus
- directionCode (*SMALLINT*): code indicating the direction of the ride (no more info available)
- distanceToNextTrip (*INT*): distance to the next trip, in which unit (number of stops or number of kilometers) is not clear.
- entitynumber (*SMALLINT*): number of the region in which the bus runs, in this case this is done at the level of the provinces
- internalLineNumber (*SMALLINT*): internal number of the route the bus is running
- loadTime (*SMALLINT*): loading time of the data (no more info available)
- nextTripId (*VARCHAR[32]*): ID (rowKey) of the next trip that the bus is driving
- p (*VARCHAR[16]*): indicates if the bus open to the public ('y' = yes)
- realTime (*TIME*): time the bus arrives at the stop

- scheduleTime (*TIME*): time the bus was scheduled to arrive at the stop
- sequence (*SMALLINT*): number of the stop, of that ride
- stopId (*ID*): ID of the stop
- t (*TIME*): current time
- timeOfFirstStop (*TIME*): time when the bus started its ride
- timeOfLastStop (*TIME*): time when the bus ended its ride
- tripStopIndex (*REAL*): number of the stop, of that ride (starts count from 0)
- tripnumber (*SMALLINT*): number on the bus during the ride, visible to travelers
- vrtnumber (*INT*): number of the bus

The following functional dependencies can be derived:

- Rowkey → blockId
- RowKey → Partition
- nextTripId → distanceToNextTrip
- internalLinenumber → entitynumber

8.2.2 Implementation

The implementation of the system, as described in the Chapter 7, is the result of the tests with the event data of the buses. Since the system was designed for the event data of the trains, some small additions were needed to fully support the event data of the buses.

- Supporting multiple Parquet files, which are later merged into one file
- Adding the scores for the column types that did not appear in the event data of the trains (INT, TEXT, VARCHAR [32], ...)

8.2.3 Results

DeLijn's experiment uses the available event data from the period August 18, 2019 - September 14, 2019. This initial data set consists of 28 files, so no files are missing. None of these files are later marked as possible outliers.

Performance

The event data of the buses consists of more than 1000000 events daily, which is 17 times more than in the case with the trains. In addition, 6 additional attributes are kept per event. This means that all steps in the developed system, logically, take much longer. Although the time required during initialization is not of such great importance, for data sets of this size it may be interesting to add parallelizations. Obviously, the most important thing is that interesting queries are presented at the end of the process.

Intitialisation

As indicated earlier, it is difficult to say how well the filtering is done. But for this data, with 24 attributes and more than a 1 million events per day, this step is of utmost importance. Otherwise, it would not be feasible to test and use all possible interesting queries in a considerable amount of time, taking into account different columns or different types in the following steps. Just as with the data of the trains, about 84% of the queries can be excluded.

Clustering works very well, as was the case for the event data of the trains. This time however, the weekend days are placed in separate clusters. In the end, the following clusters are identified:

- Weekdays
- Weekdays during a holiday period
- Saturdays
- Sundays

Some of the queries are able to distinguish even Wednesdays from the rest of the weekdays. While this clustering is not considered the most common, it does show how using query results can lead to very solid clustering results.

Next, if we look at the queries presented by the system, we see that a noticeable number of queries score high and are therefore recommended. The main reason for this is that they achieve a very high consistency, which is closely related to the fact that almost all find the same clustering as the one selected in the previous step. The fact that this data set is more complete compared to *NMBS*'s data set certainly plays a role. In addition, it can be assumed that the events are generally more consistent in this data set. At *NMBS*, 20 queries were recommended, here are 100 queries with a significant score. Below is an overview of some noteworthy queries.

- Some queries that are interesting according to our point of view and also are recommended by the system:

```
(A) SELECT entitynumber, COUNT(DISTINCT internalLinenumber)
    FROM bus_event_data
    GROUP BY entitynumber
```

  Query A is one of the highest recommended queries, with almost a maximum score. This query indicates how many line numbers there are per region.

```
(B) SELECT internalLinenumber, COUNT(DISTINCT stopId)
    FROM bus_event_data
    GROUP BY internalLinenumber
```

  This query above indicates how many stops there are on each line. The *data score* of this query is not that high (56%). But due to its high consistency (98%) and similarity (100%), it is still recommended.

```
(C) SELECT COUNT(DISTINCT internalLinenumber)
    FROM bus_event_data
```

For now we only showed recommended *GBC-queries*, but there are also other types of queries that are recommended. Query C indicates how many different line numbers there are in total.

- A query whose meaning is not entirely clear to us, but is recommended:

```
(D) SELECT directionCode, COUNT(DISTINCT internalLinenumber)
    FROM bus_event_data
    GROUP BY directionCode
```

The meaning of this last query is not very clear to us. Of course it indicates how many line numbers there are per direction code. But the exact meaning behind each direction code is unknown to us. All we know for sure is that the query gives very consistent results, and therefore, is recommended. A domain expert of *DeLijn's* data will most likely be able to clarify this query.

Monitoring

If high-scoring queries are selected, the monitoring of queries from the event data of buses differs little from that of the trains. The correct clusters are suggested when adding new days, and the results are correctly displayed in the form of histograms. As a test, we looked at a query that is not recommended by the system, i.e. a query with a low end score: the *GBA-query* (GroupBy-Average query) that gives the average delay per bus stop.

```
SELECT stopId, AVG(delay)
FROM bus_event_data
GROUP BY stopId
```

This is not a good idea, especially because the consistency of this query is far too low. This entails that, with the addition of a new day, many outliers are found, and for all of these a seperate histogram must be drawn. This, obviously, takes a lot of time and requires much memory. If you, then, want to look at the outliers of the query for that day, it can be seen that there are outliers for almost every key (bus stop). In addition, there are many bus stops, which means that the overview is completely lost. A query that focuses more on bus stops within a certain area might be a better option. However, this type of query is currently not supported by the system.

```
SELECT stopId, AVG(delay)
FROM bus_event_data
GROUP BY stopId
WHERE entitynumer = 4
```

8.3 Conclusion

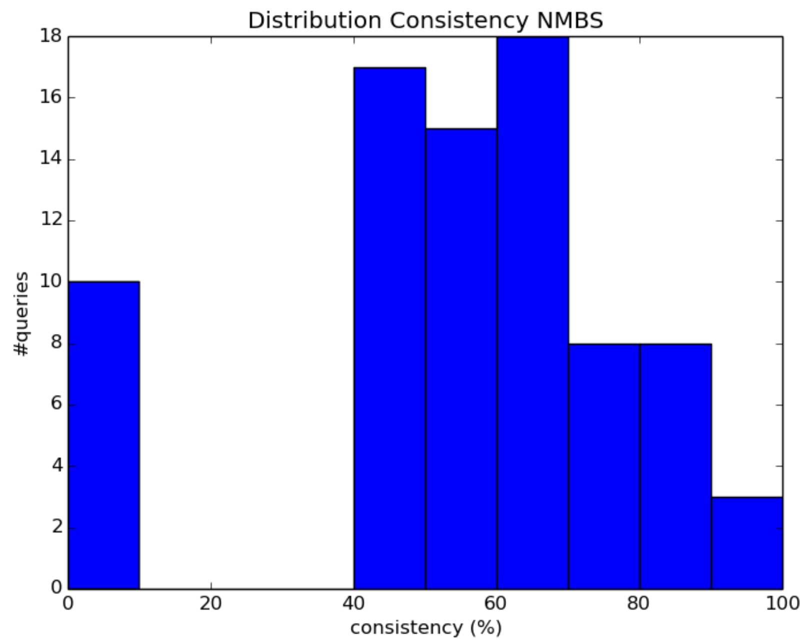
Both data sets obtained very similar results for both filtering (Table 8.1) and searching for clusters. However, the biggest difference with the *NMBS* was the number of queries that received a high recommendation in the last step of the initialization. As indicated earlier, this is due to the fact that there is more consistency in the daily event data (Figure 8.1). This was certainly not the case with *NMBS*, so consistency was implemented as a crucial factor in the calculation of the final score. As a result, many queries with the data from *DeLijn* now receive the same assessment (Figure 8.2). We are sure that among these queries there are certainly some very interesting ones. A specification of the data, which we do not currently have available, could

provide a better picture of how many of the suggested queries are meaningful. In addition, it was possible for *NMBS* to determine similar queries manually. For *DeLijn* this is an impossible task because of the larger number of queries.

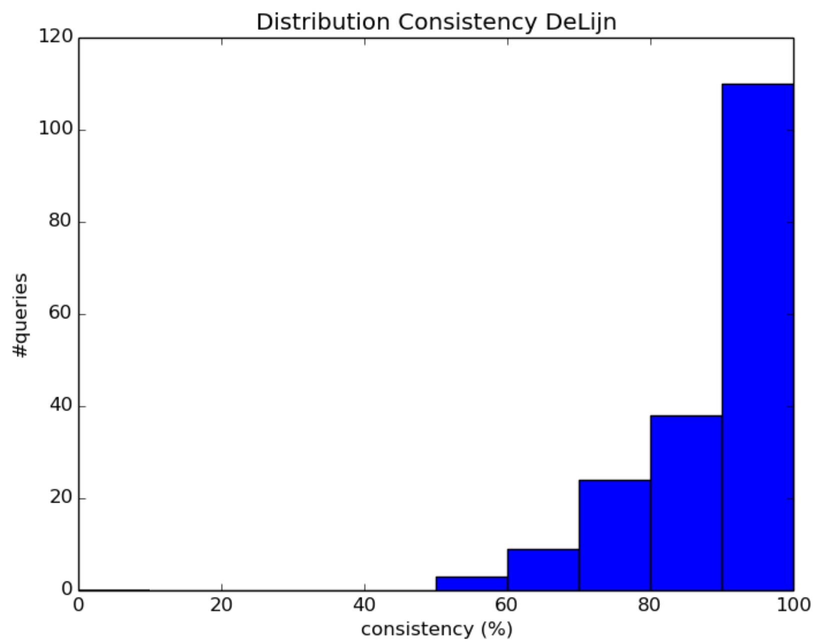
	#columns	#queries	#queries (after filtering)	queries filtered out
NMBS	18	666	79	88%
DeLijn	24	1176	184	84%

Table 8.1: Overview of filter results of data sets NMBS and DeLijn

Furthermore, we could see that a query with a low score at *DeLijn* is neither interesting nor feasible to monitor clearly. A new threshold to prevent such queries from being suggested by the system seems necessary. It makes sense to do this only based on consistency, as it has a direct influence on the number of outliers. However, this value should also depend on the data set or the column on which the *GROUPBY* operation is applied. For *NMBS*, a threshold of 85% would exclude many queries that can still be clearly monitored. The data set is much smaller and contains many columns with a limited number of different values. So, a lower threshold still provides a number that can be monitored. This is not the case with *DeLijn*, this data set is much larger and contains a lot of columns with many different values. Therefore, a higher threshold is more desirable. In short, the threshold should depend on how many values outliers occur. As long as this number can be clearly displayed, the query can be recommended. After this, it is up to the user to select his preferences.

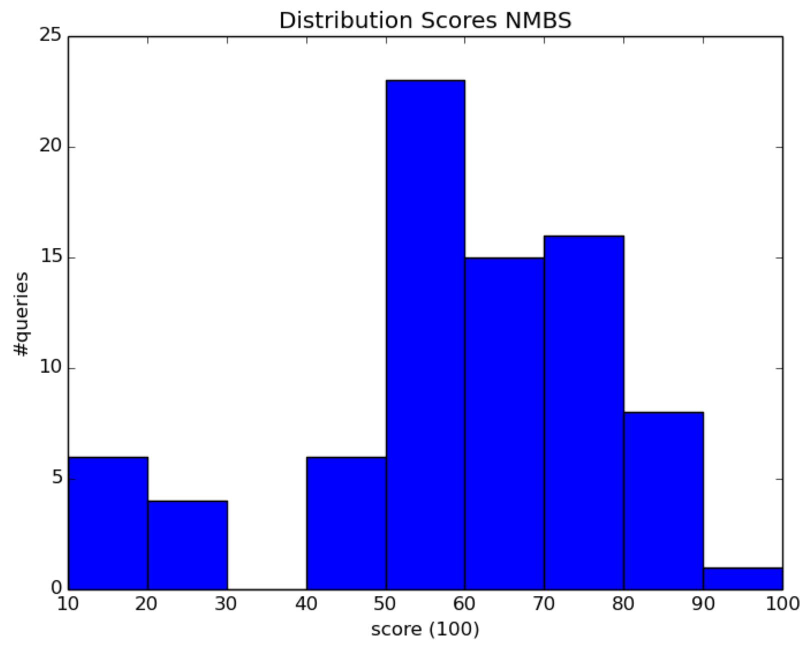


(a) NMBS

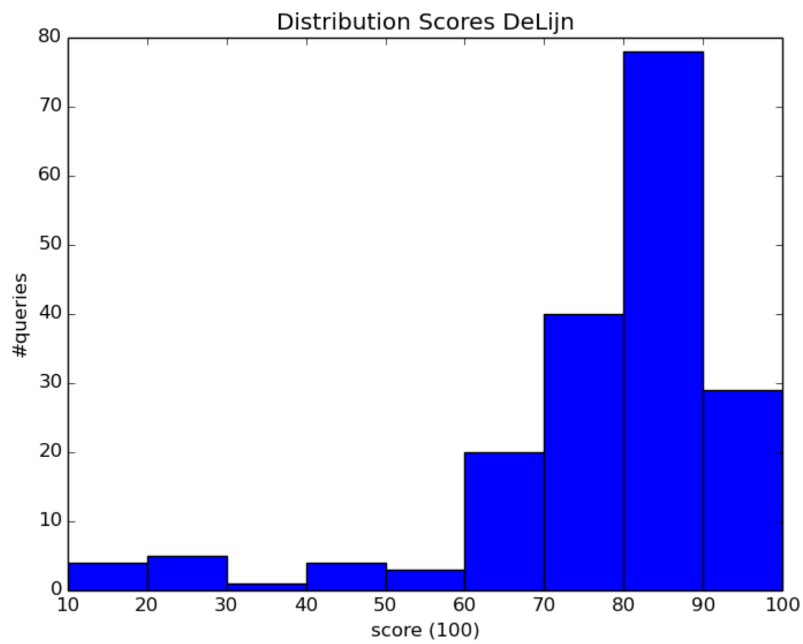


(b) DeLijn

Figure 8.1: Distribution consistency



(a) NMBS



(b) DeLijn

Figure 8.2: Distribution scores

Chapter 9

Conclusion

Current data cleaning tools lack the functionality to write and/or detect rules in the form of aggregations. However, based on these aggregations, more substantive deviations can be detected in data. In this work, we have attempted to develop a system that is able to detect interesting aggregations in event data (Section 5.1.1) and monitor their outliers. The aggregations can be expressed by use of queries (Section 5.2). In order to subsequently determine whether the result of the query contains outliers, historical data is needed. This historical data then consists of previous results of this query performed on the data.

The final developed system consists of two parts: initialization and monitoring. The first part searches for interesting queries. This process mainly consists of applying 3 algorithms: filtering, clustering and consistency calculation. At the end of this process it is known what the clusters of the data set are and which queries need to be monitored. The second part consists of monitoring the queries selected. First a new day is added. Subsequently it is determined, among other things, to which cluster this new day belongs. Outliers are then detected based on the historical data of the corresponding cluster.

In the first step of the initialization, all possible queries are filtered (Section 6.1.1), because it is not feasible nor interesting to test them all (Section 5.3). By applying this filtering, an average of 80% of the queries can be classified as uninteresting (Section 6.1.3). These are therefore no longer considered in the following steps. This step is crucial, especially as the data sets increase in size, to complete the process in an acceptable time. Even though the tests we performed did not filter out interesting queries in this step, we assume that there is a small chance that this can happen. To avoid this, it might be possible to test some of these filtered queries after the last step. Suppose it is still unfeasible to test them all, we see two possible options. The first option is to lower the threshold, so that more queries still reach this threshold. The other option is to select a random sample of queries from the set of excluded queries. Both options can be repeated if necessary, as long as interesting queries are found.

The next step is to look for the clusters of the data set, by use of the remaining queries (Section 6.2.1). The clusters are calculated for each query, the most common clustering is then suggested to the user. The results of this were very good (Section 6.2.3). Many queries are able to find the same and logical clusters. As we increased the threshold in the filtering step, the percentage that found the same and expected clusters increased. If this is an indication that there are more interesting queries left, we leave it in the middle, but we do assume that there are more queries that are meaningful. Furthermore, we could see that this technique also has its limits. For example, no distinction can be made between the different weekdays in the case of train days. The main reason for this, we believe, is similarity and inconsistency. The different weekdays differ very little from each other, which makes it difficult to distinguish them by data.

In addition, we noticed that there is little consistency in the data set, which means that, for example, two Tuesdays can already differ a lot from each other. If the found clusters are not sufficiently detailed or even completely wrong according to the user, he has the option to change the clusters. If the clusters are changed by the user, chances are that there is no good query that will also find these clusters. This can then cause that, when adding a new day, the correct cluster is not always suggested by the system. This can possibly be improved by continuously recalculating the centroids, after adding a new day. This allows the system to better estimate which cluster it concerns over time.

In the last step the consistency is calculated, which plays a crucial role in determining whether a query is highly recommended or not. The final formula used for this is the result of a few iterations and takes all necessary factors into account (Section 6.3.1). The calculation is done for all queries by the same algorithm, so even if there is an error margin here, it is the same for all queries.

After the consistency has been calculated, the scores of all queries that remained after the filtering in the first step are displayed. This was mainly to get a better idea of which queries were highly recommended and which were not, and how much difference there is in scores between the different queries. For the data from *NMBS*, very good queries, with a clear meaning, were highly recommended (Section 8.1). This is not illogical, since the system was built on the basis of this data. The recommended queries from *DeLijn* (Section 8.2.3) were not always clear due to the cryptic form of the data (Section 8.2.1). A domain expert is required to get a better idea of what each query is and if it is meaningful.

Very remarkable in the tests with *DeLijn* was that more queries achieved a high score compared to the data from *NMBS*. The main reason for this is that these queries all achieve a high consistency and therefore score high. The high consistency is mainly due to the fact that the events at *DeLijn* are more consistent. It was also noticed that certain queries are similar or even give completely the same result. This will automatically ensure that when such queries are monitored, they will always give outliers at the same time. If only a limited number of queries are recommended, this can still be checked manually. From the moment there are a more, as was the case with *DeLijn*'s data set, this is impossible to do manually. So it would be good if similar queries could be indicated and/or grouped by the system.

In addition, if this system were to be used effectively, there is a need for a new threshold that the queries must reach. For example, we saw that queries with too low a consistency should not be selected. The main reason is that these will give too many outliers on a daily basis, making it difficult to maintain an overview. So, a consistency threshold seems to be the best option, as the other factors do not play a direct role in the number of outliers. The value of this threshold is difficult to determine in advance, as it depends on the data set used. For smaller data sets, this threshold can be a bit lower. As long as it remains feasible to clearly display all outliers.

To summarize it all, the developed system is able to find interesting queries by use of the implemented process and certainly has a lot of potential. But there are clearly a number of points for improvement, which we have mentioned throughout this chapter.

Chapter 10

Future work

In the conclusion, but also throughout the thesis, a lot of points for improvement have already been mentioned. In addition, we see some more general things that can happen in further research to derive interesting queries from data. First and foremost, the system must be tested with other data sets, so that we get a better idea how robust the system is and which adjustments must be made to support more types of data. In addition, it must be examined which other types of queries are interesting to support. For example, queries that further split data based on an element from an extra column (Section 7.2). This may increase consistency, so that this data can ultimately be monitored. If one wants to support more queries, filtering becomes even more crucial in the process. So in the field of filtering, it must be further investigated which other factors determine whether a certain column is suitable for use in an aggregation. Another direction one can take to support more types of queries is to improve speed in the other steps of the process. For example, parallelizations can be applied in the other phases of the process. Such as, for example, when searching for the clusters or calculating the consistency of the queries.

Appendices

Appendix A

System specifications

This appendix first provides a brief explanation in A.1 of how the system is structured and how it can be executed. Then in A.2, an overview is given where one can find the implemented algorithms in the back-end and how all information of a project is stored. Next, A.3 gives an overview of the available services of the API and how to call them. Finally, A.4 shows the visualization of the different steps of the implemented system.

A.1 Setup

A.1.1 System overview

The system, as shown in Figure A.1, consists of 2 main parts: a back-end and a front-end. The back-end consists mainly of the previously described algorithms that were implemented in Python. In addition, the Flask framework [12] is used to provide an API that offers the functionality. The front-end is written in the Angular framework [6] and makes additional use of Primeng [15]. The latter provides ideal support for the implementation of various UI components.

A.1.2 Installation

To start the back-end, run the following commands in the corresponding folder:

1. Install **virtualenv**
2. Create a new virual environment
3. Run *pip3 install -r requirements.txt*
4. Run *source env/bin/activate* (so that the virtual environment is started)

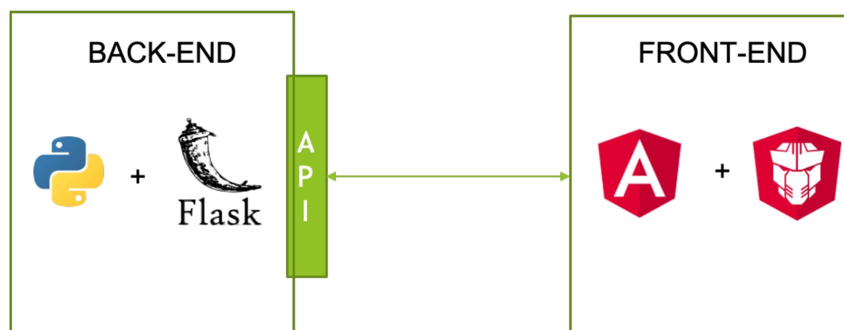


Figure A.1: System structure with used frameworks for each part

5. Run *python3 api.py*
6. The back-end starts up on (port 5000)

REMARK The back-end can also be run without virtual environment, please install the missing libraries specified in *requirements.txt*.

To start the front-end, run the following commands in the corresponding folder:

1. Install **Angular**
2. Run *ng serve* --*open*
3. Run *python3 api.py*
4. The Front-end starts up (port 4200)

A.1.3 Metanome

To obtain the necessary files from Metanome for a data set, following Metanome built-in algorithms are recommended:

- Basic statistics: SCDP
- Functional dependencies: HyFD (with the maximum determintant size at 1)

A.2 Structure

A.2.1 Algorithms

The following python scripts can be found in the back-end folder:

- *check.py*: checks the files, in the initial data set, for any outliers (#columns, #rows)
- *ranking.py*: calculates the *data score* for columns and the consistency of queries for a set of days.
- *clustering.py*: clusters the files
- *process.py*: processes the selected queries and generates, for each query, the centroid and its history.
- *process.py*: processes the selected queries and generates, for each query, the centroid and its history.
- *new.py*: processes new files and determines the outliers, of the selected queries.

A.2.2 Projects

All created projects are kept in the *projects folder* (Back-end). Each project has an ID that is also used to name the folder of each project. After the initialization process, the project contains the following files:

- *columns.json*: overview of the columns that occur in the data set.
- *outliers.json*: overview of the files that were identified as possible outlier in the initial data set.

- *cluster.json*: the selected clustering by the user.
- *clustering.json*: overview of the different clusterings found and through which queries this happened.
- *r_scores.json*: overview of the calculated *data score* per query.
- *c_scores.json*: overview of the calculated consistency per query.
- *scores.json*: overview of all proposed queries with the different scores and their example output.
- *days.json*: overview of the days added after the initialization process, each with an ID, a filename, a date and the cluster it belongs to.

A.3 API

This section gives an overview of all implemented API calls.

/project

- GET: get list of all projects with ID and name
  - Responses:
    - 200 OK:

```
{
    "projects": [
        {
            "id": project.id,
            "name": project.name
        },
        ..
    ]
}
```

/project/create

- POST: create a new project
  - Responses:
    - 200 OK:

```
project.id
```

/project/{id}

- GET: get project name
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
project.name
```

    * 404 NOT FOUND

- DELETE: delete project
  - Parameters: ID of project
  - Responses:
    * 204 NO CONTENT
    * 404 NOT FOUND

/project/{id}/name

- PUT: set project name
  - Parameters: ID of project
  - Body:

```
{
    "name": project.name
}
```

  - Responses:
    * 200 OK
    * 400 BAD REQUEST

/project/{id}/days

- GET: get all added days of the project
  - Parameters: ID of project
  - Responses:
    * 200 OK:

```
{
    "days": [
        {
            "id": day.id,
            "filename": day.filename,
            "date": day.date,
            "cluster": day.cluster
        },
        ..
    ]
}
```

    * 404 NOT FOUND

/project/{id}/day/{dayid}

- GET: get an overview of the outliers per monitored query for a day
  - Parameters: ID of project, ID of day
  - Responses:
    * 200 OK:

```
{
    "queries": [
        {
            "queryType": query.type,
            "queryParams": [paramA,..],
            "nStdType": [A,B,C],
            "outliers": [
                {
                    "columns": [columnA,..],
                    "index": key,
                    "values": [X,..],
                    "type": outlier.type,
                    "mean": mean,
                    "std": std,
                    "min": min,
                    "max": max
                },
                ..
            ]
        },
        ..
    ]
}
```


    * 404 NOT FOUND

- PUT: create a new day to be processed/monitored (search for outliers)
  - Parameters: ID of project, ID of day
  - Body:


```
{
    "date": date,
    "cluster": cluster,
    "add": true
}
```


  - Responses:
    * 200 OK
    * 400 BAD REQUEST
- DELETE: delete a processed day
  - Parameters: ID of project, ID of day
  - Responses:
    * 204 NO CONTENT
    * 404 NOT FOUND

/project/{id}/file/new/{delimiter}

- POST: add new file for monitoring
  - Parameters: ID of project, delimiter (necessary to parse the files correctly)

- Body: Files in CSV or parquet format
- Responses:
  - 200 OK:

```
{
    "id": file.id,
    "cluster": file.cluster
}
```

  - 400 BAD REQUEST

/project/{id}/file/data/{delimiter/id}

- POST: add a file to the initial data set
  - Parameters: ID of project, delimiter (necessary to parse the files correctly)
  - Body: Files in CSV or parquet format
  - Responses:
    - 200 OK:

```
1
```

    - 400 BAD REQUEST
- DELETE: delete a file from the initial data set
  - Parameters: ID of project, ID of File
  - Responses:
    - 204 NO CONTENT
    - 404 NOT FOUND

/project/{id}/file/data/info

- GET: get all info from files from the initial data set
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "files": [
        {
            "id": file.id,
            "name": file.name,
            "filename": file.filename,
            "date": file.date
        },
        ..
    ]
}
```

    - 404 NOT FOUND
- PUT: set all info form files from the intial data set

  - Parameters: ID of project, ID of File
  - Body:

```
{
    "files": [
        {
            "id": file.id,
            "name": file.name,
            "filename": file.filename,
            "date": file.date
        },
        ..
    ]
}
```

  - Responses:
    - 200 OK
    - 400 BAD REQUEST

/project/{id}/file/example/{delimiter}

- POST: add a representative sample file of the data set
  - Parameters: ID of project, delimiter (necessary to parse the file correctly)
  - Body: File in CSV or parquet format
  - Responses:
    - 200 OK:

      ```
      1
      ```

    - 400 BAD REQUEST

/project/{id}/file/statistics

- POST: add file with statistics (from sample file)
  - Parameters: ID of project
  - Body: File in JSON format
  - Responses:
    - 200 OK:

      ```
      1
      ```

    - 400 BAD REQUEST

/project/{id}/file/fds

- POST: add file with functional dependencies (from sample file)
  - Parameters: ID of project
  - Body: File (*Metanome* format)
  - Responses:
    - 200 OK:

      ```
      1
      ```

    - 400 BAD REQUEST

/project/{id}/file/setup

- GET: get info about the setup files
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "stats": stats.json,
    "fds": fds.fd,
    "example": example.parquet
}
```

    - 404 NOT FOUND

/project/{id}/columns

- GET: get columns of the data set
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "number_of_columns": number,
    "column_names": [columnA,..]
}
```

    - 404 NOT FOUND

/project/{id}/outliers

- GET: get outliers (files) of the data set
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "outliers": [
        {
            "type": type,
            "mean": mean,
            "std": std,
            "files": [
                {
                    "name":file.name,
                    "deviation": deviation
                },
                ..
            ]
        },
        ..
    ]
}
```

    * 404 NOT FOUND

/project/{id}/cluster

- GET: get the clustering of the project
  - Parameters: ID of project
  - Responses:
    * 200 OK:

```
{
    "cluster": [
        {
            "name": name,
            "files": [
                {
                    "id": id,
                    "name": name,
                    "filename": filename,
                    "date": date
                }
                ,
                ..
            ]
        },
        ..
    ]
}
```

  - 404 NOT FOUND

- PUT: set the clustering of the project
  - Parameters: ID of project
  - Body:

```
{
    "cluster": [
        {
            "name": name,
            "files": [
                {
                    "id": id,
                    "name": name,
                    "filename": filename,
                    "date": date
                }
                ,
                ..
            ]
        },
        ..
    ]
}
```

  - Responses:
    - 200 OK
    - 400 BAD REQUEST

/project/{id}/queries

- GET: get suggested queries
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "queries": [
        {
            "type": type,
            "params":[paramA,..],
            "consistency": consistency,
            "simalarity": simalarity,
            "rScore": rscore,
            "fScore":fscore,
            "example": [
                {
                    "index":index,
                    "value": value
                },
                ..
            ]
        },
        ..
    ]
}
```

    - 404 NOT FOUND

/project/{id}/queries/selected

- GET: get selected queries
  - Parameters: ID of project
  - Responses:
    - 200 OK:

```
{
    "queries": [
        {
            "type": type,
            "params":[paramA,..],
            "consistency": consistency,
            "simalarity": simalarity,
            "rScore": rscore,
            "fScore":fscore,
            "example": [
```

```
                        {
                            "index":index,
                            "value": value
                        },
                        ..
                    ]
                },
                ..
            ]
        }
```

  * 404 NOT FOUND

- GET: set selected queries
  - Parameters: ID of project
  - Body:

```
{
    "queries": [
        {
            "type": type,
            "params":[paramA,..],
            "consistency": consistency,
            "simalarity": simalarity,
            "rScore": rscore,
            "fScore":fscore,
            "example": [
                {
                    "index":index,
                    "value": value
                },
                ..
            ]
        },
        ..
    ]
}
```

  - Responses:
    * 200 OK
    * 400 BAD REQUEST

A.4 Screenshots

The visualizations of the different steps in the system are shown in this section.

- Figure A.2: Shows the overview of the created projects. Here, the user gets the possibility to view/delete a project or to create a new one.
- Figure A.3: Shows the first step during the initialization process of a new project. In this step, the user can name the project and upload the necessary files.
- Figure A.4 & A.5: Show the second step during the initialization process of a new project. Here, the user must associate the uploaded files with a date. In addition, it is given the option to remove any files from the data set that are considered as outliers by the system.
- Figure A.6: Shows the third step during the initialization process of a new project. In this step, the user must validate the suggested cluster.
- Figure A.7: Shows the last step during the initialization process of a new project. Here, the user can choose from the various queries proposed by the system, which must be monitored later. For each suggested query one can view the different scores and an example output.
- Figure A.8 & A.9 & A.10: Show all steps to add a new day. The date of the new day must be specified in the first step. Then, one or more files can be uploaded. Finally, the cluster (name) proposed by the system can be changed. In this last step, the user can also indicate if this day should be included in the history of the cluster.
- Figure A.11: Shows the overview of already added days. Per day you can immediately see both the date of the day and the cluster to which the day belongs. The user is given the option to view/delete a day or add a new day.
- Figure A.12: Shows the overview of all queries that were monitored for the selected day. Per query you can immediately see in which category, and how many, outliers occurred.
- Figure A.13: Shows the history of values per outlier by use of a histogram.

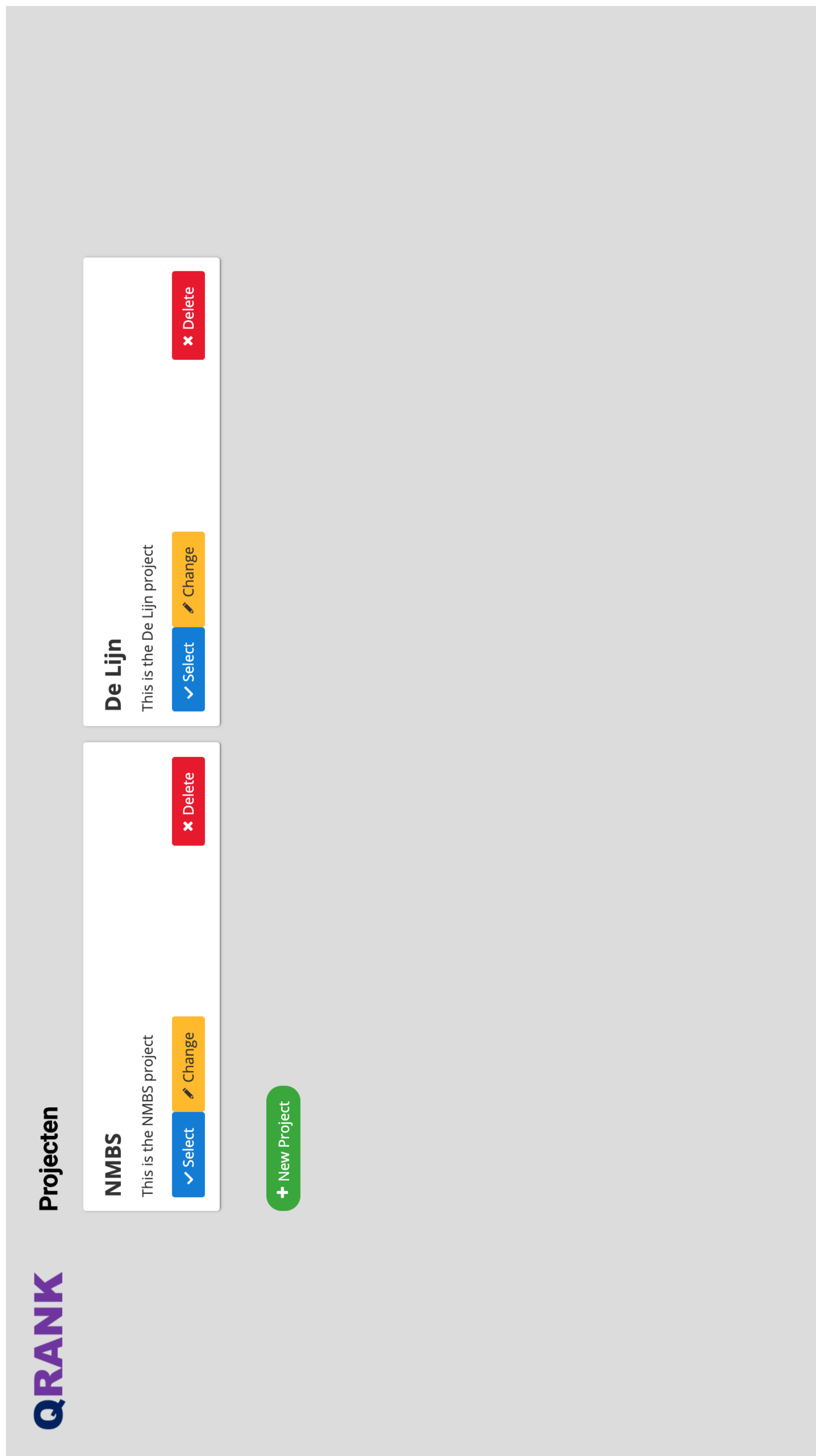


Figure A.2: Overview of the different projects

Figure A.3: Creating a new project: setup project (step 1)

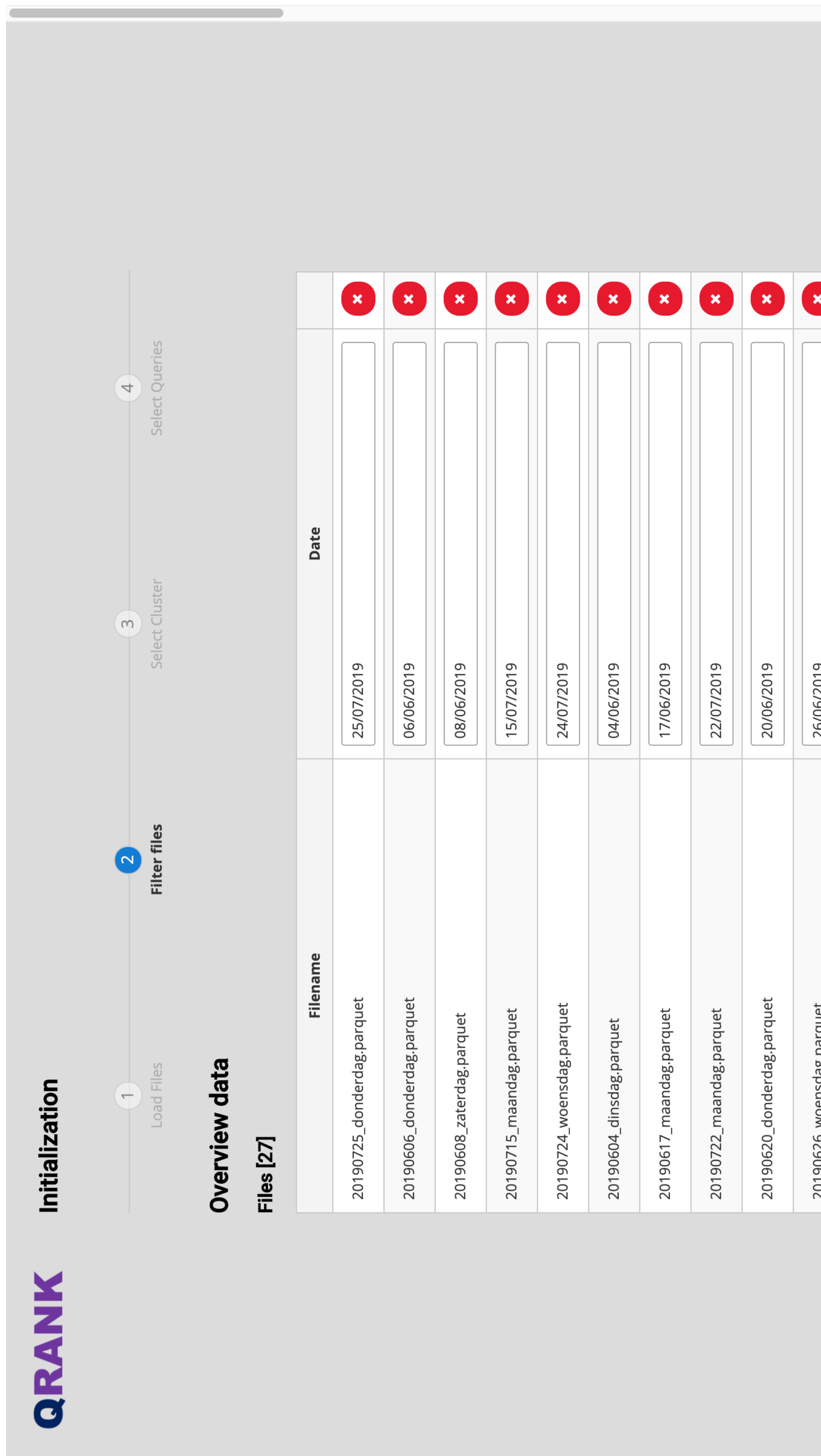


Figure A.4: Creating a new project: naming files (step 2)

20190624_maandag.parquet	24/06/2019	×
20190613_donderdag.parquet	13/06/2019	×
20190708_maandag.parquet	08/07/2019	×

Columns [18]

- Uur_van_geplande_vertrek
- Uur_van_reele_vertrek
- Treinnummer
- Vertraging_bij_aankomst
- Datum_van_geplande_vertrek
- Spoorwegoperatoren
- Richting_van_de_relatie
- Spoorlijn_van_vertrek
- Uur_van_geplande_aankomst
- Naam_van_de_halte
- Date_van_vertrek
- Spoorlijn_van_aankomst
- Datum_van_reele_vertrek
- Uur_van_reele_aankomst
- Vertraging_bij_vertrek
- Datum_van_reele_aankomst
- Relatie
- Datum_van_geplande_aankomst

Outliers

row

Expected value: 52275 - 75681

Filename	Value	
20190608_zaterdag.parquet	39170	×
20190609_zondag.parquet	37292	×

Figure A.5: Creating a new project: filtering files (step 2)

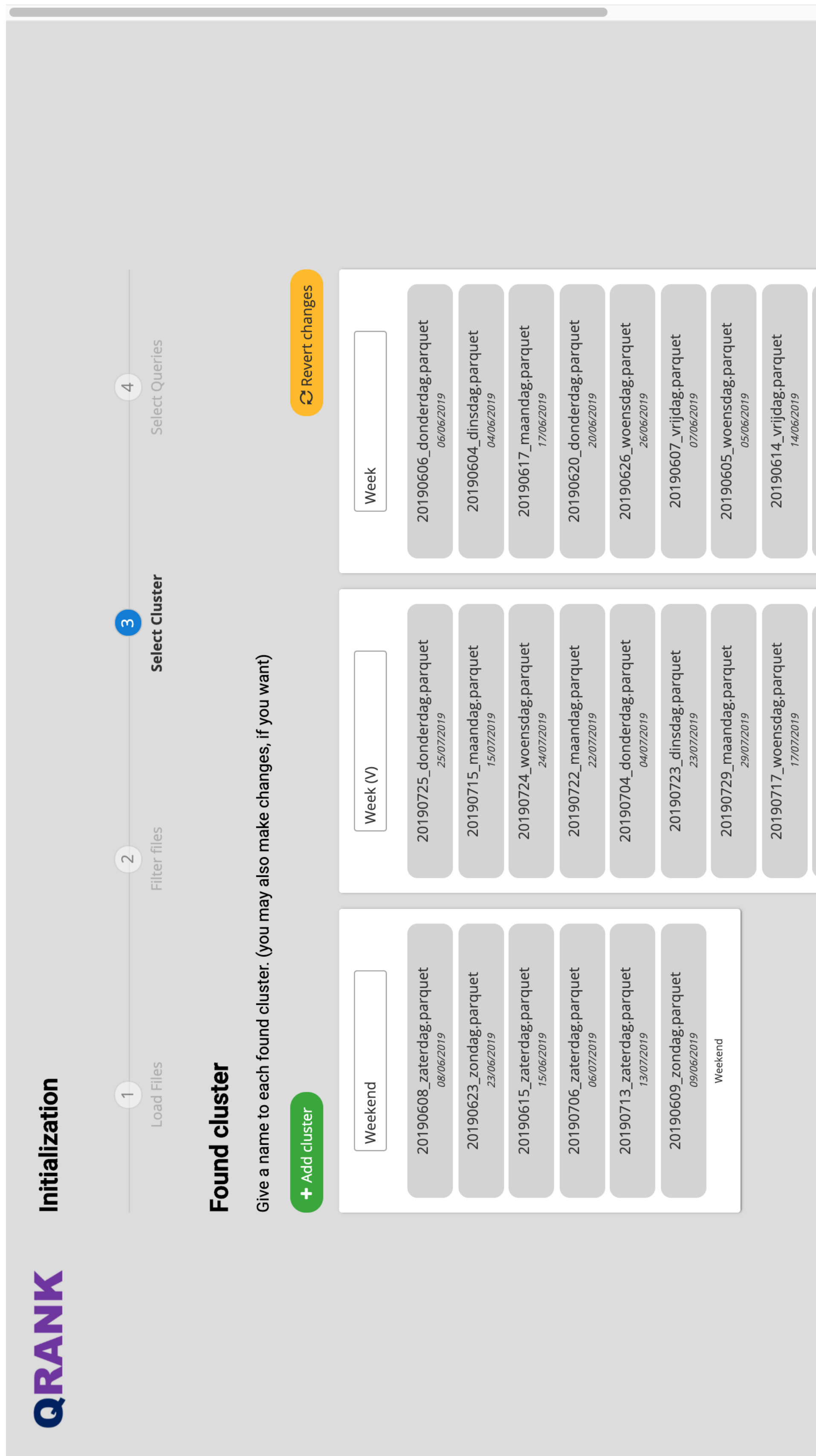


Figure A.6: Creating a new project: clustering files (step 3)

Figure A.7: Creating a new project: selecting queries (step 4)

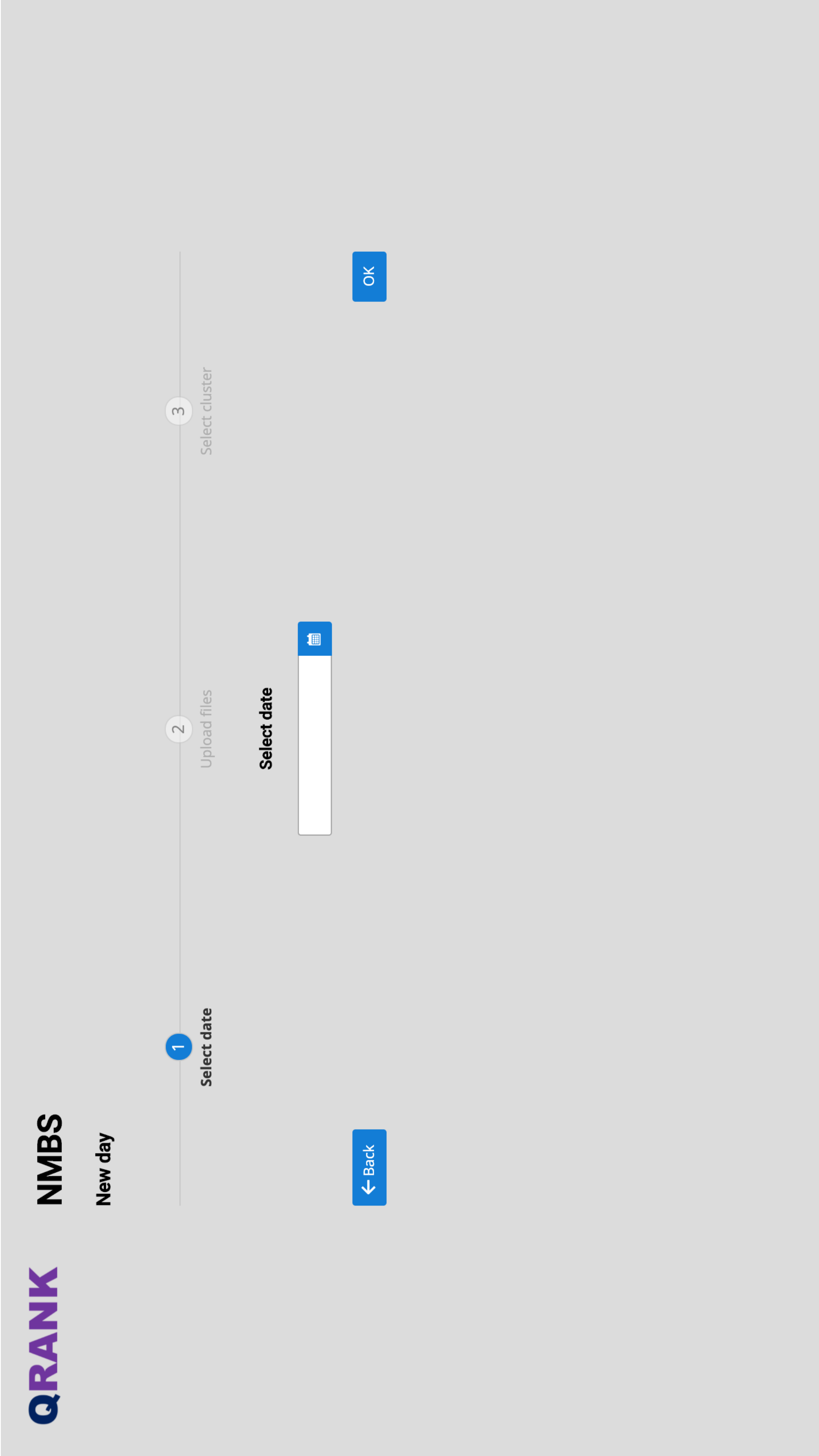


Figure A.8: Adding a new day: setup day (step 1)

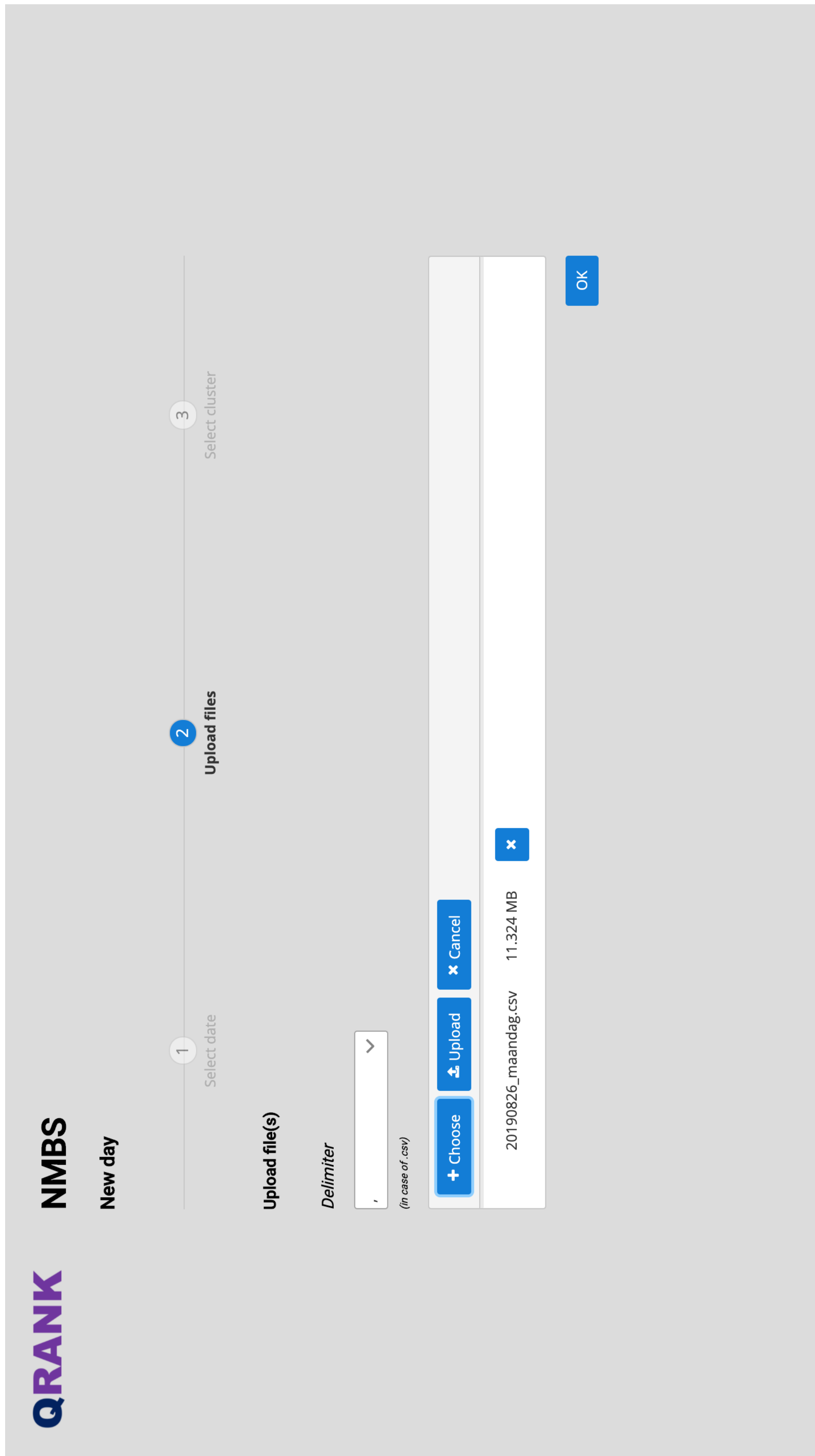


Figure A.9: Adding a new day: loading file(s) (step 2)

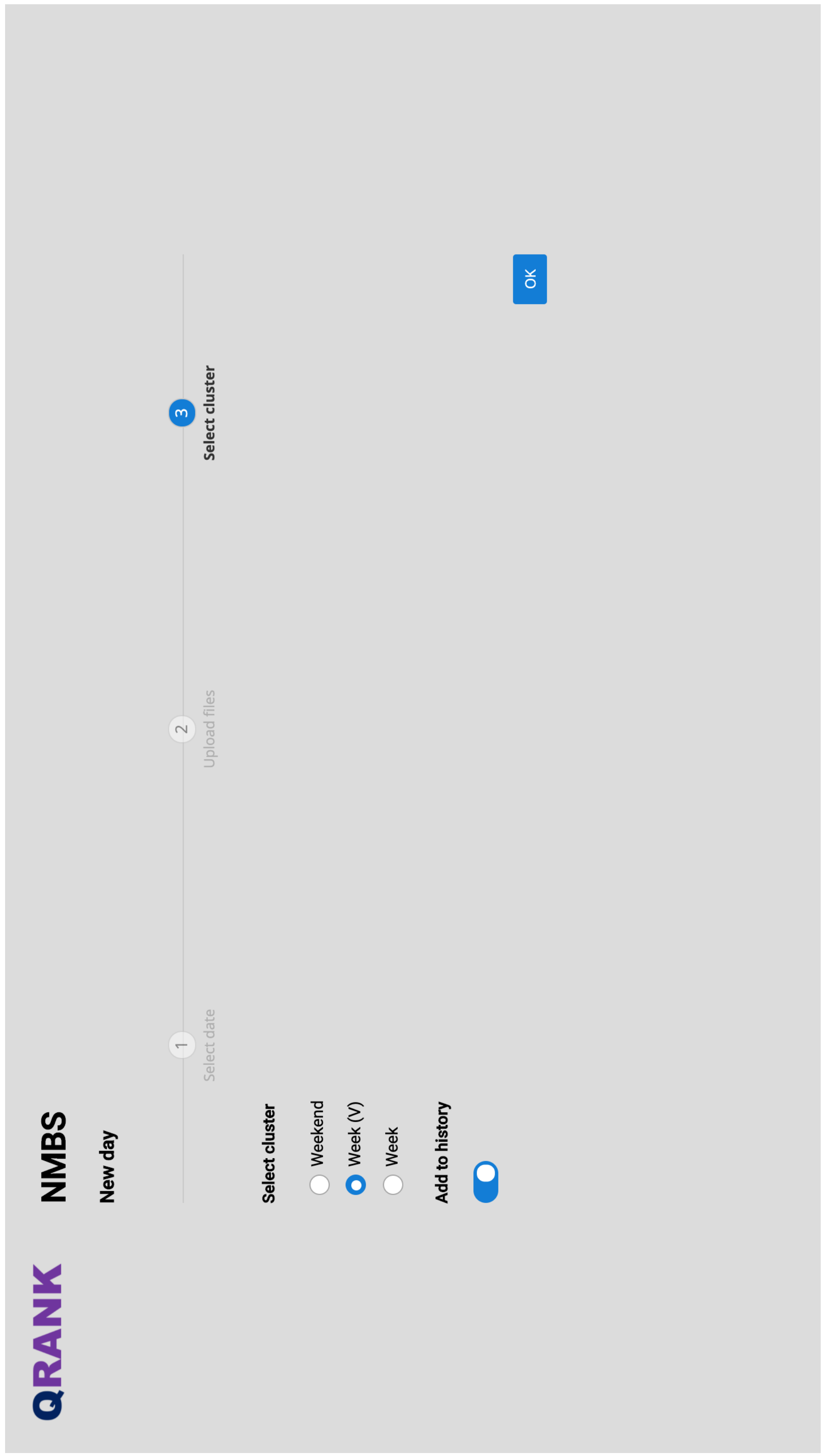


Figure A.10: Adding a new day: selecting cluster (step 3)

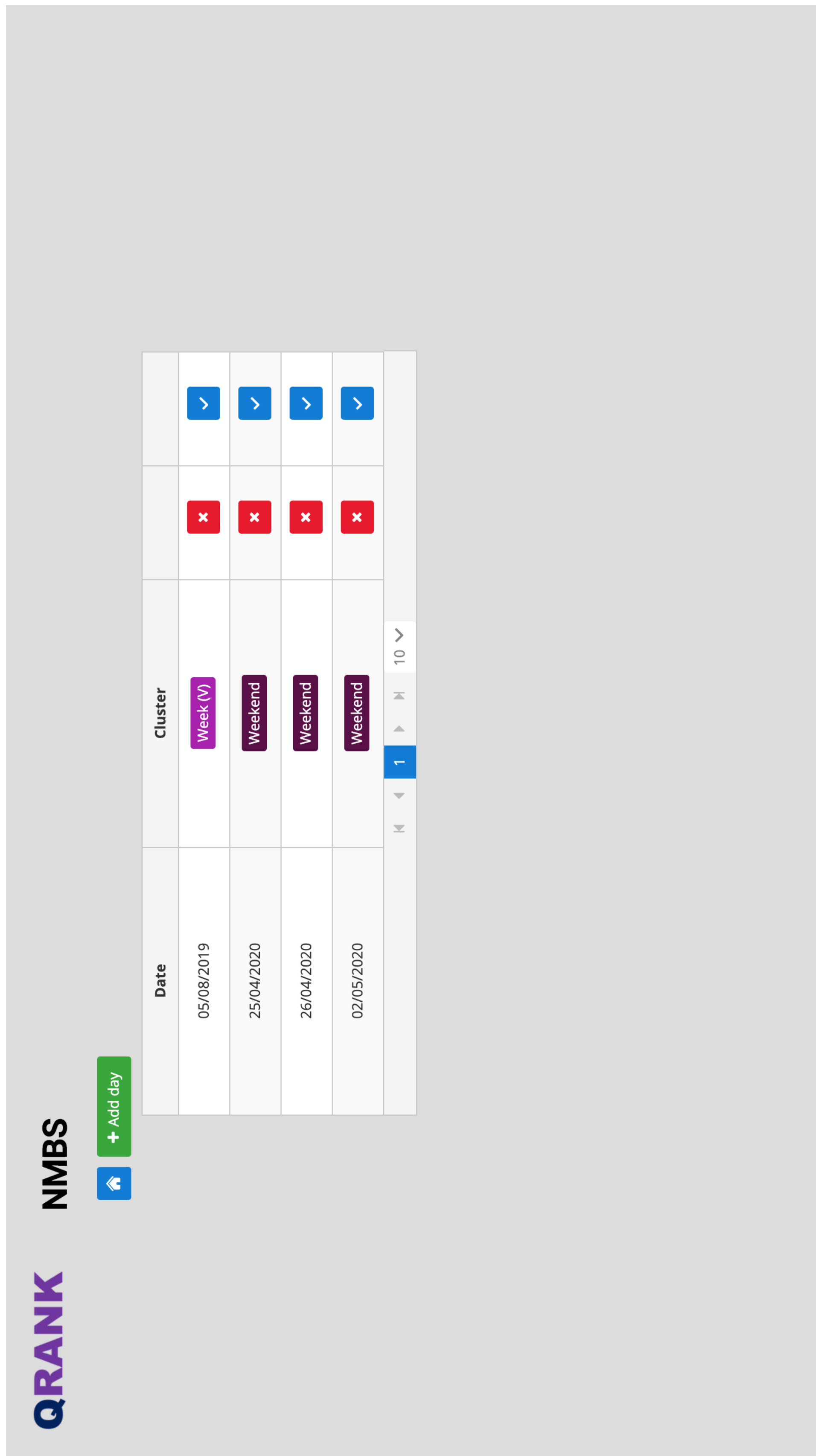


Figure A.11: Overview of the different days of a project

Figure A.12: Overview of the monitored queries (with outliers)

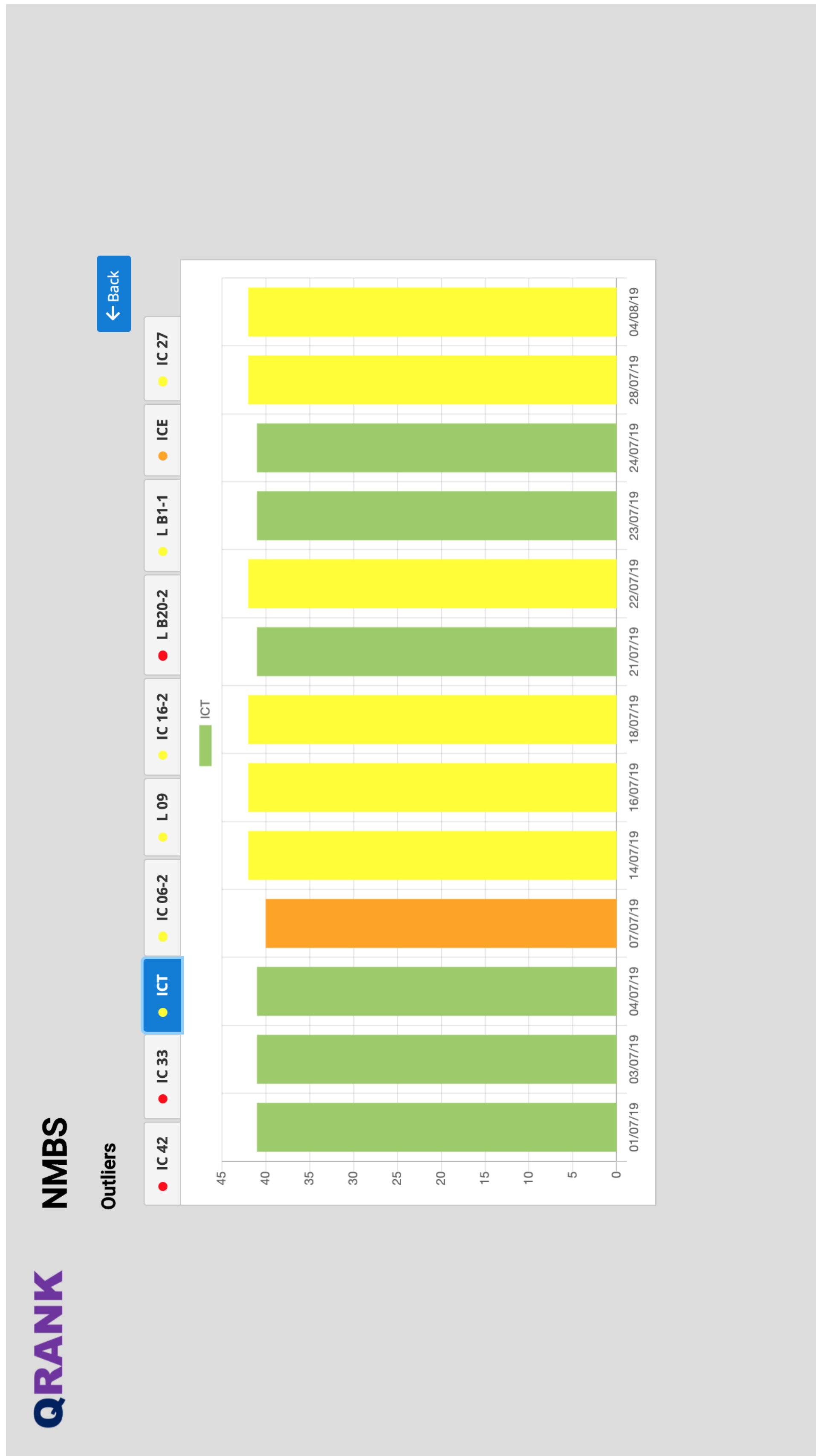


Figure A.13: Overview of the outliers of a monitored query

Appendix B

Nederlandse Samenvatting

B.1 Inleiding

De dag van vandaag produceert elk bedrijf data en dit in allerlei structuren, komende van de verschillende afdelingen (sales, marketing, ...) uit het bedrijf. Tegenwoordig in de zogenaamde 'moderne' bedrijven komt daar ook nog eens *internetdata* bij, zoals onder andere *clickstreams*, die het gedrag van bezoekers op de bedrijfswebpagina bevatten. Al deze data komt samen terecht in *data warehouses* of *data lakes*, die beide gezien kunnen worden als één grote database. Deze data kunnen dan dienen voor analyse of voor het gebruik in *machine learning* (ML), maar daarvoor moet deze data eerst bruikbaar gemaakt worden. Het bruikbaar maken bestaat voornamelijk uit het proper maken van de data. Dit is voor beide toepassingen zeer belangrijk. De data die gebruikt worden voor analyse, beïnvloeden namelijk mogelijk indirect toekomstige bedrijfsbeslissingen. Bij gebruik in ML kan de data echter gebruikt worden in meerdere fases, in alle fases heeft niet propere data een slechte invloed op de accuraatheid van het model. Zo zijn de functies die gevonden bij het trainen van het model, afhankelijk van de training data die gebruikt worden. En in een latere fase is het noodzakelijk dat de data die aan het model gevoerd worden, gelijkaardig zijn aan de gebruikte training data. Om de data proper te maken bestaan er tientallen tools, die opgedeeld kunnen worden in meerdere categorieën, de ene focust zich namelijk meer op een specifiek aspect van data cleaning, terwijl een andere dan weer meer algemene functionaliteiten aanbiedt. Een voorbeeld hiervan is *Trifacta*, een zeer populaire tool met een breed aanbod:

- Uitschieters detecteren in een numerieke kolom.
- Afwijkingen in lengte detecteren in een alfanumerieke kolom.
- Afwijkingen van datatype detecteren in een kolom (bv. een string in een numerieke kolom).
- Een veelvoorkomend patroon herkennen en afwijkingen op dit patroon detecteren.
- ...

Deze tool wordt dan ook door veel bedrijven dagdagelijks gebruikt voor data cleaning. Een onderdeel van data cleaning, is dus de detectie van uitschieters. Van elke kolom wordt dan onder andere het datatype of het aantal (unieke) waarden gecontroleerd. Verder kunnen er afhankelijkheden, in de vorm van regels, tussen kolommen worden opgelegd die in de gehele data set moeten gelden. Echter ontbreekt in deze tools de mogelijkheid om regels omtrent de eigenlijke inhoud van de data set te definiëren. Bijvoorbeeld: '*de trein van Genk naar Blankenberge rijdt elke dag 10 keer*' of '*elke maand wordt er op het bankaccount van Jef, gemiddeld 100 euro uitgegeven aan sport en cultuur*'. De regels kunnen deels uitgedrukt worden aan de hand van *queries*. Je kan namelijk uit de data halen hoeveel keer de trein van Genk naar Blankenberge heeft gereden met volgende *query*:

```
SELECT COUNT(DISTINCT treinnummer)
FROM data
GROUP BY traject
WHERE traject = "Genk naar Blankenberge"
```

Vervolgens heb je nood aan een vooraf bepaalde dienstregeling of historische data om te controleren of het resultaat van deze *query* als 'normaal' beschouwt kan worden. Deze aanpak zorgt voor enkele problemen:

- Er is in de meeste gevallen nood aan een *domeinexpert* om de interessante *query* te selecteren.
- De data sets van tegenwoordig nemen al snel een aanzienlijke grote aan, waardoor het niet haalbaar is om alle interessante *queries* manueel te zoeken en te definiëren.
- Er kunnen tal van soorten *queries* geschreven worden, gebruik makend van verschillende kolommen in data set. Het is niet haalbaar al deze *queries* dagelijks (semi-) automatisch te controleren en te kijken als er iets interessant voorvalt.

Het doel van deze thesis is om te onderzoeken of het mogelijk is om een algoritme te vinden dat, (semi-) automatisch, interessante *queries* kan detecteren. Zodat vervolgens op basis van deze *queries* uitschieters gedetecteerd en gemonitord kunnen worden. Hierbij zullen we ons focussen op één bepaald type data, namelijk *event data*. Dit geeft ons volgende onderzoeksvragen:

- Kunnen we *queries* filteren zonder deze uit te voeren?
- Kunnen we de data (van meerdere dagen) clusteren op basis van *queries*?
- Kunnen we interessante *queries* detecteren?
  - Hoe relevant zijn aanbevolen *queries*? Zijn deze betekenis vol?
  - Hoe uniek zijn deze *queries*? Zijn er *queries* met dezelfde betekenis?
- Kunnen we deze *queries* monitoren?

B.2 Aanpak

In dit deel wordt besproken hoe we anomaliedetectie proberen toe te passen op *event data*. Eerst wordt het begrip *event data* gedefinieerd. Daarna wordt uitgelegd hoe we juist anomalieën in deze data proberen te detecteren. Als laatst kijken we naar de verschillende uitdagingen.

B.2.1 Event data

In dit werk lag de focus op het detecteren van afwijkingen in *event data*. De *event data* die beschouwd werd, bestaat uit events die gelogd worden gedurende een bepaalde tijdsperiode, meestal gebeurt dit per dag. Deze events worden elke dag (meermaals) herhaald. Een goed voorbeeld hiervan is de data van treinen of bussen. Elke dag rijden namelijk (meestal) dezelfde treinen en bussen en stoppen deze aan dezelfde stations en haltes. Deze gebeurtenissen worden tegenwoordig allemaal geregistreerd en vormen de *event data*.

In de eerste fase van het ontwikkelingsproces werd de treindata van de *NMBS* gebruikt als een representatief voorbeeld van *event data*. Deze data werd gebruikt om ideeën te testen en het uiteindelijke systeem uit te werken.

B.2.2 Aggregaties

Gegeven de *event data* van de treinen, zijn we geïnteresseerd in het detecteren van bijvoorbeeld volgende afwijkingen:

- Normaal rijden er op zondag 10 treinen van Genk naar Brugge. Deze zondag reden er slechts 8 treinen.
- Trein 25 stopt normaal 5 keer op maandag. Echter, deze maandag stopte deze 7 keer.
- Gemiddeld rijden er 5000 treinen op een weekdag en 3000 in het weekend. Op deze dag reden er minder dan 2000 treinen.

We kunnen de gegevens van deze voorbeelden uit de *event data* halen door het toepassen van een aggregatie. Een aggregatie kan het toepassen zijn van een operatie op een of meerdere kolommen (b.v. de tel operatie, om het aantal voorkomens van elke entiteit in een gespecifieerde kolom te berekenen). Om vervolgens na te gaan of er sprake is van anomalieën vergelijken we het resultaat van de aggregatie met de historische data. De historische data bestaat steeds uit de resultaten van deze aggregatie toegepast op alle of een vooraf bepaald aantal voorgaande dagen. In de vergelijking wordt gebruikt gemaakt van een gepast uitschietersdetectietechniek om na te gaan of we al dan niet te maken hebben met een afwijkende waarde.

Om de aggregaties toe te passen op de data, wordt er gebruik gemaakt van SQL-*queries*. Om bijvoorbeeld het eerste voorbeeld van daarjuist te controleren, moet het aantal ritten berekent worden die trein 25 gemaakt heeft op maandag. Deze aggregatie ziet er als volgt uit in SQL-formaat:

```
SELECT relatie, COUNT(DISTINCT treinnummer)
FROM event_data_dinsdag
GROUP BY relatie
```

B.2.3 Uitdagingen

Om anomalie detectie dag per dag te kunnen uitvoeren, moet er eerst bepaald worden welke zaken juist opgevolgd en vervolgens gemonitord moeten worden. Elke dag, elke mogelijk aggregatie controleren is niet haalbaar en niet interessant. Het eerste omdat er een enorm aantal aan mogelijke aggregaties zijn: elke aggregatie bestaat immers uit een operator die toegepast wordt op één of meerdere kolommen. Het aantal mogelijke aggregaties loopt dus al snel op als het aantal kolommen in de data set toeneemt. Er is dus nood aan een manier om oninteressante aggregaties vooraf uit te sluiten zonder de resultaten ervan te bereken. Het tweede omdat het mogelijk is dat aggregaties nooit oftewel altijd uitschieters geven. Dus moet er opzoek gegaan worden naar een manier om enkel de interessante aggregaties op te volgen. Aangezien "interessantheid" iets subjectief is, bestaat er geen algemene definitie van een interessante aggregatie. In dit werk, is een interessante aggregatie als volgt gedefinieerd: *Een aggregatie wordt als interessant beschouwd als deze bijna altijd hetzelfde resultaat geeft, maar soms niet.* Met andere woorden, er moet voor iedere aggregatie gecontroleerd worden als deze consistent is. Om dit te kunnen doen, is er nood aan historische data (die later ook kan dienen om eventuele afwijkingen op te sporen) en een manier om de consistentie van een aggregatie te bepalen. Stel dat we een manier hebben om de consistentie van een aggregatie te bereken, is er nog een bijkomend probleem: we kunnen niet zomaar elke dag als gelijke beschouwen. Als we naar de treindata kijken, zien we dat niet elke dag dezelfde hoeveelheid treinen ingepland staan. Zo rijden bijvoorbeeld in het weekend veel minder treinen als door de week. Indien we dus wel alle dagen als gelijke zouden beschouwen, zouden we potentiele interessante aggregaties, wegens inconsistentie, mogelijk onterecht uitsluiten.

Dus er is nood aan:

- Filteren van oninteressante aggregaties.
- Clusteren van dagen op basis van *event data*.
- Selecteren van interessante aggregaties, m.a.w. het berekenen van de consistentie van de resultaten een aggregatie.

B.2.4 Query

Om de complexiteit van dit werk enigszins te verminderen, hebben we ons gefocust op een bepaald type aggregatie (vanaf nu *query* genoemd). Hiervoor hebben we gekeken naar wat voor soort informatie in de *event data* van de treinen interessant is vanuit het oogpunt van een datawetenschapper en hoe de *queries* naar deze informatie eruitzien. Hieruit bleek dat een bepaalde soort vorm van *query* vaak nodig is:

```
SELECT kolomA, COUNT(DISTINCT kolomB)
FROM event_data
GROUP BY kolomA
```

Voor de rest van dit werk wordt dit type *query* de *GBC-query*, GroupBy-Count-*query* genoemd. In een later stadium zijn de soorten *queries* uitgebreid naar *queries* van een vergelijkbaar formaat.

B.3 Algoritmen

Dit deel behandelt het proces dat de interessante *queries* probeert te achterhalen. Dit proces maakt gebruik van verschillende technieken, waarvan de algemene aanpak in de volgende secties besproken wordt.

B.3.1 Filteren

Het eerste wat we willen doen in de zoektocht naar interessante aggregaties, is het vooraf verminderen van aantal mogelijke aggregaties die geëvalueerd moeten worden (omdat zoals eerder aangehaald, het er al snel te veel zijn). Een eerste filtering die toegepast wordt, gebeurt op basis van kolomstatistieken. Hierbij kijken we naar elke kolom apart en gegeven we deze een score voor elk mogelijke positie in de aggregatie. Dus voor onze *GBC-query* kan het zijn, dat het niet interessant is om op een bepaalde kolom een $GROUPBY$-operatie uit te voeren, omdat de kolom enkel unieke waarde bevat en de operatie dus geen invloed heeft. Deze kolom is mogelijks dan wel weer interessant om de $COUNT$-operatie op toe te passen. Zoals men kan zien, wordt in deze fase geen exacte wetenschap toegepast en zullen we *queries* intuïtief uitsluiten. Of een *query* al dan niet wordt uitgesloten, is gebaseerd op de volgende elementen van de kolom: data type, entropie, aantal verschillende waarden, aantal nul-waarden en functionele afhankelijkheden van en tussen kolommen. Deze elementen bepalen, samen met de positie van de kolom in de *query*, de score van de kolom. Nadat voor elke kolom zijn scores bekend zijn, kan de score van de *query* worden berekend door de scores van de kolommen te vermenigvuldigen. De vermenigvuldiging wordt toegepast zodat een lage score van een kolom, een grote invloed heeft op de uiteindelijke score van de *query*. Op basis van een vooraf gedefinieerde drempel, kunnen we de *queries* wegfilteren waarvan de score onder deze drempel valt. In deze fase is het belangrijkste doel om *queries* uit te sluiten die zeker niet interessant zijn, dus het kan zijn dat we alsnog enkele *queries* opnemen die later niet interessant blijken te zijn.

B.3.2 Clusteren

Nadat er voldoende *queries* uitgesloten werden in vorige stap, is het nu wel haalbaar om de overgebleven *queries* uit te voeren op de data. Op basis van deze resultaten kan er dan bepaald worden of een *query* al dan niet consistent is. Aangezien er bij het controleren van consistentie geweten moet zijn welke dagen met elkaar vergeleken moeten worden, moeten de databestanden van de verschillende dagen eerste geclusterd worden, als de clusters niet gekend zijn. Om de verschillende dagen op te delen in clusters moet er eerst bepaald worden op basis van welke 'deeldata' dit gebeurt. Dit aangezien, het niet haalbaar is om de volledige data sets steeds met elkaar te vergelijken. Er is vervolgens gekozen om de resultaten van de aggregaties zelf, te beschouwen als 'deeldata'. Dus, iedere *query* wordt uitgevoerd om de reeks dagen die geclusterd moeten worden en de clusters worden berekend op basis van de resultaten van elke *query*. Dit geeft vervolgens een goed beeld op basis van welke data, welke clusters berekend kunnen worden. Dus in deze fase worden de clusters van iedere aggregatie berekent om:

- De consistentie te berekenen.
- Om (mogelijks) nog aggregaties uit te sluiten.
  - Geval 1: Indien geen clusters gevonden kunnen worden, betekent dit dat de resultaten steeds verschillend zijn.
  - Geval 2: Indien één cluster gevonden wordt, betekent dit dat de resultaten steeds gelijk zijn.

  Beide gevallen zijn wat we zoeken in de zoektocht naar interessante aggregaties.

B.3.3 Consistentie

Nu geweten is welke bestanden tot welke cluster behoren en dus welke bestanden met elkaar vergeleken moeten worden, kan de consistentie van de *query* berekend worden. Er zijn verschillende manieren mogelijk om de consistentie van elke *query* te bepalen. Op het niveau van de clusters moet er bepaald worden als de consistentie voor één of alle clusters berekend wordt. In het geval van één cluster, wordt deze waarde genomen wordt als consistentie voor de gehele clustering. In het geval van alle clusters, moet er vervolgens bepaald worden als het gemiddelde, minimum, maximum of een andere aggregatie van alle waarden gekozen wordt als de uiteindelijke consistentie, die geldt voor de volledige clustering van de *query*. Als er naar de resultaten binnen een cluster gekeken wordt, dan hebben we in het geval van de *GBC-query*, te maken met *key-value* paren. Dit zorgt opnieuw voor enkele keuzes die gemaakt moeten worden:

- Welke maatstaf wordt gebruikt om de consistentie van waarden te berekenen?
- Hoe wordt consistentie van de key bepaalt?
- Wanneer is een key inconsistent?
- Hoe bereken je de uiteindelijke consistentie van de cluster?

Op al deze vragen zijn verschillende antwoorden mogelijk en het is moeilijk te bepalen welke nu meer juist is dan de andere. Belangrijk is wel dat er rekening gehouden wordt met alle factoren. Bijvoorbeeld, als je enkel naar de waarden gaat kijken van de *keys* om de consistentie te bereken, dan zou je wel eens vertekende resultaten kunnen krijgen. Zo kan het zijn dat *keys* zelf niet consistent zijn en dat ze bijvoorbeeld, maar eenmaal op de tien keer voorkomen. Indien er dan enkel naar die ene waarde gekeken wordt, dan kan de *key* op basis van deze waarde niet aangeduid worden als inconsistent. Dus het bepalen of de resultaten van een *query* consistent zijn in de loop van tijd is niet zo voor de hand liggend.

B.4 Conclusie

In dit werk hebben we getracht een systeem te ontwikkelen die in staat is om interessante aggregaties te detecteren in *event data* en de uitschieters hiervan vervolgens te monitoren. Huidige data cleaning tools missen namelijk de functionaliteit om regels in de vorm van aggregaties te schrijven en/of te detecteren. Op basis van deze aggregaties zouden er echter meer inhoudelijke afwijkingen gedetecteerd kunnen worden in de data. De aggregaties kunnen aan de hand van *queries* uitgedrukt worden. Om vervolgens te bepalen of het resultaat van de *query* uitschieters bevat, is er nood aan historische data. Deze historische data bestaat dan uit eerdere resultaten van deze *query* uitgevoerd op de data.

Het uiteindelijke ontwikkelde systeem bestaat uit 2 grote delen: initialisatie en monitoring. In het eerste deel wordt er gezocht naar de interessante *queries*, dit proces bestaat voornamelijk uit het toepassen van 3 algoritmes: filteren, clusteren en consistentie berekening. Op het eind van dit proces is geweten wat de clusters van de data set zijn en welke *queries* gemonitord moeten worden. Het tweede deel bestaat uit het monitoren van de *queries* die geselecteerd werden. Eerst wordt een nieuwe dag toegevoegd. Hierbij wordt onder andere bepaald tot welke cluster deze nieuwe dag behoort. Vervolgens worden uitschieters gedetecteerd op basis van de historische data van de overeenkomstige cluster.

In de eerste stap van de initialisatie worden al de mogelijke *queries* gefilterd, dit omdat het onhaalbaar en oninteressant is om ze allemaal te testen. Door het toepassen van deze filtering, kan er gemiddeld 80% van de *queries* gemarkeerd worden als oninteressant. Deze worden dan ook niet meer beschouwd in de volgende stappen. Deze stap is cruciaal, zeker naarmate de data sets in grote toenemen, om de volgende stappen in aanvaardbare tijd af te werken. Ook al werden in de door ons uitgevoerde testen, geen interessante *queries* weg gefilterd in deze stap, nemen we aan dat er en kleine kans bestaat dat dit toch gebeurt. Het zou dan eventueel mogelijk zijn om na de laatste stap toch enkele van deze weg gefilterde *queries* te testen. Stel dat het onhaalbaar om ze allemaal te testen, zien we twee mogelijk opties. De eerste optie is om de drempel te verlagen, zodanig dat een meer *queries* toch deze drempel halen. De andere optie is, om een willekeurige sample van *queries* te selecteren uit deze set. Beide opties kunnen eventueel herhaald worden, moesten er toch nog interessante *queries* gevonden worden.

In de volgende stap gaat men opzoek naar de clusters van de data set, waarbij gebruik gemaakt wordt van de overgebleven *queries*. Voor elke *query* worden de clusters berekent, de meest voorkomende clustering wordt vervolgens voorgesteld aan de gebruiker. De resultaten hiervan waren zeer goed. Veel *queries* zijn namelijk in staat om dezelfde en verwachte clusters te vinden. Naarmate we de drempel in de filteringstap verhoogde steeg zelfs het percentage dat dit deed. Als dit een indicatie is dat er meer interessante *queries* overhouden laten we in het midden, wel nemen we aan de er meer *queries* overblijven die betekenisvol zijn. Verder konden we zien dat deze techniek ook zijn limieten heeft. Zo kan er bijvoorbeeld geen onderscheid gemaakt worden tussen de verschillende weekdagen in geval van de treindagen. De voornaamste reden hiervoor zijn volgens ons gelijkheid en inconsistentie. De verschillende weekdagen verschillen namelijk zeer weinig van elkaar, wat maakt dat er een moeilijk een onderscheid gemaakt kan worden. Daarnaast hebben we gemerkt dat er in de data set weinig consistentie aanwezig is, wat maakt dat bijvoorbeeld twee dinsdagen al veel van elkaar kunnen verschillen. Indien de gevonden clusters niet voldoende gedetailleerd zijn of zelfs helemaal verkeerd zijn volgens de gebruiker, heeft deze de mogelijkheid om de clusters te wijzigen. Indien de clusters door de gebruiker gewijzigd worden, is de kans groot dat er dan geen goede *query* bestaat die deze clusters ook vindt. Dit kan er vervolgens voor zorgen, dat bij het toevoegen van een nieuwe dag, niet altijd de juiste cluster wordt voorgesteld door het systeem. Dit kan mogelijk verbeterd worden door de centroids die hiervoor gebruikt worden, na het toevoegen van een dag, steeds opnieuw te berekenen. Hierdoor

kan het systeem na verloop van tijd waarschijnlijk beter inschatten om welke cluster het gaat.

In de laatste stap wordt de consistentie berekent, deze speelt een cruciale rol in het bepalen of een *query* hoog aanbevolen wordt of niet. De uiteindelijke gebruikte formule hiervoor is het resultaat van enkele iteraties en houdt rekening met alle nodige factoren. De berekening gebeurt voor alle *queries* door hetzelfde algoritme, dus ook al zou hier eventueel een foutmarge inzitten, dan is deze voor alle *queries* gelijk.

Nadat de consistentie berekend is, worden de scores van alle *queries* weergegeven die na de filtering in de eerste stap overbleven, dus niet alleen dit interessante. Dit was voornamelijk om een beter beeld kregen welke *queries* hoog aanbevolen werden en welke niet, en hoeveel verschil er is in scores tussen de verschillende *queries*. Voor de data van *NMBS*, werden zeer goede *queries* hoog aanbevolen met een duidelijke betekenis. Dit is ergens niet onlogisch, aangezien het systeem gebouwd werd op basis van deze data. De aanbevolen *queries* van *DeLijn*, waren door de cryptische vorm van de data, niet altijd even duidelijk. Een domeinexpert is vereist om een beter idee te krijgen wat elke *query* juist inhoud en als ze betekenisvol zijn.

Zeer opvallend bij de testen met *DeLijn* was dat veel meer *queries* een hoge scoren haalde in vergelijking met de data van *NMBS*. De voornaamste reden hiervoor is dat deze *queries* allemaal een hoge consistentie behalen en bijgevolg ook hoog scoren. De hoge consistentie is op zijn beurt te wijten aan het feit dat de events bij *DeLijn* meer consistent zijn. Verder viel ook op dat bepaalde *queries* gelijkaardig zijn of zelfs helemaal hetzelfde resultaat geven. Dit zal er automatisch voor zorgen dat wanneer zo *queries* gemonitord worden, deze altijd op hetzelfde moment uitschieters zullen geven. Indien er slechts een beperkt aantal *queries* worden aanbevolen, kan dit nog manueel nagegaan worden. Vanaf dat dit er wat meer zijn, zoals het geval was bij de data set van *DeLijn*, is dit manueel ondoenbaar. Het zou dus goed zijn als gelijkaardige *queries* door het systeem aangegeven en/of gegroepeerd kunnen worden.

Daarnaast, als dit systeem effectief gebruikt zou worden, dan is er nood aan opnieuw een drempel die de *queries* moeten halen. Zo zagen we dat *queries* met een te lage consistentie best niet geselecteerd worden. De voornaamste reden is dat deze bij het monitoren dagelijks te veel uitschieters zullen geven, waardoor het moeilijk wordt om het overzicht te bewaren. Dus een drempel op consistentie lijkt de beste optie, de andere factoren spelen namelijk geen directe rol in het aantal uitschieters. De waarde van deze drempel is moeilijk om op voorhand vast te leggen, aangezien we aannemen dat deze afhankelijk is van de gebruikte data set. Voor kleinere data sets, mag deze drempel gerust wat lager liggen. Zolang het maar overzichtelijk blijft om alle uitschieters weer te geven.

Om het allemaal samen te vatten, het ontwikkelde systeem is in staat om interessante vragen te vinden door gebruik te maken van het geïmplementeerde proces en heeft zeker veel potentie. Maar er zijn duidelijk een aantal verbeterpunten, die we in dit deel hebben benoemd.

B.5 Vervolg

In de conclusie, maar ook doorheen de hele thesis, werden al een heleboel verbeterpunten aangehaald. Daarnaast, zien we nog aantal meer algemenere zaken die kunnen gebeuren in verder onderzoek om interessante *queries* uit data af te leiden. Zo moet het systeem eerst en vooral getest worden met andere data sets, zodat duidelijker wordt hoe robuust het systeem is en welke aanpassingen moet gebeuren om meer soorten data te ondersteunen. Daarnaast moet er gekeken worden welke andere soorten *queries* interessant zijn om te ondersteunen. Bijvoorbeeld *queries*, die data nog eens verder opsplitsen op basis van een element uit een extra kolom. Hierdoor stijgt mogelijks de consistentie, waardoor uiteindelijk deze data toch gemonitord kan worden. Als men meer *queries* wil ondersteunen, wordt de filtering nog cruciale in het proces. Dus op

vlak van filtering moet verder onderzocht worden welke andere factoren bepalen of een bepaalde kolom geschikt is om te gebruiken in een aggregatie. Een andere richting die men kan uitgaan als men meer soorten *queries* wil ondersteunen, is verbeteren op vlak van snelheid in de andere stappen van het proces. Zo kunnen er in de andere fases van het proces, parallellisaties toegepast worden. Zoals bijvoorbeeld, bij het zoeken naar de clusters of het berekenen van de consistentie van de *queries*.

Bibliography

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004, August 2016.

[2] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. Data profiling. *Synthesis Lectures on Data Management*, 10(4):1–154, 2018.

[3] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA, 2006.

[4] Zhamak Dehghani. How to move beyond a monolithic data lake to a distributed data mesh. Blog published on website martinfowler.com. https://martinfowler.com/articles/data-monolith-to-mesh.html (Accessed: 2020-01).

[5] Great expectations. Always know what to expect from your data. https://greatexpectations.io/. Accessed: 2020-03.

[6] Google. Angular: One framework. mobile & desktop. https://angular.io/. Accessed: 2020-04.

[7] Hasso-Plattner-Institut. Metanome: Data profiling. https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html. Accessed: 2020-03.

[8] Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Databases*, 5(4):281–393, October 2015.

[9] Ihab F. Ilyas and Xu Chu. *Data Cleaning*. Association for Computing Machinery, New York, NY, USA, 2019.

[10] Amazon Web Services Labs. Deequ: Unit tests for data. https://github.com/awslabs/deequ. Accessed: 2019-10.

[11] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[12] Pallets. Flask: Web development, one drop at a time. https://flask.palletsprojects.com/en/1.1.x/. Accessed: 2020-04.

[13] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with metanome. *Proc. VLDB Endow.*, 8(12):1860–1863, August 2015.

[14] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Rec.*, 47(2):17–28, December 2018.

[15] PrimeTek. Primeng: Angular ui component library. https://www.primefaces.org/primeng/. Accessed: 2020-04.

[16] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2011.

[17] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, page 269–278, New York, NY, USA, 2002. Association for Computing Machinery.

[18] Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat, Phillipp Schmidt, Stephan Seufert, Pierre Brunelle, and Andrey Taptunov. Unit testing data with deequ. pages 1993–1996, 06 2019.

[19] Tableau Software. Business intelligence and analytics software. https://www.tableau.com/. Accessed: 2020-02.

[20] Tamr. Enterprise data mastering at scale. https://www.tamr.com/. Accessed: 2020-02.

[21] Metaweb Technologies. Openrefine: A free, open source, powerful tool for working with messy data. https://openrefine.org/. Accessed: 2020-02.

[22] Trifacta. Data wrangling software and tools. https://www.trifacta.com/. Accessed: 2020-02.