Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2017 Open / ~~Confidential~~
Author: Julian Minde	 (signature author)
Instructor: Prof. Chunming Rong Supervisor: Prof. Chunming Rong Additional contacts: Trond Linjordet, Aryan TaheriMonfared	
Title of Master's Thesis: Automatic collection and storage of smart city data with semantic data model discovery and sample data analysis. Norwegian title: Automatisk innsamling og lagring av smartbydata med semantisk datamodelloppdagelse og prøvedataanalyse.	
ECTS: 30 sp	
Subject headings: Smart city data, big data, data collection, Internet of things, Elasticsearch, Logstash.	Pages: 118 Stavanger, 15.06.2017

Automatic collection and storage of smart city data with semantic data model discovery and sample data analysis

Julian Minde

June 15, 2017

Contents

List of Figures

List of Tables

List of Algorithms

List of Examples

Abstract

Collecting and storing smart city data is a task that requires thorough data exploration, configuring and testing to be of value. Configuring a data collection pipeline for data from a new data provider needs to take into account what the various fields represent, what parts of the data is of interest, which data fields should be stored, and more. In some cases the data follows a predefined, and known schema, in other cases the data may be undocumented.

This thesis presents a framework and a software for automating the process of collecting and storing smart city data, and other event based data sets. The problem, and solution is illustrated in this thesis by a use case where the task consist of storing public transportation data in a structured way in a storage system that can handle big data.

Chapter 1

Introduction

With the recent advances in computer technology, generating data is easier than ever before. More data is generated and at a faster pace. For example about 72 hours of video was uploaded to YouTube every minute on average in 2014 [1].

Traditional computer systems are not well suited to such large amounts of data, nor the rapid speed at which they are generated. This gave rise to the term big data, defined by Apache Hadoop as, "datasets which could not be captured, managed, and processed by general computers within an acceptable scope" [1]. In 2011, the International Data Corporation (IDC) redefined big data as "describing a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis" [2]. This definition implicitly refers to the characteristics of big data commonly known as "the four Vs, i.e. Volume (great volume), Variety (various modalities), Velocity (rapid generation) and Value (huge value but very low density)" [1].

Through analysis, statistics, and machine learning, new and previously hidden insights can be found in big data sets. One of the key challenges in big data applications today is the data representation. Different datasets have different structures, semantics, organisation, granularity and accessibility [1]. This presents a problem when combining data from different sources for analysis.

Data generated from actual events are normally collected where the event happens, encoded, and sent over the internet to a system that processes and stores the data. A data collection framework is a software system that is responsible for acquiring and storing data in an efficient and reliable manner. The system must be able to store data arriving at a high pace, while making sure all data is stored correctly. It must also make the data easy to work with for data scientists, developers and other users of the data. Such a framework can handle data of varying structure from multiple different data sources in the same running instance of the system. The part of the instantiated system that handles one data source with one structure is referred to as a data collection pipeline.

The data, in this context, is represented as a set of data fields, i.e. combinations of field names and field values. One such set of data fields is in this thesis referred to as a data point, thus a data set is a collection of many data points. If a data set follows a given schema, most data points will have the same field names, and their values will be of the same types. However, if

there is no value for a field in a data point, that field can either have the `null` value, or the field might be missing all together.

The part of the data collection framework that stores the data is in most cases some type of database. Traditionally, structured data would be stored in a relational database, like MySQL or PostgreSQL. In the case of storing structured, big data, the storage system must be able to scale. Relational databases are not built to scale, and are therefore unfit for the job [3]. However, this has caused a rapid development of a category of databases referred to as NoSQL databases. NoSQL, as in "Not Only SQL", refers to a group of non-relational databases that don't primarily use tables, and generally not SQL for data manipulation [4]. NoSQL databases are "becoming the standard to cope with big data problems" [3]. This is in part "due to certain essential characteristics including being schema free, supporting easy replication, possessing a simple Application Programming Interface (API), guarantee eventual consistency and they support a huge amount of data"[3]. NoSQL databases are said to not primarily use tables, instead of rows of data, as one would have in a relational database, one often has data objects in a NoSQL database. A relational database is constrained by the schema of the tables, i.e. to store a row of data to a table, it has to conform to the format of the table it is stored to. NoSQL databases are generally "schema-free" meaning the documents stored in the database need not follow the same, or any, schema. Partly because of their schema free operation, NoSQL databases are relatively well suited for fault-tolerant distributed operation. This again enables NoSQL databases to, generally speaking, store big amounts of data.

Configuring a pipeline for data from a new data provider can be a complex task. It needs to take into account what the various data fields represent, what parts of the data is of interest, whether there is information that should not be stored, and more. In many cases, the data will follow a set schema that defines names and types of the various data fields. Data that is just dumped to file with no schematic information can be difficult to use later, unless the meaning of the data fields is documented adequately.

The term metadata is defined as "data that provides information about other data" [5]. In the context of this thesis, metadata refers to information about the data that is not represented by the values of the data. For example, the units of a value. Metadata can be implied as part of the field name, it can be stored in a separate documentation file, or it can exist only in the mind of the domain expert.

Semantics is the study of meaning in languages. The semantic information about the data is information that clarifies the meaning behind the data, and their relationship to the real-world entities that the data represent. A semantic data model is a data model that includes and focuses on the semantics of the data. Both metadata and semantic information also contribute to interpret the data and extract the value of the data.

Raw data from sensors will sometimes include redundant and useless data that do not need to be stored [1]. Therefore, the data will typically be transformed at the data collection pipeline, to fit the storage system. The data is transformed in order to follow the same encoding as other data in the same system, to make the data easy to retrieve, to remove unwanted data and in some cases to make the data more structurally similar to other semantically similar data in the system. Through this transformation and cleaning of the data, the data representation is altered. This altering of data representation must be done with a strong focus on making the data more meaningful for computer analysis and user interpretation. Improper data representation can reduce the value of the data [1]. In this transformation of the data, there is a risk of losing valuable information and at the same time storing data that is of less value. An

example of data with no value might be identifier numbers referring to the data provider's internal systems that are inaccessible, or model numbers of sensors that could be inferred from table or index names instead. If the data provider's internal systems are accessible on the other hand, that identifier field could be valuable and it should possibly be stored. The task of configuring a data collection and storage pipeline is a task of fitting data into a useful structure.

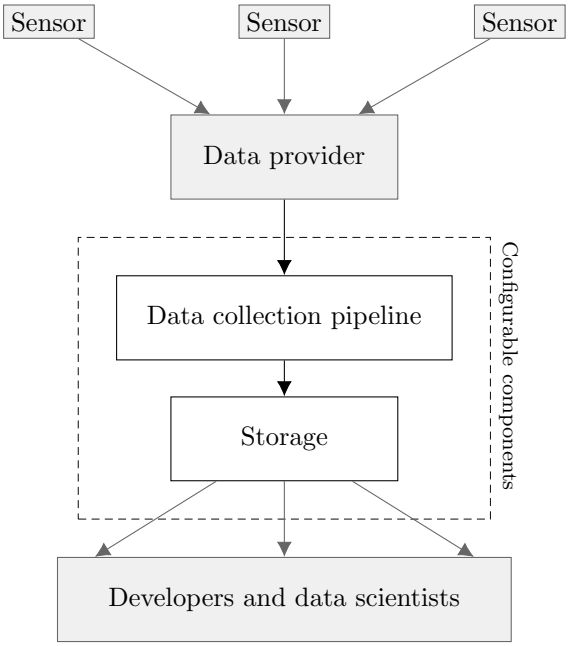

Figure 1.1: Overview of the data collection framework that this thesis will work with. The data provider collects data from various sensors or other event sources and sends it to the data collection and storage pipeline which transforms the data and store it. Developers and data scientists retrieve data from the storage.

1.1 Motivation

More data is being generated and collected now than ever before, and from that data new insights can be gathered. According to a forecast done by Gartner, the quantity of Internet of Things (IoT) sensors will reach 26 billion by the year 2020, thus making IoT data the most important part of big data [6]. Collecting and storing data can be quick and simple. However, structuring the data in a way that achieves a good data representation is a more complex task.

Currently, in order to ensure good data representation, the data and its structure must be analysed and fully understood by the person configuring the data collection pipeline. Valuable data must be interpreted and stored using data types that reflect the semantics of the data as well as possible. And data that is of less value to the objective of the collection can be

removed to save storage space and make the data easier to work with. In some cases, a data point can benefit from being split into smaller data points, e.g. if one data point contains several real-world entities. Other cases call for data points to be combined into bigger batches. Data might also need to be converted into another format to better fit the storage system.

This thesis is motivated by the above challenges, and the idea of addressing these by automating the exploration and analysis of the data, and by generating templates for the configuration of the data collection pipeline. This approach could reduce the need for human labour, and make data representations more accurate and more consistent across data collections.

The challenges that motivate this work come in part from the Triangulum smart city project and the data collection platform being developed at the University of Stavanger (UiS). The data collection platform will collect and store large amounts of data from a variety of smart city data providers. The structures of these data sets will vary. Some data sets follow strictly defined and documented schemas, while others may come with no documentation or semantic information at all. The data to be stored will be used for smart city research and development.

1.2 Motivating use case, part 1

One of the data providers that is working with the Triangulum smart city project is Kolumbus, the public transportation administration in Rogaland. They monitor the location of around 400 busses that are in traffic in the region, in real time [7]. The data provided by the Kolumbus web service is collected in batches of data that contains both valuable, and redundant information. The data has a complex structure, and the data collection pipeline should be configured properly in order to make the data both storage efficient, and easily retrievable. This use case will present a very simplified process of acquiring and storing data that represents the location of a bus.

Imagine a service that sends a text string to a pipeline once a minute. Let the data provider in this case be an external data provider, i.e. Kolumbus. The string is an encoded object, containing a set of data fields encoded as key value pairs. The object contains fields that are named `id`, `timestamp`, `route`, and `bus_loc`. The first two fields contain numeric values, `route` contains a string, and the `bus_loc` contains two numeric values, named `lat` and `long`. How can a data collection pipeline best handle this type of data? What does the various fields represent, and what are possible values they can have? What data types are best suited for the fields?

The first step is to look at the fields of the sample data, and make some qualified assumptions. The field `id` is probably an identifier for the event. Unless confirmed, however, it cannot be confidently assumed to be unique. A new identifier field, that is unique in the pipeline context, will be assigned by the storage system by default. It might still be of value to store the original field but then it should perhaps be renamed to reflect that it is the identifier used by the data provider, for example '`provider_side_id`'.

The `timestamp` is a numeric value representing some point in time. While the name of the field, `timestamp`, says nothing about what happened at this point, it could be inferred that this is the time of measurement at the sensor. Alternatively, this could be the time when the event was recorded at the data provider's data centre. Storing this field can either be done by specifying this field as the main timestamp for the event, or rename the field to a name that implies more clearly the field's semantic meaning, e.g. '`provider_timestamp`' or '`measurement_time`'.

The `bus_loc` field is different, because its value is an object containing two fields, `latitude` and `longitude`. This structure can be easily maintained in most NoSQL storage systems. If, on the other hand, it is safe to assume that `bus_loc` represents a location in the real world encoded as GPS coordinates, there may be a data type designed specifically for this in the data storage system of the data collection framework.

These are a few of the challenges one faces when configuring a data collection pipeline for a new data source. This concrete use case will be revisited throughout this thesis to illustrate the relevance and implications of the work presented. It will be continued in Section 2.5.

1.3 Project description

The project description as it was presented at the start of this project was to "study the possible approaches for designing and implementing a module for dynamic schema modelling and data type mapping," i.e. a module that can discover the schema of the data and the data types of the fields. Furthermore, "the module should use data from external data providers, and also interact with the metadata provided through a data intake form."

This thesis will present the architecture, design, and implementation of a system that automate some parts of the process of configuring a data collection pipeline. By discovering the underlying data model, and analysing sample data, suggested mappings will be generated, i.e. likely data types for the data field in the storage system. For example a field that is a string in the original data set might contain only float values, and thus might benefit from conversion to float upon storage. The system also enables renaming field names, with the purpose of adding semantic information to the field name. However, this part requires domain expertise, and is therefore not automated.

Goal Develop a software system that can automatically generate a set of configuration files for a data collection pipeline, based on some sample data that is representative of the data to be collected.

Research question 1 Can a model of the expected data, including field types, be discovered by reviewing a representative set of sample data?

Research question 2 Can data collection pipeline configuration files be generated automatically, based on a model of the expected data?

This thesis develops the design and implementation of a software system that can handle the challenges of automatic pipeline configuration, specifically for the Triangulum smart city project context. The software system here developed is called SDModel, and is a system that can generate configuration files based on sample data.

1.4 Thesis structure

This thesis presents some of the challenges, and possible solutions for collecting and storing smart city data. It also presents the architecture, design, implementation, and testing of a

software system for generating configuration files for a data collection framework.

The thesis start after this introduction with some background and a presentation of the Elastic stack in Chapter 2. Chapter 3 sets the requirements and provides an overview of a possible solution to the problem. In Chapter 4 the design of the system is presented. The concept behind a set of analysers that will analyse the sample data are presented, and the approach to suggest a list of likely data types for each data field as well. Chapter 5 presents in technical detail the implementation of the software, including how it is made and how it is used. Chapter 6 tests and evaluates the software through some experiments and present the results. Chapter 7 gives the overview and evaluation of the work presented as well as contributions and ideas for future work.

Chapter 2

Background

Working with data intensive systems is different from traditional data systems. This is much due to the architectural differences big data storage systems require to handle the fast pace and large amounts of data. Many new systems have been created to handle the challenges of big data. However, it turns out there exists older systems too that could handle the challenges presented by big data.

This chapter will present some theory and background, first data storage in Section 2.1. Data modelling is presented in Section 2.2. Finite automaton and their relationship with Regular expressions is presented in Section 2.3. Related work is presented in Section 2.4, before the data provider for the motivating use case is presented in Section 2.5. Elasticsearch, a search engine and document store is presented in Section 2.6. Logstash, a data collection engine is presented in Section 2.7.

2.1 Theory: Data storage

The data points of a data set are often stored in some type of database. A database stores the data in a way that makes it easy and fast to query and retrieve specific data. Relational databases, like MySQL or PostgreSQL have long been the standard. They store data as rows in predefined tables. This makes it easy to retrieve data based on the values of one or more specific columns in the table. And it makes it possible to retrieve only the requested columns, and combinations of columns from several tables. Unfortunately, this architecture is unfit for the challenges that big data present [3]. In the case of storing structured, big data, the storage system must be able to scale. and relational databases are not built for that.

NoSQL, as in "Not Only SQL", refers to a group of non-relational databases that do not primarily use tables, and generally not SQL for data manipulation [4]. NoSQL databases are "becoming the standard to cope with big data problems" [3]. This is in part "due to certain essential characteristics including being schema free, supporting easy replication, possessing a simple API, guarantee eventual consistency and they support a huge amount of data"[3].

There are three primary types of NoSQL databases:

Key-value store databases are simple but fast databases, where each data point is a key-value

pair, i.e. a unique key identifies the value [3]. To retrieve a data point stored in such a database, one must know the key of the object, and make a query based on that. This makes more advanced queries or searches impossible. On the other hand, this fact make key-value stores very fast since the database never evaluates the value of the data point. Among the various key-value store databases that have emerged recently, most seem to be heavily influenced by Amazon's Dynamo [3]. Another popular key-value store is Redis, "an open source, in-memory data structure store, used as a database, cache and message broker" [8].

Column-oriented databases store and process data by column instead of by row [3] like traditional relational database systems do. This type of database is very similar to the relational database, and often use SQL as its query language. However, column-oriented databases are normally more efficient than relational database equivalents when searching for data and queries are often faster.

Document-store databases "support more complex data structures than key-value stores, and there are no strict schema to which documents must conform" [3]. MongoDB, SimpleDB, and CouchDB are examples of document-store databases [3]. Another example is Elasticsearch, "a real-time distributed search and analytics engine" [9].

2.2 Theory: Data modelling

Data modelling is the process of creating a data model. A data model is an abstract model of some data.

The American National Standards Institute (ANSI) suggests [10] that a data model can be one of three kinds:

1. A conceptual data model, where the semantics of the domain is the scope of the model and entities are representations of real world objects without implementation considerations.
2. A logical data model, where the semantics are described relative to a particular manipulation technology.
3. A physical data model in which the physical storage is the main focus.

These represent levels of abstractions of the data model, different applications may require different levels of abstraction, and thus different kinds of data models.

Hammer and McLeod [11], describe a semantic data model as "a high-level semantics-based database description and structuring formalism (database model) for databases". Their idea is to create a language for database modelling that is based on real world-entities and the relationships between them.

Semantic web is an approach to make the web, and more generally, human speech and writing, comprehensible for computers. The semantics is expressed by the Resource Description Framework (RDF), a language for defining objects and their relationships. The RDF content is serialised to eXtensible Markup Language (XML) and placed on the web page, hidden from the general user, but visible to computers.

An entity is "something that has separate and distinct existence and objective or conceptual reality" [12]. In the semantic web specification, entities are mapped to collections of informa-

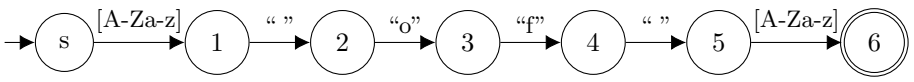

Figure 2.1: Finite automaton for the regular expression `[A-Za-z]+\ of\ [A-Za-z]+`

tion, called ontologies. In the context of information theory, an ontology is defined as “an explicit specification of a conceptualisation” [13]. In this context, a conceptualisation is defined as an “abstract, simplified view of the world that we wish to represent for some purpose” [13]. The purpose of ontologies is to define a method for which different data providers within the same domain can use the same formally defined terms in their semantic web definitions.

Ontology alignment, or ontology matching, is the process of determining the ontology represented by a field’s value [14].

In contrast, schema matching, as defined by [15], is to find a mapping between the data elements of two different schemas. Schema integration is the process of integrating data with different schemas, into one unified schema [16].

These subjects mentioned above can in some aspects seem similar to the part of the problem this thesis seeks to solve, the discovery of a data model from some sample data. However, they all seek to map between an implementation format, via some abstract and generic format, and then to some other implementation format. For this thesis, it might be easier to map the data types directly from one implementation format to the other, and drop the detour via the generic format.

2.3 Theory: Finite automaton and Regular expressions

When searching a text one often specifies a pattern to search for. For example, to search this thesis for the word “of” one would specify that word itself as the pattern to search for, and expect all places where it occurs to be in the results. The results will include many hits that are not just the word “of”, but words that include “of”, like “pr**of**iling”. To avoid this one might add spaces in front and back, and search for “ of ” instead. However, if every page of this report had the page number printed as “page X of Y” where X and Y are numbers, one might wish to omit those matches. This could be accomplished by defining that there needs to be a letter, and not a digit, before and after too.

Imagine a computer program that could search for this pattern in a collection of text, given that the text is presented one character at a time.

A “finite automaton is a graph-based way of specifying patterns” [17]. The finite automaton for the pattern above is depicted in Figure 2.1.

The finite automaton in Figure 2.1 can be understood as a computer program. When it starts it is in ‘state s’, illustrated by the circle with the ‘s’ inside. The automaton then reads the characters of a string one by one. If there is a transition, illustrated by an arrow, that has a label that includes the input character, the automaton will move to the state one at the other end of the arrow. In this example, any letter between a and z, capitalised or not, will move the automaton from ‘state s’ to ‘state 1’. However, should the input contain any character that does not match a transition from the state, like a digit, the automaton moves back to the starting position, and evaluates the character from there. If the automaton is in ‘state 1’ and

the input contains a space character, the automaton will move to 'state 2'. When in 'state 2', the automaton needs to see the letter 'o' in the input to move to 'state 3'. When in 'state 3', it needs to see the letter 'f' to move to 'state 4', anything else will move it back to start. When in 'state 4', it needs to see a space character to move to 'state 5', and from there, one letter is enough to move it to 'state 6'. 'State 6', however, is accepting, meaning that if a string reaches this state, it is accepted by the automaton.

Imagine a string '`3 of 55 or one of the best`' being fed to the automaton. Seeing the character '3' will move it nowhere, the space character neither. When the automaton sees the 'o' it moves to 'state 1'. The 'f' will cause a reset and also a move to 'state 1', thus no change. This will continue for all the letters, numbers, and spaces up until just after the letter 'e' has been seen. The automaton is now in 'state 1', and seeing the space character moves it to 'state 2'. The next character is the letter 'o', the only letter not causing a reset of the automaton, but rather a move to 'state 3'. From 'state 3', the character 'f' is seen, moving it to 'state 4', and then the space character moves it to 'state 5'. The next character that is seen is the letter 't' which is between a and z and it moves the automaton to 'state 6'. 'State 6' is accepting, so the given string is said to be *accepted* by the automaton, and the rest of the string need not be evaluated.

A finite automaton is a *deterministic* finite automaton if, for all the states there is at most one transition that include each character [17], i.e. there is no character that can move the automaton into two states at the same time. A finite automaton can also be *non-deterministic*, in which case it is allowed to have transitions with the same characters in their labels, thus leading to several states at the same time [17]. By definition, a deterministic automaton is also a non-deterministic automaton [17].

Another feature of the non-deterministic automaton is the ϵ transition, it represents a silent transition, or empty transition [17]. For example if the automaton in Figure 2.1 had an ϵ transition between 'state 4' and 'state 2', and it moved to 'state 4' it would simultaneously be in 'state 2'. This would cause the automaton to accept the same strings as before, only now the characters 'of' can appear any number of times before the last space character. For example the string 'a man ofofofof honour' would be accepted.

"Regular expressions are an algebra for describing the same kind of patterns that can be described by automata" [17]. It is analogous to arithmetic algebra. An operand can be one of the following [17]:

- A character
- The symbol ϵ
- The symbol $\emptyset$
- A variable whose value can be defined by a regular expression.

Operators in regular expressions can be one of the following [17]:

- *Closure*, denoted by a star, e.g. `R*`, whose effect is "zero or more occurrences of strings in R".
- *Concatenation*, has no symbol, e.g. `ab` is the concatenation of `a` and `b`.
- *Union*, denoted by `|`. E.g. `a|b`, effectively `a` or `b`.

UNIX systems use regular-expressions like these to describe patterns not only in search tools like *grep* but also in text editors and other tools [17]. However, in UNIX there has been added some convenient additions to regular-expressions. Character classes are groups of characters inside square brackets that are interpreted as any of these characters. For example, `[aghinostw]` is the same as "any of the characters in the word 'washington' " [17]. If a dash is put between two characters, like `[a-z]`, all the characters between `a` and `z` in the alphabet is denoted [17].

UNIX also has symbols for start and end of a line, `^` denote start, and `$` end of the line. For example `^[a-z]*$` will only match if the line consists of only letters between `a` and `z`. To 'escape' characters, i.e. use the literal meaning, UNIX uses the backslash. For example to match on an amount given in dollars one could use `^\$[0-9]*$`. Here the first `$` is escaped by the backslash and is not interpreted as the end of line requirement, but the actual character `$`.

The example in Figure 2.1 is equivalent to the regular expression

```
[A-Za-z]+\ of\ [A-Za-z]+
```

Here the space characters are escaped. In clear text this regular expression accepts a string consisting of "one or more letter between a and z, capitalised or not, then a space character, then the word of, then a space character and in the end one or more letter between a and z, capitalised or not".

2.4 Related work

The SEMantic INTegrator (SEMINT) tool uses *neural networks* to identify attribute correspondences between databases [18], i.e. attributes that have semantically equivalent values. This is done by analysing metadata values extracted from relational databases [18].

The Automatch system uses *bayesian learning* to match the attributes of two database schemas [19]. The Automatch system relies heavily on knowledge provided by domain experts that to find the matching attributes [19].

The *Learning Source Descriptions (LSD)* system for automatic data integration is presented in a paper from 2001 [20]. It performs data integration, the process of creating a "mediated table" that contain virtual columns that are mapped to several data sets through semantic mappings. In this context, a semantic mapping is a definition of the relationship between a column in the mediated table and each of the data source's tables. The LSD system attempts to automate the process of finding semantic mappings. For example, for a column named 'phonenumber' in one data source's table and 'phoneno' in another data source's table, the semantic mapping would define that the two columns have the same semantic meaning and therefore should both be mapped to the same column in the mediated table. This enables the user to make one query and get results from several different data sources.

The LSD system uses a set of *base learners*, a *meta-learner* and a *prediction converter* to automatically discover the semantic mappings [20]. The LSD system operates in two phases, training phase and matching phase.

The training phase starts with the user providing the semantic mappings manually for a small set of data sources, then the system uses these mappings together with data from the data sources to train the learners [20]. Each of the learners learn from different parts and characteristics of the source data. For example there is a name matcher that is a base-learner which

learn from the name of the XML tag. Another base-learner is the County-Name Recogniser which searches a database extracted from the web to verify if a value is a county name [20].

In the matching phase, LSD applies the learners to the new data set and combine their predictions in a 'meta-learner' [20].

The LSD system solves a different problem than this thesis seeks to solve, however, parts of the approach can be used in this thesis. The architecture presented in Chapter 3 is, on an abstract level, inspired by the base-learner, meta-learner and prediction converter used in the LSD system.

2.5 Motivating use case, part 2

Continuing the use case presented in Section 1.2.

Kolumbus Real Time Open Data is an open web service created by Kolumbus. It utilises the Service Interface for Real Time Information (SIRI) to provide a Simple Object Access Protocol (SOAP) web service. The web service exposes three services defined by SIRI.

- The Vehicle Monitoring Service (VM) provides information about the current location and expected activities for the buses [21].
- The Stop Monitoring Service (SM) serves information about vehicles arriving to and departing from a bus stop [21].
- The Situation Exchange Service (SX) serves deviation messages [7].

Requesting data from such a service is a complicated and tedious task and the response is composed by many complex objects. It can prove difficult to make sense of the data and its structure without the right tools.

Any SOAP web service is by definition self-documented. The specifications of all the data fields exist in the Web Service Definition Language (WSDL) file. However, the data types defined in the WSDL file are not generally available storage systems. Knowledge about how to map these values to the available types in any specific storage system is needed in order to store this data properly.

In 2011, Roy Fielding proposed an alternative to the SOAP protocol in his PhD thesis, the Representational State Transfer (REST) principles [22]. Compared to SOAP, the new approach was much easier and less verbose, but also more prone to application errors. Mulligan and Gracanin [23] present a set of tests that prove REST to be "more efficient in terms of both the network bandwidth utilised when transmitting service requests over the Internet and the round-trip latency incurred during these requests" [23]. This is probably parts of the reason why REST seem more popular than SOAP.

While REST does not enforce any restrictions on serialisation of the data, the most commonly used serialisation of transmissions is probably JSON.

With a view toward the general applicability of the solution presented in this thesis, beyond this use case, the Kolumbus data will be retrieved, converted to JSON, and then used as input to the system. The definitions provided by the WSDL files will act as a reference to what the system should discover about the data.

2.6 Elasticsearch

"Elasticsearch is a real-time distributed search and analytics engine" [9]. It stores documents in a flat-world, document-oriented database. An data point stored in Elasticsearch is referred to as a *document*. This convention comes from the early days of Elasticsearch, when it was used to store mostly documents. Elasticsearch differs from classical relational databases in several ways. For example, there are no tables in Elasticsearch. One document is stored in one place, instead of being spread over multiple tables or columns. This makes searching the documents fast. However, the result of a query will consist of complete documents, not parts or aggregations of documents like one can get from relational databases [9].

Elasticsearch runs on top of Apache Lucene core, "a high-performance, full-featured text search engine library written entirely in Java" [24]. Elasticsearch was first created as libraries and helpers to make it easier for developers to work with Apache Lucene core.

Communicating with an Elasticsearch instance is done through one of two ports. Port `9300` for Java applications through Java specific protocols, and `9200` for other languages through a RESTful interface.

2.6.1 Distributed operation of Elasticsearch

Elasticsearch normally runs distributed in a cluster. A node in this context is one running instance of Elasticsearch. While several instances of Elasticsearch can run on the same physical (or virtual) machine, a node is typically one instance of Elasticsearch running on one machine. Several nodes can form a cluster where one node is elected leader following the Paxos algorithm [25]. When a leader node, for any reason, becomes unresponsive, another node will be elected leader. Any node in the cluster can try to become the leader, but in order accomplish this, the node must have the votes of at least half of the other nodes, i.e. it must have votes from a quorum. In the context of the Paxos algorithm, a quorum is the number of votes, or a group of voters that form a majority in the system, i.e. more than half of the voters form a quorum. The leader, or master node, is in charge of coordinating the cluster structure. However, queries does not need to go via this node, as every node can handle any request from a client, ensuring high availability. [9]

The data stored in an Elasticsearch system is divided into `shards`, yet another type of containers of data. Because an index potentially could contain more data than any hardware has capacity to store, the index can be divided into shards. Shards can live on several nodes, and one node can have several shards. Each shard is responsible for its own set of data, and handles storing, searching, and retrieving the data. And while one node might have several shards it does not need to have all the shards, but can make other nodes search through other shards. There are two kinds of shards in the system, `primary shards` and `replica shards`. Each replica shard is a replica of a primary shard. While write operations must involve the primary shard, a read operation can safely be done off a replica shard. This division makes the system highly available and fault-tolerant. [9]

The address of a shard is based on some routing value, `_id` by default. Given a document with routing value R, being stored to an instance with a total of N_{shards} primary shards, the shard that the document is stored in, S_{doc} is given by

$$S_{doc} = hash(R)\%N_{shards} \tag{2.1}$$

where the % denotes the modulo operator and $hash()$ is some hashing function. [9] This makes it easy to find which shard a document is located in, but makes the number of shards to split an index into, immutable. If the number of primary shards were to ever change, then all the documents already indexed would have to be re-indexed, which is possible but costly to do.

2.6.2 Storing data in Elasticsearch

When a document is stored in Elasticsearch, it is stored as a specific 'document type' to an index. The name of the document type is stored in a property, `type`. An index in Elasticsearch is a container that is searchable and can contain documents of multiple document types [9].

A document in Elasticsearch is a data point that is serialised into JSON and stored to an index under a unique identifier. When you store a JSON object, or document, in Elasticsearch, it is indexed, and some metadata is added to the object. Elasticsearch creates a document with a field `_id` that contains a unique identifier, and stores the original data under the property `_source` of this document. Then at the root level of the object, other metadata is added, like `_index` (the name of the index where the document is stored), and `_type` (the user defined type of the document) [9].

A document is indexed in Elasticsearch by sending a PUT request to a node in the cluster with the URL `/{index}/{type}/{id}`, where `{index}` is replaced by the index name, `{type}` by the document type, and `{id}` by the id that the document should be stored with. The document is encoded as a JSON string, and passed in the request body. By using a POST request and omitting the identifier in the URL, Elasticsearch will generate a 20-character, URL-safe, Base64-encoded GUID-string, and use that as the identifier. As specified by the REST protocol, retrieving a document should be done using the GET request method, and the URL should specify which document to retrieve according to the same pattern as above. Elasticsearch also supports the HEAD request to check if a document exists, and the DELETE request to delete a document.

When a field in a given document is stored in Elasticsearch, the values of the field will be added to an inverted index, i.e. a list of values with references to the documents and fields where each value can be found. In the case of a full-text field, it will be analysed and split up into single words that are added to the inverted index.

Documents in Elasticsearch are immutable once they are indexed. Therefore, updating a document is not possible without re-indexing the entire document. Elasticsearch does, however, hide this fact. If a PUT request is sent to an URL that contains a document, the values in the request body will be updated on the document, and also the `_version` property will be incremented. But underneath, the document is in fact copied and re-indexed as a new document. The `_version` property is used by Elasticsearch to handle concurrency control.

Elasticsearch is said to use *optimistic concurrency control*. A document is updated only if the version property in the update request is the same as the highest one in the system for the given document. For example if one user, or application instance, retrieves a document with `_version` = 4, and another user updates the document, causing the `_version` in the system to increase to 5. Then the first user cannot use his or her document for updating, since it may contain outdated data. In a system that uses *pessimistic concurrency control*, the second user would not get his or her update committed because of a lock placed by the first user. In Elasticsearch, being a system that focuses on fast search and high availability, an optimistic approach is chosen.

When a document is created or deleted in Elasticsearch, the node that receives the request forwards it to the node that holds the primary shard for the document. If this request is successful at the primary shard, the node that has the primary shard will forward the request to the node(s) that hold the replica shards. When the replica shards also are successful in performing the operation, a confirmation is sent back to the node that received the request, and back to the client. By default, the node with the primary shard must first check that it has contact with a quorum of the replica shards before starting the write operation. This behaviour can be controlled by the `consistency` option in the system configuration, but it could cause major consistency problems to turn it off. There is also a `timeout` option available, in which a request will fail if there is a problem with the request.

To retrieve a document, on the other hand, the receiving node will just need to find a shard that has the document requested, and send the request to that node, regardless of whether it is a primary or replica shard. The nodes that have the requested shard will take turns in handling retrieval requests in a round-robin fashion.

Elasticsearch is optimised for search, and to do this well the data must be mapped and analysed before indexing. A GET request to `/_search` is the simplest way of searching in Elasticsearch, and it will return a list of hits, by default ten hits. Each of these hits will have a property `_score` which represents how good a match the hit is for the query. A GET request to `/{index}/{type}/_search` forms a query that will search only within the specific index given by `{index}`, and only for documents that are of the document type given by `{type}`. [9]

2.6.3 Index mapping

Elasticsearch is made to be simple and require as little configuration as possible. A mapping in Elasticsearch is a set of properties that define how Elasticsearch should handle the field that the mapping applies to. The main purpose of a mapping is to define what data type the value of the field should be stored as. However, other properties can also be defined, like what language the values of a string field is expected to be in, which could in turn improve searchability. Searchability is in this context understood as the ability for the field to appear in search queries where it is expected to appear, and to not appear in searches where it does not belong.

If a document is stored to an Elasticsearch cluster with the name of an index that does not already exist, that index is automatically created. The mappings for the index is generated from the data using Elasticsearch dynamic mapping. If a document is added later with an extra field, the dynamic mapper will identify the field and add it to the index. This means the extra field will be present in all documents of the same document type that is indexed after this addition. The dynamic mapping can find mappings for the general types of values based on the JSON field types. It detects properly formatted dates and detects numbers represented as strings [26].

The mapping of the fields in the data, can have great effect on searchability. For the user to gain control of the mapping, an index must be created explicitly. This is done by a PUT request to the index name. The content of the request should contain the index specification, i.e. a JSON-encoded string with an outer object that has properties `settings` and `mappings`.

The value of the `settings` property is an object that defines some system properties for the index. For example the number of shards to spread the index on, or the number of replicas to maintain of the indexed data.

The value of the `mappings` property is an object that defines the mappings of the data that is to be stored. A simple example of a mapping can be seen in Example 3.1. If dynamic mapping is enabled on the Elasticsearch instance, only the fields that are expected to not be identified properly by the dynamic mapper need to be specified. A mapping can be updated for a field, after index creation, but that will only have an effect on future indexing of either new or re-indexed documents [9].

2.6.4 Data types in Elasticsearch

Elasticsearch supports a number of different data types for the fields in a document [26]. Some are obvious counterparts to the data types defined by JSON, while others are more complex and specialised data types. The choice of data type for a field in Elasticsearch is important for the performance of the whole Elastic stack.

The main benefit from choosing the correct data type is increased searchability. There are also general performance benefits gained from choosing the data type. For example if storing a number with a defined number of decimals. Here Elasticsearch has a data type `scaled_float`, that stores the number as a `long` with a scaling factor. The value is scaled up upon storage and down on retrieval. "This is mostly helpful to save disk space since integers are way easier to compress than floating points" [27]. A data field that has the wrong data type will, generally speaking, be less searchable than if it had the correct type. In many cases though, there is not as clear a distinction between the various data types, and it can be hard to find the correct data type for a data field.

The following general data types can be mapped to a variety of possible Elasticsearch data types.

A string can either be a `text` or `keyword` in Elasticsearch. While `text` is considered to be a full-text field and can be searched accordingly, a `keyword` is searchable only by the exact value [26].

Numbers can be either `long`, `integer`, `short`, `byte`, `double`, `float`, `half_float`, or `scaled_float` in Elasticsearch. The integer types, `long`, `integer`, `short` and `byte`, i.e. the whole number types, differ in possible maximum values. Choosing the smallest integer type according to its maximum value compared to the maximum value of the data will help the indexing and searchability. However, storage is optimised for the actual values that are stored, and not by the capacity of the data type, so the choice of data type will not affect the storage demand [26].

A date can be represented either by a string containing formatted dates, a long number representing milliseconds-since-the-epoch, or an integer representing seconds-since-the-epoch. Elasticsearch will convert the date to UTC, if the timezone is specified, before storing it as the Elasticsearch data type `date` [26].

A Boolean in JSON can be stored as `boolean` in Elasticsearch. The Boolean type also accepts the strings "true" and "false" as Boolean values. Elasticsearch versions prior to 5.3.0 also

accepted strings “off”, “no”, “0”, ”” (empty string), 0, 0.0 as Boolean false and all other as Boolean true values, but this is deprecated in newer versions.

Range data types are also supported in Elasticsearch, and can be either `integer_range`, `float_range`, `long_range`, `double_range`, or `date_range`. The field that represents a range should be a JSON object with any number of range query terms, like `gte` and `lte` representing “greater than or equal” and “less than or equal”, respectively [26].

An array of values in JSON can be a list of values in Elasticsearch. Actually, there are no explicit data type for arrays in Elasticsearch. Instead, all Elasticsearch fields can contain lists of values, as long as the values are all the same Elasticsearch data type [26].

Objects are not supported by the Lucene core, since it only handles one level of values. However, Elasticsearch hides this fact by flattening objects using dot notation prior to storing them [26].

Nested objects is a specialised version of the `object` datatype that allows arrays of objects to be indexed and queried independently of each other in Elasticsearch [26].

A location represented by a latitude-longitude pair, can be stored in Elasticsearch using the data type `geo_point`. This enables searching and ordering by location. A `geo_point` can be specified in four different formats, an `object` with `lat` and `lon` keys, a `string` with the format `"lat,lon"`, a geohash, or an `array` with the format `[lon,lat]` [26]. A geohash is a geocoding system that creates a URL-friendly string to represent a location [28].

GeoJSON types can be stored using the datatype `geo_shape`. It is used to represent a geographic area. Elasticsearch supports the GeoJSON types `point`, `linestring`, `polygon`, `multipoint`, `multilinestring`, `multilinepolygon`, `geometrycollection`, `envelope` and `circle` [26].

In addition to the mentioned types, there also exists a list of specialised types, IP, Completion, Token count, mapper-murmur-3, Attachment and Percolator .

2.7 Logstash

While it is possible to add documents directly to Elasticsearch using the REST API, it is often preferable to use a program like Logstash to collect and prepare the data before it is stored. Logstash is a “data collection engine with real-time pipelining capabilities” [29]. One running instance of Logstash can take input from different sources, prepare the data as defined by the configuration, and send the data to storage. Logstash can send the data to a variety of different storage systems, however, it is developed and marketed as part of the Elastic stack.

A Logstash pipeline consists of three different stages, input, filter and output. Inputs generate events, filters modify them and outputs ship them elsewhere [29]. The three stages are all configured in the same configuration file.

The format of the Logstash configuration file looks like it might be inspired by JSON, but it is not valid JSON. According to the source code the format is custom-made using Treetop, a Ruby-based package for parsing domain-specific languages, based on parsing expression grammars [30].

2.7.1 Input

The input section of the configuration file defines how and where the input data will arrive and how it is to be handled. There are a number of input plugins available that enable Logstash to read events from specific sources [29]. For example the `file` plugin stream events from file, `http` will receive events sent over http or https. An example of the input section can be found in Example 2.1. This configuration enables Logstash to receive JSON encoded events using TCP on port 5043.

```
input {
  tcp {
    port => "5043"
    codec => json
  }
}
```

Example 2.1: An example of the input section of a Logstash configuration file. The Logstash instance will here be receiving events using the `tcp` plugin on port 5043 and the event data is expected to be encoded as JSON

2.7.2 Filter

There are a number of filter plugins that can be used in the filter section of the configuration. Filter plugins perform intermediary processing of an event [29]. In Example 2.2 one plugin is used, `mutate`, and from that plugin two operations are performed. First four of the field names are changed, and then the types of the fields are changed, or at least set explicitly.

2.7.3 Output

As for the output section of the configuration, it too uses plugins. In Example 2.3 two plugins are used. First the `stdout` plugin will output events to the std out of the program using the codec `rubydebug`. Finally the Elasticsearch plugin is used to ship the data to an Elasticsearch instance at localhost:9200. The event will be indexed at the index named `testindex` and be of type `testtype`.

```
filter {
  mutate {
    rename => {
      "id" => "provider_side_id"
      "timestamp" => "measurement_time"
      "[bus_loc][latitude]" => "[bus_loc][lat]"
      "[bus_loc][longitude]" => "[bus_loc][lon]"
    }
    convert => {
      "[bus_loc][lat]" => "float"
      "[bus_loc][lon]" => "float"
    }
  }
}
```

Example 2.2: The filter section of a Logstash configuration file for the motivating example. This configuration renames the field `id` to `provider_side_id`, the `timestamp` to `measurement_time` and `bus_loc` to `location`.

```
output {
  stdout {
    codec => rubydebug
  }
  elasticsearch {
    hosts => ["localhost:9200"]
    index => "testindex"
    document_type => testtype
  }
}
```

Example 2.3: An example of the output section of a Logstash configuration file. This output section uses two plugins. First the `stdout` plugin sends events to the std out, and the second plugin sends the same data to the Elasticsearch instance at `localhost:9200`. The event will be indexed at the index named `testindex` and be of type `testtype`.

Chapter 3

Architecture

To collect and store data using a data collection pipeline it is important to know how the data is structured and what the various fields represent. Some information can be derived from the field names, and there might also be other documentation available. However, assumptions made from field names and dataset documentation might not be enough. Looking at sample data is often a good way to get more knowledge, but it can be very time consuming.

To automate the process of a data collection pipeline configuration, the data must be explored and analysed automatically. The domain expert must be given a chance to edit the results of this analysis. And the results can be used to generate configuration files.

The goal for this project is to develop a software system that can automatically generate a set of configuration files for a data collection pipeline, based on some sample of the expected data. The main focus is the challenge of automatically configuring new data collection pipelines for the data collection platform being developed at UiS.

One option for software to be used in the data collection platform, is the Elastic stack. Elastic stack is a software stack consisting of several programs, most important is Elasticsearch, Logstash and Kibana. Elasticsearch is "a real-time distributed search and analytics engine" [9]. Logstash is "a data collection engine with real-time pipelining capabilities" [29]. Kibana is "an analytics and visualisation platform designed to work with Elasticsearch" [31]. All three are open source projects. The Elastic stack provides a system well suited for collecting, transforming, storing, searching and visualising data.

Another document-storage based software is Apache CouchDB [32]. CouchDB is easy to use, schema free, scales linearly and focuses on being an 'offline-first' type of database [32]. It supports offline operation, and can run offline for example on a smartphone, and also synchronise with the main database when it is back online. However, CouchDB does not support data types other than those of JSON [32], which would make the data less structured than it would be in Elasticsearch.

This thesis describes a system that can automate the collection and storage of smart city data in the Elastic stack. Mapping definitions for Elasticsearch and the filter section of the Logstash pipeline configuration, are generated based on sample data.

An architectural overview of the solution is presented in Section 3.1, including the software system called SDModel. The name comes from the idea of making a Semantic Data Modelling

tool. The motivating use case is continued in Section 3.2, where the data provider is presented. The process of analysing the data is presented in Section 3.3. Reviewing and editing the data model, will be presented in Section 3.4, before generating the output files is presented in Section 3.5.

3.1 Overview of the system architecture

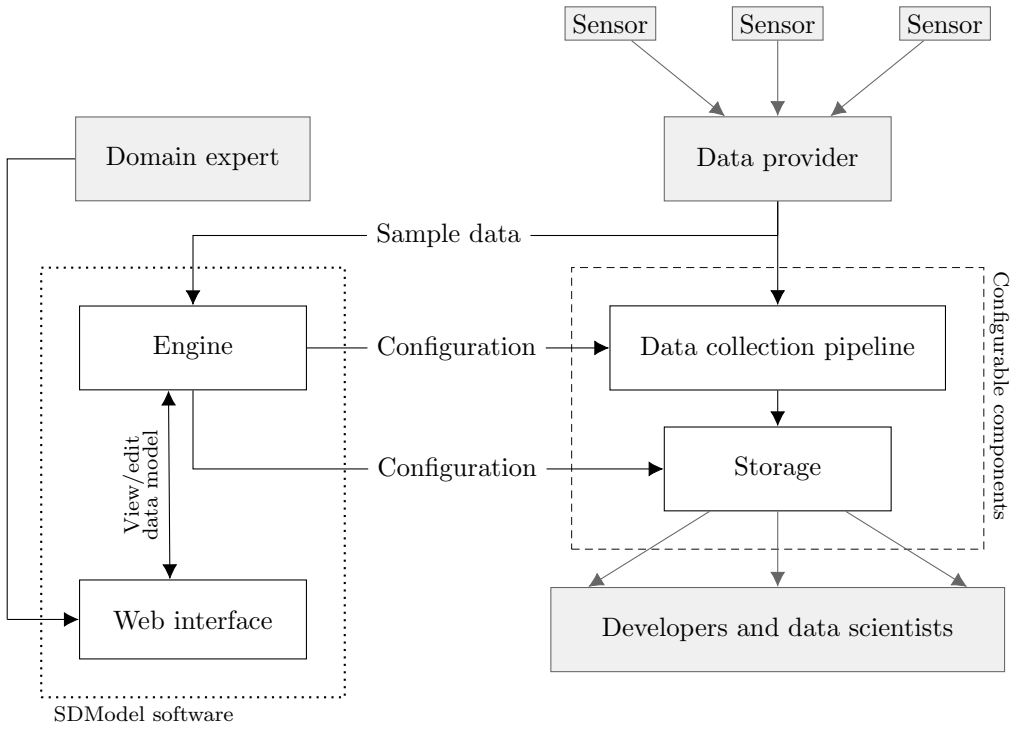

Figure 3.1: Overview of the system architecture. The data provider collects sensor data and delivers data to the data collection and storage pipeline which passes the data on to the storage. Here the data provider also sends some sample data to the engine. The engine serves the discovered data model through a web interface where domain experts can view and edit the data model and generate configuration files for the pipeline.

Figure 3.1 shows an overview of the system architecture of the system. The data provider collects sensor data and delivers data to the data collection and storage pipeline which passes the data on to the storage. The data provider also sends some sample data to the engine. The engine analyses sample data provided by the data provider. From this it generates a data model that can be viewed or edited by a domain expert, through a web interface. The data model shows how the data is structured, what data types the various fields are, and results of the analysis. The domain expert can then generate configuration files that can be used in the data collection and storage pipeline configuration.

3.2 Motivating use case, part 3

(Continuing on the use case presented in Section 2.5.)

```
PUT /busdataindex
{
  "mappings": {
    "bus_event_doc_type": {
      "properties": {
        "bus_loc": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

Example 3.1: JSON structure for creating an Elasticsearch index that has a document type `bus_event_doc_type` that is of type `geo_point`. The index is created by sending this structure in a `PUT` request to `/busdataindex`, which is the new index name.

When storing an object to Elasticsearch, it is stored to an index. An index in Elasticsearch is like a container for documents, it can contain many documents of multiple document types. The data types of the field in a document in Elasticsearch can be defined explicitly, or Elasticsearch can infer them automatically. Whenever Elasticsearch sees a new field, it will assign to that field a data type based on its value, and the data type of that value. If Elasticsearch gets data belonging to an index it has not already seen it will give create a new document type based on this first data object, that specifies the data types for all the fields of the document. However, it does not recognise a GPS location as a `geo_point`, this need to be specified explicitly in advance. [26]

Example 3.1 shows an example of a request to the Elasticsearch Index API, that would create a new index called `busdataindex`. In this index, any document of type `bus_event_doc_type` is expected to have a field named `bus_loc` that is of data type `geo_point`, and thus have the numeric fields `lat` and `lon`.

In the original data however, the fields of `bus_loc` are not named `lat` and `lon`. Renaming and converting the fields can be done by Logstash. Example 2.2 shows an example of a Logstash filter for the situation described here. First, the `id` and `timestamp` fields are renamed to `provider_id` and `provider_timestamp`, respectively. This is done for the field names to imply a more definite semantic meaning. The `bus_loc` fields are renamed, and data types converted explicitly to floats, to fit the format of the `geo_point` type.

3.3 Analysing sample data

The process of establishing a data model starts with finding the schema of the sample data. This is done using an algorithm that recursively traverses the data, and analyses each data

```
filter {
  mutate {
    rename => {
      "id" => "provider_side_id"
      "timestamp" => "measurement_time"
      "[bus_loc][latitude]" => "[bus_loc][lat]"
      "[bus_loc][longitude]" => "[bus_loc][lon]"
    }
    convert => {
      "[bus_loc][lat]" => "float"
      "[bus_loc][lon]" => "float"
    }
  }
}
```

Example 3.2: The filter section of a Logstash configuration file for the motivating use case. This configuration renames the field `id` to `provider_id`, the `timestamp` to `provider_timestamp` and `bus_loc` to `location`. The convert plugin enables the value of the location field to be interpreted as a `geo_point` in the Elastic stack, however, it must also be explicitly assigned that data type (as in 3.1)

point in the sample data set and creates a schema of this data that will be the basis for the data model. The next step is then to find statistical characteristics of the sample data, and make an ordered list of possible storage level data types for each of the data fields.

3.4 User editions and additions

The data model with results of analysis is available for review through a web interface, and can be reviewed and edited there. This provides a good presentation of the data set without having to manually explore the data. The data model may also be edited using the web interface. For example, data type assumptions that are not correct can be changed, fields can be renamed upon storage, and descriptions of data fields can be added to the model directly.

3.5 Generating output files

After the initial discovery and analysis process, the data model has enough information to create configuration files for the data collection pipeline. There are two configuration files that can be generated by the system, and they should be used together.

One is the data preparation stage configuration, a Logstash filter configuration. This file will configure Logstash to rename data fields that have new names, change the data type of fields that have new data types, split arrays and more.

The second configuration file that the system can generate is the storage configuration, the Elasticsearch mappings. This will tell Elasticsearch what data types the various data fields are, and how they should be stored. One of the reasons these should be used together is that a

data field that has been renamed will have the new name in the mappings, and if the name has not been changed by Logstash Elasticsearch will not find these fields.

Chapter 4

Design

To automatically collect and store data, the system must get to know the data, the same way a domain expert would if the process was done manually. The SDModel system must check every value of every field in the sample data set, before forming an opinion on what data type field represents. When all fields have been identified by data type, instructions on how to transform the data and handle the values, can be generated.

This chapter will present an overview of the design of the data collection system. The concept of discovering the schema of the sample data is presented in Section 4.2. The concept behind each part of the data analysis process is presented in Section 4.3. Section 4.4 presents the process of limiting, prioritising, and selecting an appropriate data type for each field. The presentation of the model, and use of the system is presented in Section 4.5. Section 4.6 concludes the chapter with the concepts behind generating the configuration files.

4.1 Overview of the design

It is assumed by the system that the data provider sends sample data in JSON format to the engine. In cases where the sample data is of a different format, it must be converted to JSON before entering the system, e.g. the Kolumbus data that is initially XML formatted. JSON is a text format for the serialisation of structured data, it can contain values that have data types `string`, `number`, `boolean`, `array` and `object`, in addition to `null` [33]. It is assumed in this thesis that the sample data is represented as a JSON `array`, that contains any number of similar objects, i.e. data points.

Figure 4.1 shows an overview of the design of the system presented in this thesis. The 'data provider' sends data to the 'data collection pipeline', and also to the 'Schema discoverer', a part of the SDModel software. In the 'Schema discoverer' in Figure 4.1, each data point in the array of sample data is used to discover a schema that all the data points can conform to. The sample data is then run through a set of analysers, where every value of every field is analysed. After that the 'Data type inferrer' uses the collected data to infer a data type, and a list of alternative data types, for each of the data fields. Through this first stage a model of the data is built. It is presented through the web interface for the domain expert to review and edit. From the data model representation, configuration files can be generated.

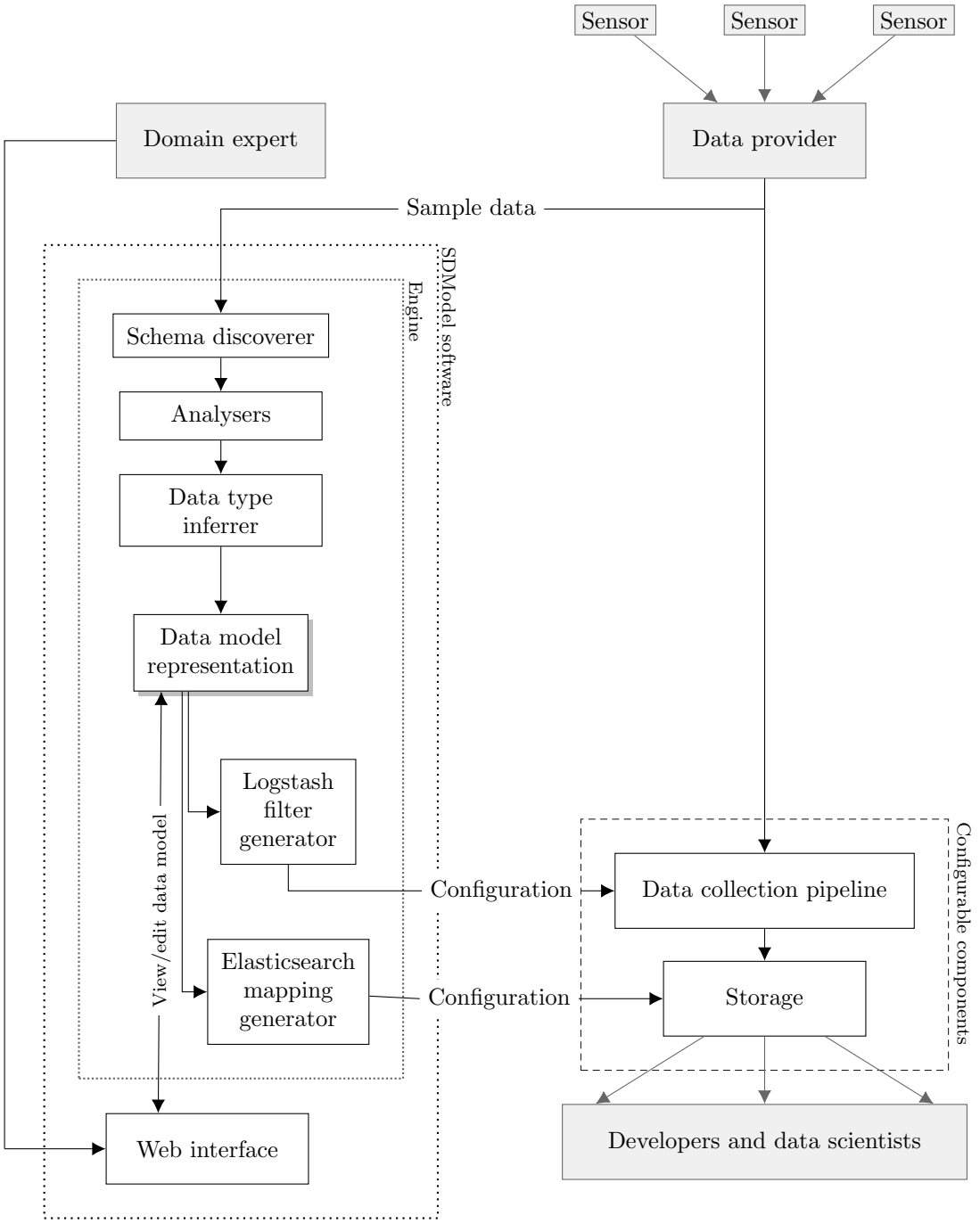

Figure 4.1: The overall design of the system. The data provider sends some sample data to the engine, it is passed to the 'Schema discoverer'. Next, the sample data is run through a set of 'Analysers' that analyses the values of every field of every data point. After this, the 'Data type inferrer' will suggest a list of possible data types for each of the data fields. The data model is served through a web interface where domain experts can view and edit the data model. Logstash filter configuration and Elasticsearch mappings can be generated from the data model.

4.2 Discovering the schema

The first step is to find the schema of the data. This information is then used to create a data structure that represents the data model. This data model is structured so that each data object can have a name, a data type, an key-value list of metadata, and a list of child data objects. The property name on the data object reflects the name of the field in the data model it represents. The data type is the name of the type that the field has in the JSON encoding. The metadata list is a JSON object that have properties with values. It holds information discovered about the values of the field, such as units of the field or relative frequencies of patterns. The data objects list is a list of child objects in case the field is of type `object` or `array`, and thus have nested values.

```
{
  "RecordedAt" : "2017-04-11T09:27:31.6814296+02:00",
  "Temperature" : "21",
  "DeviceType": "ManufacturerModel5100",
  "Location" : {
      "Latitude" : 58.938334,
      "Longitude" : 5.693542
      },
      "BatteryLevel" : 65
}
```

Example 4.1: Example of JSON encoded event data from a temperature measurement device.

Given a data set consisting of JSON formatted data from some temperature measurement device, where a random reading from the sensor looks like the one in Example 4.1, how can its schema be discovered? Finding the schema for this structure is done recursively by passing the data point, a parent object and a key to a function `discoverSchema`. The data point is the root of the data object, i.e. the outermost object in Example 4.1. The parent object can be an empty data model object in the initial call, however, the data model that is created will be passed along here in the recursive calls to the algorithm. If the data point that is passed is a `string`, `number` or `boolean`, the algorithm will add a metadata key called `samplevalue` with the value of the document and then add this data object to the parentObject and return it. If the data point that is passed is of type `object` the algorithm will call itself recursively, with the key and value of each or the properties as key and document, and the newly created data object as the parentObject. If the document is an `array` the algorithm will call itself recursively once for each element in the `array`, but with no key. This results in every recursive call overwriting the last and only the last item in the `array` will be used to form the data model below the `array`.

In the case in Example 4.1, the root object would be passed in first, and since it is an object the algorithm would call itself recursively once for each of its properties. All values except `Location` is made into data objects and returned directly. The data objects is then added to the root object's data objects list. `Location` becomes the parameter to another recursive call to the algorithm, which in turn returns `Latitude` and `Longitude` as data objects. The `Location` object is then added to the root object's data objects list. This would result in a data model represented by Figure 4.2.

```
1:  procedure DISCOVERSCHEMA(document, parentObject, key)
2:      data_object =create_new_data_object(key, primitive_type(document))
3:      data_object.data_path = parentObject.data_path +key
4:      if data_object.primitive_type in ['string','number','boolean'] then
5:          data_object.add_meta_data('samplevalue', document)
6:      else if data_object.primitive_type =='object' then
7:          for doc_val, doc_key in document do
8:              data_object = DISCOVERSCHEMA(doc_child, data_object, doc_key)
9:      else if data_object.primitive_type =='array' then
10:         for item in document do
11:             data_object = DISCOVERSCHEMA(item, data_object)
12:     parentObject.add_data_object(data_object)
13:     return parent_object
```

Algorithm 4.1: Recursively discovering the schema and types of a single document of sample data

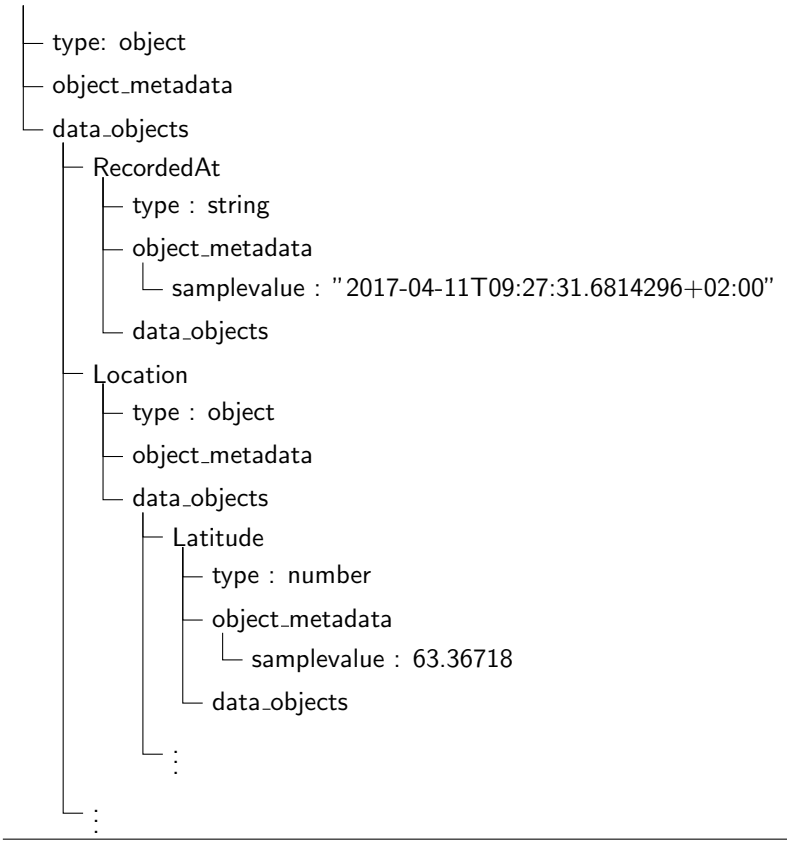

Figure 4.2: Partial data object created from the JSON encoded event data generated by a temperature measurement device shown in 4.1.

This algorithm only requires one document to generate a schema, but that one document might not be an adequate representation of the dataset as a whole. In order to make the data model a more comprehensive representation of the available sample data, all the documents of the sample data is analysed, and the model must be updated accordingly. This is done by the extended algorithm listed in Algorithm 4.2

```
1:  procedure DISCOVERSCHEMA(document, parentObject, key)
2:      if data_object = parentObject.has_data_object_with_name(key) then
3:          if data_object.primitive_type == primitive_type(document) then
4:              if data_object.primitive_type in ['string', 'number', 'boolean'] then
5:                  do.add_meta_data('samplevalue', document)
6:              else if data_object.primitive_type == 'object' then
7:                  for doc_val, doc_key in document do
8:                      data_object = DISCOVERSCHEMA(doc_child, do, doc_key)
9:              else if data_object.primitive_type == 'array' then
10:                 for item in document do
11:                     data_object = DISCOVERSCHEMA(item, data_object)
12:         else
13:             if data_object.primitive_type == null then
14:                 data_object.primitive_type = primitive_type(document)
15:         parentObject.replace_data_object(data_object)
16:     else
17:         data_object =create_new_data_object(key, primitive_type(document))
18:         data_object.data_path = parentObject.data_path +key
19:         if data_object.primitive_type in ['string', 'number', 'boolean'] then
20:             data_object.add_meta_data('samplevalue', document)
21:         else if data_object.primitive_type == 'object' then
22:             for doc_val, doc_key in document do
23:                 data_object = DISCOVERSCHEMA(doc_child, data_object, doc_key)
24:         else if data_object.primitive_type == 'array' then
25:             for item in document do
26:                 data_object = DISCOVERSCHEMA(item, data_object)
27:         parentObject.add_data_object(data_object)
28:     return parent_object
```

Algorithm 4.2: Recursively discovering the schema and types of sample data accepting previously seen fields. Allowing the algorithm to iterate over many documents, and many values in an array, to generate a more comprehensive data model.

The benefit of passing the parent object as an argument is more clear here as most of the fields analysed will be fields that already exist in the parent object. It is easy to imagine that a field may have a null value in some documents and another value in another document. If only the null-valued document is explored, the field will have no data type, but if the field is explored again with for example a `string` value, the field will indeed be marked as a `string` type field. In the case where all fields in the parent object are already explored, there is still value in iterating over more sample data, as these extra sample data objects will confirm that the model is correct. The more sample data that is being used to generate the data model, the more probable it is that unseen data will be properly represented by the data model.

4.3 Analysing the fields

In order to form an opinion about the field type, some characteristics about the values of the field must be discovered. Once the schema is discovered each field can be analysed by iterating over the values in the sample data.

4.3.1 Estimated probabilities from empirical relative frequencies

The first set of analysers find the empirical relative frequency of some characteristic of a field. The characteristic is defined by a discriminant function, f, that takes one value as input and returns a 1 if the value satisfies the characteristic, and a 0 if not. The empirical relative frequency over the sample data $X = [x_0, x_1, \ldots, x_n]$, is used as an estimator $\hat{P}$, of the probability that the field has the characteristic given by the function f. This estimator is defined as

$$\hat{P}(f) = \frac{\sum_{x \in X} f(x)}{n} \tag{4.1}$$

The empirical relative frequency, $\hat{P}(f)$, is the probability that a randomly selected data point in the sample data has this characteristic. The probability of this data field having the same characteristic in a new data point from the same data source is also estimated to be $\hat{P}(f)$.

This is further illustrated by Example 4.2.

Number of null values

The fact that a field exists and has a string value in one data point does not necessarily mean that it will exist in all data points in the data set. The number of null values can be a good measure for how many data points have values for the specified field.

Counting null values is done by iterating over all the data points in the sample data, and count the number of null values and the total number of values for each field. If a field is missing in a data point, this counts as a null value. The algorithm that finds the empirical relative frequency of null or missing values is shown in Algorithm 4.3. The result value is added as a metadata value under the key `p_null`.

In this case the discriminant function, f_{null} is defined by

$$f_{null}(x) = \begin{cases} 1, & x = null \\ 0, & x \neq null \end{cases} \tag{4.2}$$

As an example, imagine a data set with 1000 data points, one of the data fields have the value null in 440 of the data points. According to Example 4.2, the empirical relative frequency of null values, $\hat{P}(f_{null})$ is 0.44 in this case.

$$\hat{P}(f_{null}) = \frac{440}{1000} = 0.44$$

Example 4.2: Example of data set where 440 of 1000 values are null. The empirical relative frequency of null values is then calculated to 0.44.

```
1: procedure countNullValues(dataSource, obj_model)
2:     total_values = 0
3:     null_values = 0
4:     for document in dataSource do
5:         target_data = dataSource.get_data(obj_model.path)
6:         if target_data is array then
7:             for item in target_data do
8:                 if item is null then
9:                     null_values + = 1
10:                total_values + = 1
11:        else if target_data is null then
12:            null_values + = 1
13:            total_values + = 1
14:        else
15:            total_values + = 1
16:    obj_model.object_metadata['null_values'] = null_values/total_values
17:    if obj_model.has_data_objects then
18:        for child_obj in obj_model.data_objects do
19:            child_obj = countNullValues(dataSource, child_obj)
20:    return obj_model
```

Algorithm 4.3: Algorithm for counting the number of null values in the sample dataset and storing the empirical relative frequency of null values as the value of a metadata field for the data object.

Numbers disguised as strings.

Sometimes fields that are of a numerical nature gets stored as strings, perhaps by mistake or for simplicity. However, if the field always contains numbers, it might be valuable to convert the value of the field to a numeric data type.

To check if a field might be numerical the sample data is iterated using the same approach as Algorithm 4.3. The discriminant function in this case is one that checks if a string value is accepted by the regular expression `"^((\+-)*[.,]*[.,]+)$"`—. The corresponding finite automaton is shown in Figure 4.4. This regular expression accepts numbers, signed numbers, numbers with comma as decimal mark, and numbers with punctuation mark as decimal mark. If a value is `null` it will be ignored and thus neither counted as a numeric value nor in the total number of values.

The relative frequency of numeric fields in the sample data set is added to the `object_metadata` under the key `p_number`.

Booleans disguised as strings

In some systems, Boolean values are represented by strings, or they may have been converted to strings at some point. Strings that represent boolean values are easy to find, at least if there is a limited set of strings that are expected to be interpreted as booleans. The algorithm that discovers possible boolean values run through the dataset using the same procedure as Algorithm 4.3, only here it checks if string values are in the lists of true or false boolean val-

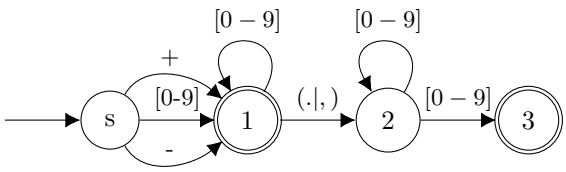

Figure 4.3: Non-deterministic finite automaton modelling the regular expression for recognising a number disguised as a string. The input string must start with a digit, plus sign, or minus sign. This moves the finite automaton to state 1, which is an accepting state. If there is a comma or punctuation mark, the automaton will move to state 2, which is not an accepting state. State 2 demands at least one number to move to state 3, which is an accepting state.

ues. According to the documentation for Elasticsearch [34], it would previously accept a list of values, `false`, `"false"`, `"off"`, `"no"`, `"0"`, `""` (empty string), `0`, `0.0`, interpreted as the Boolean value false. Any other value was interpreted to be the Boolean value true. This feature was deprecated in version 5.3. The strings from this list, however, are used to build the list used in this system, $TrueBooleans$ = {'true','t','yes','on','1'} and $FalseBooleans$ = {'false','f','no','off','0'}. These are not exhaustive lists of possible representations of boolean values, but they cover common variants.

If a field is of data type `number` and the sample data only has values 0 or 1 for the field, that will also cause it to be determined as a possible boolean value.

The empirical relative frequency across the sample data set of possible boolean values in a field, will be added to the metadata under the key `p_bool`.

Recognizing dates

Date or time values are usually represented as either a string or a number. In the case of string there are a lot of different formats depending not only on locality, different generating systems use different representations as well. Some values represent only the date while others contain the time as well. In the case of the value being a number, the most used format is to give the number of seconds since the UNIX epoch January 1st 1970 [35]. Some systems provide the time since the epoch in milliseconds. Recognising that a string represents a date is a tedious task, however most modern programming languages have some date parser built in, and most of them recognise dates on the most used formats. This thesis will thus use a date parser for Python called `dateutil` [36] to check if strings are dates.

The function that checks if a value might be a date will try to parse a date from the value using the `dateutil` date parser. If the parser is successful, it returns a 1, or true, if it is not successful a 0 or false is returned. The empirical relative frequency of a field being a date will be set on the object metadata under the key `p_date`.

If `p_date` is between 0.8 to1.0, the value of the metadata property `samplevalue` will be converted to unix time and stored in the metadata property `_date_unix`. This conversion to a unified format is done to make it easier to present the sample value in a human-readable format when presenting the data model.

If the field is a numeric type it might be a unix timestamp, and according to the unix time definition [35] all real numbers represent some time, e.g the number 1 is January 1st 1970

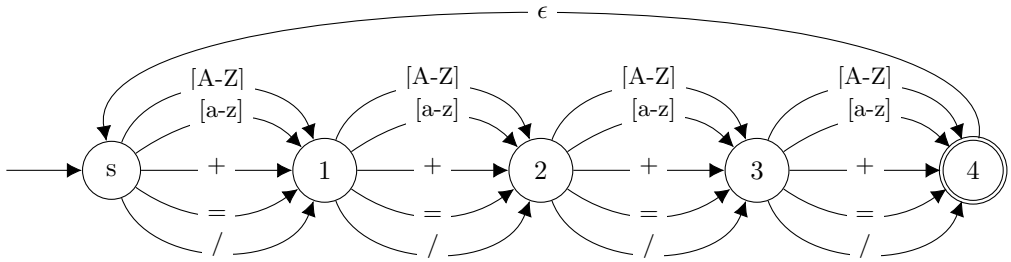

Figure 4.4: The regular expression that accepts strings that could be Base64 encoded binary strings, represented by a non deterministic finite automaton. The regular expression will accept any string that consists of the characters `A-Z`, `a-z`, `0-9`, `+`, `=` and `/`. The total number of characters in the input string must also be a multitude of 4.

00:00:01. However the number 1 might also be a boolean, a route number or any other numeric value. To handle this problem with a simple solution the number must be between 473385600 and 2051222400 to be considered a date. The result being that only numbers representing dates between January 1st 1985 and 2035 will be considered. This can lead to wrong assumptions, and should be checked extra carefully by the domain expert.

Bytes strings

Base encoding of data is to convert binary data to an ASCII string, so any binary data can be stored as a string. Base64 encoding uses 65 characters of US-ASCII character set, to encode 6 bits in each character, the 65th character is = and is used for padding [37]. The Base64 encoding process represents 24-bit groups of input as output strings of 4 encoded characters [37]. If the input is only 8 bits the rest will be padded with = [37]. The length of a valid Base64 string must then be a multiple of 4, and it can only consist of characters from the US-ASCII character set. To check if a string might be a Base64 encoded byte value, the string is tested against the regular expression `"^([A-Z0-9a-z=/\+]{4})+$"`, which accepts characters, numbers, + and =, in groups of four. This regular expression does not accept a string with less than four characters.

The relative frequency of fields that might possibly be strings representing byte values is added to the object with the key `p_bytes`.

Numbers with or without decimals.

An easy way to differentiate numeric values is if they have decimals or not. A number that is without decimals, e.g. an integer, performs better in most systems. Each numeric value in the sample data is checked for decimals. The empirical relative frequency of integers is stored under the object metadata key `p_integer`.

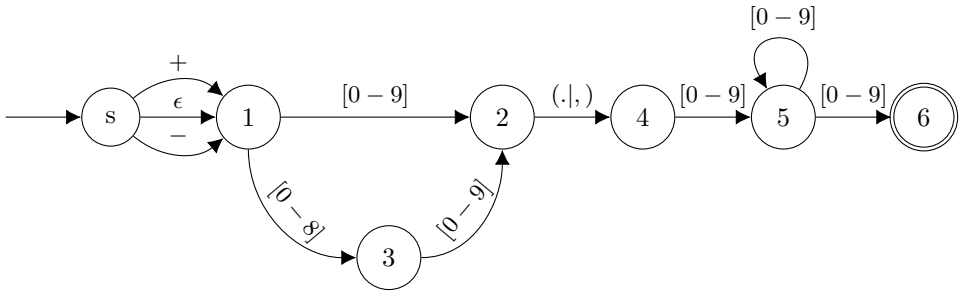

Figure 4.5: Non deterministic finite automaton modelling the regular expression for recognising a latitude coordinate. The input string can start with +, - or nothing to move to state 1. A digit between 0 and 9 will take it to state 2, and if the digit is between 0 and 8 it will also move to state 3. Should the next character be a punctuation mark or comma the automaton will move to (only) state 4, if it is a number it will move to (only) state 2. To get from state 4 to state 6 two or more digits between 0 and 9 are needed. State 6 is the accepting state.

Latitude and or longitude value.

Geographical locations represented by GPS coordinates, can be indexed in Elasticsearch using the data type `geo_point`. One benefit of this data type is that `geo_point` enables geographical searches. A string or a number can be a part of, or a complete representation of, a `geo_point`.

The coordinate reference system used by GeoJSON is WGS84 [38], and that is the type of values that this test will look for in string and number fields. WGS84 has its coordinate origin at what is believed to be the earths centre of mass. Zero longitude is 102.5 m east of the Greenwich meridian, and zero latitude is the equator. Any position on earth, can be given by the angle, in the longitude and latitude direction, between two lines that both start at the centre of the earth. One line goes through the zero longitude and zero latitude point, while the other goes through the position that is to be represented [39]. These two angles can be given in several formats, in Elasticsearch the numerical representation is used. Together with the altitude this gives an exact position. However, if the altitude is omitted, ground-level altitude is inferred.

To check if a field contains part of a coordinate, the simplest constraint is that it has to be a number. Latitude angles can be from -180.0 to 180.0 degrees, while longitude can only be from -90.0 to 90.0 degrees. This approach will also assume that a coordinate will be given with two or more decimals. If a value can be a latitude, its string value must be accepted by the regular expression `^(\+|-)?([0-9]|[0-8][0-9])\.{1}[0-9]{2,}$`. It must consist of a number larger than -90 and lower than 90, and have at least 2 decimals. Figure 4.5 shows the nondeterministic finite automaton representing the regular expression. For longitude the regular expression will be very similar, but now it will accept values between -180.0 and 180.0,

`^(\+|-)?([0-9]|[0-9][0-9]|[0-1][0-9][0-9])\.{1}[0-9]{2,}$`.

One important thing to note here is that all strings that match for latitude will also match for longitude. Another approach will be needed to decide which is which.

The relative frequencies of fields that might be latitude and longitudes will be stored in the object metadata under `p_lat` and `p_lon` respectively.

Figure 4.6: Example of a box plot. The horizontal lines on the left and right side represent the min and max values. The grey box is defined by the fifth and ninety fifth percentile values, and the red horizontal line represents the median. It is easy to see that the values are relatively evenly distributed and the median, is in the middle of the figure. This example was generated from the values min: 1495540962 max: 1495541000 mean: 1495540981 median: 1495540981 variance: 240.6667 5th percentile: 1495540964 95th percentile: 1495540998.

Elasticsearch also accepts a string with latitude and longitude values separated by a comma as a valid `geo_point`. These values are found by grouping the regular expressions and require a comma between them.

The relative frequency of values that might be the string representation of a `geo_point`, are stored in the object metadata under `p_geo_point_string`.

Uniqueness

How many unique or distinct values a field has can be a good measure for discovering fields that represent categories or that otherwise can be used for grouping. Uniqueness can also reveal fields where all the values are the same. A measure called uniqueness [40], is given by the number of distinct values divided by the total number of values. This is the same as the empirical relative frequency of unique values.

The algorithms suggested for this in [40] use either hashing or sorting. To sort all the values of the field, they would need to be stored in memory, full length. This can be very memory intensive. Adding just the hashes of the values to a set would require less memory, but more CPU power. In the analysis of the sample data set, capacity for a bigger sample data set is more important than the speed of the process, so hashing the values is definitely the best procedure.

Three values are stored to the `object_metadata`, `unique_num`, the number of unique values, `unique_total_values` the total number of values counted, and the `unique_uniqueness` the former divided by the latter.

4.3.2 Summary statistics for box plots

One way of visualising the statistical properties of a numeric value is through a box plot, such as in Figure 4.6. The box plot is useful in the web interface, for the domain expert to get a quick sanity check and an overview of the field's values in the sample data set. A box plot will intuitively reveal if there are outliers in the values, if the values are evenly distributed around the median, how widely spread the values are compared to the minimum and maximum borders, and more. Summary statistics can also be created for aggregated values, like the number of characters, or number of words in a string field.

To create a box plot describing a set of values one needs to know the minimum and maximum values, the mean value, the median, and the 5th and 95th percentiles. Given a set of values $X = x_0, x_1, \ldots, x_n$, the minimum value is the value in the set that has the lowest value, and

the maximum value is the value with the highest value. The mean value $\bar{x}$ is given by

$$\bar{x} = \frac{\sum_{x \in X} x}{n} \tag{4.3}$$

The median, $\tilde{x}$, is the middle element, and if n is odd, the median is given by

$$\tilde{x} = x_{\frac{n}{2}} \tag{4.4}$$

If n is even, the median is given by

$$\tilde{x} = \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2} \tag{4.5}$$

The 5th percentile, $5p$, is the value that 5% of the samples are less than, and is given by

$$5p = x_{\lceil n*0.05 \rceil} \tag{4.6}$$

Accordingly the 95th percentile, $95p$, is the value that 95% of the samples are less than, and is given by

$$95p = x_{\lceil n*0.95 \rceil} \tag{4.7}$$

Summary statistics for numeric fields

For a numeric field all the values are are calculated and stored as one JSON object under the key `numeric_value_summary` in the object metadata. This provides data to show a box plot in the web interface as well as the statistics.

String length summary statistics

How long the strings are can give contribute to intuitive understanding of a field, in further analysis of the dataset. To find the summary of the statistics of the string length, all the string lengths for a given field in the sample data are collected, and the values are calculated and stored like in section 4.3.2, and stored under the key `string_length_summary`.

Number of words in strings summary statistics

The number of words in a string can also contribute to the understanding of a field, and can help decide whether the string represents a keyword or is a full text field. Counting the number of words in a string is done by splitting it on space and counting the number of results. The values are calculated and stored like in section 4.3.2, under the key `word_count_summary`.

4.4 Inferring Elasticsearch data types

Once each field in the model has been analysed and metadata values added, there are some Elasticsearch datatypes that are possible matches and some that are not possible matches. By analysing each field for match with each Elasticsearch data type, an ordered list of possible matches is made, and the data types that are not possible matches are discarded. The following section will go through the data types defined by the JSON format, and present an approach for how to generate a list of possible matching Elasticsearch types for data of that JSON data type.

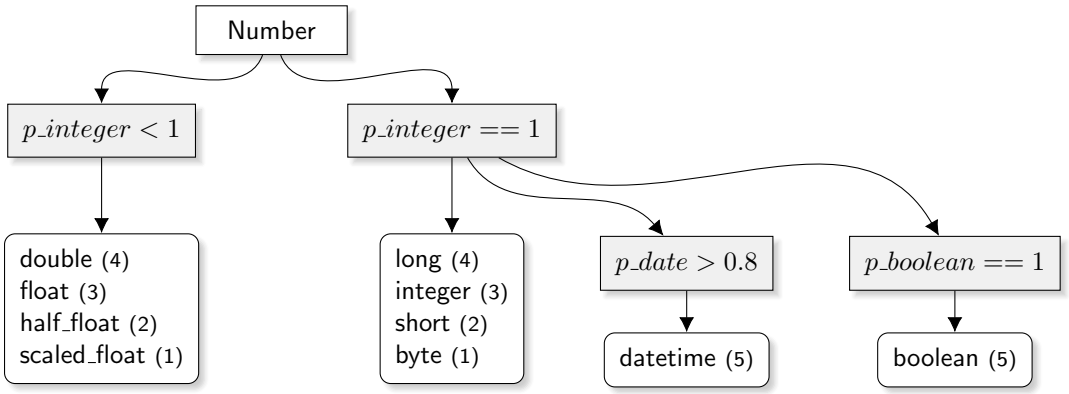

Figure 4.7: Identification tree for a JSON number type field. The grey boxes are tests, based on previous analysis. The numbers in parenthesis are the weight of each suggestion. For example, if the number is not an integer, the best suggestion is double because it has the highest weight. The rest of the types in that box are also possibilities.

4.4.1 Boolean type

If a field is of type Boolean in the JSON representation, it is very probable that it is best stored as a Boolean in Elasticsearch. However, one might want to store it as a keyword or a text. The list of possible Elasticsearch field types for a JSON boolean field is *Boolean* on top, then *keyword* as a second possibility, and *text* in case the domain expert insists.

4.4.2 Number type

If a data field has original JSON data type `number` there are several possible Elasticsearch type matches. Figure 4.7 shows an identification tree for JSON values. The gray boxes represent constraints and the white rounded boxes show the possible data types that will be added should they be reached. The number behind the data types is the weight of the suggestion. The weight of a suggested data type is a number that is used to sort the suggestions, i.e. the data type with the higher weight is more probable. It shows that if `p_integer` is less than 1, i.e. there is at least one value that has decimals, the list of floating point data types is suggested. If `p_integer` is 1, the list of integer data types is added, and `p_date` and `p_boolean` is tested as well.

The list of integer number types is ordered after maximum number capacity in descending order, so if the top one is chosen that will accept the largest range of numbers. If the domain expert knows that values in a field will never be larger than the maximum value of a `short` in Elasticsearch, choosing this will give better performance. If `p_date` is bigger than 0.8, the data type date is added to the list of suggestions. `p_date` only has to be bigger than 0.8 because dates are often misinterpreted by the date parser. Since this suggestion has weight 5, it will be added above the integer types. There is still a possibility that the field represents a Boolean value, and if `p_boolean` is 1, the data type `boolean` will be added to the top of the list.

4.4.3 Array type.

In Elasticsearch, there is no explicit data type "array". Instead all types can contain lists of values, as long as they are of the same type [26]. In other words, the Elasticsearch data type of a field with arrays of values, should be the data type shared by all the values in the array.

However, in the case of an array containing objects this might not be the best approach. Lucene core does not support nested objects. An array of objects will be flattened internally, and each data field of the inner object will become one array containing the values from all the object's corresponding fields. Thus the relationship internally in each object is lost.

For example, take the case of storing an object that represents two bus trips that are planned for the next hour. Each has an origin and a destination field. The request to store the object is given in Example 4.3. This example has been adapted from [41].

```
PUT trips/batch/1
{
  "group" : "planned-trips-next-hour",
  "trips" : [
    {
      "origin" : "Haugesund",
      "destination" :  "Aksdal"
    },
    {
      "origin" : "Forus",
      "destination" :  "Stavanger"
    }
  ]
}
```

Example 4.3: JSON structure for the indexing of an array of two trips, each with an origin and a destination.

Because of the Lucene core, Elasticsearch will flatten these and store them as in Example 4.4.

```
"group" : "planned-trips-next-hour",
"trips.origin" : [ "Haugesund", "Forus" ],
"trips.destination" : [ "Aksdal", "Stavanger" ]
```

Example 4.4: Example of how Elasticsearch would store the request from Example 4.3

The relationships inside the independent objects are then removed. A search for a trip from Stavanger to Haugesund, i.e. a trip with `origin` Stavanger and `destination` Haugesund, would be successful, even though the original data specifies no such trip.

To avoid this type of behaviour, and make each object in the array searchable and independent, there are two possible approaches. One is the Elasticsearch `nested` data type, and the other is Logstash `split` plugin. In the case of Elasticsearch's `nested` data type, Elasticsearch will put each trip in separate hidden documents and search them when the main object is queried in a nested search. Through this approach it would appear to the user that the trips

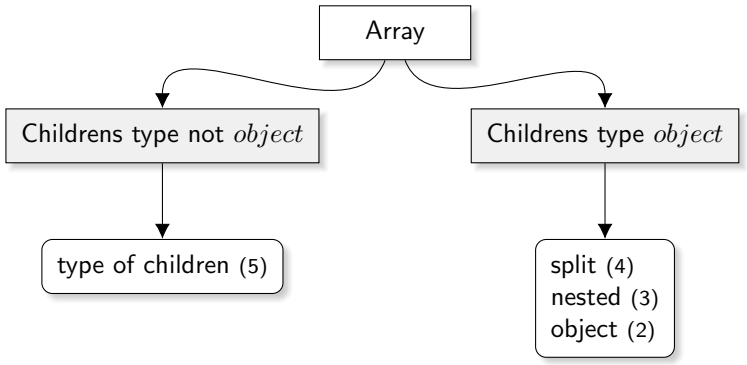

Figure 4.8: Identification tree for a JSON array type field. If the objects contained in the array are of type `object`, suggest `split`, `nested` or `object`, if not, use the type of the objects contained in the array.

are objects in a list that is a field on the parent object. However, when used exhaustively, this approach can damage performance.

Logstash's `split` plugin is not a data type, but a filter plugin for Logstash that will split the document into two separate documents, duplicating all properties that the documents are expected to have in common. However, for simplicity in the development of this thesis, `split` will be considered to be a special case data type. The result of the `split` plugin on this example is two independent objects with the same schema, but different values for the fields `trips.origin` and `trips.destination`. The result is shown in Example 4.6.

```
{
"group" : "planned-trips-next-hour",
"trips.origin" : "Haugesund",
"trips.destination" : "Aksdal"
},
{
"group" : "planned-trips-next-hour",
"trips.origin" : "Forus",
"trips.destination" : "Stavanger"
}
```

Example 4.5: Example of how Logstash's `split` plugin would transform the request from Example 4.3

The possible Elasticsearch types for what is an array field in JSON is `split` with the highest weight, `nested` as the alternative, and `object` lowest.

4.4.4 Object type.

A field with JSON data type `object` can be have data type `object` in Elasticsearch too, even though it will be flattened, Elasticsearch maintains the object properties. However, when it is an object in the JSON representation, it could also be a representation of a value that would be better stored as one of the Elasticsearch data types `range`, `geo_point` or `geo_shape`. A `range` in Elasticsearch is an object that represents a range on some scale, by defining specific boundaries. A `geo_point` in Elasticsearch is a type that represents a geographical location, while a `geo_shape` represents a geographic area.

If the `object` is an Elasticsearch data type `range`, it must either be `integer`, `float`, `long`, `double` or `date range`. Ranges are recognised by the properties they must contain, the range query parameters. The parameters are "greater than or equal", "less than or equal", "greater than" and "less than". These are abbreviated *gte*, *lte*, *gt* and *lt* respectively. The data type of the range is defined by the data types of the range boundary parameter values.

The object might also be a `geo_point`. Elasticsearch accepts `geo_point`s in four different formats:

- An object containing a `lat` and a `lon` property.
- A string in the format `lat,lon`.
- A geohash string.
- An array with `lat` as the first item and then `lon`.

If the object is an object representation of a `geo_point` it must have one field with $p\_lat == 1$ and one with $p\_lon == 1$. The two properties must also have names that contains 'lat' for the latitude, and 'lon' or 'lng' for the longitude.

An object might also be one of the `geo_shape` data type. Elasticsearch accepts `geo_shape`s encoded by the GeoJSON format. According to [38] all GeoJSON object types that are accepted by Elasticsearch have a data field `type` that describes the GeoJSON type of the object, e.g. `line`, `polygon`, or `rectangle`. If the object in question has a field named `type` and the value of that field exists in the list of GeoJSON types accepted by Elasticsearch then `geo_shape` will be suggested as an Elasticsearch data type for the object.

The identification tree for the JSON type `object` is shown in Figure 4.9.

4.4.5 String type.

Fields that are strings in the JSON representation of the data have the largest list of possible Elasticsearch types. A string can be a `text`, `keyword`, `date`, a `number`, `boolean`, `range`, `geo_point` or `geo_shape`.

The identification tree for strings is shown in 4.10. If the `p_number` is 1, then the sample data values should be converted to numbers and run through the analysers again. The resulting suggestions should be added to this list of suggestions. The same approach goes for

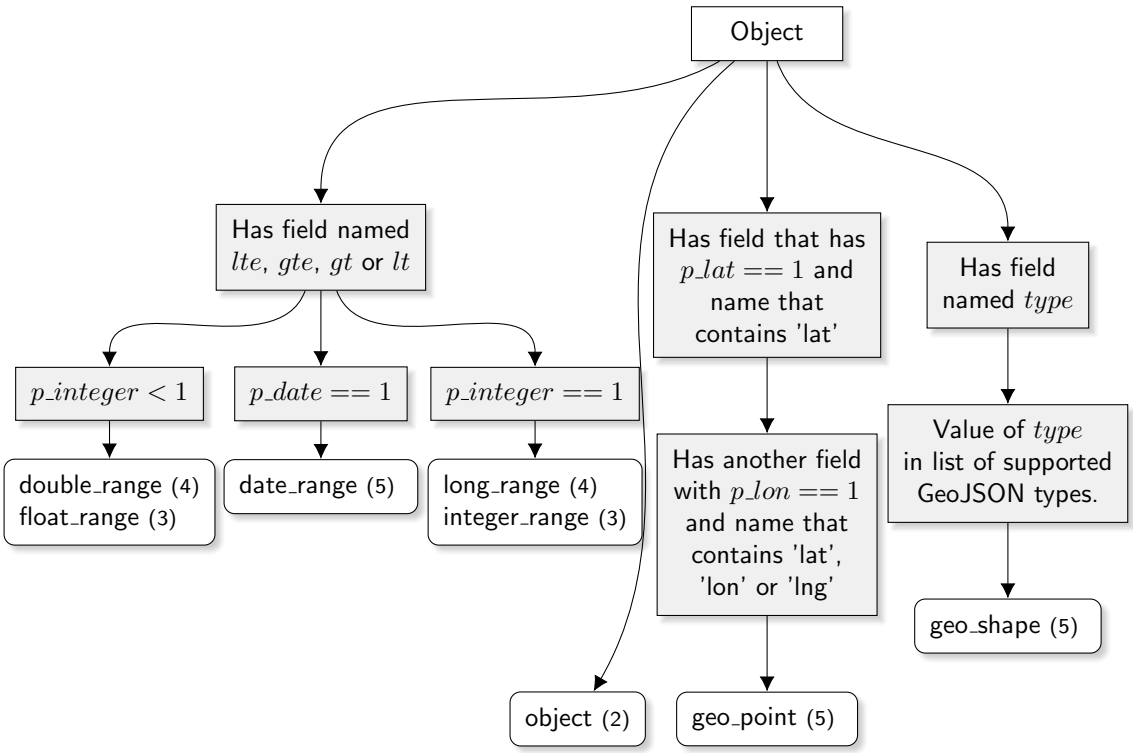

Figure 4.9: Identification tree for a JSON object type field. If the object has properties named `lte`, `gte`, `gt` or `lt` it is probably some kind of range. If it has a property type with a value corresponding to one of the geo shapes accepted by Elasticsearch it is probably a geo shape. Checking *p_lat* and *p_lon* can tell if this is a geo point, and at the bottom of the list is always the object type.

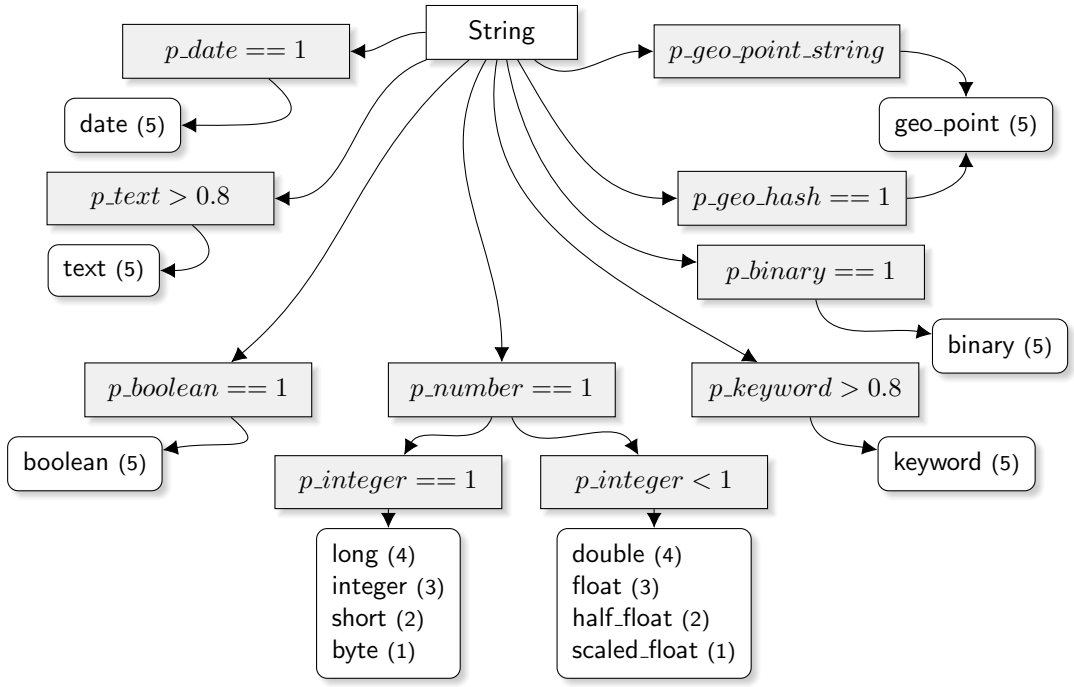

Figure 4.10: Identification tree for a JSON string type field. The relative frequencies determine the data type suggestions, if $p\_date == 1$ `date` is suggested with weight 5. p_text and $p\_keyword$ need only be more than 0.8 to suggest `text` and `keyword` respectively.

4.5 Presenting the data model

When a data model has been created, and its metadata has been populated with results from analysers, it can be presented to the user for review. The starting point for any use of the system is the command line interface. For example, the web interface is started from the command line interface. Through it, the domain expert can review and edit the data model. When the data model is complete a Logstash filter configuration and an Elasticsearch index mapping request can be generated.

4.5.1 Command line interface

All actions in the system are started from the command line interface. All interaction with the command line interface is dependent on the current working directory, i.e. the directory from which the command is run. Any command will use the current working directory to save or load the data model file. For example, to create a new data model, move to the directory where the data model file should be stored and run `sdmcli init`. An empty model will be created in that directory. The data models are always named `sdmodel.json`, therefore two data models can never reside in the same directory. The model file represents the current state of the system, all other events in the system happen in separate and independent operations, and all store their results in the model file.

Discovering a new model from sample data is done from the command line interface. It takes the location of a JSON file that contains an array of documents as an argument. The model file will be stored in the folder the command was run from. To review the discoveries of the discoverer, one could print to screen the data model in the command line window, or start the web interface.

4.5.2 Web interface

Starting the web interface starts a local web server. It takes the data model as a JSON file and serves this through a REST API. The API has an endpoint `/model` through which the data model can be interacted with. It can be retrieved by a GET request to this endpoint, and edited by sending an edited version as the body of a PUT request. To create a new data object in the model, the POST method is used. It is also possible to delete an object using the DELETE method, but in most cases it is better to mark the data object as deleted.

The server also serves a Javascript front end application that uses the REST API and lets the user interact with the data model.

Viewing the model

The data model is presented in the web interface as a tree containing data objects. At the root is the `_root_object`. An example of a data object is shown in Figure 4.11. All data objects has an *Original Name* that is the field name in the JSON representation. *New Name* is initially set to the same as the original name, and can be edited to give the field a different name in storage. The field *JSON Type* shows the JSON data type for the field, while *Elasticsearch type* shows the top suggestion for Elasticsearch data types.

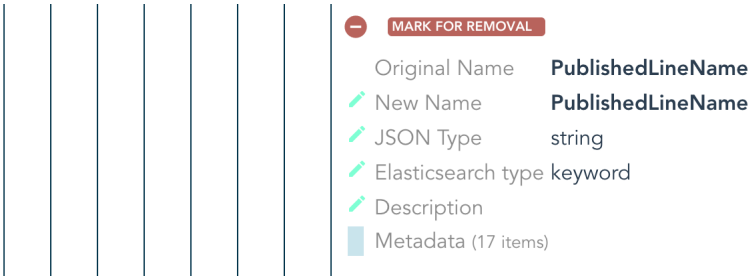

Figure 4.11: Example of a data field represented in web ui. The vertical lines represent parent objects. The original name of this field is "PublishedLineName". The field "New Name" can be edited to give the field a new name in storage. The JSON type of the field is `string` and the system has suggested `keyword` as the best choice for Elasticsearch data type. A description can be added and there are 17 `object_metadata` items.

Metadata values are viewed as they are, however some metadata properties have graphical representations, for example summary statistics is presented as a box plot. There is also a set of 'internal metadata' properties, which are hidden by default, but can easily be made visible. Figure 4.12 shows the data object from Figure 4.11 with the metadata section expanded and some values edited.

Making changes to the model

Some of the values in the data model can be edited in the web interface by clicking the pen icon next to the field in question. When a field is edited, and the save button is clicked, the new edition of the data object is sent to the backend and then the interface is updated. The object is sent through the REST API by an Ajax (Asynchronous Javascript and XML) engine that handles the transaction with the server without blocking the user interface. While waiting for the request to be returned however, the field that has been edited is blocked from further editing. The back end handles an update request directly on the model file and saves this to disk for every request. Once the request is returned, the front end will reload the model from the back end. In Figure 4.12, the *New Name* has been changed to "PublishedRouteIdentifier", and this is the name it will be stored as. A short description of what the field represents has also been added.

4.6 Generating configurations

Configuration files for Elasticstack and Logstash can be generated from the web interface. Since the data model stored on disk always represents the current state of the system, the configuration files is generated in the backend with minimal input from the user.

4.6.1 Filter section of Logstash configuration file

The user can request the filter section of a Logstash configuration from the web interface without passing any arguments. Since input and output parts of the configuration file mainly con-

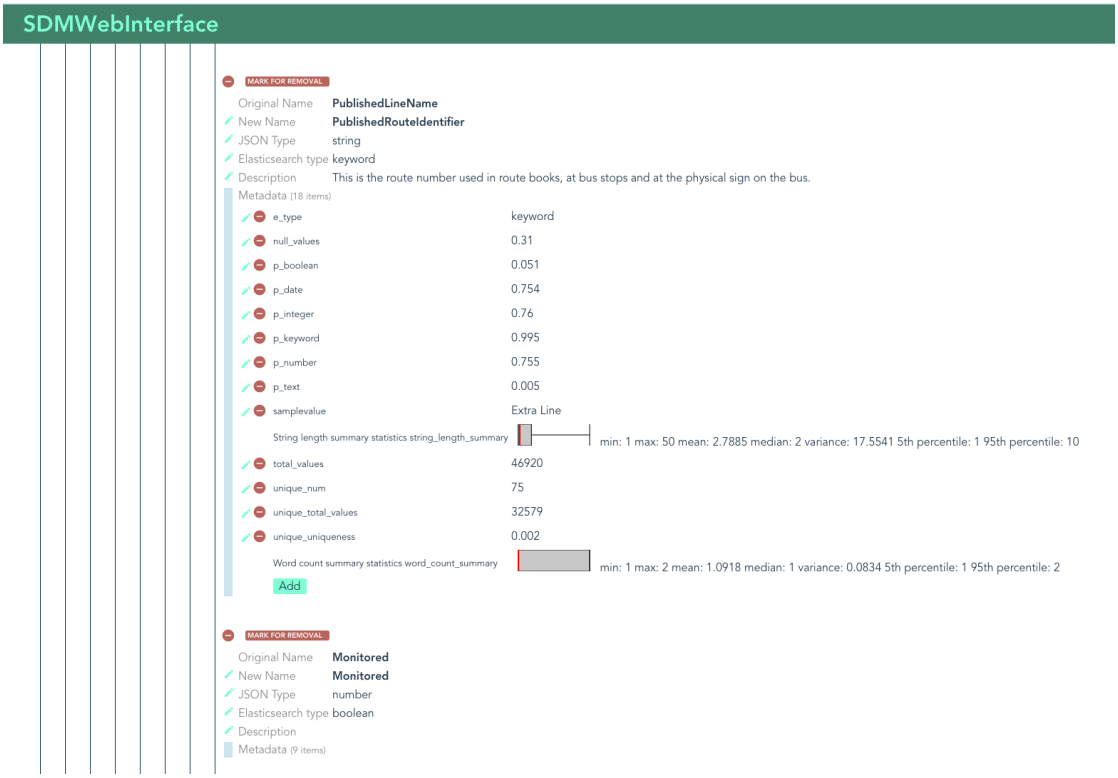

Figure 4.12: Example of a data field represented in web ui, with the "Metadata" field expanded. In this case the "New Name" has been changed, and this field will get the name "PublishedRouteIdentifier" upon storage. The estimated probabilities are visible in the metadata section, as they have been presented previously in this chapter. Since this field has data type `string` in the sample data there is two box plots in the metadata. String length summary statistics show that most values have very few characters, while only a few values have many characters. From the word count box plot it is obvious that there are few words in each of the values, over half have only one word, and a value has at most two words.

tain data provider and data storage specific information, it is not included in the output.

The Logstash filter generator will iterate over the data model recursively several times and add configuration snippets to the string that will become the response. The first step is to traverse the model recursively looking for arrays where the `split` option is chosen. This must be done first for the addressing of underlying elements to work correctly later. An example of the snippet that is added to the filter configuration is shown in Example 4.6.

```
if [Answer] {
  split {
    field => "[Answer][VehicleMonitoringDelivery]"
  }
}
```

Example 4.6: An example of usage of the Logstash `split` plugin. First it checks that the field `[Answer]` exists, if it does, each value in the `array` at `[Answer][VehicleMonitoringDelivery]` is split into its own document before storing.

The second step is the transformations. Snippets configuring the transformation of any data field that require explicit data type transformation will be added to the filters string at this step. For example if a field is a `string` in the JSON representation and `integer` is chosen to be the Elasticsearch data type. An example of the string that is added to the filter configuration is shown in Example 4.7. The snippet in Example 4.7 will convert all the values of the data field `[Answer][StopPointRef]` to integers.

```
mutate {
  convert => {
    "[Answer][StopPointRef]" => "integer"
  }
}
```

Example 4.7: An example of usage of the Logstash `mutate` plugin. The value at `[Answer][StopPointRef]` is explicitly converted to an `integer`.

Dates are also in some cases transformed. If the date in the JSON representation is given in UNIX time, i.e. seconds from the epoch, this must be converted to milliseconds from the epoch before Elasticsearch can use them. Example 4.8 shows an example of the string that is added for such dates. The line starting with `match` defines which data field to match on, and what plugin-specific data type the data field is expected to have. The line starting with `target` is required to store the resulting value in the same field.

```
date {
  match => ["[Answer][DestinationAimedArrivalTime]", "UNIX"]
  target => "[Answer][DestinationAimedArrivalTime]"
}
```

Example 4.8: An example of usage of the Logstash `date` plugin. The line starting with `match` defines that the `date` plugin should act upon fields named `[Answer][DestinationAimedArrivalTime]` and expect it to be of type `UNIX`. This type is specific to this plugin. The line starting with target sets the output of the plugin, storing it in the same field must be defined explicitly.

If the name of a data object has been changed in the web interface, that must result in a transformation in the Logstash filter configuration. To rename a field, the `mutate` plugin is used. The `rename` operation takes as arguments a key-value list, where the current name is key, and the new name is value. An example is shown in Example 4.9.

```
mutate {
  rename => {
    "[VehicleLocation][Longitude]" => "[VehicleLocation][lon]"
  }
}
```

Example 4.9: An example of renaming a field using the Logstash `mutate` plugin. The plugin has an operation called rename that takes a key-value list as argument. Here with the current name `[VehicleLocation][Longitude]` as key, and the new name `[VehicleLocation][lon]` as its value.

In the web interface each object can be 'marked for removal', the result is a metadata property `_marked_for_removal` that is set to true. If this metadata property exitsts and is set to true, the field and its value is to be removed in the Logstash filter. This is also a job for the `mutate` plugin. The operation `remove_field` takes one string argument. An example of removing a field can be seen in Example 4.10.

```
mutate {
  remove_field => "[VehicleLocation][srsName]"
}
```

Example 4.10: An example of removing a field using the Logstash `mutate` plugin. The plugin has an operation called remove that takes a string as argument. Here the field with the name `[VehicleLocation][srsName]` will be removed from all data points.

4.6.2 Elasticsearch mappings

Elasticsearch mappings can also be requested from the web interface. However, in this case, the user must pass the wanted 'index name' and 'document type name' as arguments. The

mappings are a JSON-encoded object containing an object for each of the fields in the data model. The object have a field named 'type' that contains the data type of the data that is expected in that field. It also has a field 'properties' that is a list of objects representing any child objects. Generating these mappings is done by traversing the data model, and adding each of the data objects from the data model representation to the JSON object. If the data object has an Elasticsearch type that corresponds to one of the Elasticsearch types that expect object data types, like `range` or `geo_point` the recursion will stop. The content of such objects can not be defined in the mappings, but are defined implicitly by the data type.

```
{
  "mappings": {
    "exampletype": {
      "properties": {
        "RecordedAt": {
          "type": "date"
        },
        "BatteryLevel": {
          "type": "long"
        },
        "DeviceType": {
          "type": "keyword"
        },
        "Temperature": {
          "type": "long"
        },
        "Location": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

Example 4.11: An example of an Elasticsearch mapping creation request that was generated by the system. The document type that the mappings are created for is named `exampletype`.

Chapter 5

Implementation

A system for automatic data exploration and configuration generation will in some cases be required to process large amounts of data. There might also be several domain experts cooperating on the same data set configuration. Therefore, such a system needs to be portable in terms of running environment. It also might be beneficial if the system could be operated from another machine than the one that it runs on.

This chapter presents an overview of the implementation of the SDModel software system in Section 5.1. The process of building up the data model is presented in Section 5.2. Section 5.3 presents the data type inferrer, and its implementation. Section 5.4 presents the implementation and use of the command line interface from where all use of the system start. The web interface is presented in Section 5.5, before the output plugins' implementation is presented in Section 5.6.

5.1 Overview of the implementation

The SDModel system consists of two parts, a back end that does the heavy lifting, and a front end web interface for controlling the back end. The back end is implemented in Python. A third party Python package CherryPy is used to serve a REST API and the front end app. The web interface is written in HTML and JavaScript. It uses the JavaScript library Vue.js, with a set of additional packages.

Figure 5.1 shows an implementation overview of the system. The sample data is first used by an instance of the `SDMDiscoverPlugin` to discover the schema and do statistical analysis. `SDMDiscoverPlugin` inherits from `SDMInputPlugin`, therefore the discover plugin can be invoked from the web interface. The sample data and data model is then run through an instance of the `SDMEtypePlugin`, which creates a list of possible Elasticsearch data types, based on the analysis done by the `SDMDiscoverPlugin`.

When a data model is created or loaded from file, it is made into an instance of the `SDModel` class. The `SDModel` class inherits from `SDMDataObject`. An UML diagram of the relationship between `SDModel` and `SDMDataObject` can be seen in Figure 5.2. The `SDModel` instance is the root object in the data model, and the main difference, separating it from the `SDMDataObject` is its ability to save and load to and from file.

An instance of the class `SDMServer` exposes a REST API that handles serving and editing the data model. The input plugins communicate through a REST API exposed by an instance of the `SDMPluginServer`, and the output plugins using `SDMOutputPluginServer`. All server classes use functionality from the CherryPy package.

The output formats are defined by output plugins that inherit from the `SDMOutputPlugin`. The class `SDMElasticsearchOutputPlugin` generates the request needed to create a mapping for the data in Elasticsearch. The `SDMLogstashOutputPlugin` generates the filter section of a Logstash configuration file.

CherryPy also serves the static files that make up the front end app. The Javascript front end app is built by Node using the JavaScript module bundler Webpack that builds the Vue.js app from the Javascript source files.

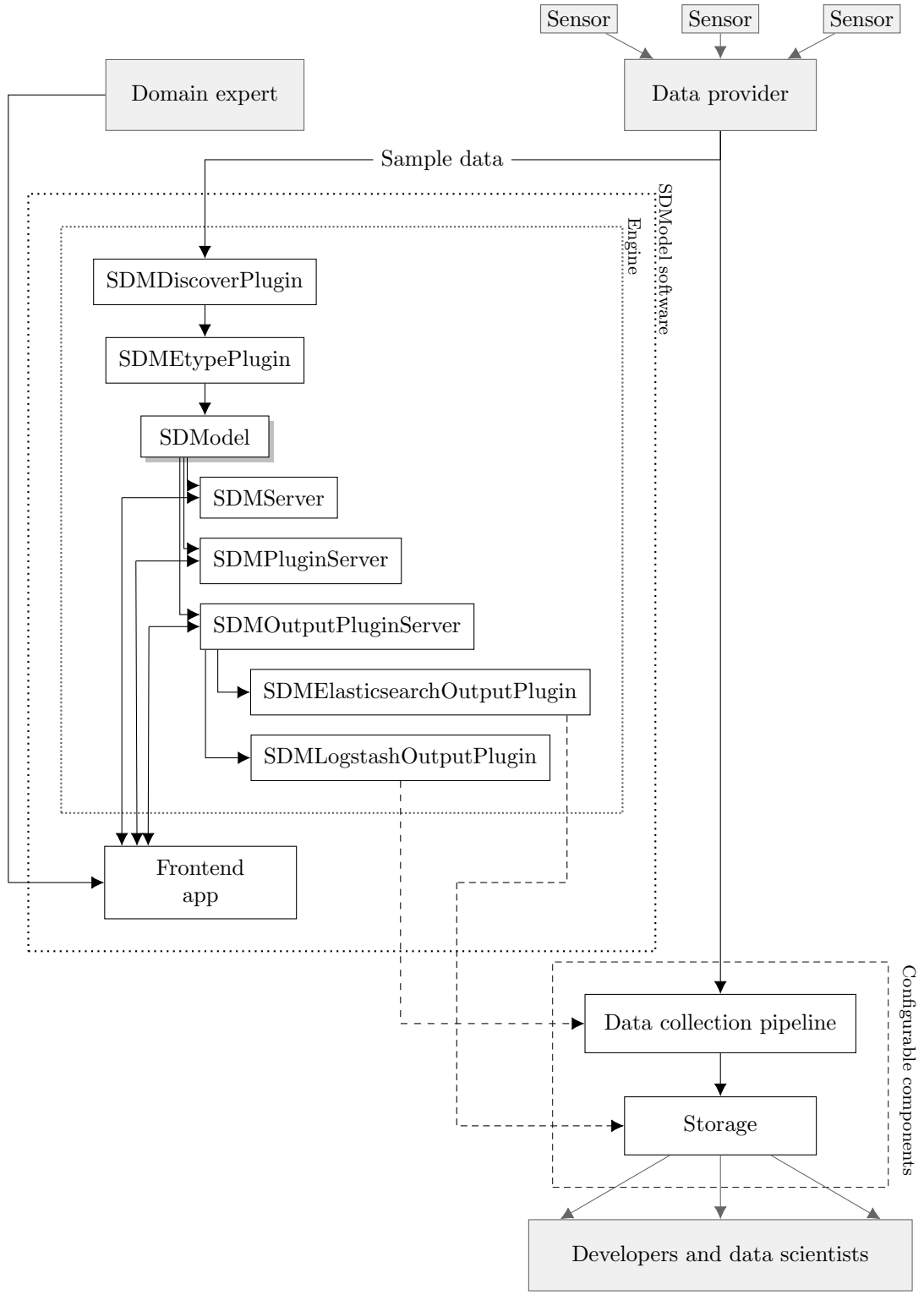

Figure 5.1: The implementation overview of the system. The data provider sends some sample data to the engine that is passed to the `SDMDiscoverPlugin`. It discovers the schema and populates a data model with results from analysers. The preliminary data model and data is then passed to `SDMEtypePlugin`, which suggest a list of Elasticsearch types for each data field in the model. The data model, now an instance of the `SDModel` class, is then served served through the `SDMServer`. The frontend app lets domain experts review and edit the data model. The `SDMPluginServer` and `SDMOuptutPluginServer` serve, and handle the use of, a lists of available plugins. The output plugins generate configurations from the data model. The domain expert can download, review, and add these to the data collection pipeline and storage manually.

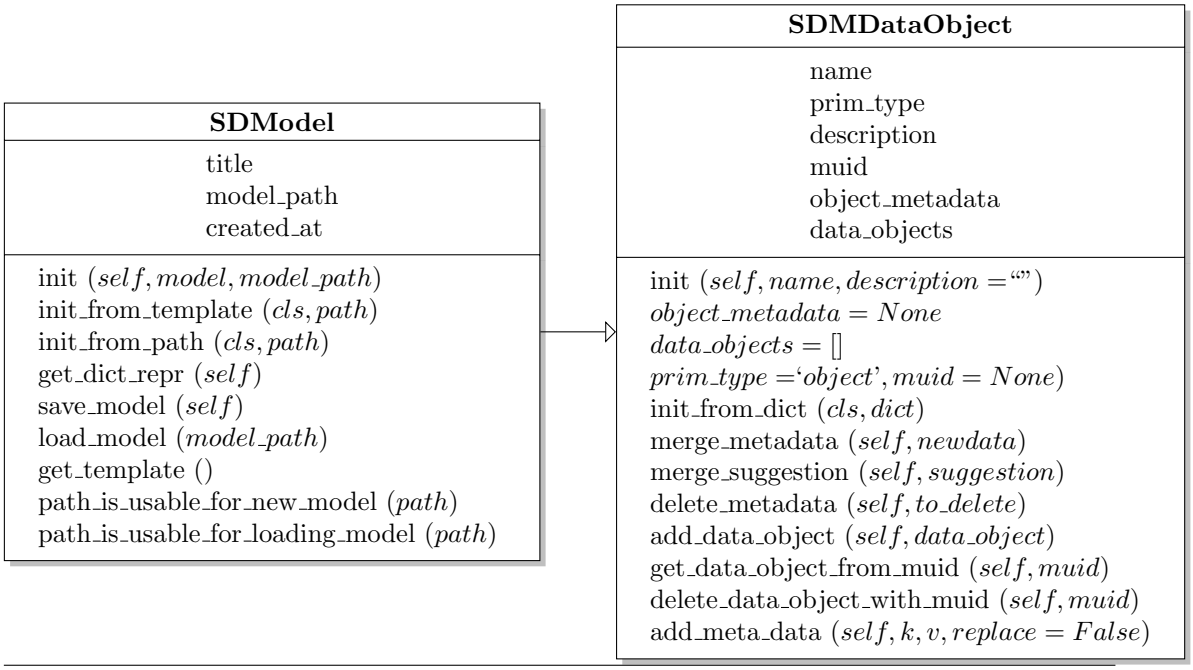

Figure 5.2: UML diagram showing the classes SDModel and SDMDataObject and their relation.

5.2 Building up the data model

An instance of the class `SDMDataObject` represent a data object in the data model. A data object is the data model representation of one data field from the sample data. When the schema of the sample data set is discovered, the result is the data model represented by an instance of the `SDModel` class, which has a list of child objects that are instances of the `SDMDataObject` class. The data objects can also have a list of child objects, which are instances of the `SDMDataObject` class.

5.2.1 SDMDataObject

An instance of `SDMDataObject` have an instance variable `name`, which reflects the field name in the sample data. It also has a variable `prim_type` that contains a string that identifies the JSON data type of fields values in the sample data. These types must be one of the following values, `string`, `number`, `boolean`, `array` or `object` in addition to `null` [33].

If the field is of type `object` it can have other data objects in its list of data objects. However, it may also reflect a field which has a `string`, `number`, `boolean` or `null` value, in which the data objects list is empty. If the field is of type `array`, it will have one data object representing all the items in the `array`. The data object representing such members of an `array`, will have an empty string as its name.

The class `SDMDataObject` also defines a variable `description`. It is meant to hold a short human readable description of the data object and what it represents.

Instances of the `SDMDataObject` class also have the variable `object_metadata`, a list of key-value pairs that can hold information about the data object. The key must be unique to the data object, and the value can be any of the JSON defined types. When analysers find information about a data field it will be stored in the list of object metadata. Metadata pairs that are intended only for use by the system itself and that normally need not be exposed to the user has their keys prefixed by an underscore, inspired by the Python convention for private variables [42]. These metadata pairs are hidden from the user by default, but can easily be displayed for a more verbose user interface.

Every data object also has the variable `muid`, a model unique identifier. When a data object is first initialised, a Universally Unique IDentifier (UUID) Version 4, is generated. This is is a 128-bit number that is pseudo randomly generated and that "guarantee uniqueness across space and time" [43]. There are five versions of UUIDs specified in RFC4122 [43], and they differ on what kind of input they use. Version 1 uses the current time and MAC address, version 2 is very similar to 1, but replaces a random part with a local domain value that is meant to reflect the system that generated the value. Versions 3 and 5 are name based and thus embeds some name, often used in distributed systems with some node name as the name preventing two nodes producing the same UUID at the same time. Version 3 uses the MD5 hashing algorithm version 5 uses SHA-1. Version 4 uses only a pseudo random value to generate the UUID. Since this system runs on one node version 4 has been chosen. Version 1 would also work.

A new instance of `SDMDataObject` can be initialised either with just a name, or with any of the values described above. There is also a class method for creating an instance from a dictionary, making initialisation from JSON a lot simpler. By using the `@classmethod` decorator, a method is treated as a static method that has access to the class they belong to through the first argument to the function. This argument is by convention named `cls`. The static method can then initialise and return an instance of the class, i.e. act as an initialiser for the class. This is the Python approach for 'overloading constructors', as seen for example in Java. In Java, two constructor methods can have the same name but different input parameters. This enables the software to let the data type of the arguments determine which constructor to use, and thus how to instantiate the class. In Python, method names must be unique to the class, and a class can only have one instantiating method, the `__init__()` method. However, by using the `@classmethod` decorator on a method, it can act as an initialising method, and return a new instance of the class, but take different input arguments.

Retrieving a data object that is child of an instance of `SDMDataObject`, can be done either by its `name`, or by `muid`. Because the `name` of a data object is unique only within the parent object's list, only the current data object's list of data objects will be searched when searching by `name`. A NameError is thrown if it is not found. The `muid` however, is unique in the entire data model. If the object is not found within the current object, the method is called on all its data objects, so that all the data objects in data model will be searched for the object with the correct `muid`.

The `SDMDataObject` has a method for JSON serialisation `__jsonobj__` that can be used to serialise the instance. This ensures the entire instance is serialised and unserialised correctly.

5.2.2 SDModel

The `SDModel` class inherits from `SDMDataObject` and is the root object in any data model. It can be initialised either with a path to an existing model or as an empty model. The additions

that this class have over the `SDMDataObject` class are the ability to save and load itself from file. When the `SDModel` writes itself to the `sdmodel.json` file it uses its own and its child objects' `__jsonobj__` methods.

For JSON serialising, the Python `json` package is used. The result from serialising with this library is by default a JSON structure with no whitespace and no newlines [44]. This gives a little increase in performance for machine readability. In this implementation, the indentation is set to 4, making the serialiser add newlines and 4 spaces for each indentation. This contributes to making the file more human readable.

5.2.3 Schema discovery

Discovering the schema of the sample data is done by the `SDMDiscoverPlugin`. It iterates over all the data points in the sample data, and passes each data point to a method `_discover_schema` together with an instance of the `SDMDataObject` class (or one that inherits from it), named `parent_object`. The first call to `_discover_schema` is done with an empty `parent_object`. The method then traverses the data object recursively in a depth first manner. An instance of `SDMDataObject` is instantiated for each of the data fields. The name and JSON data type of the field is set, and the instance is added to the list of data objects on the parent object. If a field is of type `object`, each of its data fields are passed recursively to the `_discover_schema` method. The data object, i.e. the instance of the `SDMDataObject`, that was created for the object is passed as the `parent_object`.

The `_discover_schema` function will first check if the object is already discovered in the model before adding it. If the `parent_object` has a data object corresponding to the data field, it will verify that the data type is the same and move on.

5.2.4 Analysing data and adding metadata to the model

Once the schema is discovered, and a preliminary data model is instantiated, the `SDMDiscoverPlugin` finds the relative frequencies of a predefined set of characteristics for the data fields.

The method `_mark_string_field_as_possible_property` takes three mandatory arguments, `obj_model`, `prop_key` and `disc_function`. The first, `obj_model`, is the instance of `SDMDataObject` that will get a new metadata property. The `prop_key` is the key that metadata will get. The `disc_function` is the discriminant function for the characteristic in question. In Python, functions are "First-class citizens", meaning a function can be assigned to a variable, and passed to another function. The `disc_function` is an example of this, and it can discriminate between data points that have a characteristic and those that do not. In addition there are two optional arguments. `prim_types` a list of primitive data types, the data object must be one of the data types in order to be checked. And `threshold` which determine how high the relative frequency of a characteristic must be to add it to the object metadata list. By default, the threshold is zero, resulting in that a characteristic with relative frequency higher than zero is added to the metadata list of the data object.

5.3 Inferring data types

The `SDMEtypePlugin` adds a list of possible Elasticsearch data types to each data object in the model, based on the analysis stored in the data object's metadata. The root object is passed to a method `_add_e_types_to_model`. This method will retrieve a list of possible types from `_get_e_types_for_object` and add the first to a metadata value with key `e_type`. The list of possible types is serialised to a JSON string and added as the value of the metadata key `_e_types`.

The `_add_e_types_to_model` method consists of if clauses, one for each possible JSON datatype. Prior to this, an empty list, `e_types`, has been initialised, and inside each of the if clauses, possible Elasticsearch data types are added to the list. For example if the primitive type is `boolean`, the types `boolean`, `keyword`, and `text` are added to the list of possible data types. The `e_types` list is then returned.

5.4 Command line interface

All parts of the SDModel system are started using the `sdmcli` command line interface. The file `sdmcli` is a Python file that will call the method `check_args(args)` on the class `SDMCli` if started from the command line. When a Python program is started from the command line, a list of the arguments passed along can be retrieved by using `argv` property of the Python package `sys`. For example if the command "`sdmcli init`" is run, the `sys.argv` will be the list `['sdmcli', 'init']`.

The first argument is always the name or path of the file that was executed, while the rest are arguments to the application. In `sdmcli` the second argument refers to the command to run, and the rest are arguments to the method that will execute the given command. For example to initialise a new model the command is `init` and it takes one optional argument, title given either by `-t` flag or the more verbose `--title` flag. The full command to initialise a model named 'test' then becomes `sdmcli init --title=test`. The `args` that are passed to the `check_args` method in this case is `['sdmcli','init','--title=test']`. The first argument is discarded, the second argument refers to the command which to run, in this case `init`. The init method is called with `args` as the argument.

The init method will check the arguments for the flag `-t` or `--title`. If present this will be used as the title of the model object. If no title is given, `sdmcli` will prompt the user for a title. Then the method creates an instance of the `SDModel`, saves it to the current location and exits.

A list of all commands and their arguments are available in Table 5.4

5.5 Web interface

The web interface provides a visual representation of the data model, and options for editing the model. It consists of two parts, a server module that serves the model through a REST API, and a client side JavaScript application that consumes this API through Ajax - Asynchronous Javascript and XML calls.

Command	Arguments	Description
`init`		Iinitialise a new model in the current directory.
	[-t –title]	Title of the model.
`addobj`		Add object to the model.
`addobject`	-n –name	Name of the object.
	-d –description	Description text for the object.
	[-o –object]	Path to the parent object that this object should be attached. Names separated by `/`.
`addmeta`		Add metadata to object.
	-k –key	Key of the the metadata.
	-v –value	Value of the metadata.
	[-o –object]	Path to the object that this metadata should be added. Names separated by `/`.
	[-r –replace]	Replace the current value with same key. Default is false.
`editmodel`		Change the title of the model.
	-t –title	The new title.
`print`		Prints the model to the console.
`wsdl`		Retrieves a WSDL file and builds a data model based on that.
	-p –path	The path or url to the WSDL file.
	[-q –quiet]	Merge all suggestions without asking.
	[-v –verbose]	Ask about all merges.
`serve`		Start the web interface server.
	[-p –port]	Port number to run server on. Default : 8080.
	[-c –cors]	Set to allow cross origin requests from anywhere. Mainly used for development or situations where the web app runs on a different server than the backend.
`discover`		Discover a model from some sample data.
	-f –file	Path to a JSON formatted file containing an array of sample data documents.
	[-n –name]	Name to give the new model.

Table 5.1: List of all the available commands and their arguments for the `sdmcli` command line interface. Arguments placed inside [square brackets] are optional.

5.5.1 Server

The server is started from the command line interface using the command `serve`. This will start four server modules, each with its own endpoint. The server modules use decorators from CherryPy, a lightweight, object oriented web framework [45], to serve content.

The main module is the `SDMServer`, it provides a REST API at the `/model` endpoint. This endpoint supports GET, POST, PUT and DELETE requests in addition to the OPTIONS preflight request to allow Cross Origin Resource Sharing (CORS) requests (mainly used for development). All methods take one parameter; `muid`, which identifies the object to perform the requested action on.

A GET request will return the data object with the matching `muid`, or the whole model if the `muid` parameter is set to the string "top". A POST request will add a `SDMDataObject` to the object that corresponds to the given `muid`. If the `muid` is the string "top" then the object will be added to the `SDModel` directly. The body of the request must consist of a JSON encoded version of the `SDMDataObject` that is to be added.

A PUT request updates the data object that has the corresponding `muid` with the JSON encoded values in the body of the request. DELETE is for deleting objects, a DELETE request thus will delete the data object with the corresponding `muid`. If the `muid` parameter is missing or set to top, this will not delete the model as one might expect, rather nothing will happen. To delete the model one has to delete the `sdmodel.json` file from disk.

The static files for the frontend are served by CherryPy directly at the `/` endpoint. Navigating the browser to `localhost:8080` after starting the server loads the frontend app.

The `SDMPluginServer` provides an API to handle input plugins at `/plugins`. A GET request will return a list of the available plugins. Each plugin in the list is an object with four properties. `name` is the display name for the plugin while the class name can be found in `class_name`. A short description is also included, as is a list named `arg_components` that specifies the arguments needed to use the plugin.

To run an input plugin on the model, a POST request is sent to the `/plugins` endpoint with the plugin object from the GET request as the body of the request. The argument components should of course be edited as seen fit. This will return a list of merge suggestions, objects that represent a suggested change in the model.

The merge suggestions that are accepted is then sent in a PUT request to `/plugins` which will execute them on the model.

The output plugins are served by the `SDMOutputPluginServer`. A list of available output plugins is available through the endpoint `/outputplugins`. The usage of the output plugins are the same as the input plugins, except that the response of the output plugins is typically the text of a configuration file.

5.5.2 Front end

The front end is built with Vue.js, a "progressive framework for building user interfaces" [46]. Vue.js is a Javascript framework that let developers easily develop progressive single page web apps, web pages that resemble mobile or desktop apps [46].

In traditional web pages, the page is loaded synchronously, i.e. the user enters a url and the

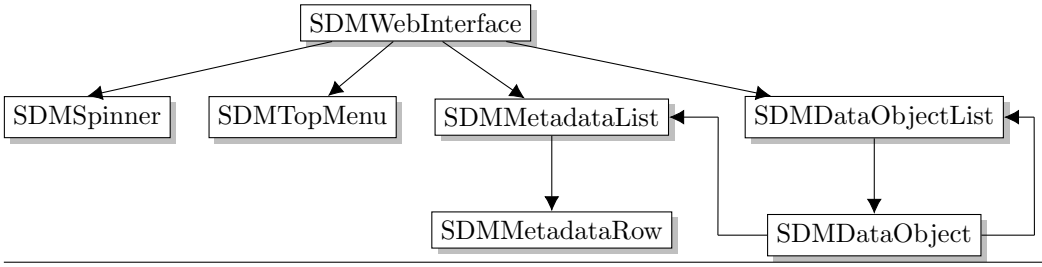

Figure 5.3: The component tree showing the components of the Vue.js application and how they are connected.

content on that page is loaded from the server and presented to the user. When a link is clicked, the linked page is loaded from the server and presented. This means that the user has to wait with no page showing while next page loads. In a progressive single page app the goal is to load as little as possible before presenting it to the user, and then load the rest of the content asynchronously. This way the user can start consuming the content faster. The content of connected pages can be loaded once the current page is done. When the user clicks a link to another page, the content is already loaded and the transition is a matter for the user interface and need no network interaction. This in turn cause no change in the url, thus the name "single page app".

Vue.js is a component based system, i.e. any component can contain a HTML, CSS styles and a JavaScript, and is reusable. Components use other components to lay out their content. The app can then be viewed as a tree of components. Data is passed down this tree. If a component lower on the tree wishes to change some piece of data it will do this locally and either send it to the server or to the state manager. The state manager is a global state management object that keeps track of shared data in the application [46]. Figure 5.3 shows the component tree for the frontend app presented here.

Vue.js also implements a reactive data structure for the data in the web app. This means that when a piece of data is changed by some part of the app, any view underneath it in the tree that displays this data will also update [46]. When the data is changed in the state manager all views will update.

The rest of this section will present each of the components used in the frontend app.

SDMWebInterface is the root element of the component tree. It lays out a component `TopMenu`, some model information, and the components `spinner`, `MetadataList` and `DataObjectList`. When this component is loaded and presented to the user it will set the variable `loading` to true. This causes the spinner to be visible. It then dispatches an asynchronous action `getModel` on the state manager, that will get the data model from server. When the state manager has received the response, it will call the function that was passed with the dispatch to stop the spinner.

Once the model is updated on the state manager, all components will get the updated version and can present it to the user.

SDMMetadataList takes one property, `metadata` and displays a list of `SDMMetadataRow` components by iterating over the metadata and feeding one metadata pair to each row.

An empty row is added to let the user add metadata to the data object.

SDMMetadataRow displays a metadata pair, and takes three parameters, `metakey`, `metaval` and `muid`. If the user clicks the edit icon the state of the component will change to edit mode and the key and value will be editable. When saved, the component will dispatch a `setMetadata` action on the state manager that will send the request to the server. If the request is successful the state manager will dispatch another action `loadSDModel` that will load and update the model. This causes the change to be visible in the view.

SDataObjectList is very similar to the metadata list, it displays a list of `SDataObject` components, and an empty one for adding extra data objects to the model.

SDataObject is the component that displays the data object. It has buttons for editing fields and functions that will update the data object on a per field basis. This means that if a user updates the description of a data object, that change will be sent to the server. The user can now edit the type or any other part of the data object while still waiting for the name change to be returned by the server. This is just one of the parts where asynchronous server communications show its benefits in user experience.

This component also uses the component that is its parent, the `SDataObjectList` to show a data object's data objects. And also a `SDMMetadataList`.

SDMTopMenu is the component that handles the top menu and the plugin interactions. When the user choose to use a plugin from the list in the top menu, a modal overlay is added that covers the screen and lets the user configure the plugin. When the plugin is initiated a list of merge suggestions can be accepted or rejected from the top menu's modal element.

SDMSpinner is just a way of making an animated spinner easily reusable.

5.6 OutputPlugins

The output plugins inherit from the `SDMOuptutPlugin` which in turn inherits from `SDMPlugin`. These relationships provide a basis for developing new plugins, and also encourages a similar implementation for each of the plugins.

5.6.1 SDMLogstashOutputPlugin

The `SDMLogstashOutputPlugin` class is initialised by passing the arguments, `model` and `plugin_opts`, to its superclass, the `SDMOutputPlugin`. The arguments become the values of the instance variables `self.model` and `self.plugin_opts`. The `SDMLogstashOutputPlugin` does not require any arguments passed from the user, so the `plugin_opts` could be discarded.

The `get_output` method on the class is called by the `SDMOutputPluginServer` to get the output from the plugin. In the case of the `SDMLogstashOutputPlugin` the method is responsible for building up the filter section of a Logstash configuration file. First a local variable `filters` is initialised with the string '`filter {\n`'. Then all the data objects at the root of the data model are iterated over, and filter configurations are appended to the `filter` variable.

As described in Section 4.6.1, the first filter configurations that are added are the uses of the Logstash `split` plugin. This is done by passing the data object that is currently being evaluated to the `_get_splits` method. The `_get_splits` method instantiates an empty string '`split_string`', and checks if the `prim_type` of the object is `array` and the `e_type` is '`split`'. If both these are true, it will add to the `split_string` a snippet that will split the array on the current data object. It will then iterate over the data objects of the current data object, and pass each as the data object in a recursive call to itself, `_get_splits`. The result of each of these calls will be added to the `split_string` before it is returned.

The next configurations are the transforms. This is done by a similar approach as above, the root object is passed as argument to `_get_transforms`. The method `_get_transforms` will instantiate an empty string `t_string` and check the `prim_type` of the data object.

If the `prim_type` is '`string`', it will check the `e_type` for any of the floating point number types, `double`, `float`, `half_float` and `scaled_float`. If this test passes, a filter configuration for the plugin `mutate` will be appended to the `t_string` variable. The filter plugin will in this case be set to convert the current object to a `float`. The same approach is used if the `e_type` is any of the integer types, `long`, `integer`, `short` or `byte`. Only here the filter plugin will be set to convert the value to an `integer`.

If the `prim_type` is '`number`', the `e_type` is '`date`' and the metadata value for `samplevalue` is between 473385600 and 2051222400, it would seem to be a numeric UNIX timestamp. The original UNIX timestamp, as defined in 4.3.1, is given in seconds since the epoch. However, Elasticsearch does not recognise a number as a date unless it is given in milliseconds since the epoch. Therefore, if a field is numeric, has `e_type` '`date`' and is within the 'seconds from epoch' interval, it must be converted to milliseconds since the epoch. This is done by the Logstash filter plugin `date`, and the snippet that configures it will be added to the `t_string`.

If the `prim_type` is '`object`' and the `e_type` is '`geo_point`' several filter configurations will be added. The data objects of the data object is iterated over, and each will be checked. If the current data object's name has the string 'lat' in it, case insensitive, that field will be renamed to '`lat`'. If the current data object's name has the string 'lon' or 'lng' in it, case insensitive, that field will be renamed to '`lon`'. If neither of the two apply, the field will be removed. That is, a Logstash filter `mutate` will be used to remove the field from the data. This is required because Elasticsearch does not accept any other objects within a `geo_point` object than the `lat` and `lon`. After this, a filter configuration using the `convert` operation from the `mutate` plugin is added that will explicitly set the `lat` and `lon` data types to `float`.

The `_get_transforms` method is then called recursively for all data object in the data objects list.

The field removals are the next set of configurations to be added to the filter configuration. The method `_removal_for_data_object` is called with the data object as argument. It will check if the date object's object metadata has a field `_marked_for_removal`. If it does, and it is `true`, the Logstash `mutate` filter is used to set up Logstash to remove the field from the data.

The last filter configuration type that is added is the renaming of values. This has to be last, because after a field is renamed, the new name is the only way to address the element, and all the previous filter configurations operate with the original names. Renaming a data field is done by the `mutate` plugin using the operation `rename`. It takes an array of key-value pairs as argument. Here the key is the original name, and the value is the new name.

5.6.2 SDMElasticsearchOutputPlugin

The `SDMElasticsearchOutputPlugin` is initialised in the same way as `SDMLogstashOutputPlugin`, by passing `model` and `plugin_opts` to its superclass, `SDMOutputPlugin`. However, after this, the `SDMElasticsearchOutputPlugin` also extracts some information from the `plugin_opts`. The instance variables `self.index_name`, `self.type_name`, and `self.mapping_mode` are set from their corresponding values in the `plugin_opts` key-value dictionary.

The `SDMElasticsearchOutputPlugin` also has the `get_output` method. It starts by instantiating an empty dictionary called `mappings`. The value of `type_name` becomes the key for a new dictionary added to the `mappings`. The new dictionary then gets another dictionary under the key '`properties`'. Then the data objects are iterated over and each object is passed as an argument to the method `_get_mappings_for`. The returned value is stored in the '`properties`' dictionary under the key corresponding to the name of the data object.

The method `_get_mappings_for` checks first if the name of the data object is 'null' or `None` ('not defined' in Python). If this is the case, the data object is the item in an array, and thus could safely be ignored. Therefore the variable `data_obj` is given the value of its first data object.
`data_obj = data_obj.data_objects[0]`
If it has no data objects, `None` is returned.

The variable `type` is set to the value of the object metadata value for the key `e_type`. If the type is `split`, which is not a type in Elasticsearch, it is given the value '`object`'. The `type` is then added to the mappings dictionary, under the key with the same name.

The data objects of the data object are iterated and passed as arguments to a recursive call to `_get_mappings_for`, unless the data object has an `e_type` that should be treated differently, or has the metadata key `_marked_for_removal` with its value set to `true`. If the `e_type` is one of the types represented by objects in Elasticsearch, i.e. `geo_point`, `geo_shape`, `long_range`, `integer_range`, `date_range`, `float_range` and `double_range`, its data objects should not be explored. A mapping for an object in the list above that contains 'properties' will cause an error in Elasticsearch, they are implicitly defined by Elasticsearch.

If the `self.mapping_mode` is set to the string '`newindex`', a dictionary containing predefined settings will be added, otherwise, the mappings dictionary is returned as is.

Chapter 6

Experiments and Results

Data can come in many different formats and encodings, it can have different structures and schemas, and it can be small or big. To test how well a system performs, one should, ideally, test it against all the data that it is made to handle. This is usually close to impossible. However, testing the system on a smaller set of data that represents the various data it is made to handle might be possible, but this is usually still a very large amount of data.

Testing the system presented in this thesis is done using two different data sets. First it will be tested against a simulated data set that is based on the temperature measurement example in Section 4.2. The second test will be real-world data from the Kolumbu VehicleMonitoring service.

This chapter will present the approach used to create a simulated data set for testing the system in Section 6.1. Section 6.2 will present the results of testing the system on the simulated data set. The results from the real-world example of collecting data from Kolumbus' Vehicle Monitoring Service is presented in Section 6.3.

6.1 Simulating test data

To test the system, a data set that resembles the temperature example in Section 4.2 has been developed. To make the data more realistic, the values of the fields are randomised and will vary within certain constraints.

`RecordedAt` is increased by 1 minute for each document, simulating a measurement every minute.

The value of the field `Temperature` is given by a random sample from a gaussian probability distribution with mean = 15 and standard deviation = 5. The result is that 68.2% of the measurements lie between 10 and 20.

The value of the field `Location` is given by the location in Example 6.1, plus a random sample from a gaussian distribution with mean = 0 and standard deviations 0.06 for latitude and 0.03 for longitude. This causes the majority (68.2%) of the values to fall inside a ellipse, approximately 7 by 5km, and the University of Stavanger in the centre.

The battery level will decrease from 100 in the first document down towards zero in the last.

An example of the generated data can be seen in Example 6.1.

```
{
  "RecordedAt" : "2017-04-11T09:27:31.6814296+02:00",
  "Temperature" : "21",
  "DeviceType": "ManufacturerModel5100",
  "Location" : {
      "Latitude" : 58.938334,
      "Longitude" : 5.693542
      },
  "BatteryLevel" : 65
}
```

Example 6.1: JSON encoded event data from temperature measurement device.

6.2 Simple temperature example

The first experiment is based on the temperature example Section 4.2. A sample data set of 1000 documents is generated using the simulation technique described in section 6.1.

Letting Elasticsearch dynamically map the data types

The first approach is to let Elasticsearch dynamically map the data types, i.e. not create any index beforehand, just let Elasticsearch create the index and all the mappings. In this approach there are no filters in the Logstash configuration.

Getting mappings from SDModel

The second approach is to use the SDModel software presented in this thesis to discover the data model. In this approach all the sample data is run through the system, and the resulting data model is used to generate both Logstash filter configuration and Elasticsearch index mappings.

Results

Some of the results of the mappings are presented in Table 6.1. The complete results can be seen in Appendix B.

The table presents the JSON type of the original data field in the first column. The second column, *Dynamic type*, represents the result of dynamic mapping done by Elasticsearch. The third column, *SDModel type*, represents the number one suggestion from the SDModel system. The last column, *GT type*, is the ground truth, i.e. the perfect suggestion for the field.

The first field, `RecordedAt`, represents a date, and both methods suggested the `date` data type, so both are correct. The field `Temperature` is a JSON `string` type field, however, in

Field name	JSON type	Dynamic type	SDModel type	GT type
RecordedAt	`string`	`date`	`date`	`date`
Temperature	`string`	`text`	`float`	`float`
DeviceType	`string`	`text`	`keyword`	`keyword`
Location	`object`	`object`	`geo_point`	`geo_point`
Location.Latitude	`number`	`float`	part of geo_point	part of geo_point
Location.Longitude	`number`	`float`	part of geo_point	part of geo_point
BatteryLevel	`number`	`float`	`long`	`float`

Table 6.1: Mappings for example data set. JSON type is the data type the field has in JSON encoding. Dynamic type is the result of dynamic mapping done by Elasticsearch. SDModel type is the result from the SDModel software described in this thesis. GT type is the ground truth type, i.e. the perfect result.

all data points in the sample data, it contains only numbers, so it should be converted into numeric values. Dynamic mapping suggested `text`, while SDModel suggested `float`, which is the GT type. If the field is `text` it will be searchable as a text, and in some cases that is the required behaviour. However, if the field is converted to a `float`, one can for example do range searches, e.g. get all the measurements where the temperature is above 17 degrees. The `DeviceType` is suggested to be data type `text` by dynamic mapping, while SDModel wants to use the `keyword` data type. The GT type is `keyword`, but the difference here is not a big one, `keyword` is searchable only by the complete term, while `text` is searched like a full text field with partial hits.

The `Location` field is interpreted as an `object` by dynamic mapping, causing it to not be searchable at all. SDModel however, suggests using the `geo_point` data type, which also is the GT type. One of the benefits of the `geo_point` data type, is that Elasticsearch can do queries based on the geographic location.

The `BatteryLevel` field is a number in JSON and is interpreted as a `float` by dynamic mapping and a `long` by the SDModel. In this case dynamic mapping is more correct than the SDModel, the battery level is given with 1 decimal in all of the data points and should thus be a `float`. However, in all of the data points in the sample data, the decimal is 0, causing the SDModel to see it as an `integer`. If the SDModel had used the decimal it would have suggested `double`, and been more correct in this case. However, dynamic mapping still wins, as the `float` type can store values up to 2^{32}, the `Temperature` field probably won't need more.

This experiment was specifically designed to make the SDModel software shine, and make Elasticsearch's dynamic mapping the loser. However, as will be shown in later examples, this experiment is not that far from the real world.

6.3 Kolumbus VM data

A request to the Kolumbus Vehicle Monitoring Service (VM) is shown in Example 6.2. The request consists of a header and a body part. Inside the header, the `Action` is set to `GetVehicleMonitoring`. There are other actions available on the service, however, this is the only one used in this experiment. The field `MessageID` has a value generated by `zeep`, the Python SOAP client, and is not required by the Kolumbus VM Service. The field `To` gives the URL to which this request is sent.

The body of the request has one object, `GetVehicleMonitoring`. It again has a `ServiceRequestInfo`, a `Request`, and a `RequestExtension`.

The `ServiceRequestInfo` contains information related to the request. It has the property `RequestTimestamp`, reflecting when the request was sent, according to the sender, and a `RequestorRef`. The latter field is required to not be empty, here it is set to the authors email address.

Next in the body is the `Request`, that contains information about what specific data is requested. It has the same property `RequestTimestamp` as above. It also has a property `VehicleMonitoringRef` that defines what vehicles to get monitoring data for, in this case `ALL` refers to all available vehicles. The property `MaximumVehicles` could limit the request to a number of vehicles to get data for, in this case it is set to 0, meaning there is no limit.

```
<?xml version='1.0' encoding='utf-8'?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsa:Action>GetVehicleMonitoring</wsa:Action>
    <wsa:MessageID>urn:uuid:4e9a31eb-a79b-45a9-a44c-67d7d62f95f3</wsa:MessageID>
    <wsa:To>http://sis.kolumbus.no:90/VMWS/VMService.svc</wsa:To>
  </soap-env:Header>
  <soap-env:Body>
    <ns0:GetVehicleMonitoring xmlns:ns0="http://www.siri.org.uk/siri">
      <ServiceRequestInfo>
        <ns0:RequestTimestamp>2017-06-06T16:41:50.021341</ns0:RequestTimestamp>
        <ns0:RequestorRef>mindejulian@me.com</ns0:RequestorRef>
      </ServiceRequestInfo>
      <Request version="1.4">
        <ns0:RequestTimestamp>2017-06-06T16:41:50.021341</ns0:RequestTimestamp>
        <ns0:VehicleMonitoringRef>ALL</ns0:VehicleMonitoringRef>
        <ns0:MaximumVehicles>0</ns0:MaximumVehicles>
      </Request>
      <RequestExtension/>
    </ns0:GetVehicleMonitoring>
  </soap-env:Body>
</soap-env:Envelope>
```

Example 6.2: XML encoded request to Kolumbus Vehicle Monitoring Service. The action is `GetVehicleMonitoring`, and it is sent to the Kolumbus' url. The body of the request has only the required parameters. The `ServiceRequestInfo` object has two parameters. `RequestTimestamp` is the current time, `RequestorRef` can have any value, in this case the authors email address. The `Request` object is required to have the version, at the moment `1.4`. Furthermore it has the `RequestTimestamp` again. The parameter `VehicleMonitoringRef` can be used to narrow down the number of results, in this case, the status of all vehicles are requested. `MaximumVehicles` and `VehicleMonitoringDetailLevel` have their default values, namely "0" and "normal" respectively.

The response from the service can be acquired by sending this request to the url given. The response is XML encoded as well, however, to conduct this experiment it is converted directly to JSON.

For this experiment, the Kolumbus VM Service was called 120 times with one minute between each call. Due to the time spent by the requests themselves, the total time for the requests was closer to 130 minutes. The expected structure of the response is available in Appendix A.

Because the structure of the response consists of several layers of nested objects at the root level, Elasticsearch's dynamic mapping will map only the three properties directly below the root node, namely `Answer`, `AnswerExtension`, and `ServiceDeliveryInfo`. The values of these properties are treated as un-indexed objects. Therefore the results of dynamic mapping is omitted in this experiment.

Getting mappings from SDModel

The data from the Kolumbus VM Service was saved to a file, as a JSON array with 120 items. This data was used as the sample data for the SDModel system, and it suggested a data model. The generated data model was then used to generate an Elasticsearch index request, and the filter section of a Logstash configuration file.

Results

Table 6.2 shows the structure of parts of the response from the Kolumbus VM Service, and results from the SDModel system. The complete results are available in Appendix B. The first column is the field name, here the caracter `-` indicates that the field is a subfield of the previous field with one less `-`. The second column shows the JSON data type of the value in the field. The third column is the SDModel suggestion for Elasticsearch data type for the field, while the fourth column indicates the best choice for the Elasticsearch type in the authors opinion. The choice of data type often a subjective one, and therefore there is no 'ground truth' in this example. The last column indicates if the SDModel was correct or not.

From Table 6.2 it is apparent that the data model is not a flat one, in fact there are up to five levels of nesting. If the fields that have subfields are not counted, the data model has 110 fields. The analysis of the sample data show that 75 of these fields had null values. That is approximately 68% of the data fields in the data set had no value in the 120 requests that were made for this experiment. Further investigation into the dataset documentation issued by Kolumbus [7] show that these fields are either not used, or not mentioned in the documentation. The SDModel system marks these fields for removal by default, this can be corrected through the web interface. Marking a field for removal results in a Logstash filter that removes the data field from the data point before this is sent to Elasticsearch.

The fields that are arrays in the data set are given the data type `split` by default. `split` is not really an Elasticsearch data type, but a Logstash filter that will split the data point into several data points, one for each element in the array that is split. All values other than those in the array will be duplicated and appear in each of the new data points.

The first field in Table 6.2 that the SDModel system did not get correct is the field `VisitNumber`. The JSON type for this field is `number` but the SDModel system suggests `boolean`. The correct suggestion would be `integer`. The value of this field indicate how many times on the current trip the vehicle has visited the upcoming stop [7]. One might imagine a situation where this number was higher than 1, but in the Kolumbus route system, the buses visit each stop at most once per trip. In the sample data set, this value was 1 or `null` in all the 46920 data points. The number of data points at this point is so high because there are 391 vehicles that

Field name	JSON	SDModel	Best	Result
Answer	object	object	object	correct
- VehicleMonitoringDelivery	array	split	split	correct
- - VehicleActivity	array	split	split	correct
- - - MonitoredVehicleJourney	object	object	object	correct
- - - - MonitoredCall	object	object	object	correct
- - - - - VisitNumber	number	boolean	integer	incorrect
- - - - - AimedDepartureTime	string	date	date	correct
- - - - - StopPointName	string	text	text	correct
- - - - - BoardingStretch	string	boolean	null	correct
- - - - - ExpectedArrivalTime	string	date	date	correct
- - - - - ⋮				
- - - - Delay	number	long	integer	partial
- - - - DestinationAimedArrivalTime	number	date	date	correct
- - - - PublishedLineName	string	keyword	keyword	correct
- - - - Monitored	number	boolean	boolean	correct
- - - - VehicleLocation	object	geo_point	geo_point	correct
- - - - - srsName	null	null	null	correct
- - - - - Precision	string	long	null	correct
- - - - - Coordinates	null	null	null	correct
- - - - - Longitude	number	double	float	partial
- - - - - Latitude	number	double	float	partial
- - - - - Id	null	null	null	correct
- - - - DestinationName	string	text	text	correct
- - - - JourneyNote	array	null	null	correct
- - - - OriginName	string	text	text	correct
- - - - ⋮				
- - - ProgressBetweenStops	object	object	object	correct
- - - - Percentage	number	double	float	partial
- - - - LinkDistance	number	long	integer	partial
- - - RecordedAtTime	number	date	date	correct
- - - ⋮				
- - ValidUntil	number	date	date	correct
- - ⋮				
AnswerExtension	null	null	null	correct
ServiceDeliveryInfo	object	object	object	correct
- ResponseTimestamp	number	date	date	correct
- ⋮				

Table 6.2: Partial response data from the Kolumbus VM Service using the request in 6.2. First column gives the name of the field, the character - indicates that the field is a subfield of the previous field with one less -. The second column shows the JSON data type, while the third shows the number one suggestion by SDModel and the next the ground truth (GT). The last column indicates whether the SDModel suggestion was correct, incorrect or partially correct. Partially correct is used in cases where the suggestion is close and the difference has little impact on the system, e.g. long instead of integer. The complete results are available in Appendix B.

report their status in each of the 120 requests, resulting in $391 * 120 = 46920$ data points of the `VehicleActivity` field, and its child fields. The SDModel analysis reported the estimated probability of null values to be 0.47 for this field. The SDModel system would by default suggest that any field that contains only values 0 or 1 to be `boolean` data type.

The next suggestion that is not entirely correct is the field `Delay`. It is a `number` in the JSON representation, and the SDModel system suggested the data type `long`, while the correct answer was `integer`. This happens to several of the fields, `double` is suggested over `float` too. The reason for this is that the SDModel system does not know the maximum value for the field and thus chooses the data type with the highest capacity.

The response consisted of a total of 118 data fields, including data fields with subfields. Of these, 75 never had a value in the sample data, thus making the data type suggestion close to impossible. Of the 43 remaining data fields, 30 got correct suggestions, 13 got partially correct suggestions, and 1 got an incorrect suggestion.

Chapter 7

Conclusion

This thesis presents the architecture, design, and implementation of a software system, *SDModel*, that can generate configuration files for big data collection pipelines, based on a set of sample data. In terms of selecting the data types and transformations of the data, the system is not fully automatic. It requires that a domain expert validates, and sometimes corrects the suggestions given by the system. The renaming of data fields, for more semantic data field names, is a task left for the domain expert to do while reviewing the data types.

The person that is to configure a system like the Elastic stack to store data from a new data set, must get familiar with the data set. Knowledge about the data set is important in order to minimise the information loss, as result of transforming the data from one system to another. This can also contribute to avoiding the storage of redundant information. In the Elastic stack, searchability is a valued feature, and optimising the configuration for the data to be easily searchable requires knowledge not only about the actual data, but also about what it represents, and how it will be used.

The SDModel framework for automating configuration of data collection pipelines presents the summary of a data set through a web interface, with an intuitive and informative design. All the collected statistics are available through the web interface, and some properties leverage graphical components to present the results of the analysis. This makes it easy for the person configuring the data collection pipeline to get familiar with the data set, and reduces the need to explore large amounts of sample data manually.

As the results of the experiments in Chapter 6 shows, the SDModel is not always correct in its number one suggestion. Suggesting data types is a complex task. Data representing the same real-world entity can come in so many formats, and so many different entities can be represented by the same format. However, the SDModel narrows down the list of possible data types for a data field, and if the number one suggestion was not correct, another suggestion may be.

In many cases, transforming data using Logstash is a necessity. For example, the case of the Elasticsearch type `geo_point`, where the properties of the object must be named `lat` and `lon`. Here a renaming operation is in many cases required. If the original fields are named `Latitude` and `Longitude`, this is a relatively safe transformation to make, with no apparent loss of information. However, there might be other data fields in the same object, as is the case in the Kolumbus VM experiment. In this data set, there are data fields `Precision` and `Coordinates`

on the `Location` field as well as `Latitude` and `Longitude`. The requirements for the Elasticsearch data type `geo_point` is that `lat` and `lon` are the only properties of the object. Removing these two extra fields could cause an information loss, and requires the domain expert to get familiar with the situation before removing the fields, or choosing a different approach like splitting the object into two objects. In the case of the Kolumbus VM, the extra fields proved to bear no information in the sample data, i.e. the values were null at all data points, so removing them appears to be a safe decision.

The process of collecting and storing smart city data is a task that still requires that the person doing the configuration understand the data and how it is structured. However, the system presented in this thesis can provide very valuable insights and templates for the configuration of data collection pipelines.

7.1 Evaluation

The SDModel system does reach the goal of automatically generating the configuration files for a data collection pipeline based on sample data. However, suggesting field types was harder than the author of this thesis anticipated. There are many cases not covered by the SDModel system, however, the results of the experiments in Section 6 proves that the system work well even with a real-world data set.

In the beginning of this project, the user experience was an important part of the system, unfortunately solving the problem has taken priority, and thus the user experience is not optimal.

The SDModel system provides an intuitive and good summary of the contents of a data set, and for this the system could prove useful even in situations where the configuration files are not used.

7.2 Contributions

The author of this thesis, Julian Minde, has contributed with

- The exploration into some of the alternative choices of software for collecting and storing smart city data and the description of one such system, the Elastic stack.
- The development of the SDModel software system. A software that analyses sample data and generates a data model that show the structure and properties of the sample data.
- The development of a specific solution to storing data from the Kolumbus Vehicle Monitoring Service, with configurations generated by the SDModel system.

7.3 Future work

The SDModel system has been developed as a ‘proof-of-concept’ and thus may need some more development in terms of user interface and experience in order to be usable to general users.

The SDModel system does not configure systems, rather it presents files that a user can use to configure the system. This might be automated when the SDModel system is more mature and reliable.

The SDModel system, in its current version, uses a finite set of sample data to create configurations. One possibly valuable feature would be if the SDModel system could receive bits of sample data from the data provider throughout the whole process of the data collection. This data could be analysed, and should the system discover a discrepancy between the current configurations and the data that is received, it could either raise an alert to a technician, or simply update the configuration files to account for the discrepancy. For example, if a field that has only null values in the sample data suddenly starts carrying valuable information, it could be critical to start collecting said field.

The SDModel system provides a good overview of the data set. However, this data model presentation, the results of the analysis, and descriptions of the data fields, might be valuable information to the data scientists, developers and other users of the data, as well as the technician(s) configuring the data collection pipeline.

The collection and storage of smart city data is constantly evolving, and so must the system presented in this thesis. There are many corner cases and more common cases that the SDModel system will not handle in its current implementation. However, by continuing the development of solutions within the system, it will be able to handle similar cases later, and also for other users.

Appendices

Appendix A

Kolumbus Vehicle Monitoring Service Response

The structure of the response from the Kolumbus VM Service is presented in Figure 1.1.

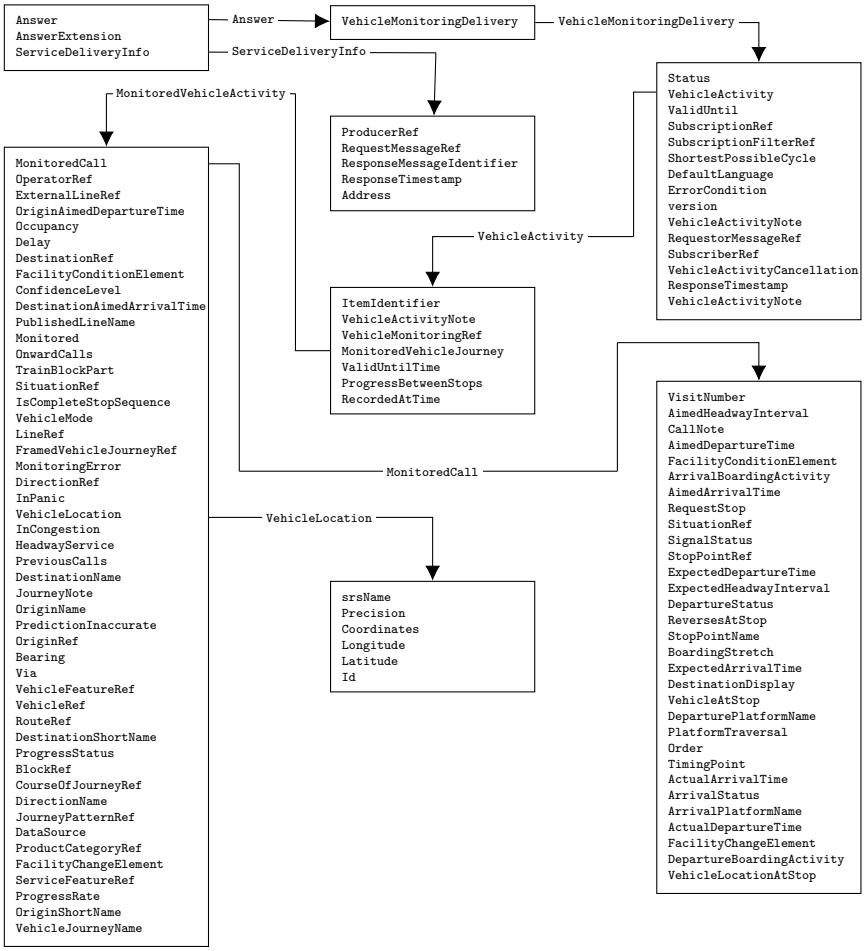

Figure 1.1:
Sketch of the structure of the data fields in the Kolumbus Vehicle Monitoring service's response to the action `GetVehicleMonitoring`. The root object is in the upper left corner, and this figure shows that there are several levels of nesting in this data.

Appendix B

Kolumbus VM Service Example, Complete Results

This appendix present the complete results of the experiment presented in Section 6.3. Table B.1 through B.5 show the fields present in the data set, and their types. It also show the SD-Model systems suggestions and the authors opinion on what Elasticsearch data types are best fit for the fields.

Example B.1 show the suggested Elasticsearch mapping. And Examples B.2 and B.3 show the filter section of a Logstash configuration file.

Field name	JSON	SDModel	GT	Result
Answer	object	object	object	correct
- VehicleMonitoringDelivery	array	split	split	correct
- - Status	null	null	null	correct
- - VehicleActivity	array	split	split	correct
- - - *See B.3*				
- - ValidUntil	number	date	date	correct
- - SubscriptionRef	null	null	null	correct
- - SubscriptionFilterRef	null	null	null	correct
- - ShortestPossibleCycle	number	long	integer	partial
- - DefaultLanguage	null	null	null	correct
- - ErrorCondition	null	null	null	correct
- - version	string	long	float	partial
- - VehicleActivityNote	null	null	null	correct
- - RequestorMessageRef	null	null	null	correct
- - SubscriberRef	null	null	null	correct
- - VehicleActivityCancellation	null	null	null	correct
- - ResponseTimestamp	number	date	date	correct
- - VehicleActivityNote	null	null	null	correct
AnswerExtension	null	null	null	correct
ServiceDeliveryInfo	object	object	object	correct
- ProducerRef	string	keyword	keyword	correct
- RequestMessageRef	null	null	null	correct
- ResponseMessageIdentifier	string	keyword	keyword	correct
- ResponseTimestamp	number	date	date	correct
- Address	null	null	null	correct

Table B.1: The response data from the Kolumbus VM Service using the request in 6.2. First column gives the name of the field, - indicates that the field is a subfield of the previous field with one less -. The second column shows the JSON data type, while the third shows the number one suggestion by SDModel and the next the ground truth (GT). The last column indicates whether the SDModel suggestion was correct, incorrect or partially correct. Partially correct is used in cases where the suggestion is close and the difference has little impact on the system, e.g. long instead of integer.

Field name	JSON	SDModel	GT	Result
MonitoredVehicleJourney	object	object	object	correct
- MonitoredCall	object	object	object	correct
- - *See Table B.5*				
- OperatorRef	null	null	null	correct
- ExternalLineRef	null	null	null	correct
- OriginAimedDepartureTime	number	date	date	correct
- Occupancy	null	null	null	correct
- Delay	number	long	integer	partial
- DestinationRef	string	long	integer	partial
- FacilityConditionElement	array	null	null	correct
- ConfidenceLevel	null	null	null	correct
- DestinationAimedArrivalTime	number	date	date	correct
- PublishedLineName	string	keyword	keyword	correct
- Monitored	number	boolean	boolean	correct
- OnwardCalls	null	null	null	correct
- TrainBlockPart	null	null	null	correct
- SituationRef	array	null	null	correct
- IsCompleteStopSequence	string	boolean	null	incorrect
- VehicleMode	array	keyword	keyword	correct
- LineRef	string	long	integer	partial
- FramedVehicleJourneyRef	null	null	null	correct
- MonitoringError	string	keyword	keyword	correct
- DirectionRef	string	keyword	keyword	correct
- InPanic	null	null	null	correct
- VehicleLocation	object	geo_point	geo_point	correct
- - *See Table B.4*				
- InCongestion	null	null	null	correct
- HeadwayService	null	null	null	correct
- PreviousCalls	null	null	null	correct
- DestinationName	string	text	text	correct
- JourneyNote	array	null	null	correct
- OriginName	string	text	text	correct
- PredictionInaccurate	null	null	null	correct
- OriginRef	string	long	integer	partial
- Bearing	null	null	null	correct
- Via	array	null	null	correct
- VehicleFeatureRef	array	null	null	correct
- VehicleRef	string	long	integer	partial
- RouteRef	null	null	null	correct
- DestinationShortName	null	null	null	correct
- ProgressStatus	null	null	null	correct
- BlockRef	null	null	null	correct
- CourseOfJourneyRef	string	long	integer	partial
- DirectionName	null	null	null	correct
- JourneyPatternRef	null	null	null	correct
- DataSource	null	null	null	correct
- ProductCategoryRef	null	null	null	correct
- FacilityChangeElement	null	null	null	correct
- ServiceFeatureRef	null	null	null	correct
- ProgressRate	null	null	null	correct
- OriginShortName	null	null	null	correct
- VehicleJourneyName	null	null	null	correct

Table B.2: Mappings for example data set. JSON type is the data type the field has in JSON encoding. Dynamic type is the result of dynamic mapping done by Elasticsearch. SDModel type is the result from the SDModel software described in this thesis. GT type is the ground truth type, i.e. the perfect result.

Field name	JSON	SDModel	GT	Result
VehicleActivity	array	split	split	correct
- ItemIdentifier	null	null	null	correct
- VehicleActivityNote	null	null	null	correct
- VehicleMonitoringRef	null	null	null	correct
- MonitoredVehicleJourney	object	object	object	correct
- - *See Table B.2*				
- ValidUntilTime	number	date	date	correct
- ProgressBetweenStops	object	object	object	correct
- - Percentage	number	double	float	partial
- - LinkDistance	number	long	integer	partial
- RecordedAtTime	number	date	date	correct

Table B.3: Mappings for example data set. JSON type is the data type the field has in JSON encoding. Dynamic type is the result of dynamic mapping done by Elasticsearch. SDModel type is the result from the SDModel software described in this thesis. GT type is the ground truth type, i.e. the perfect result.

Field name	JSON	SDModel	GT	Result
VehicleLocation	object	geo_point	geo_point	correct
- srsName	null	null	null	correct
- Precision	string	long	null	correct
- Coordinates	null	null	null	correct
- Longitude	number	double	float	partial
- Latitude	number	double	float	partial
- Id	null	null	null	correct

Table B.4: Mappings for example data set. JSON type is the data type the field has in JSON encoding. Dynamic type is the result of dynamic mapping done by Elasticsearch. SDModel type is the result from the SDModel software described in this thesis. GT type is the ground truth type, i.e. the perfect result.

Field name	JSON	SDModel	GT	Result
MonitoredCall	object	object	object	correct
- VisitNumber	number	boolean	integer	incorrect
- AimedHeadwayInterval	null	null	null	correct
- CallNote	array	null	null	correct
- AimedDepartureTime	string	date	date	correct
- FacilityConditionElement	array	null	null	correct
- ArrivalBoardingActivity	string	keyword	keyword	correct
- AimedArrivalTime	string	date	date	correct
- RequestStop	string	boolean	null	incorrect
- SituationRef	array	null	null	correct
- SignalStatus	null	null	null	correct
- StopPointRef	string	long	integer	partial
- ExpectedDepartureTime	string	date	date	correct
- ExpectedHeadwayInterval	null	null	null	correct
- DepartureStatus	string	binary	null	incorrect
- ReversesAtStop	string	boolean	null	correct
- StopPointName	string	text	text	correct
- BoardingStretch	string	boolean	null	correct
- ExpectedArrivalTime	string	date	date	correct
- DestinationDisplay	null	null	null	correct
- VehicleAtStop	string	boolean	null	incorrect
- DeparturePlatformName	null	null	null	correct
- PlatformTraversal	string	boolean	null	incorrect
- Order	null	null	null	correct
- TimingPoint	string	binary	null	correct
- ActualArrivalTime	string	date	null	correct
- ArrivalStatus	string	binary	null	correct
- ArrivalPlatformName	null	null	null	correct
- ActualDepartureTime	string	date	null	correct
- FacilityChangeElement	null	null	null	correct
- DepartureBoardingActivity	string	binary	null	correct
- VehicleLocationAtStop	null	null	null	correct

Table B.5: Mappings for example data set. JSON type is the data type the field has in JSON encoding. Dynamic type is the result of dynamic mapping done by Elasticsearch. SDModel type is the result from the SDModel software described in this thesis. GT type is the ground truth type, i.e. the perfect result.

```
{
  "mappings": {
    "busdata": {
      "properties": {
        "Answer": {
          "type": "object",
          "properties": {
            "VehicleMonitoringDelivery": {
              "type": "object",
              "properties": {
                "VehicleActivity": {
                  "type": "object",
                  "properties": {
                    "ProgressBetweenStops": {
                      "type": "object",
                      "properties": {
                        "Percentage": { "type": "double" },
                        "LinkDistance": { "type": "long" }
                      }
                    },
                    "RecordedAtTime": { "type": "date" },
                    "MonitoredVehicleJourney": {
                      "type": "object",
                      "properties": {
                        "MonitoredCall": {
                          "type": "object",
                          "properties": {
                            "RequestStop": { "type": "boolean" },
                            "VisitNumber": { "type": "boolean" },
                            "TimingPoint": { "type": "binary" },
                            "ActualArrivalTime": { "type": "date" },
                            "ReversesAtStop": { "type": "boolean" },
                            "StopPointRef": { "type": "long" },
                            "ExpectedDepartureTime": { "type": "date" },
                            "AimedDepartureTime": { "type": "date" },
                            "DepartureStatus": { "type": "binary" },
                            "PlatformTraversal": { "type": "boolean" },
                            "StopPointName": { "type": "text" },
                            "ActualDepartureTime": { "type": "date" },
                            "ArrivalStatus": { "type": "binary" },
                            "BoardingStretch": { "type": "boolean" },
                            "DepartureBoardingActivity": { "type": "binary" },
                            "ArrivalBoardingActivity": { "type": "keyword" },
                            "AimedArrivalTime": { "type": "date" },
                            "VehicleAtStop": { "type": "boolean" },
                            "ExpectedArrivalTime": { "type": "date" }
                          }
                        },
                        "OriginRef": { "type": "long" },
                        "VehicleLocation": { "type": "geo_point" },
                        "Monitored": { "type": "boolean" },
                        "VehicleRef": { "type": "long" },
                        "CourseOfJourneyRef": { "type": "long" },
                        "VehicleMode": { "type": "keyword" },
                        "OriginAimedDepartureTime": { "type": "date" },
                        "DestinationName": { "type": "text" },
                        "Delay": { "type": "long" },
                        "DestinationRef": { "type": "long" },
                        "LineRef": { "type": "long" },
                        "DestinationAimedArrivalTime": { "type": "date" },
                        "PublishedLineName": { "type": "keyword" },
                        "OriginName": { "type": "text" },
                        "MonitoringError": { "type": "keyword" },
                        "IsCompleteStopSequence": { "type": "boolean" },
                        "DirectionRef": { "type": "keyword" }
                      }
                    },
                    "ValidUntilTime": { "type": "date" }
                  }
                },
                "ShortestPossibleCycle": { "type": "long"},
                "version": { "type": "long" },
                "ValidUntil": { "type": "date" },
                "ResponseTimestamp": { "type": "date" }
              }
            }
          }
        },
        "ServiceDeliveryInfo": {
          "type": "object",
          "properties": {
            "ProducerRef": { "type": "keyword" },
            "ResponseTimestamp": { "type": "date" },
            "ResponseMessageIdentifier": { "type": "keyword" }
          }
        }
      }
    }
  }
}
```

Example B.1: Elasticsearch mapping for Kolumbus VM Service response.

```
filter {
  # Split arrays into separate events if the parent field is present
  if [Answer] {
    split { field => "[Answer][VehicleMonitoringDelivery]"}
  }
  if [Answer][VehicleMonitoringDelivery] {
    split { field => "[Answer][VehicleMonitoringDelivery][VehicleActivity]"}
  }
  # Transform values to better fit with storage
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][StopPointRef]" => "
          integer" }
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OriginAimedDepartureTime]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OriginAimedDepartureTime]"
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DestinationRef]" => "integer" }
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DestinationAimedArrivalTime]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DestinationAimedArrivalTime]"
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][LineRef]" => "integer" }
  }
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][srsName]
      "}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][
      Precision]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][
      Coordinates]"}
  mutate {
    rename => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][Longitude]" =>  "[
        Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][lon]" }
  }
  mutate {
    rename => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][Latitude]" => "[
        Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][lat]" }
  }
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][id]"}
  mutate {
    convert => {
      "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][lat]" => "float"
      "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][lon]" => "float"
    }
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][Precision]" => "
          integer" }
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OriginRef]" => "integer" }
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleRef]" => "integer" }
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][CourseOfJourneyRef]" => "integer" }
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][VehicleActivity][ValidUntilTime]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][VehicleActivity][ValidUntilTime]"
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][VehicleActivity][RecordedAtTime]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][VehicleActivity][RecordedAtTime]"
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][ValidUntil]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][ValidUntil]"
  }
  # Convert JSON type string to float. Elasticsearch type is long
  mutate {
    convert => { "[Answer][VehicleMonitoringDelivery][version]" => "integer" }
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[Answer][VehicleMonitoringDelivery][ResponseTimestamp]", "UNIX"]
    target => "[Answer][VehicleMonitoringDelivery][ResponseTimestamp]"
  }
  # Convert UNIX time to UNIX_MS time
  date {
    match => ["[ServiceDeliveryInfo][ResponseTimestamp]", "UNIX"]
    target => "[ServiceDeliveryInfo][ResponseTimestamp]"
  }
  # To be continued ...
```

Example B.2: Filter section of a Logstash configuration file for Kolumbus VM Service response part 1, continued in B.3

```
  # Continuing ...
  # Remove fields that are marked for removal
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][Status]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][ItemIdentifier]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][VehicleActivityNote]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][VehicleMonitoringRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      AimedHeadwayInterval]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][CallNote]"
      }
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      FacilityConditionElement]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      SituationRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      SignalStatus]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      ExpectedHeadwayInterval]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      DestinationDisplay]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      DeparturePlatformName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][Order]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      ArrivalPlatformName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      FacilityChangeElement]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][MonitoredCall][
      VehicleLocationAtStop]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OperatorRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ExternalLineRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][Occupancy]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][FacilityConditionElement]
      "}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ConfidenceLevel]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OnwardCalls]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][TrainBlockPart]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][SituationRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][FramedVehicleJourneyRef]"
      }
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][InPanic]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][srsName]
      "}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][
      Coordinates]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleLocation][id]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][InCongestion]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][HeadwayService]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][PreviousCalls]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][JourneyNote]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][PredictionInaccurate]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][Bearing]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][Via]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleFeatureRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][RouteRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DestinationShortName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ProgressStatus]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][BlockRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DirectionName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][JourneyPatternRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][DataSource]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ProductCategoryRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][FacilityChangeElement]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ServiceFeatureRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][ProgressRate]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][OriginShortName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivity][MonitoredVehicleJourney][VehicleJourneyName]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][SubscriptionRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][SubscriptionFilterRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][DefaultLanguage]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][ErrorCondition]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivityNote]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][RequestMessageRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][SubscriberRef]"}
  mutate { remove_field => "[Answer][VehicleMonitoringDelivery][VehicleActivityCancellation]"}
  mutate { remove_field => "[AnswerExtension]"}
  mutate { remove_field => "[ServiceDeliveryInfo][RequestMessageRef]"}
  mutate { remove_field => "[ServiceDeliveryInfo][Address]"}
}
```

Example B.3: Filter section of a Logstash configuration file for Kolumbus VM Service response, part 2. Continued from B.2

Appendix C

Source Code for SDModel system

The source code for the SDModel system is attached in this PDF. Sourcecode for the SDModel system.

Acronyms

ANSI American National Standards Institute. 8, 91, *Glossary:* American National Standards Institute

API Application Programming Interface. 2, 7, 17, 23, 45, 46, 51, 52, 57, 59, 91, *Glossary:* Application Programming Interface

CORS Cross Origin Resource Sharing. 59, 91, 94, *Glossary:* Cross Origin Resource Sharing

DDL Data Definition Language. 91, *Glossary:* Data Definition Language

IoT Internet of Things. 3, 91, *Glossary:* Internet of Things

JSON JavaScript Object Notation. 7, 12, 14–18, 21, 27, 29, 38–46, 48, 50, 54–59, 66–71, 82–85, 91, 93, *Glossary:* JavaScript Object Notation

RDF Resource Description Framework. 8, 91, *Glossary:* Resource Description Framework

REST Representational State Transfer. 12, 14, 17, 45, 46, 51, 52, 57, 59, 91, *Glossary:* Representational State Transfer

SIRI Service Interface for Real Time Information. 12, 91, *Glossary:* Service Interface for Real Time Information

SOAP Simple Object Access Protocol. 12, 67, 91, 95, *Glossary:* Simple Object Access Protocol

UiS University of Stavanger. 4, 21

UUID Universally Unique IDentifier. 55, 91, *Glossary:* Universally Unique IDentifier

WSDL Web Service Definition Language. 12, 58, 91, *Glossary:* Web Service Definition Language

XML eXtensible Markup Language. 8, 91, *Glossary:* eXtensible Markup Language

Glossary

American National Standards Institute A private, not-for-profit association that administers U.S. private-sector voluntary standards [47].. 8, 91

Application Programming Interface Application Programming Interface. 2, 91

Asynchronous Javascript and XML An engine for doing http requests from the browser asynchronously, so as to not block the user interface while waiting for the server to respond [48].. 46, 57, 94

Cross Origin Resource Sharing A website can by default not request resources from other origins than its own unless the resource opens for it through its headers. This mechanism is called Cross Origin Resource Sharing.. 59, 91

Data Definition Language A data definition language or data description language (DDL) is a syntax similar to a computer programming language for defining data structures, especially database schemas. [49]. 91

data model A data model is an abstract model that organises elements of data and standardises how they relate to one another and to properties of the real-world entities. [50] . 22–24, 28, 29, 51, 53

DELETE The HTTP DELETE method requests that the server deletes the resource identified by the URI [51]. . 14, 45, 59

Elastic stack Distributed framework for document oriented search, storage and visualising of big data using Elasticsearch, Logstash and Kibana. 6, 16, 21, 24, 73, 74

eXtensible Markup Language A markup language that defines a set of rules for encoding documents in a format that is both human- and machine-readable. [52]. 8, 91

geohash A geocoding system invented by Gustavo Niemeyer and placed into the public domain, its purpose is to encode GPS coordinates in an URL friendly manner [28]. . 17

GeoJSON A geospatial data interchange format based on JSON. It defines several types of JSON objects and the manner in which they are combined to represent data about geographic features, their properties, and their spatial extents. GeoJSON uses a geographic coordinate reference system, World Geodetic System 1984, and units of decimal degrees [38]. . 17, 36, 42

GET Using the HTTP GET method in a request will retrieve whatever information is identified by the URI [51]. This is the method used for standard webpage retrieval. . 14, 15, 45, 59, 94

HEAD The HTTP HEAD method requests that the headers and not the body of a GET request is returned. It can be used for checking the validity of an url or obtaining meta information about the entity without transferring the entity body itself [51]. . 14

Internet of Things "A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies" [53].. 3, 91

inverted index A data structure which is designed to allow very fast full-text searches. An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears. [9] . 14

JavaScript Object Notation JavaScript Object Notation (JSON) is a text format for the serialisation of structured data. [33]. 7, 91

Lucene core A high-performance, full-featured text search engine library written entirely in Java [**?**]. . 13, 17, 40

ontology "An explicit specification of a conceptualisation" [13]. In this context, a conceptualisation is defined as an "abstract, simplified view of the world that we wish to represent for some purpose" [13].. 9

OPTIONS The HTTP OPTIONS method is used by a client to request information about communication options available on the request/response chain identified by the URI without implying a resource action or initiating a resource retrieval [51]. This is often used by Asynchronous Javascript and XML libraries before performing a CORS request to some resource. . 59

Paxos A family of protocols for solving consensus in a network of unreliable processors. Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures [54].. 13

POST The HTTP POST method is used to request that the entity enclosed in the request is accepted as a new subordinate of the resource identified by the URI [51]. For example when a webpage sends a form to the server, the forms content is enclosed in the request, and it is requested that this submission is accepted in the list of form submissions. . 45, 59

PUT The HTTP PUT method requests that the enclosed entity be stored at the URI, and if there is already an entity at the URI the enclosed entity should be considered as a modified version of the one residing on the server [51]. . 14, 15, 45, 59

Representational State Transfer Architectural style for distributed hypermedia systems [22] where a resource is identified by a URI and can be edited using the HTTP methods as they were intended GET to get a resource, POST to set a resource, DELETE to delete and PUT to update a resource.. 12, 91

Resource Description Framework The Resource Description Framework (RDF) is a framework for representing information in the Web. This document defines an abstract syntax (a data model) which serves to link all RDF-based languages and specifications [55].. 8, 91

Service Interface for Real Time Information XML protocol to allow distributed computers to exchange real time information about public transport service and vehicles [56].. 12, 91

Simple Object Access Protocol A protocol intended for exchanging structured information in a decentralised, distributed environment [57].. 12, 91

Triangulum smart city project "The three point project Triangulum is one of currently nine European Smart Cities and Communities Lighthouse Projects, set to demonstrate, disseminate and replicate solutions and frameworks for Europe's future smart cites" [58]. . 4, 5

Universally Unique IDentifier A UUID is a 128-bit value that can guarantee uniqueness across space and time [43]. Often used in systems where an unique id is needed but a global authority is not available or preferred.. 55, 91

Web Service Definition Language Language used to define SOAP web services.. 12, 91

References

[1] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] J. Gantz and D. Reinsel, "Extracting value from chaos," *IDC iview*, vol. 1142, no. 2011, pp. 1–12, 2011.

[3] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE Access*, vol. 2, pp. 652–687, 2014.

[4] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[5] Merriam-Webster, "metadata." [Online]. Available: https://www.merriam-webster.com/dictionary/metadata

[6] "Gartner says the internet of things installed base will grow to 26 billion units by 2020." [Online]. Available: http://www.gartner.com/newsroom/id/2636073

[7] Kolumbus, "Kolumbus real time open data." [Online]. Available: https://www.kolumbus.no/globalassets/sanntid-siri/kolumbus-real-time-open-data.pdf

[8] "Redis." [Online]. Available: https://redis.io

[9] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide.* " O'Reilly Media, Inc.", 2015.

[10] "Final report of the ansi/x3/sparc dbs-sg relational database task group," *SIGMOD Rec.*, vol. 12, no. 4, pp. 1–62, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/984555.1108830

[11] M. Hammer and D. Mc Leod, "Database description with sdm: A semantic database model," *ACM Trans. Database Syst.*, vol. 6, no. 3, pp. 351–386, Sep. 1981. [Online]. Available: http://doi.acm.org/10.1145/319587.319588

[12] Merriam-Webster, "Entity." [Online]. Available: https://www.merriam-webster.com/dictionary/entity

[13] T. R. Gruber *et al.*, "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[14] "Ontology alignment." [Online]. Available: https://en.wikipedia.org/wiki/Ontology_alignment

[15] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," in *vldb*, vol. 1, 2001, pp. 49–58.

[16] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Comput. Surv.*, vol. 18, no. 4, pp. 323–364, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/27633.27634

[17] A. V. Aho and J. D. Ullman, *Foundations of computer science*. Computer Science Press, 1992.

[18] W.-S. Li and C. Clifton, "Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks," *Data & Knowledge Engineering*, vol. 33, no. 1, pp. 49–84, 2000.

[19] J. Berlin and A. Motro, "Database schema matching using machine learning with feature selection," in *Seminal Contributions to Information Systems Engineering*. Springer, 2013, pp. 315–329.

[20] A. Doan, P. Domingos, and A. Y. Halevy, "Reconciling schemas of disparate data sources: A machine-learning approach," in *ACM Sigmod Record*, vol. 30, no. 2. ACM, 2001, pp. 509–520.

[21] "Siri whitepaper," 01 2017. [Online]. Available: http://user47094.vs.easily.co.uk/siri/schema/1.0/doc/Siri%20White%20paper08.zip

[22] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[23] G. Mulligan and D. Gračanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Winter Simulation Conference*. Winter Simulation Conference, 2009, pp. 1423–1432.

[24] "Apache lucene core." [Online]. Available: https://lucene.apache.org/core/

[25] "Leader election, why should i care?" [Online]. Available: https://www.elastic.co/blog/found-leader-election-in-general

[26] "Elasticsearch reference." [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

[27] "Numeric datatypes - elasticsearch." [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/number.html

[28] Wikipedia, "Geohash." [Online]. Available: https://en.wikipedia.org/wiki/Geohash

[29] Elasticsearch, "Logstash reference." [Online]. Available: https://www.elastic.co/guide/en/logstash/current/index.html

[30] nathansobo, "Treetop, a ruby-based parsing dsl based on parsing expression grammars." [Online]. Available: https://github.com/nathansobo/treetop

[31] Elasticsearch, "Kibana user guide." [Online]. Available: https://www.elastic.co/guide/en/kibana/current/index.html

[32] "Apache couch db." [Online]. Available: http://couchdb.apache.org

[33] D. Crockford, "The application/json media type for javascript object notation (json)," July 2006.

[34] Elastico, "Elasticsearch reference - boolean data type," 2017. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/boolean.html

[35] T. O. Group, "The open group specifications issue 7, section 4.16 seconds since the epoch." [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16

[36] Dateutil, "Powerful extensions to datetime." [Online]. Available: https://dateutil.readthedocs.io/en/stable/

[37] S. Josefsson, "The base16, base32, and base64 data encodings," Internet Requests for Comments, RFC Editor, RFC 4648, October 2006, http://www.rfc-editor.org/rfc/rfc4648.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4648.txt

[38] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub, "The geojson format," Internet Requests for Comments, RFC Editor, RFC 7946, August 2016.

[39] Wikipedia, "World geodetic system." [Online]. Available: https://en.wikipedia.org/wiki/World_Geodetic_System

[40] Z. Abedjan, L. Golab, and F. Naumann, "Profiling relational data: a survey," *The VLDB Journal*, vol. 24, no. 4, pp. 557–581, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00778-015-0389-y

[41] Elasticsearch, "Nested datatype." [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/nested.html

[42] "Private variables." [Online]. Available: https://docs.python.org/3/tutorial/classes.html#private-variables

[43] R. S. P. Leach, M. Mealling, "A universally unique identifier (uuid) urn namespace," 2005. [Online]. Available: https://tools.ietf.org/html/rfc4122#page-7

[44] "Python json package." [Online]. Available: https://docs.python.org/2/library/json.html

[45] "Cherrypy - a minimalistic python web framework," 05 2017. [Online]. Available: http://cherrypy.org

[46] "Vue.js guide." [Online]. Available: https://vuejs.org/v2/guide/

[47] R. Shirey, "Internet security glossary, version 2," 2007. [Online]. Available: https://tools.ietf.org/html/rfc4949

[48] J. J. Garrett, "Ajax: A new approach to web applications," February 18 2005. [Online]. Available: https://pdfs.semanticscholar.org/c440/ae765ff19ddd3deda24a92ac39cef9570f1e.pdf

[49] Wikipedia. (2017, 01) Data definition language. [Online]. Available: https://en.wikipedia.org/wiki/Data_definition_language

[50] ——. Data model. [Online]. Available: https://en.wikipedia.org/wiki/Data_model

[51] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999. [Online]. Available: http://www.rfc.net/rfc2616.html

[52] (2017, 05). [Online]. Available: https://en.wikipedia.org/wiki/XML

[53] ITU, "Internet of things global standards initiative." [Online]. Available: http://handle.itu.int/11.1002/1000/11559

[54] "Paxos (computer science)." [Online]. Available: https://en.wikipedia.org/wiki/Paxos_(computer_science)

[55] "Rdf 1.1 concepts and abstract syntax." [Online]. Available: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[56] "Service interface for real time information." [Online]. Available: https://en.wikipedia.org/wiki/Service_Interface_for_Real_Time_Information

[57] W3C, "Soap version 1.2 part 1: Messaging framework (second edition)." [Online]. Available: https://www.w3.org/TR/soap12/

[58] "Triangulum project." [Online]. Available: http://triangulum-project.eu