CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Framework for Virtual Knowledge Graph Construction over Time Series Data

Master's thesis in Computer science and engineering

ERIK BERG JAKOB HENRIKSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

A Framework for Virtual Knowledge Graph Construction over Time Series Data

ERIK BERG
JAKOB HENRIKSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

A Framework for Virtual Knowledge Graph Construction over Time Series Data

ERIK BERG JAKOB HENRIKSSON

Supervisor: Martin Hilgendorf, Department of Computer Science and Engineering
Advisor: Karthikeyan Jeganathan, Volvo Group AB
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

A Framework for Virtual Knowledge Graph Construction over Time Series Data

ERIK BERG
JAKOB HENRIKSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Analyses of products during and after development are essential to improve and guarantee their quality. In the automotive industry, analyses are often based on time series data coming from numerous heterogeneous sensors equipped by vehicles. The analysis of time series and sensor data is challenging due to the high volume and high variety of the data, often leading to unstructured storage solutions. In this work, we show how these challenges can be addressed relying on data preprocessing and a Virtual Knowledge Graph (VKG) approach. The data preprocessing is used to create tables from the time series data that integrate efficiently with the VKG, and the VKG itself introduces semantics, virtualization, and a graph-representation of the data. To experiment with the setup, we preprocessed the data, developed an ontology for the virtual knowledge graph, and developed a set of mappings that connect the graph to the preprocessed data. The system is evaluated using a set of analysis questions derived from a real business use case that we have encoded as SPARQL queries. The results obtained are promising, showing that a virtual knowledge graph is a viable alternative to current analytical approaches for time series data.

Keywords: Computer, science, computer science, engineering, project, thesis, virtual, knowledge, graph.

Acknowledgements

We would like to thank our supervisor Martin Hilgendorf and our examiner, Marina Papatriantafilou, for their continued support and thoughtful feedback throughout the thesis work. Our appreciation also goes to our Volvo Group supervisors, Karthikeyan Jeganathan and Ashni Reddy, whose insights and guidance have been invaluable.

A big thank you as well to our thesis cohort for the helpful discussions and for being such excellent "rubber duckies" along the way. We are especially grateful to our opponents, Jonatan Mentzer and Gustav Bruhn, for their constructive feedback and input that truly helped improve the thesis work. Lastly, we want to extend our thanks to our families for their continued support.

Erik Berg, Jakob Henriksson, Gothenburg, 2025-06-18

Contents

List of Figures

List of Tables

1

Introduction

In today's society, information has become more critical than ever, with large amounts of data being used to make well-informed decisions. Effective data collection, management, and interpretation is crucial to derive value from the vast amounts of information collected. This is especially apparent in the automotive industry, where the volume of data is rapidly increasing as the number of parameters that can be measured in a vehicle increases [1]. Big data in the automotive industry places emphasis on time series data originating from sensors.

Sensor data is routinely used by automotive companies to test, validate, and improve their products. These sensors measure various parameters at high frequency, resulting in large volumes of heterogeneous time series data. This data is a key asset for improving products and increasing production efficiency. However, the absence of proper data modeling often leads organizations to store the data in *data lakes*. While data lakes provide a flexible and scalable storage environment, they lack semantics, making it difficult to extract meaningful insights. In particular, relationships between data points and the contextual meaning of sensor output are often undocumented or implicit, making data analysis a considerable challenge.

The lack of semantics and the challenge of utilizing the data means that data analysts spends more time integrating, cleaning, de-duplicating, and semantically homogenizing the data [2]. As a result, a lot of time is spent preparing the data instead of analysis. Volvo Group is no exception to this challenge. Despite collecting extensive time series data from sensors, utilization of this data is often limited to smaller, use-case specific applications. This requires data analysts to spend a lot of time integrating and semantically homogenizing the data to draw meaningful insights.

The problem in this thesis is characterized by four distinct challenges: **high data volume** of time series data, **high data variety** of sensor data, presence of **complex relationships** that is not modeled, and **lack of semantics**. To address the problem and these challenges, this thesis proposes a framework comprising a Virtual Knowledge Graph (VKG) and data preprocessing supporting the VKG. The VKG addresses the challenges in the following manner:

The challenge of **high data volume** is mitigated through a combination of preprocessing and the use of the VKG. Preprocessing involves sampling, which we show can reduce the volume of data by a factor of around 10^3. The VKG also introduces virtualization, where the knowledge graph is created virtually at query time, dy-

namically accessing the underlying data sources [2]. This enables data to remain in its original format, eliminating the need to transform data into a graph structure. A known limitation of VKGs is their reduced performance when interacting with unstructured data sources [3]. To address this, the preprocessed data is ingested into a relational database with relevant keys and indexing, acting as the underlying data source to the VKG.

The challenges of **high data variety**, **complex relationships**, and **lack of semantics** are mitigated by the knowledge and graph aspects of a VKG. In the graph, nodes represent entities or data points, and edges capture relationships. This structure efficiently models **complex relationships** [4]. Knowledge is added through an *ontology*, which provides semantics by enriching nodes and edges with a domain-specific vocabulary that allows data analysts to fetch data based on known concepts in the domain [5], [6]. This semantic integration addresses the challenges **high variety** and **lack of semantics** by enriching heterogeneous data with more information, enabling better interpretability and easier access to information [7].

To evaluate the proposed VKG framework, it is applied to a use case at Volvo Group. A set of domain-specific analytical questions is encoded in SPARQL and is used to evaluate the performance of the VKG in terms of three metrics: scalability, effectiveness, and extensibility.

2

Background

This section provides descriptions of important concepts for the thesis. This includes big data, time-series data, ontologies, graphs, databases and knowledge graphs.

2.1 Big Data in the automotive industry

Big data is commonly broken down as spanning three, four, or five dimensions, the so-called *five Vs of big data* [1]. These dimensions are defined as follows:

Volume refers to the size of the data. In the automotive industry, it is estimated that a customer fleet of approximately one million vehicles can generate approximately 560 TB of data per day, while a test fleet of approximately one thousand vehicles generates approximately 4.5 TB per day [1]. These estimates are from 2014, when the article was published, indicating that these volumes are larger today.

Variety refers to the diversity of the data sets. In the automotive industry, data sets can be very diverse. Common data types include time-based signals or sensor data from vehicle sensors, scalar data from diagnostic services, images, video, and administrative data, among others [1].

Velocity refers to the timing of the data, whether it needs real-time processing or near real-time processing [1]. Examples of this in the automotive industry are active safety functions or autonomous driving, where safety decisions must be made almost instantly.

Veracity refers to the ability of determining if data from multiple sources can be trusted [1]. In the context of the automotive industry, this is particularly relevant to ensuring the safety and security of vehicle drivers and passengers.

Value refers to the value that data can have in terms of increased revenue, new services, and improved products. In the automotive industry, this relates to developing better products and introducing new services, as well as generating additional revenue streams, such as selling data to insurance companies or road administration authorities [1].

In conclusion, the most relevant aspect of this thesis is the fact that big data in the automotive industry has the characteristics of high volume and high variety. This is especially true when discussing sensor data, which is a type of time-series data.

2.2 Time series data

The term time series data is ambiguous, as its definition often varies depending on the author and the domain in which it is used [8]. This ambiguity raises a problem: What is classified as time series data in one domain may be a simpler problem than what is referred to as time series data in another domain. To clarify this ambiguity, this thesis will use the following distinctions.

Time series data: Is also referred to as temporal data in the literature. Any data is stored as a series of discrete timestamp-value pairs. The data can be of any data type.

Event-based vs continuous time series: The distinction here is based on the nature of the underlying phenomenon. Event-based time series are a series of discrete phenomena in time stored as time series, e.g, a sensor logging error codes for a machine or a sensor recording stateful, discrete events. The discrete nature of these time series means that they are considerably simpler than the continuous time series, as precisely every discrete event is needed. Continuous time series are time series created through continuous processes, e.g, data generated by a temperature sensor representing engine temperature in a vehicle or a speed sensor representing the vehicular speed.

Historical vs Streaming time series: Streaming time series are any time series that is updated while it is processed, which is also known as stream processing. However, time series that are not updated in parallel to analysis tasks are considered historical time series. This typically occurs when data are first collected from sensors and subsequently, analysis is performed on that data.

This thesis's focus is on historical time series data in the form of mainly event-based time series but also some continuous time series. Their distinct properties are therefore explained below.

2.2.1 Event-based time series

Event-based time series are concerned with capturing the exact moment when an event has occurred. Any extra logging of data is redundant, as it does not provide new information. For example, a sensor logging continuously about different states at a rate of 100 Hz means that every data point that does not reflect a new state can be discarded. In the case that extra unnecessary data points are logged, simply not sending these to the server can significantly reduce the data volume.

2.2.2 Continuous time series

Time series representing continuous phenomena are captured digitally through a signal measuring these phenomena at some given frequency [8]. These signal measurements, or sampling as it's often referred to, are what is known as a continuous time series. Values are only stored when samples are taken, which means that the information between measurements is lost. From this, it follows that a high sampling

frequency yields a lower information loss.

One characteristic of continuous time series is that they have a large data volume and a high dimensionality (i.e. number of data points) [8]. This makes it infeasible to store the entire volume directly, both in terms of storage capacity and computationally. Instead, a representation that reduces the dimensions is desired. The simplest method for this is to subsample (or downsample, as it is sometimes referred to) the data. This means that instead of using all data points, only a subset of them is selected. One way to do this is to choose the data points at a rate of m/n, where m is the length of the original time series, and n is the dimensionality after subsampling. This reduces the series to a size of n samples, e.g, choosing a rate of 5, only every 5th data point would be used. This method can affect the shape of the data at high sampling rates, leading to inaccuracies when the data is analyzed.

Another approach is to use a piecewise aggregate approximation, where a rate is chosen. This rate represents the number of segments into which the series is split, and then for each of those segments, the data is aggregated using functions such as average [8].

There exist several more methods for reducing the dimensionality of continuous time series, such as singular value decomposition, laplacian eigenmap, locally linear embedding, and discrete Fourier transform [9]. All of these have their niches and reasons for use, depending on both the nature of the analysis that is performed on the time series and on the nature of the continuous time series itself.

2.3 Ontology Engineering

Ontology engineering is part of the scientific domain *Knowledge Representation and Reasoning*, which was part of the early science of Artificial Intelligence (AI). One of the areas of knowledge representation focuses on helping artificial intelligence make intelligent decisions based on knowledge in the real world [10]. This primarily relates to enabling a machine to mimic human reasoning based on the knowledge a human possesses. An example of this is that if a person is born in the city of Stockholm, one could infer that the person is Swedish based on the knowledge that Stockholm is the capital of Sweden. There exist multiple structures for knowledge representation, one of which is creating a knowledge base using an ontology [10].

2.3.1 Ontology

For the thesis, the ontology is defined more formally as a tuple of a *Terminology Box* (TBox) and an *Assertion Box*(ABox). A box can be defined as a container of information. The TBox is responsible for defining classes, relationships, and their respective constraints and properties in a domain. It can be seen as equivalent to a schema of the ontology, which provides a formal representation of how concepts are organized and connected in the domain. The ABox contains assertions about the instances of the domain, such as membership, relationships, and equalities between instances based on the schema defined by the TBox [11]. An example of this is

provided below, in the context of a university, where the TBox defines the following classes and relationships:

- Person P
- *Course* : C
- *University*: U
- Student S is a Person P: $S \subseteq P$
- *Professor* PF is a *Person* P: $PF \subseteq P$
- *Teaches* T is C: $T \subseteq PF \times C$
- *offersCourse* oC: $oC \subseteq U \times C$
- *affiliatedIn* A: $A \subseteq P \times U$
- *takesCourse* tC is C: $tC \subseteq S \times C$

The ABox defines the assertions for the instances based on the schema of the TBox. Continuing with the example of the university, the ABox is defined as:

- $C = \{Cryptography\}$
- $U = \{Chalmers\}$
- $S = \{Alice\}$
- $PF = \{Bob\}$
- $T = \{(Bob, Cryptography)\}$
- $oC = \{Chalmers, Cryptography\}$
- $A = \{(Alice, Chalmers), (Bob, Chalmers)\}$
- $tC = \{Alice, Cryptography\}$

Note that even though Alice is defined as a student and Bob is defined as a professor, they are implicitly defined as a person as well. This is further explained in section 2.3.3. The university example is a running example throughout the background chapter.

In practice, unique identifiers such as person IDs would be used instead of names for students and professors. However, in this example, names are used to make the conceptual structure easier to follow.

2.3.2 Ontology Approaches

An ontology can be created in different ways, where the two most prominent approaches are the top-down and the bottom-up approach [6]. The top-down approach first defines the vocabulary through domain experts and then relates it to the data, while the bottom-up approach is more data-driven, creating the entities based on the underlying data source. In the context of the thesis, the ontology will be developed

using a combination of the two approaches, through an analysis of the underlying data sources and workshops with domain experts.

2.3.3 Ontology Implementation

There exist multiple languages to construct an ontology. An example of this is the W3C Web Ontology Language 2 (OWL 2), which can represent knowledge of entities, their properties, and how they are related to each other. OWL 2 is maintained by the World Wide Web Consortium (W3C). It is mandatory for programs that implement OWL 2 to support the format of RDF in an XML syntax. The RDF format is further explained in section 2.4.2.

In figure 2.1, one can see an overview of the OWL 2 language model. The OWL 2 language can represent ontology structures in two ways: directly, based on semantics, and as a graph structure, with distinct features.

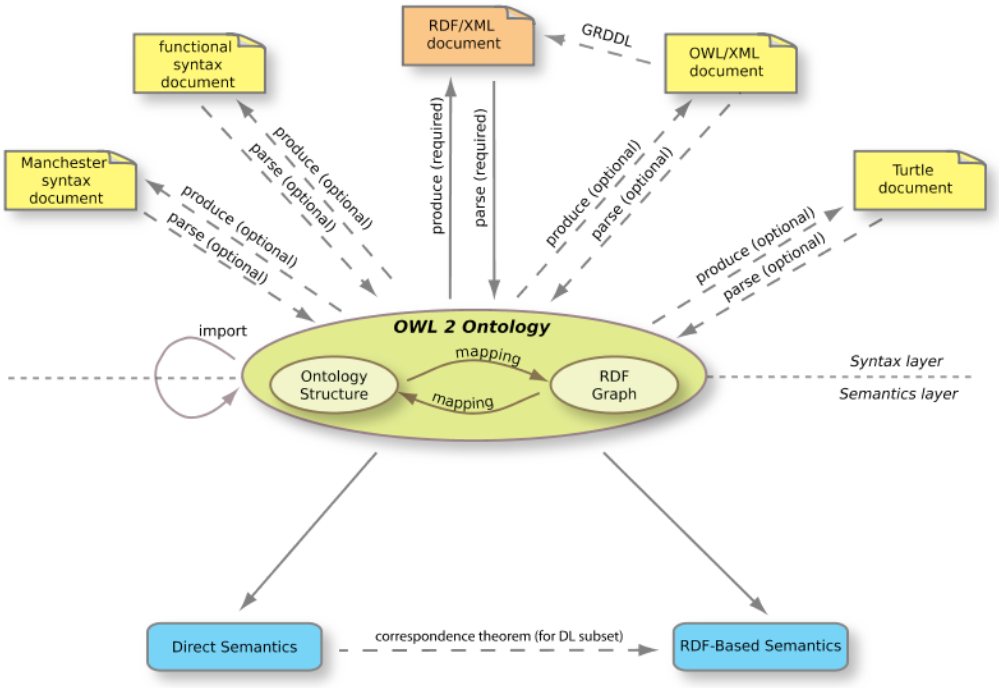


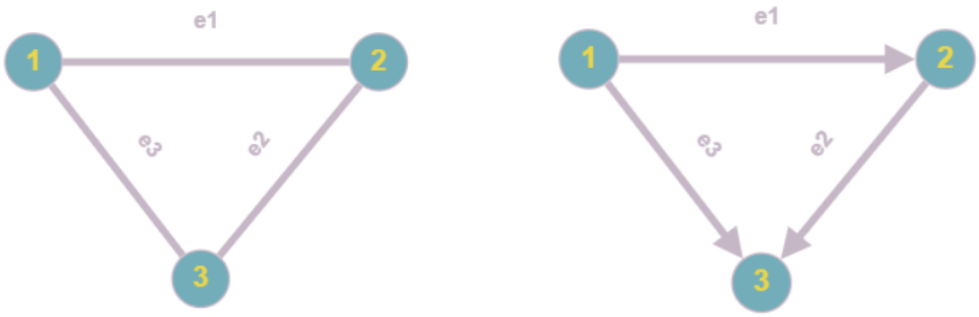
Figure 2.1: Overview of OWL 2 from *OWL 2 Web Ontology Language Document Overview (Second Edition)* [12].

Direct semantics is related to a concept called *Description Logic* [11], which is connected to a logical representation rather than a graph structure. This structure enables the possibility of inferring more relationships than defined in the ABox by leveraging the structure in the TBox using reasoning capabilities. Returning to the example of the university, when *Alice* is defined as a *Student* in ABox, implicitly she is inferred as a *Person* as well. These inferences are helpful as the model expands, both for understanding and for querying purposes, as they enable grouping multiple sub-entities under a broader entity. Using the RDF graph structure instead constructs a knowledge graph that can be queried to extract expressive relationships.

Direct semantics is also implicitly used through reasoning capabilities to infer more relations [12].

2.4 Graph

A graph can be defined as an ordered pair $G = (V, E)$ where V is a set of vertices and E is a set of edges, which is defined as a pair of vertices such that $E \subseteq V^2$. An example graph can therefore be defined as follows: $V' = \{v_1, v_2, v_3\}$, $e1 = (v_1, v_2), e2 = (v_2, v_3), e3 = (v_1.v_3)$, $E' = \{e1, e2, e3\}$ where $G' = (V', E')$. The graph is different depending on whether it is an *undirected* or *directed graph*, highlighted in figure 2.2a, 2.2b.



(a) Undirected graph

(b) Directed graph

Figure 2.2: Undirected (2.2a) and directed (2.2b) graph examples

2.4.1 Directed and Undirected graphs

The pairs defined as edges can be represented in an ordered or unordered fashion, creating two different graph structures. If they are unordered, an *undirected graph* is created, shown in figure 2.2a, which means that if $\{v_1, v_2\}$ then $\{v_2, v_1\}$ is considered the same edge. If ordered, the graph becomes a *directed graph*, which means that the order of vertices in the pair matters. This means that $\{v_1, v_2\}$ is distinct from $\{v_2, v_1\}$, which is shown in figure 2.2b.

2.4.2 Resource Description Framework and SPARQL

The *Resource Description Framework* (RDF) [13] can be formally described as a data format that captures $\langle subject, predicate, object \rangle$ where *subject* and *object* are nodes, and *predicate* is the definition of the edge that forms a graph structure.

RDF was developed by the World Wide Web Consortium (W3C) community to provide semantic meaning to web resources. This enables machines to think and reason about the data by leveraging *semantic markups* [14].

The listing 1 shows how an RDF representation can be defined given the university example from section 2.3.1.

```
#prefixes
@prefix : <http://example.org/university#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
# TBox Definitions
:Person rdf:type rdfs:Class .
:Student rdf:type rdfs:Class ; rdfs:subClassOf :Person .
:Professor rdf:type rdfs:Class ; rdfs:subClassOf :Person .
:Course rdf:type rdfs:Class .
:University rdf:type rdfs:Class .

:teaches rdf:type rdf:Property ;
    rdfs:domain :Professor ; rdfs:range :Course .
:takesCourse rdf:type rdf:Property ;
    rdfs:domain :Student ; rdfs:range :Course .
:offersCourse rdf:type rdf:Property ;
    rdfs:domain :University ; rdfs:range :Course .
:affiliatedWith rdf:type rdf:Property ;
    rdfs:domain :Person ; rdfs:range :University .

# ABox Assertions
:Alice rdf:type :Student ;
    :takesCourse :Cryptography ;
    :affiliatedWith :Chalmers .
:Bob rdf:type :Professor ;
    :teaches :Cryptography ;
    :affiliatedWith :Chalmers .

:Cryptography rdf:type :Course .
:Chalmers rdf:type :University ;
    :offersCourse :Cryptography .
```

Listing 1: University Ontology example in RDF syntax

The definition of a specific *subject*, *object*, and *predicate* is a uniform resource identifier (URI), which is an abstract data source that creates a unique concept on the web [15]. It also provides an efficient way to categorize a node in the graph into the ontology. An example of this is that a *Student* Alice and a *Professor* Alice can be defined as distinct individuals in the following way:

- `http://example.org/university#student/Alice`
- `http://example.org/university#professor/Alice`

The SPARQL Protocol was introduced in 2004 to make it easier to query RDF data. In 2008, SPARQL became the W3C recommended query language for RDF [16].

2.4.3 Knowledge Graph

A *Knowledge Graph* (KG) is generally defined as a directed graph $KG = (V, E, L)$, where V is a set of vertices representing entities, E is a set of edges that represent the relationship between entity pairs, such that $E \subseteq V \times V$ and L is a set of labels. There exist two types of labels, L_V, which is the label of an entity specifying the type and attributes, and L_E, specifying the type of edge [17]. A KG can be modeled in the form of an RDF triple, where the edge is seen as the predicate between $\langle V_{subject}, V_{object} \rangle$ pairs, highlighting that the graph is directed 2.2b. The labels remain the same, where $(V_{subject}, L_{subject})$ represent the subject and its attributes, (V_{object}, L_{object}) represents the object, and $(E_{subject,object}, L_{E_{subject,object}})$ represents the predicate [18]. A set of RDF triples represents a KG, which can be stored in a table format or in a graph database that supports this format.

A powerful application of KG is to define an ontology for it, as this gives the computer the ability to deduce knowledge [19]. Relating this to section 2.3, an ontology makes it possible to perform more complex queries on the KG based on terminology defined in the ontology, rather than the actual data in the KG [4]. An example of this can be related to the university example 1, where a query can be made on *Person*. Through deductive knowledge specified in the ontology, the KG will know that a *Professor* and *Student* are a person as well, see listing 2.

```
SELECT ?person ?name
WHERE {
  ?person a foaf:Person .
  ?person foaf:name ?name .
}
```

Listing 2: SPARQL query example using the university example.

In terms of querying on a KG, there exist different types of queries [18]. **Simple queries** (listing 3) involve a single KG relation and a single formal query pattern. In contrast, a **complex query** (listing 4) involves multiple relations, multiple patterns, and can have aggregations as well. There are also **path queries** (listing 5), where the query aims to find a path in the KG, such as the shortest path. This can be done by matching the pattern on the sequence of edge labels, L_E, using regular expressions.

```
SELECT ?person ?name WHERE {
  ?person a foaf:Person .
  ?person foaf:name ?name .
}
```

Listing 3: **Simple** query expressed in the SPARQL language using the university example. Here a single concept is queried for with a single relation to a name.

There is a major distinction in terminology regarding *querying vs. question answering* [18]. A query has a structure and is based on graph patterns, a logical query, and uses a query language such as SQL or SPARQL to formulate the query. Question

answering, on the other hand, is the task of answering an unstructured natural language question over a KG, such as *"Where is the capital of Spain?"*. This distinction is necessary, as the scope of this thesis will only focus on the former, specifically the querying part. We will **not** be discussing the question answering part.

```
SELECT ?course (COUNT(?student) as ?num_students) WHERE {
  ?student a :Student .
  ?student :takesCourse ?course .
} GROUP BY ?course
```

Listing 4: **Complex** query expressed in the SPARQL language using the university example. Here, an aggregation in the form of a count is used as well as a group by clause.

```
SELECT ?student ?course WHERE {
  ?student :affiliatedWith/:offersCourse ?course
}
```

Listing 5: **Path** query expressed in the SPARQL language using the university example. The path is expressed by the relationship path ":affiliatedWith/:offersCourse".

2.5 Databases

In this section, we will describe two different types of databases: traditional relational databases and graph databases, outlining their respective properties, advantages, and disadvantages.

2.5.1 Graph Databases

Graph database systems are commonly described in the research community as systems that are designed to manage graph-like data following basic principles of database systems, such as persistent data storage, physical/logical data independence, data integrity, and consistency [5].

Types of graph queries are defined in the paper *Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries* [5]. The types of queries are:

- Local queries involving single vertices or edges.
- Neighborhood queries retrieving all edges adjacent to a given vertex.
- Traversal queries exploring some part of the graph, usually starting at a root vertex and traversing some part of the graph
- Global graph analytics queries, which consider the whole graph but not necessarily every property therein.

An important distinction to make for graph databases is whether the graph structure is *native* or not. There exist several types of graph database systems that use a storage backend that is not a native graph format, e.g, using a relational database to store the graph. The native graph databases, on the other hand, use some graph data model to store the graph, such as a labeled property graph or an RDF graph [5]. This is important because a native graph database is significantly better at handling graph data and graph queries than a non-native graph database. However, it will still work with a non-native graph database.

2.5.2 Relational Databases

A Relational Database (RDB) is a type of database that stores data in a structured table format [20]. Each row represents one tuple, while each column represents a variable that defines that tuple. The structure of the database often relies on keys, where the *primary key* of a table is a set of columns that uniquely identifies the entries of the table. The *foreign key* is when the primary key of one table is linked to another table. This structure, comprising keys, constraints, columns, data types, and other elements, is referred to as a database schema [20].

There exist multiple kinds of relationships that can be modeled using the keys of an RDB [20], being:

- One-to-one: For each entry in one table, there exists one entry in the other table
- Many-to-one: Many entities share a relationship with another table, for example, multiple students take the same course.
- One-to-many: An entity has a relationship with multiple entries in another table. For example, a student has various courses.
- Many-to-many: Many entities can have many relationships with different entries in another table. For example, students take multiple courses, and every course has various students in it.

Constraints can also be set to restrict one or more columns for each tuple. For example, **NOT NULL** restricts null entries, **CHECK** conditions for specific conditions, and **UNIQUE**, which ensures that the columns are unique.

An RDB is queried using Structured Query Language (SQL) [21]. Furthermore, by utilizing constraints, keys, and following design patterns when developing an RDB, it is possible to decrease data redundancy and duplication [20].

2.6 Virtual Knowledge Graph

A *Virtual Knowledge Graph* (VKG) can be defined as a triple O, M, S where O is the *ontology*, M is the *mappings*, and S is the *schema* of the underlying data sources [22]. The mapping links the ontology from the underlying data sources by translating the graph query into native queries in the local data sources. This in turn, allows

graph querying over data sources that are not native graphs. This also highlights the *virtual* property, which is that the data is not ingested into a graph database, but remains in the native storage [2]. Implementing a VKG across multiple data sources, therefore, does not result in data duplication as the data remains in its original location.

An example of how a VKG system works is presented in 2.3. First, a SPARQL query is formulated by the user. The SPARQL query is subsequently rewritten into a SQL query, taking into account the mappings and ontology of the VKG [23]. This SQL query is then executed on the underlying database and the response is returned by the VKG as RDF triples.

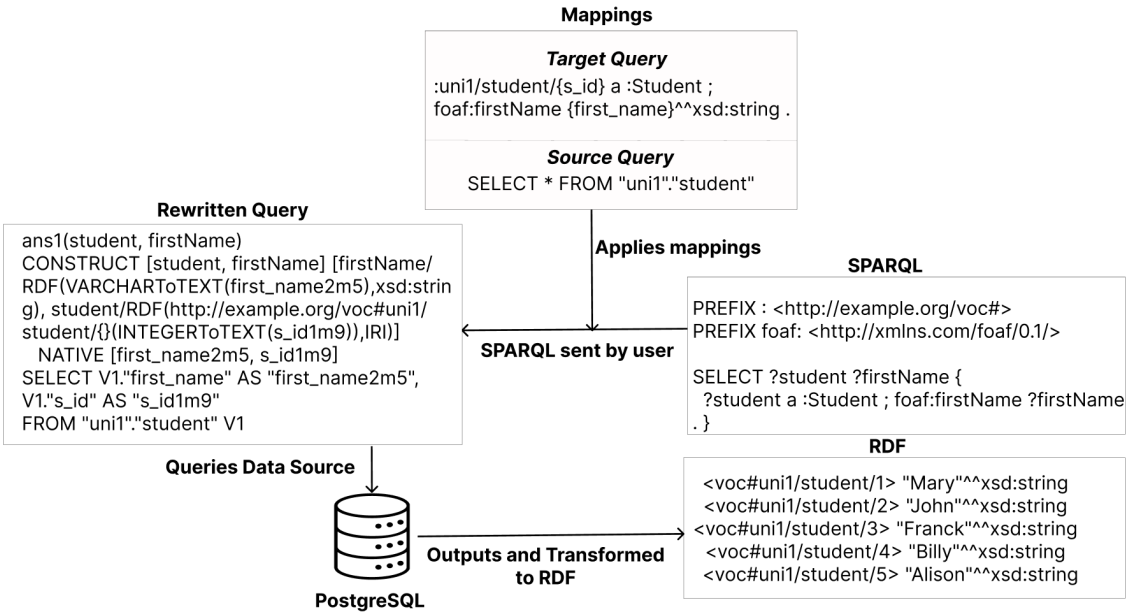


Figure 2.3: Example of SPARQL to RDF output using a VKG

2.6.1 Mappings

The mappings are essentially what populates the ABox given the TBox and the data source schema [22], which are specified using a mapping language such as R2RML [24]. An example of how a mapping can look, related to the RDF university example 1, can be seen in listing 6. The source query is what fetches the actual data from the data source, which in this example is a relational database that can be queried using SQL. The target query is then responsible for re-representing the data, according to the specifications made in the TBox. Once the mapping phase is completed, the ABox has been populated, which means that the virtual knowledge graph is now queryable.

```
#Target query in RDF format
:uni1/student/{student_id} a :Student ;
    :name {name}^^xsd:string ;
    :affiliatedWith :uni1/{university} ;
    :takesCourse :uni1/course/{course} .

#Source query in SQL
SELECT student_id, name, university, course
FROM Students;
```

Listing 6: Mapping example of Target Query and Source Query

2.6.2 Data Sources

For the VKG system to be implemented, it needs data sources, as the *virtual* part of a VKG specifically requires that the actual data be *not* stored in the KG. Instead, it is stored in its own data source or multiple data sources. A data source can have different formats, such as relational (SQL) databases, JSON documents, CSV documents, or some other NoSQL data source.

A VKG system aims to provide a queryable, unified view of the data sources. The VKG system also makes use of the metadata in the data sources, such as keys and constraints, to optimize query rewriting [25]. Conversely, if the data are not stored in a data source, but instead natively in a graph, the term *materialized* KG is used.

3

Related Work

In this chapter, some tools, frameworks, and a relevant case study are presented. First, Ontop is introduced, as it is the VKG system chosen for this thesis. Then, a framework for designing and building an enterprise knowledge graph is presented, which aligns with how the proof of concept is implemented. This is followed by an introduction to Sketching. Lastly, a case study made at Bosch is presented, along with its impact on this thesis.

3.1 Ontop

Ontop is an open-source VKG that can query multiple data sources based on the semantics captured in a KG or a VKG. This is beneficial for companies with larger datasets as it avoids data duplication. *Ontop* has seen good results in industry, for example at Siemens [26]. The problem that was solved at Siemens was a simplified way to query sensor data, relevant for monitoring and error detection. For this to work, they enhanced the database with an ontology layer, using terminology that is commonly used by Siemens engineers. This allows queries made in the vocabulary of a domain expert [25].

3.2 Designing an Enterprise Knowledge Graph

The book *Designing and Building Enterprise Knowledge Graphs* [27] proposes a methodology for constructing an enterprise KG, based on a relational data source.

Identified in the book are a number of current storage approaches, with the approach of a data lake and data warehouse being the most similar to the use-case of the thesis. The data warehouse approach is defined as a general solution that integrates data from disparate sources for the purpose of data analysis and decision-making. The data lake approach is defined as a data warehouse that allows the company to dump any type of data into it, where transformations take place after the data are in the lake, i.e. extract, load, and *then* transform [27]. This is exactly the case for this thesis use-case where data are extracted and loaded into the data lake, and then it is up to the analysts to transform it into a format that they can use.

3.2.1 Building the knowledge graph

The book proposes a process consisting of three phases for the construction of a KG. The first phase is called *Knowledge Capture* in which the business question is analyzed and understood. According to Sequeda et al. this is done in a collaborative environment where the KG schema and expected data output is found out through e.g whiteboard sketches. Typical questions that need to be answered as part of this phase can be found in table 3.1.

	Questions
What?	What is the business problem? What are the business questions?
Why	Why do we need to answer these questions? What is the motivation?
Who?	Who produces the data? Who consumes the data? Who is involved?
How?	How is this business question answered today, if at all?
Where?	Where are the data sources required to answer the business questions? Are these databases? Spreadsheets? or something else? How are the data sources accessed?
When?	When will the data be consumed? Real-time? Daily? Update criteria?

Table 3.1: Questions answered during the Knowledge Capture phase

The second phase is called *Knowledge Implementation* in which the KG schema and mappings are implemented based on the findings in the first phase. Here, the KG is supposed to be validated against the requirements established in the first phase. This is also the phase where the mappings are implemented, based on the design patterns and the whiteboard sketches from the first phase. Thus, the sanity check for the knowledge implementation is to ensure that the implementation can answer the business questions that are specified during the first phase under the ”What?” question. The knowledge implementation can be made in a multitude of different tools depending on the requirements of the graph [27].

Lastly, the third phase is called *Knowledge Access* in which the data product is made available. This product can be consumed in different ways, e.g by traditional data tools that consume the tabular output, or by graph tools that consume the graph output. The data analyst can now use their preferred tools to interact with the data product. In this phase, according to Sequeda et al. [27], the original business question from the first phase should also be analyzed and care should be taken that this phase actually provides the answers to the original question (corresponding to "What?" in table 3.1).

3.2.2 Mapping design patterns

Design patterns presented in the book is of a general kind and proposes patterns for the process of mapping relational data sources to a graph format [27]. The

design patterns are categorized into direct, complex concept, complex attribute, and complex relationship mapping patterns.

Direct mapping patterns are mappings where there is a 1:1 correspondence between either table-concept, column-(concept/relationship) attribute, or foreign key-relationship [27]. An example of 1:1 correspondence between table-concept is when a table is directly responsible for the instantiation of a concept, as can be seen in listing 7.

Complex mappings are defined as complex if an arbitrary SQL query is required in the source query. Examples of such arbitrary SQL queries are the WHERE clause, JOIN, CONCAT, CASE, etc. These can be used to define concepts, attributes, and relationships. Perhaps the simplest example, using the university example of the theory chapter 1, of a complex concept mapping is when the WHERE clause is used: A concept "Professor" is defined by the SQL query "SELECT person_id FROM persons WHERE persons.profession = 'professor' ".

The mapping patterns are important to this thesis, as they both form the basis of how mappings are defined in the mapping part of the implementation, but also because they can be used to define how complex these mapping actually are.

```
#Target query in RDF format
:uni1/{university_id} a :University

#Source query in SQL
SELECT * FROM universities;
```

Listing 7: Example of a direct table-concept mapping where the target is directly instantiated by selecting everything from one single table.

3.3 Sketches

Sketches are methods for reducing the dimensionality of data and extracting statistics from said data using, for example, linear transformations [28]. They have been introduced in the literature due to increased ability to measure and capture information, both in academia and industry, using e.g. sensors. However, processor power, memory, and disk speed are not growing at the same rate that the ability to measure information is. Therefore, sketches are introduced as a way to drastically reduce dimensionality and make the data manageable while also being able to extract statistics.

Sketches work by performing approximate queries on streamed data [29]. Typically, these queries are run with fixed memory and an approximation factor (limiting the acceptable error), and can typically be done in one pass. There are different kinds of sketches, such as frequency based sketches and distinct-value sketches. Frequency-based sketches are mainly concerned with summarizing the observed frequency distribution of the dataset. Distinct-value sketches answers queries on distinct values such as cardinality of attributes or cardinality of operations performed on certain

attributes [29]. What they all have in common is that they can only capture certain properties, such as unique occurrences, frequency estimation, set membership, etc. No single sketch can capture all properties.

Most relevant to this thesis are sketches that can be performed on time-series data, such as the Deep Time Series Sketching algorithm [30]. This sketch allows for a deeper understanding of the time-series data that industrial systems nowadays collect, where different patterns are identified and highlighted.

To summarize, sketches deal with big data and streamed time, where the data can be either event-based or continuous. Sketches summarize data and answer queries by approximating the results. While this thesis mainly focus on the historical, event-based time series data, sketches offer an alternative way to reduce dimensionality even before it gets stored in a database. Thus, sketches offer a way of understanding the data even in cases where the data is too large to feasibly access.

3.4 Case study: Bosch Manufacturing Data

One example of a successful integration of a VKG for sensor data can be found in the paper "Semantic Integration of Bosch Manufacturing Data Using Virtual Knowledge Graphs" [3]. In this paper, the methodology involves both a bottom-up and top-down approach towards the creation of an ontology for the manufacturing processes at Bosch.

This involves both the identification of core concepts, and the phases that is specific for this manufacturing process.

The data sets used for the Bosch study comprised of 3.15 GB, 31 GB, and 59 GB respectively [3], comparable sizes to a reduced subset of all available data at Volvo. The largest data set here is comprised of about 13M lines, which is comparable to one vehicle's worth of data collected over one week at Volvo. Thus, for our use case, scalability is identified as the major bottleneck and something that is dealt with in the implementation chapter.

One major difference between the study at Bosch and our data set is that at Bosch, the data generated by the machines in their manufacturing process is different based on each machine. At Bosch, one type of machine logs data of what it has processed and the other type of machine logs all faults that it can find in the manufactured products. For the use case of this thesis, all the data is stored together in a data lake which means that creating an ontology based on the inherent knowledge that different tables mean different things is very difficult. Instead, this can be done by writing SQL queries to fetch the relevant data from the data lake and thus generate results that look similar to what can populate the ontology.

4

Problem and Aim

Today, big companies tend to collect large amounts of data without properly storing it [27]. This is especially true for companies that collect data from a large number of sensors. These sensors often capture time-series data, which is complex to store by nature because of the volume of data produced by the sensors. Sensors also tend to have a high variety in the data they output [1], by measuring different, heterogeneous metrics. This further increases the complexity of storing data in an organized manner. In total, these factors influence companies to store the data in large data lakes to avoid handling these difficult tasks.

This leads to the problem of data analytics becoming more difficult. Because the data are stored in an ad-hoc manner in the data lakes, this makes it hard for data analysts to know which sensors or data to use for a given task. I.e there is a lack of semantics describing the data and how it is related. Analysts therefore, have to make a fragmented analysis focused on small use-cases, which makes it hard to extract meaningful insights and fully leverage the potential of the dataset.

The unstructured nature of data lakes makes it difficult for data analysts to make analyses based on the entire data set. Data analysts have been reported to spend 80 to 95% of the time integrating, cleaning, and transforming the data in an ad-hoc manner for specific analytical tasks [2]. Thus, data analysts are spending a lot of time preparing the data, which leaves very little time for performing the actual analysis.

4.1 Problem Statement

Given the high volume and variety of time series data and the lack of semantic context, this thesis develops a proof of concept comprising a Virtual Knowledge Graph (VKG) and data preprocessing supporting the VKG. There is research to suggest that structured data is preferable for a VKG [25], which is why there is a need for data preprocessing to support the VKG. This also addresses the problem of analyzing unstructured *Big Data*. The resulting system aims to reduce data preparation overhead for analysts and facilitate more effective, semantically driven data analysis.

Properties of interest for the problem include how to effectively handle time-series data for analytical tasks, as the data tends to be in high volume and of high variety.

Furthermore, this, in conjunction with the nature of data lakes, means that the data are categorized as Big Data [1]. This requires a different approach than if the data volume were smaller. Particularly in the domain of time series, where the proof of concept needs to find a way to keep the data volume low. See sections 2.1 and 2.2 for background on data volume and time series data.

The properties of time series and big data relate to a trade-off between materializing and virtualizing data. Materialization involves migrating data to a graph database according to the ontology of the proof of concept, while virtualization keeps the data in its original form or another relational database. Given the high volume and variety of data, migrating everything may be impractical. Thus, there exists a trade-off in which data to virtualize and which to materialize. For this thesis, we focus solely on the virtualized approach due to time constraints.

Furthermore, another interesting property of the problem is that the complexity of data handling, such as data integration, cleaning, and transforming, currently falls on data analysts. Shifting this complexity to a system such as a virtual knowledge graph or another role, such as a data engineer, could reduce the workload of the analysts. This shift would free up time for analysts to perform more analyses and conduct them in greater depth, making analytical tasks more efficient and effective.

4.2 Aim

The aim is to enable more efficient and effective data analysis by enriching time series data with semantic context, capturing knowledge inherent in the company. This is achieved by constructing a VKG that captures domain knowledge and relationships within the data points, providing data analysts with a more intuitive and expressive tool for fetching data given an analytical task, effectively reducing the manual data preparation needed.

4.2.1 Use Case

The problem and aim are related to an industry use case at Volvo Group. An overview of the use case can be seen in figure 4.1. Currently, trucks are equipped with a large number of sensors for testing and validation purposes during the development process. These sensors generate a lot of time series data, both continuous and event-based. The sensor data are heterogeneous, which is indicated by the high variety of data in the data storage.

Additionally, the data pipeline consists of data logged to MDF4 file format [31], which is then transformed into parquet files [32] and stored in a data lake. This means that performing analysis and fetching the right data for analysis becomes increasingly difficult as the number of sensors increases.

A Knowledge Graph system is known to handle heterogeneous data well. In such a system, sensors are connected via relationships, and there is also the added benefit of being able to incorporate semantics into the graph. This means that both

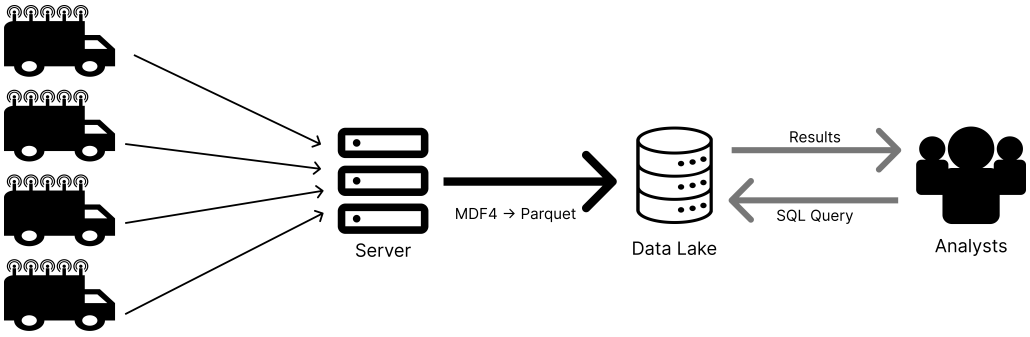


Figure 4.1: Overview of use case

descriptions and explanations of e.g. sensors are inherent in the graph, as well as the links that exist between sensors.

The analysis task that this thesis focuses on is Charging Success Rate (CSR), which is directly related to the data stored in the data lake. This is an important metric as vehicles occasionally fail to charge. If a vehicle has no battery and cannot be charged, it cannot drive, resulting in costly downtime. The data comes from around 100 different sensors, consisting of both continuous and event-based time series.

Thus, the thesis aims to produce a proof of concept for the CSR use case where the data is semantically enriched, allowing for more efficient and effective use of the data. By showing that we can solve the problem for this use case, we will also show that we are solving the more general problem specified in section 4.1.

4.3 Goals

To provide a proof of concept capable of capturing the knowledge inherent in the company, this gives the thesis a number of goals that the proof of concept should aim to achieve.

- **Scalable**: The proof of concept should demonstrate good scalability for large data volumes. That is, computational complexity must remain manageable such that query answers are obtained within seconds or minutes, rather than hours. If this is not achievable, methods to reduce the data volume must be provided to ensure the system remains scalable overall.

- **Effective**: The proof of concept should be effective in addressing the business questions defined by the analysis tasks related to the use case. The aim is that the proof of concept should be more expressive than the existing solution and provide more insight into the business questions.

- **Extensible**: The proof of concept should be easily extendable to new or similar use-cases. The solution should offer integration capabilities for other use cases, and it should also be possible to extend to more sensors.

4.4 Challenges

Three challenges have been identified with the problem and the aim of the thesis. These are based on current research and the interesting properties of the problem.

- **Modeling of time series data**: Given that the aim is to semantically enrich the data, one major challenge is how to semantically enrich time series data. Current research suggests several ways to deal with time series data, some introducing *temporal relationships* where the relationships are present at certain points in time and not at other points [26] [33] [34] [35].

  This is especially relevant to this thesis as the data that is present in the use case presented above is primarily time series based.

- **Complexity in data**: The variety of data in the automotive industry [1] can lead to high complexity in the data. Both in terms of how the data points are related to each other, but also in terms of the data handling and the data pipelines that are built. The high frequency of sensors and signal data itself also has a high variety in the type of data that is being captured, which can lead to high complexity in the data. Dealing with this complexity will be a challenge when developing the proof of concept.

  An example of this can be found in the use case presented for this thesis, where the data consists primarily of time series and sensor readings. Some concepts important to the use case can be calculated from the large dataset; the concept and its relationships are inherent in the data. However, the calculation for this has bad computational complexity of at least $\mathcal{O}(n^2)$, which means that while not intractable to calculate, it is at least not efficient when computed on a large data set. Thus, this is a challenge that needs to be overcome concerning the thesis.

- **High volume of time series data**: Given that time series data often has high volume, which is one of the properties of Big Data [1], a challenge for the proof of concept is how to address this volume. A keynote talk by Gibbons on Big Data mentions three approaches for dealing with Big Data: scale down the data, scale up the hardware, and scale out to more machines [36]. The focus for the proof of concept is on the first approach, how the data can be scaled down.

4.5 Possibilities

We also see a number of possibilities in how we can construct this proof of concept.

- **Ontology**: In order to enrich data with semantics, the state-of-the art technology is to utilize an ontology [37]. The ontology can then be used to create a knowledge graph of the data, where the semantic modeling of the data enriches the data with semantics. Utilizing an ontology is a way to move complexity from the data into the ontology, which in turn can be used to gain a deeper

semantic understanding of the data.

- **Virtual Knowledge Graph**: A VKG makes use of an ontology, and virtually maps that ontology to the underlying data sources, without actually copying the data into a new storage system [2]. It is a technique that allows for avoiding duplication of data, whilst still being able to semantically enrich said data.
- **Graph Database**: A graph database can be used in conjunction with an ontology to align the data with the semantics. The ontology can then be explicit as part of the schema for the graph database, or implicit in the sense that the relations modeled in the graph database come from the ontology. However, this comes with the drawback of migrating the data to a new storage format.

4.6 Evaluation Metrics

This section proposes the metrics that will be used to evaluate the proof of concept with respect to the goals and to determine how well the proof of concept achieves those goals. A more thorough explanation of each metric and the methodology used can be found in chapter 6. The following metrics will be used:

Scalability: This evaluation metric is related to the goal of *scalability*. The metric evaluates how the proof of concept scales in terms of query latency for different queries when the volume and content of data change. It will also evaluate how the system scales using downscaled datasets for different queries. The metric used is the latency of each query.

Effectiveness of proof of concept: This evaluation metric is related to the *effective* goal. The metric evaluates how well the proof of concept answers the business questions associated with the analytical task in the use case. Specifically, by evaluating the similarity between the output of the proof of concept and the original use case. Additionally, the metric measures the number of SPARQL queries that can be constructed to address these business questions. The metric also reflects the expressiveness of the proof of concept and the extent of information it can convey.

Extensibility: This evaluation metric is related to the *extensibility* goal. Given that there are no widely accepted standards for measuring extensibility, and its inherent subjective nature, evaluation of this metric is done through an analysis of two dimensions of extensibility: *Mapping complexity* and *structural quality*. Mapping complexity reflects the effort required to alter existing mappings, while structural quality indicates how well structured the proof of concept is. Together, these metrics indicate the overall extensibility of the proof of concept.

5

Methods

The approach to solving the problem is to implement a Virtual Knowledge Graph (VKG) along with a PostgreSQL database that acts as the underlying data source supporting the VKG. This is done to leverage semantics and to enable effective and efficient data access. An overview of the systematic approach is found in figure 5.1, where one can see that data are taken from the data lake, transformed and put in the PostgreSQL database, where the VKG is placed on top.

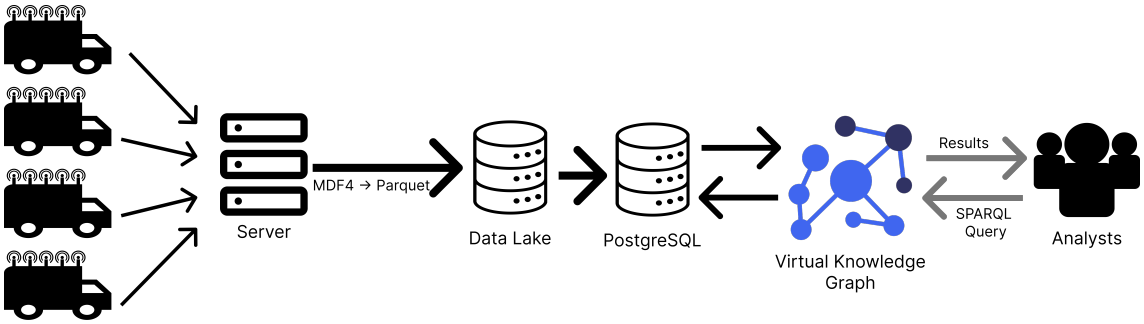


Figure 5.1: Overview of expected system. The Virtual Knowledge Graph is placed on top of a PostgreSQL database which contains sampled and processed data from the data lake.

The reason for choosing VKG as the approach, compared to e.g a native graph database, is the fact that we can make use of the virtualization aspect of the KG. This enables operations directly on the data in its original storage, eliminating the need for complex data migration. It also enables future expansion, allowing new data sources to be integrated into the VKG without relocating them, since the VKG can serve as a unified view over distributed sources. However, note that the chosen approach still moves and samples the data from the data lake to a PostgreSQL instance. This is because the format of the data lake is not an efficient format on which the VKG can operate. The chosen approach still allows for another data source to be added in a potential expansion.

The development of the VKG includes multiple steps, which are data modeling, ontology design, and integration with the underlying data sources. The tool is evaluated by benchmarking a set of evaluation queries that have been created in collaboration with Volvo Group. The examples in the method are tailored towards the *Charging Success Rate* (CSR) use-case at Volvo Group. However, the methodology used is applicable for a general use case.

5.1 Knowledge Extraction

The knowledge extraction phase involves workshops with domain experts and careful examination of the data. This relates to the top-down and the bottom-up approach for ontology creation, of which both are used in the implementation; see section 2.3.2, where the vocabulary is created first and then mapped to the underlying data source [6]. Knowledge extraction essentially boils down to what questions the data analyst wants to ask the system once it is in place. The questions then define what concepts, attributes, and relationships to extract. In addition, the questions also influence the queries defined for the evaluation metrics **effectiveness** and **scalability**. These queries are use-case specific, and it is up to the domain experts to identify and quantify how important each query is.

During the creation of the ontology, the importance of leveraging possible inferences is highlighted for the domain experts. An example of this is the definition of a broader concept, such as *Fault*, and then introducing subcategories of faults that are more specific. This allows the user to query for a *fault*, and the reasoner will still include all subcategories in the results. The ontology then not only captures domain knowledge, but also enables a more flexible and powerful querying through semantic reasoning. Furthermore, the ontology also has to be extended with semantics derived during the knowledge extraction. Examples of semantics that are applicable for the use case are names of sensors, explanations of what each sensor measures, explanations of what the values the sensor produces mean, and ensuring that all concepts are labeled in the ontology. For example, a sensor named ”charge_status” will have an explanation ”Defines the states that happened during charging”. Each distinct state is defined by a numerical value in the data, and in the VKG, this is then mapped to a string explaining what that numerical value means. The addition of semantics relates to the challenge of modeling time series data, see section 4.4, since it is difficult to label and explain sensor data in the ontology and the mappings.

5.2 Mapping creation

The mapping process involves linking the TBox with the data source to populate the ABox with real-world data. This will be done by defining a source, in the form of a query on the data source, to the TBox of the ontology.

Identifying mapping patterns is not a trivial task. A framework for mapping patterns can be found in [27], which gave inspiration for the mappings for this thesis. From this, it is apparent that having a storage in a data lake where tables have a lot of columns and rows is not sufficient to efficiently populate a VKG. Such a table has one row for every measurement taken for every vehicle. Thus, if one wishes to populate the concept of a vehicle from this table, the mapping would have to select every distinct vehicle from the enormous table. This is not optimal, as it means that every time a vehicle is queried for, a full pass of the entire table is needed. It is therefore necessary to create additional tables that are dedicated to the concepts of the ontology. These tables that are created specifically for this proof of concept

are explained in the implementation section 5.4.

Another choice made during the mapping creation is that certain preprocessing will be necessary to populate concepts that are not directly inherent in the data, an example of this is temporal concepts. A charging session is one such concept for the use-case CSR, which is defined as a time interval for a vehicle: $\langle vehicle, ts_{start}, ts_{end} \rangle$, where the charging cable is connected until it is plugged out. A charging session is a computationally expensive calculation given a set of sensor observations over time. The implementation of the preprocessing is specified in the implementation chapter 5.4.2.

5.3 Analytical considerations regarding sampling

A VKG can make use of the metadata in the underlying data source. As such it is important that the underlying data source is structured so that the VKG can operate efficiently on top of it. While it is possible to connect the VKG directly to the Data Lake, there were a number of reasons for discarding this approach and instead taking an approach of sampling and preprocessing the data.

SQL queries can be run directly on the data lake using Trino [38]. This also means that the VKG can be connected to the data lake. However, the absence of database metadata and the sheer size of the table in the data lake make performance extremely slow in the VKG. The VKG performs most *efficiently* if the data source it uses is an RDB containing a lot of metadata, where tables directly represent distinct concepts and there is a clear schema of the RDB with keys and constraints.

Furthermore, only some of the concepts found during the knowledge extraction phase, described in section 5.1, are directly possible to extract from the data lake. Even so, fetching these directly from the data lake means running a ”SELECT DISTINCT” query on the data lake, which due to the size of the data lake, takes around 15-30 minutes to complete. For the ones that are not directly extractable from the data lake, computations using SQL are needed. For those, each query on the data lake took easily over an hour to complete. As such, it was decided that using Trino as a connector to the data lake was not feasible and rather the approach of sampling and preprocessing was taken.

This approach consists of two parts; *concept preprocessing* and *sampling preprocessing*. Concept preprocessing is used for the concepts that are directly possible to extract from the data lake, while sampling preprocessing is used for the concepts that require more substantial computations to extract.

Concept preprocessing fetches the concepts directly from the data lake, making use of the column names, and ”SELECT DISTINCT” to populate concept specific tables. The VKG can then utilize these tables and, from the metadata, infer that every element in these tables is unique, meaning that what took 15-30 minutes to run on the data lake takes a few milliseconds to run using this approach. This is done in a single pass over the data lake.

Sampling Preprocessing is used for the concepts that are defined through computations in the source query. *Charging session* is one of these concepts that exists in the CSR use case, which is defined as a time interval for a vehicle: $\langle vehicle, ts_{start}, ts_{end} \rangle$, where ts_{start} is the timestamp when the charging cable is connected and ts_{end} when it is removed. The computation of a charging time is polynomial, making it sensitive to an increase in data volume.

To mitigate this, a sampling approach that runs in linear time is implemented. After sampling, the data volume is reduced by about a factor of 10^3 while still retaining the data that is important for computation. To give an idea of why this sampling is necessary, assume that the source query has a computational complexity of $\mathcal{O}(n^2)$ and that the reduction of rows (the n) is by a factor of 10^3. The sampled version then has $n_{sampled} = \frac{n_{data\_lake}}{10^3}$ rows, which in terms of Big O:

$$\mathcal{O}(n^2_{sampled}) = \mathcal{O}((\frac{n_{data\_lake}}{10^3})^2) = \mathcal{O}(\frac{n^2_{data\_lake}}{10^6})$$

This shows that the computational cost is disproportionally reduced when downsampling, especially for source queries of high complexity. Since linear sampling is performed only once, while polynomial-time computations are executed multiple times on the sampled data, the overall computational cost of the system is reduced.

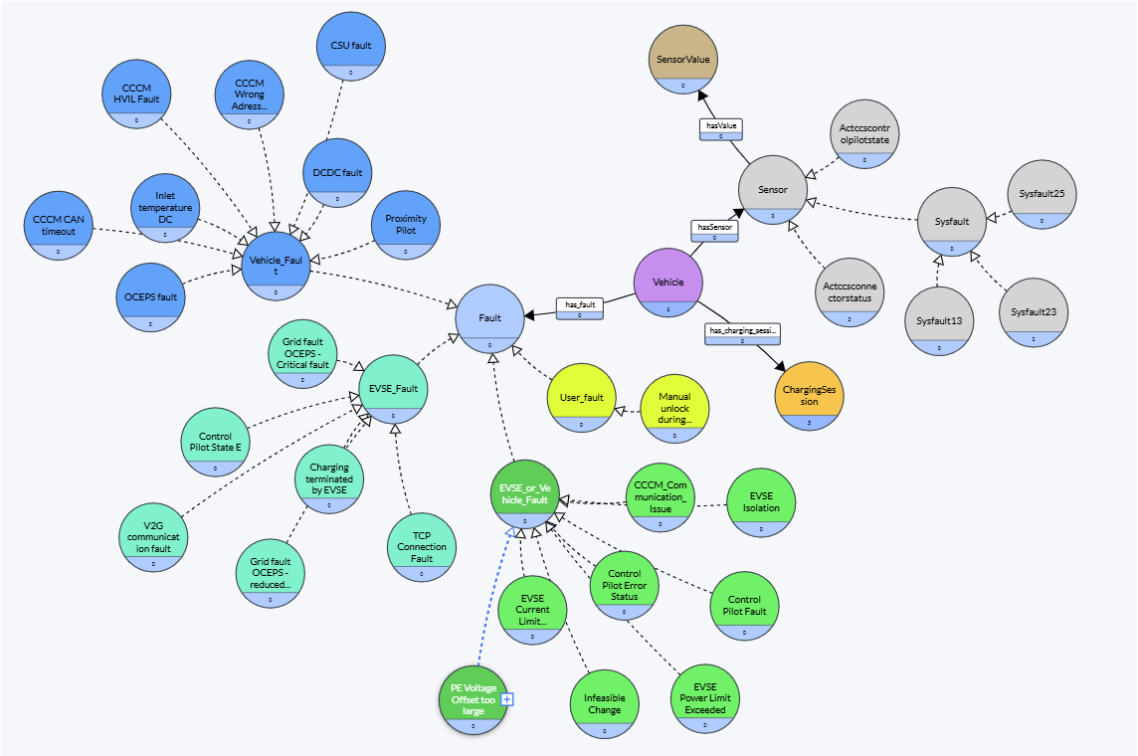


Figure 5.2: Full ontology represented as a Concept-Relationship graph. Note the absence of attributes in the graph.

5.4 Implementation

The system is implemented using Ontop, an open-source and state-of-the-art VKG system.

5.4.1 Ontology

In Ontop, the ontology is expressed in the OWL 2 language. The tool *Prótege* [39] was used to create the ontology in the correct format, following the concepts, attributes, and relationships from the knowledge extraction step 5.1.

Concepts	
Name	**Description**
Vehicle (V)	Set of Vehicles
Sensor (S)	Set of sensors in a vehicle V
Sensor Value (SV)	Sensor observation in a vehicle at a given timestamp
Fault (F)	Set of Observations for a vehicle over an interval
Charging Session (CS)	Charging timestamped interval for a vehicle
Fault Type (f_i)	A type of fault
Sensor Type (s_i)	A type of sensor
Relationships	
Caused By	F **caused by** S
Has Charging Session	V **has charging session** CS
Has Fault	V **has a** F
Has Sensor	V **has sensor** S
Has Value	S **has value** SV
Attributes	
Date Time	When SV was recorded
Date Time	When F happened
Map Value	Meaning of the SV value
Session Start	Start time of CS
Session End	End time of CS
Is Successful	Boolean of whether the CS was successful or not
Is Valid	Boolean of whether the CS is a valid CS or not
Charging Type	The charging type of the CS (AC/DC)
Value	The value that the SV took at a particular time
Name	The name of the V and the S
Label	Informative text of what the F is and what the SV mean

Table 5.2: The concepts, relationships, and attributes implemented in the VKG for the CSR use-case.

The ontology developed for the use-case *Charging Success Rate* (CSR) can be seen in the concept-relationship figure 5.2. Explanations of the concepts can be found in table 5.2. As can be seen in the figure and the table, there are multiple subclasses to both the concept *Fault* and *Sensor*, these are denoted in the table as Fault Type (f_i)

and Sensor Type (s_i), where i denotes the different subclasses that exist below Fault and Sensor. Using subclasses like this is a way to leverage the inference capabilities that exist in Ontop. Due to this, querying the graph for a Fault means that all instances of a Fault or a subclass of Fault are returned.

As mentioned in section 4.2, one of the main challenges was the creation of an ontology for time series data. Thus, in the initial development process, two versions of the ontology were created, one in which the time series is a node and the other in which the time series is an attribute. The difference between the two modeling choices is shown in figures 5.3 and 5.4. These figures show a subgraph of an instantiated knowledge graph. I.e, the nodes here are not concepts, but rather instances of concepts. As can be seen in figure 5.4, when time is converted from a concept to an attribute, this results in the subgraph being two distinct subgraphs rather than a single one. In the end, the version of time series as an attribute was chosen as the main ontology, as it has lower query latencies than the approach with time series as a node.

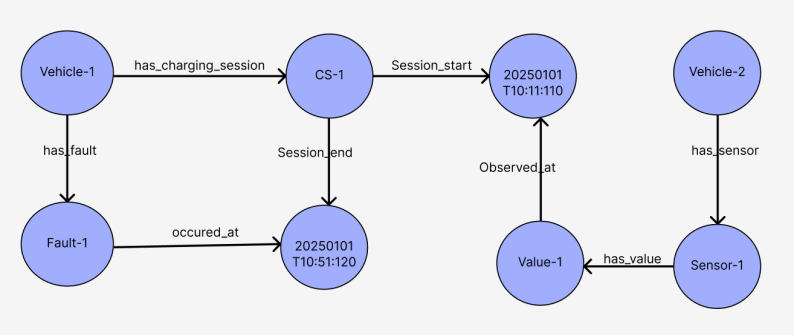


Figure 5.3: Sub graph when time series is modeled as a node

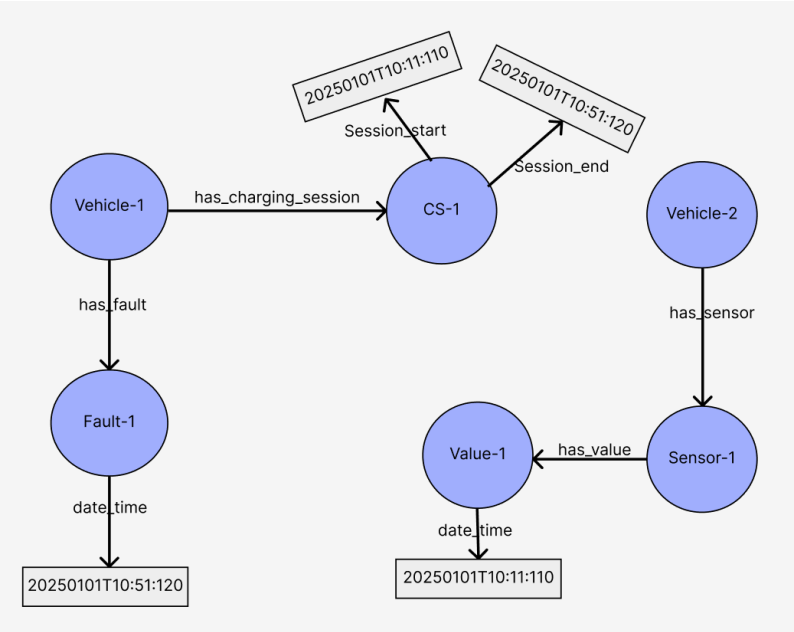


Figure 5.4: Sub graph when time series is modeled as an attribute

In total, the VKG system and its ontology consist of 38 distinct concepts, but only 33 mappings, see table 5.3. This is due to several reasons. One of them is that the Ontop mapping language allows for multiple relationships and concepts to be defined in a single mapping. Another reason is that not all of the concepts have any instantiated values. For example, the concept *Vehicle fault* does not have any own instances; instead, mappings are created for the subclasses of that concept. But the concept is still important as it allows reasoning over the graph and queries directly for the concept vehicle fault, which will return all instances of the subclasses of that concept.

Ontology part	**Amount**
Concepts	38
Relationships	5
Attributes	10
Mappings	33

Table 5.3: The amount of definitions for different parts of the ontology.

5.4.2 Sampling from Data Lake to PostgreSQL database

The sampling is made according to the considerations made in section 5.3. A PostgreSQL database is chosen as the intermediate storage between the data lake and Ontop. This database is designed and implemented to utilize the advantages that an RDB provides to a VKG system as Ontop in terms of performance.

Five data sets of different sizes are chosen to evaluate the metric **scalability**. These are extracted from the data lake, and using the approach specified in section 5.3, preprocessed and sampled to the PostgreSQL database. As can be seen in table 5.4, the frequency of sensor measurements in the data lake is 100 Hz, which seems to be the standard at Volvo Group. The varying parameters are the number of vehicles and the timeframe. Mathematically, they are related to each other as $ds1 \subseteq ds2 \subseteq ds3$ and $ds4 \subseteq ds5$, also $ds2 \subseteq ds4$, which can be seen in figure 5.5. However, note that $ds3 \not\subset ds4$ since ds3 contains a longer timeframe than ds4 and therefore contains elements that ds4 does not have.

Data Set	**Number of Vehicles**	**Timeframe**	**Number of rows in data lake**	**Measurement Frequency**
$ds1^+$	1	1 day	$3.3 \cdot 10^6$ rows	100 Hz
$ds2^+$	2	1 week	$37.6 \cdot 10^6$ rows	100 Hz
$ds3^+$	2	2 months	$266.5 \cdot 10^6$ rows	100 Hz
$ds4^+$	125	1 week	$2.4 \cdot 10^9$ rows	100 Hz
$ds5^+$	125	2 months	$16.7 \cdot 10^9$ rows	100 Hz

Table 5.4: Overview of the datasets used for the evaluation of the thesis.

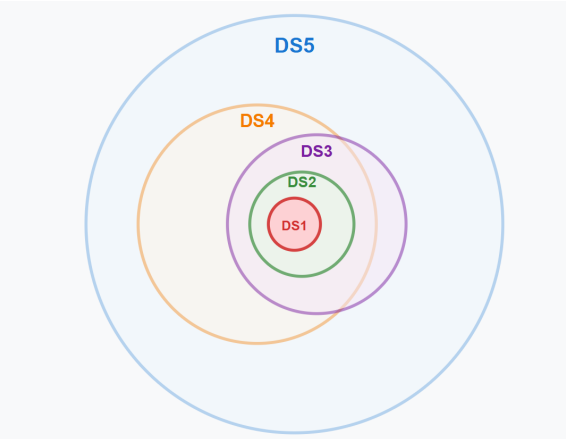


Figure 5.5: A Venn diagram of the data sets. Note that this is not proportional to their actual sizes.

Database Design

The database is designed using keys, constraints and is structured using good practices, described in 2.5.2. The tables are populated according to the approach described in section 5.3, where *Concept Preprocessing* is responsible for tables that directly map to concepts and *Sampling Preprocessing* is responsible for the downsampling to a single table that maps to multiple concepts.

Concept preprocessing

The following tables *csr_vehicles*, *csr_sensors*, *csr_values*, and *csr_map_values* are created, which can be seen in table 5.5 along with the column names. These tables represent the following concepts:

- *Vehicles:* The vehicles that are relevant for the use-case. There are about 125 vehicles in this table.
- *Sensors:* The sensors that are relevant for the use-case. About 100 sensors are per every vehicle are in this table.
- *Values:* These are all values that the sensor takes. The unique constraint of this table is the tuple consisting of vehicle, sensor_name, and timestamp.
- *Map Values:* This is an auxiliary table for the Values table that consists of all distinct values that a sensor can take, and a text explaining what each value means in plain text.

Table name	Columns
csr_vehicles	vehicle
csr_sensors	vehicle, sensor
csr_values	vehicle, sensor, timestamp, value
csr_map_values	id, sensor, value, explanation

Table 5.5: Overview of Concept preprocessing tables

Among the tables, *csr_values* sticks out. This table holds all sensor measurements. This table is populated using the LAG() function in SQL, which keeps the previous value of the row in memory. Using this function, only values that changed compared to the previous row are saved in this table. I.e event-based sensors keep only the values when an event occurs, and continuous sensors keep almost all values as they are seldom the same for two measurements in a row.

Concept preprocessing is performed *once* for tables *csr_vehicles*, *csr_sensors*, and *csr_map_values*, since they do not depend on the data sets. The table *csr_values*, is done *once per data set* since the actual values of the sensors change based on the timeframe.

Sampling preprocessing
Sampling of the data lake is performed in a single pass during the preprocessing stage. The original data lake consists the columns vehicle and timestamp, followed by approximately 50,000 sensor columns. The measurements stored in the data lake are taken at a rate of 100 Hz. The sampling approach combines two strategies: event-based selection and subsampling.

Event-based selection selects a row if any event-based sensor changes its value. In that case, all measurements for all sensors in the entire row are retained. Additionally, one row per minute is also selected as a baseline. This is to ensure that we do not lose too much information from the non-event based sensors. The result is a sampled table containing the columns vehicle and timestamp, followed by columns of approximately 100 sensors relevant to the use case.

Data Set	**Number of Vehicles**	**Timeframe**	**Number of rows in PostgreSQL**	**Table name**
ds1	1	1 day	$4.7 \cdot 10^3$ rows	ds1_subsampled
ds2	2	1 week	$42.4 \cdot 10^3$ rows	ds2_subsampled
ds3	2	2 months	$223.9 \cdot 10^3$ rows	ds3_subsampled
ds4	125	1 week	$746.9 \cdot 10^3$ rows	ds4_subsampled
ds5	125	2 months	$7.8 \cdot 10^6$ rows	ds5_subsampled

Table 5.6: Overview of the row count after sampling the datasets.

As can be seen from the tables 5.6 and 5.4, the sampling has a significant impact on the number of rows in all data sets, lowering the number of rows for most of the data sets by a factor of 10^3. The sampling is performed five times, once for each data set as seen in table 5.6. Note that all of these tables have the same structure in their output. The reason for creating five tables instead of a single one is for the evaluation of the scalability metric. If the VKG were in production, the preprocessing would instead be part of an extract-transform-load pipeline that feeds the VKG with data, and only a single table needed.

5.4.3 Mappings

For creating the mappings, there exist two different languages in Ontop, which can connect the ontology to the underlying data source, R2RML and the Ontop mapping

language. The Ontop mapping language was chosen for the implementation because it is more compact and user-friendly compared to the R2RML mapping language.

```
Target (Triples Template):
:sensor_value-{vehicle}/{sensor}/{time_abs_utc} a :Sensor_value ;
:date_time {time_abs_utc}^^xsd:dateTime ; :value {value}^^xsd:decimal .

Source (SQL Query):
SELECT vehicle, sensor, time_abs_utc, value
FROM data.csr_signals
```

Listing 8: Mapping between a table dedicated to a concept and the corresponding concept in the ontology.

```
Target (Triples Template):
:fault-control_pilot_state_e/{vehicle}/{time_abs_utc}/{data_filename}
a :Control_Pilot_State_E ; :date_time {time_abs_utc}^^xsd:dateTime ;
rdfs:label "control_pilot_state_e"^^xsd:string ;
:caused_by :sensor-actccscontrolpilotstate_ep2_x_cc/{vehicle} .
:vehicle-{vehicle} :has_fault
:fault-control_pilot_state_e/{vehicle}/{time_abs_utc}/{data_filename} .

Source (SQL Query):
SELECT *
FROM (
    SELECT
    vehicle,
    time_abs_utc,
    data_filename,
    actccscontrolpilotstate_ep2_x_cc,
    LAG(time_abs_utc) OVER (PARTITION BY vehicle ORDER BY time_abs_utc)
    AS prev_time,
    LAG(actccscontrolpilotstate_ep2_x_cc) OVER
    (PARTITION BY vehicle ORDER BY time_abs_utc) AS prev_value
    FROM "data"."dsX_subsampled"
    ) AS t
WHERE actccscontrolpilotstate_ep2_x_cc = 4
    AND prev_value = 4
    AND time_abs_utc - prev_time > INTERVAL '3 seconds'
```

Listing 9: Mapping between the sampled table and a concept in the ontology.

The mappings are dependent on the concepts identified in the knowledge extraction step, section 5.1, and also on the underlying data source, which in this case is the PostgreSQL instance, since the mappings are what connect the ontology to the data.

An example of a mapping between one of the tables created specifically for the concepts and the corresponding concept can be found in listing 8. However, for

the concepts found in the sampled data where one table maps to several concepts, the source query is sometimes more complicated, as can be seen in listing 9. The more complicated source queries typically involve one or more subqueries and aggregations and/or functions. However, not all mappings on the sampled table are complicated mappings. The majority of the mappings for fault concepts are on the form: "SELECT sensor_1, sensor_2, ... WHERE sensor_1 = 'some value' AND sensor_2 = 'some value' AND ...". I.e a select statement with a where clause with a boolean expression which are quite simple queries for an RDB to handle.

5.4.4 SPARQL

Once the ontology and mappings are defined, *SPARQL* can be written to query the VKG system, see section 2.4.2. These queries serve multiple purposes: verifying if the intended knowledge is accurately captured, identifying potential bugs in the mappings, and uncovering misunderstandings in the modeling. This is the stage when the VKG becomes fully operational and where graph queries can be leveraged to answer analytical business questions.

Using the tool Prótege, one can also see how the SPARQL query is translated into SQL. This translation provides insight into the quality of the mappings. For instance, if the generated SQL query contains a **SELECT DISTINCT** clause, it suggests that the uniqueness of the target triple is not guaranteed by the database metadata or the source SQL query. In such cases, the system must enforce uniqueness explicitly, since Knowledge Graphs require every instance of a concept to be unique, as is explained in sections 2.4.2 and 2.4.3.

The absence of unique instances for concepts forces the system to add a global **SELECT DISTINCT** to the SQL translation. This is undesirable because in SQL, a DISTINCT operation means that a full pass of the data must be done to guarantee that the result set is distinct, leading to reduced performance. If uniqueness can be guaranteed without the distinct search, the result set is simply the first matching answers that the data source could find.

6

Evaluation Methodology

Evaluation of the thesis is based on the stated goals of the thesis. The overarching goal is to make analysis tasks easier to do. In order to evaluate this goal, it is broken down into three subgoals: (1) Scalability of proof of concept, (2) Effectiveness of analytical tasks, (3) Extensible solution for more usecases. Subgoal (1) measures how scalable the solution is by testing the evaluation queries on data sets of different sizes to see how the latency scales. (2) is evaluated based on how similar the evaluation queries are to already pre-computed values of the use-case, and to which degree the evaluation queries are answerable. Lastly, (3) tests the system to see how extensible it is by measuring two related dimensions; mapping complexity and structural quality.

6.1 Baseline - Volvo CSR Use Case

Volvo is currently developing a solution that calculates the success rate of vehicle charging sessions. This is calculated based on data retrieved from a data lake, which can be read more about in Section Use Case 4.2.1. In order to evaluate the tool, the VKG uses the same data to be able to compare the results of the two solutions. The modeling of the proof of concept mimics the analytical tasks of Volvo to ensure that we model the concepts in the same way. The primary difference is that we do this using a VKG while Volvo's solution is implemented in Python and Apache Spark.

6.2 Evaluation Queries

The queries used to evaluate the thesis are written in SPARQL. There are 14 main evaluation queries, accompanied by three auxiliary queries that are closely related to three of the main ones. The auxiliary queries allow the retrieval of intermediate data that can be directly compared to Volvo's results for the effectiveness metric. Each auxiliary query is derived from its corresponding evaluation query, but without any aggregations, and one of the auxiliary queries is constructed as the inverse of its original. Auxiliary queries are named as $qx*$, where x refers to the main query it is related to. A full list and descriptions of all evaluation queries can be found in the Appendix A.

All evaluation queries are developed in collaboration with domain experts at the Volvo Group, with the intention that the queries should be representative of typical

questions that data analysts want answers to regarding the CSR use case. These are both questions that they currently have answers to, that are mimicked in the proof-of-concept, and questions that they currently do not have answers to but wish to have answers to. The queries are also developed with the complexity of the queries in mind, as for the scalability metric, it is important to have different complexities to show how the VKG system works with different complexities in the queries. In total, this amounts to 14 evaluation queries. The queries are sorted by the magnitude of their latency when executed on dataset ds5.

For the sake of readability, only three of these evaluation queries are presented below. They are formulated in both natural language and in *SPARQL*. The queries are selected to display the possible degrees of complexity in the VKG system and are representative of the complexity of the full list of evaluation queries, found in the Appendix A.

Query q2: *Return the name of all sensors that cause the fault 'DCDC Fault'*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT DISTINCT ?fault_name ?sensor_name WHERE {
  ?fault a :DCDC_Fault .
  ?fault :caused_by ?sensor ;
         rdfs:label ?fault_name .
  ?sensor :name ?sensor_name .
}
```

Listing 10: SPARQL query for q2.

Query q11: *Return sensor names, timestamp, values, and the explanation for the values, filter on vehicle = 'CUST-111'*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?sensor_name ?time ?value ?explanation WHERE {
    ?vehicle :has_sensor ?sensor ;
             :name ?vehicle_name .
    ?sensor  :has_value ?val ;
             :name ?sensor_name .
    ?val :value ?value ;
         :date_time ?time ;
     rdfs:label ?explanation .
     FILTER(?vehicle_name = "CUST-111"^^xsd:string)
}
```

Listing 11: SPARQL query for q11.

The simplest query is q1, see listing 10 for full SPARQL, where a certain type of fault is queried for. This is done through inference using the concept ”DCDC_Fault” and a link called ”caused_by” to the sensors.

A slightly more complicated query is query q11, see listing 11 for full SPARQL, where three different concepts are searched for with relationships ”has_sensor” and ”has_value”. This is then filtered on the vehicle name such that only answers belonging to this vehicle name are returned.

Query q9: *Calculate an aggregated charging success rate for every vehicle and filter the results for any rate > 95% or a rate < 50%, group by vehicle, and order by success rate.*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle ?rate WHERE {
  {
  SELECT ?vehicle (COUNT(?cs) as ?success) WHERE {
        ?vehicle :has_charging_session ?cs .
        ?cs  :isValid ?validity ;
             :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean &&
               ?isSuccess = "true"^^xsd:boolean) .
    } GROUP BY ?vehicle
  }
  {
  SELECT ?vehicle (COUNT(?cs) as ?fail) WHERE {
        ?vehicle :has_charging_session ?cs .
        ?cs :isValid ?validity ;
            :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean &&
               ?isSuccess = "false"^^xsd:boolean) .
     } GROUP BY ?vehicle
  }
  BIND(?success / (?success + ?fail) AS ?rate).
  FILTER(?rate > 0.9 || ?rate < 0.5)
} order by ?rate
```

Listing 12: SPARQL query for q9.

The most complex query in terms of SPARQL is *q9*. See listing 12 for full SPARQL. This query features two subqueries, both utilizing a filter clause and the group by clause. The outer query computes the rate using the BIND() function, which binds an expression to a variable. The final result is then filtered and ordered on this rate.

6.3 Scalability

The Scalability of the system will be evaluated by testing data sets of different time intervals and different amounts of vehicles. The data sets and their differences in the previously mentioned parameters can be found in table 5.4. The evaluation queries used for the scalability metric are the 14 queries in Appendix A. The auxiliary queries are only for the effectiveness metric and are not of importance for scalability. All queries are run as they are presented for each of the data sets. In total, every query is run five times for every data set. Then the average latency for every query is taken.

There are four different evaluations for the scalability metric:

(1) a result evaluation based on how the system performs as a whole for all of the evaluation queries. This is related to the goal of being able to run each query in isolation under one hour.

(2) a comparison in latency between ds3 and ds4, ds5 to see if the latency is strictly based on the size of the table, or if the content matters as well. The size of the data sets scales as follows $|ds1| < |ds2| < |ds3| < |ds4| < |ds5|$, and the difference in contents can be read about in section 5.4.2.

(3) a comparison between ds1, ds2 and ds3 for all queries, as within these data sets there is only one variable change, the volume of data. The evaluation analyzes how the latency of queries scales with increasing data volume. The queries are separated into three different figures that depend on the magnitude of the latency when executed on ds3. This is to be able to plot the queries on a linear scale.

(4) a comparison between a non-sampled dataset and the sampled dataset to investigate if sampling is necessary for the tool. Dataset ds2 was selected for this comparison, primarily due to the size of the original data combined with the complexity of the queries, as the latency would be too high for ds3, ds4 and ds5 without the sampling. The queries for this comparison were selected based on the latency and scaling, as it was unfeasible to run the most complex queries due to the non-sampled data set is too large.

6.4 Effectiveness of analysis task

To evaluate the effectiveness of the VKG system, a set of evaluation queries has been defined during the knowledge extraction step, which is written about in Section 6.2. Evaluation queries are important because they shows that the tool can answer relevant questions in the domain.

A very important aspect for the effectiveness metric is that queries return the correct result, therefore, the system output will be compared with the solution with the solution at Volvo, mentioned in 6.1. The queries we have compared are *q3*, *q10* and all the auxiliary queries. The reasons why the other queries were not compared were for three reasons: (1) The evaluation queries cannot be directly compared against

Volvo's solution, as the output of them is not captured in Volvo's database. (2) Evaluation queries include aggregation, which means that individual records are lost. (3) Adding the queries does not add more value, as the chosen queries have covered that part of the VKG system. A similarity score is used to compare the query results with those produced by Volvo engineers' queries.

The similarity score will be computed with the following metrics: *Agreement Rate*, *Jaccard Index*, *Dice Coefficient* and a *Average*. Jaccard Index and Dice Coefficient are metrics for set comparison [40]. By viewing the output of the queries as sets, a similarity comparison using Jaccard Index and Dice Coefficient is an effective comparison between the VKG and Volvo's solution. The main difference between the two values is that the matching elements have a higher weight using the Dice Coefficient measurement compared to the Jaccard Index. The metric Agreement Rate is also used, which measures how similar the two outputs are. Lastly, an average is calculated which is a comparison of the total number of elements in both sets, to highlight if there exist extra or missing values in the set.

Formulas for all equations are presented below. Set A is the output from the system and set B is the output from Volvo's solution.

$$jaccard\_index(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.1}$$

$$dice\_coefficient(A, B) = \frac{2|A \cap B|}{|A| + |B|} \tag{6.2}$$

$$agreement\_rate(A, B) = \frac{|A \cap B|}{|A|} \tag{6.3}$$

$$average(A, B) = \frac{|A|}{|B|} \tag{6.4}$$

6.5 Extensibility

The extensibility metric is evaluated according to an analysis of two quality dimensions of KGs; mapping complexity and structural quality. Mapping complexity relates to extensibility, as the less time and effort needed to create or adapt mappings, the easier it will be to integrate new data sources or make structural changes to the VKG. Structural quality can be seen as a metric of how well the VKG represents the domain. An ontology that broadly covers general concepts within that domain has a higher structural quality, and will thus require fewer schema-level changes when extended to a new domain or dataset. Together, these two metrics will give a good indication of how extensible the VKG is.

6.5.1 Mapping Complexity

Mapping complexity is indicative of how extensible the VKG system is, since a lower complexity in the mappings translates to easier integration of new or altered mappings. This is measured by a qualitative score for every mapping, weighted by the complexities found in the source query of the mappings. The weight is based on the complexity of common mapping patterns [22] [27], where common mapping patterns found in current research are given a weight based on their individual complexity and then every occurrence of such a mapping pattern is counted and multiplied by the weight. The weights and the patterns can be found in table 6.1, and the calculation in equation 6.5.

$$\sum_{i=0}^{n} count(m_i) \cdot w \tag{6.5}$$

Equation 6.5: Calculation of mapping complexity where m_i is the mapping pattern, $count(m_i)$ is the count of a mapping pattern, w is the weight assigned to that mapping pattern, and n is the amount of mappings in the VKG system.

A lower score means less complexity, while a higher score indicates a more complex mapping. An average and median of all complexity scores are computed to represent the total mapping complexity of the VKG.

Complexity mapping pattern	Weight
Direct concept	1
Direct relationship	1
Distinct	1.5
# of conditions	1
# of joins	3
# of arithmetic operations	1.5
# of case statements	1.5
# of nested queries	3
Use of functions/aggregations	2
Use of unions	1.5

Table 6.1: Weights assigned to different complexity types. The weights are proposed by us but inspiration is taken based on common mapping patterns in the literature [22] [27].

Furthermore, an analysis of source queries in SQL is performed using the PostgreSQL query planner [41]. The query planner returns the operations that PostgreSQL estimates have to be done, a startup cost which is a mandatory cost every time the query is run, and a total cost which is the cost it would take to fetch every element of the query. The total cost can be reduced by limiting the number of rows returned. Note that the costs have no unit; they are estimates that the query planner uses to decide which operations to apply when optimizing the query. The analysis is then compared with the mapping complexity scores for further insight.

6.5.2 Structural Quality

Structural quality measures the internal quality of the VKG according to the structure of the ontology and the use of the ontology in the VKG [42]. The structural quality metrics are proposed by Seo et. al. who measured several different knowledge graphs using these metrics [42]. The metrics used are defined in equations 6.6, 6.7, 6.8, and 6.9.

Instantiated Class Ratio, equation 6.6, refers to the ratio of classes with instances between all classes defined in the ontology. This indicates how well the classes in the ontology are actually being used.

Instantiated Property Ratio, equation 6.7, refers to the ratio of properties used in RDF triples among the properties defined in the ontology. This indicates how well the properties of the ontology are being used.

Class Instantiation, equation 6.8, refers to how detailed the classes defined in the ontology are and to which degree they are instantiated. For every class in the knowledge graph, class instantiation is calculated and summed, which is indicative of how well the ontology is utilized.

Lastly, *Inverse Multiple Inheritance* equation 6.9, refers to the simplicity of the knowledge graph. In the case where multiple inheritance occurs where a class has numerous superclasses, it can be challenging to use the knowledge graph because of the complexity of the class relationship. The higher the inverse multiple inheritance, the simpler the knowledge graph is and, by extension, the more extensible it is.

$$ICR(Ontology) = \frac{N(IC)}{N(C)} \tag{6.6}$$

Equation 6.6: *Instantiated Class Ratio*, where $N(C)$ is the total number of classes in the ontology, $N(IC)$ is the number of classes in which instances exist

$$IPR(Ontology) = \frac{N(IP)}{N(P)} \tag{6.7}$$

Equation 6.7: *Instantiated Property Ratio*, where $N(P)$ is the total number of properties in the ontology, $N(IP)$ is the number of properties used in RDF triples

$$CI(Class) = \sum_{i=1}^{n_c} \frac{ir(c_i)}{2^{d(c_i)}} \tag{6.8}$$

Equation 6.8: *Class Instantiation*, where n_c is the number of subclasses that the class has, $ir(c)$ is the instantiated ratio which is "number of instances of class/number of all instances in knowledge graph", c_i is the i-th subclass the Class has, d is the distance between the class and c_i

$$IMI(Ontology) = \frac{1}{\sum_{i=1}^{N_c} \frac{nsup(C_i)}{N_c}} \tag{6.9}$$

Equation 6.9: *Inverse Multiple Inheritance*, where N_c is the total number of classes in the ontology, C_i is each class in the ontology, $nsup(C)$ is the number of direct superclasses in the class

6.6 Evaluation Setup

The evaluation has been run on two Virtual Machines (VM1) and (VM2). VM1 running Ontop and VM2 running the PostgreSQL instance for the data source. VM1 with Ontop is only responsible for rewriting SPARQL queries into SQL, as such its hardware specifications have a lower impact on the system. VM2 with PostgreSQL executes the actual queries on the database, as such it is more reliant on the hardware.

VM1 features an AMD 32-core processor and 32 GB of RAM, running Ubuntu Version 22.04.5 as its operating system. The software used on VM1 is Ontop Version 5.2.1, Java JDK 21.0.7, Python 3.10.12, and GNU bash 5.1.16. VM2 is a virtual machine that hosts a PostgreSQL instance, with a CPU limit of 4 cores and a memory limit of 15 GB. The software is Postgres 14.15.

The evaluation was run on VM1, which communicated with VM2 through Ontop and PostgreSQL. To calculate the metrics, the following was made:

Scalability
Bash is used in two steps: (1) Configurations for all data sets were created. (2) Separation of evaluation queries into different files.

For each configuration, an Ontop instance is started and all queries are executed. For each query, the start timestamp is recorded when the query is sent to Ontop, the end timestamp is recorded when the result is received. The test is repeated 5 times and an average is taken. In addition, the output of every query is saved. Between the configurations, the Ontop instance was terminated and then started with the next configuration.

Effectiveness
This metric uses the ds5 configuration, query q3, q10, and the auxiliary queries. The results were collected in the same manner as for the scalability evaluation. However, the focus is on the output of the queries rather than the latency.

The output of every query is then compared with the result from Volvo's approach of the CSR use case, which we fetched through SQL queries. The result was then parsed into the same format and compared using Python.

7

Results

This section includes the results of the evaluation methodology. The query examples used in this section are those presented in Chapter 6, and all are related to the business use case Charging Success Rate at Volvo.

7.1 Scalability for larger data sets

The scalability of the proof-of-concept is measured as specified in section 6.3. The latencies of the 14 evaluation queries are presented in figure 7.1. Given the large disparities between the query latencies, the graph in the figure is displayed on a logarithmic scale to ensure that all values are visible. Note that $\langle ds1, ds2, ds3 \rangle$ and $\langle ds4, ds5 \rangle$ are grouped in different colors. Within the same color scheme set, the only variable change is the period. The difference between the two groups lies in the varying number of vehicles. Within each group, the variable change is time. However, when specific sets between the two groups are compared, there can be at most two variables that change: time and the number of vehicles. All data sets are ordered by size, both in terms of GB and number of rows.

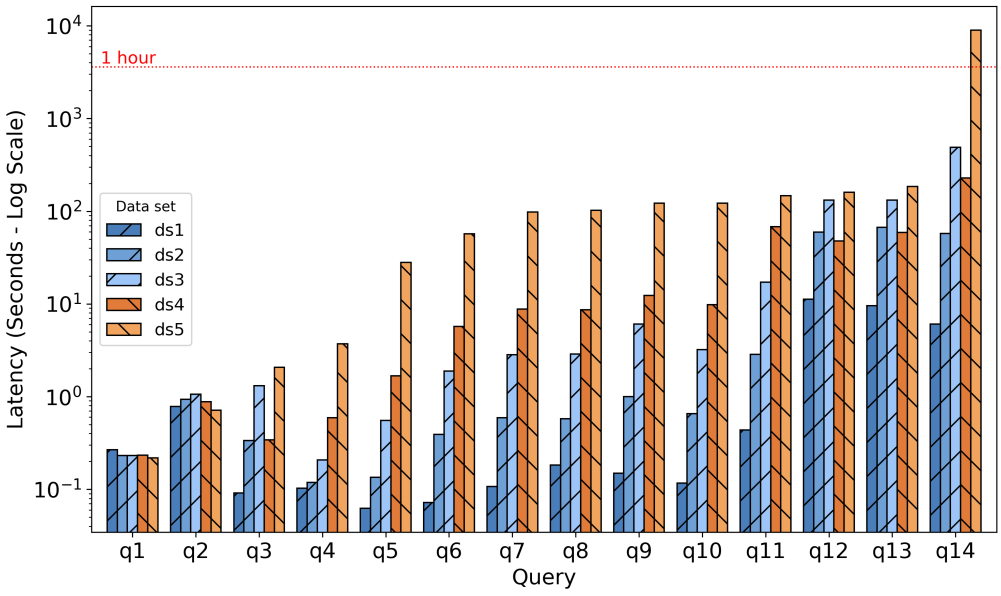
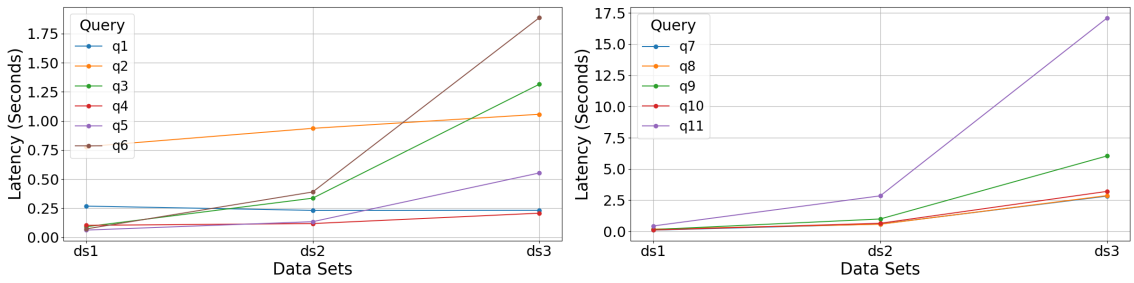


Figure 7.1: Query latency for the five data sets on a logarithmic scale.

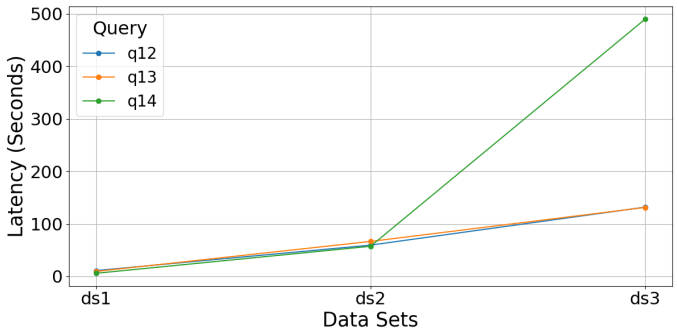
Given the goal of having all queries run in under one hour separately, the result of figure 7.1 reaches the goal for all queries except *q14*, listing (28), which finished in about 140 minutes. However, note that the queries differ in scalability. For example, for *q1* and *q2* the latency is stable over all data sets, as the latencies do not show tendencies of scaling when data sets change. Queries *q4*, *q5*, *q7*, *q8*, *q9*, *q10*, and *q11* have an increased latency as the data volume increases. This is to be expected, as the input size increases when executing the SQL queries, which in turn increases the latency. However, queries *q12*, *q13*, and *q14* do not follow this pattern. When comparing ds3 and ds4 for these queries, ds3 has a higher latency, although ds4 contains a larger data set size. Note that ds3 and ds4 have both time and vehicle as changing variables, which means that the result is unclear regarding which of the two variables caused this result.

The reason for this result is that queries depend on different parameters because of how the underlying data is represented. This is apparent from figure 7.1, where the queries with the highest latency revolve around the concept *Charging Session* in conjunction with *faults*. These concepts are computed when queried, severely affecting latency. For the charging session, the VKG system cannot guarantee that the instance is unique. Therefore, it must add the 'DISTINCT' operator to the SQL query, which further increases latency.



(a) Latency 0-1.75s

(b) Latency 0-17.5s



(c) Latency 0-500s

Figure 7.2: Queries grouped by magnitude of latency for dataset ds1, ds2, and ds3.

Figures 7.2a, 7.2b, and 7.2c present the query results for data sets ds1, ds2, and ds3 in isolation, where the only variable is the period covered by each data set. These figures show how query latency scales with increased data volume, plotted on a

linear scale. The queries are grouped by the order of magnitude of their respective latencies.

Figure 7.2a shows the results for queries *q1*, *q2*, *q3*, *q4*,*q5*, and *q6*, all of which exhibit low latency for the three data sets. However, *q6* stands out due to its increase in latency from ds2 to ds3, showing a fourfold increase in latency. *q6* does have a low latency in the beginning, which means that the latency does not scale out of hand, remaining under 2 seconds for ds3. Nevertheless, as data set sizes continue to grow, this scaling may significantly affect performance. This is evident in figure 7.1, where for ds5, *q6* takes around 60 seconds, compared to about 13 seconds for *q5*, a fivefold difference. For ds3, the latency between *q5* and *q6* is around a threefold increase, suggesting that *q6* experiences considerably higher latencies at larger data set sizes. Especially in the absence of sampling, which will be discussed later in this section.

Figure 7.2b includes results for queries *q7*, *q8*, *q9*, *q10*, and *q11*. A similar scaling trend is observed for *q11* as with *q6* in figure 7.2a, with latency increasing nearly linearly with data volume. Although *q11* shows higher overall latency, the scaling is consistent with that of *q6*. Specifically for ds1, *q11* takes approximately six times longer to execute than *q6*. A similar trend is observed between *q5* and *q9*, where the scaling is similar but they differ in their starting latency magnitudes.

Figure 7.2c includes results for queries *q12*, *q13* and *q14*, which display the highest latency across ds1, ds2, and ds3. Among them, *q14* shows signs of scaling nearly linearly, similar to *q6* and *q11*. In particular, *q14* is the only query that exceeds the latency goal of sub one hour queries, taking more than 140 minutes to complete.

The analysis of the scalability metric shows how well a query can handle increasing data volumes. If a query has high latency even for small data sets such as ds1, the performance will deteriorate more rapidly with larger data set volumes. This explains why *q14* has a significantly higher latency than *q6* and *q11*, even though the scaling trend is similar. Although all three queries scale in a comparable manner, *q14* has a higher computational cost at the beginning, making its performance less sustainable as the volume of data increases.

A comparison between sampled dataset ds2 and the same dataset but non-sampled, $ds2^+$, is seen in figure 7.3. The figure shows queries *q2*, *q3*, *q4*, *q5* and *q10*. The reason these queries are selected is that they scale the best, given the result of figure 7.1. I.e., these queries were better suited to run on $ds2^+$ in a reasonable time. The results of this demonstrate that subsampling is necessary for the system to run in a reasonable time. Note that the y-axis is once again on a logarithmic scale, further emphasizing the magnitude of performance differences.

For the sampled data sets, all queries except for *q14* execute in less than one hour, which shows that all queries have a reasonable latency with the sampling approach. However, given the approach of computing charging sessions and faults at query time, the non-sampled data sets are unfeasible to query on as the data volume is too large. This highlights a clear need to improve scalability, especially for queries involving *charging sessions* and *faults*, which exhibit high latencies due to their

computational complexity.

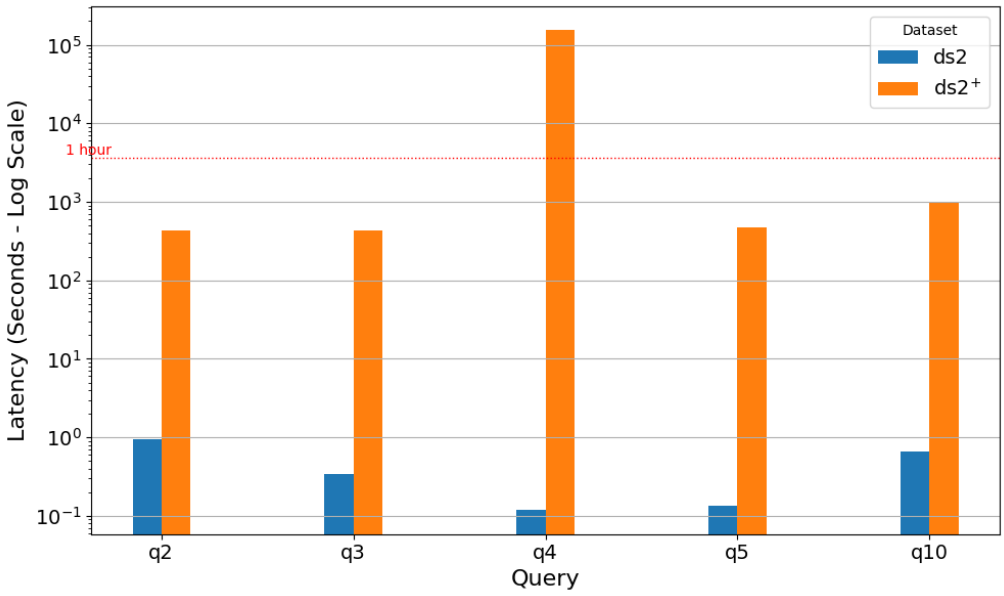


Figure 7.3: Comparison between ds2 and ds2$^+$

7.2 Effectiveness of analysis task

The effectiveness of the system is evaluated by how similar the output of the VKG is to the output of the charging success rate (CSR) use-case at Volvo. It also assesses the effectiveness of the VKG in answering analytical business questions. Dataset ds5 was used for the effectiveness evaluation, as it is the super set of ds1, ds2, ds3, and ds4. This means that all datasets are tested when using ds5.

The first conclusion of this metric is that every evaluation query is runnable, as seen in figure 7.1. This means that the VKG can be used to answer analytical business questions, as the evaluation queries are based on actual analytics questions for the CSR use case. Thus, the VKG can provide answers to analytical questions.

Query	Agreement Rate	Jaccard Index	Dice Coefficient	Average Similarity
q3	0.98	0.89	0.94	0.90
q6*	0.85	0.74	0.85	0.85
q10	0.96	0.91	0.95	0.94
q10*	0.55	0.44	0.61	0.55
q14*	0.9	0.81	0.9	0.89
VKG Average	**0.848**	**0.758**	**0.85**	**0.826**

Table 7.1: Similarity measurements between the system and the solution at Volvo.

The similarity of the output between the VKG and the CSR solution at Volvo is measured through the output of the auxiliary queries and *q3*, *q10*, which are

compared against the output of the CSR use-case. The similarity scores are shown in table 7.1.

In general, the average of all queries per metric gives a fairly high similarity. Jaccard Index is the lowest at 75.8% similarity, which is still quite similar. The Dice Coefficient score gave even better results, with an average of 85%. The fact that the Dice Coefficient is higher than the Jaccard Index indicates that there is a high degree of overlap between the sets, as the Dice Coefficient weights common elements more than the Jaccard Index. This shows that there are more shared elements between the two sets than unique elements. This is further highlighted by the fact that the Agreement Rate is around 85%, defined as the number of shared elements between the two sets, divided by the size of the output produced by the VKG. However, the results are not as high as they could have been. This is mainly due to two reasons: (1) The implementation of the concepts is not 100 percent conceptually correct. (2) The output of the CSR use-case contains only the results of the computations, and the exact time intervals of the original data are not available. This leads to slight time differences for some charging sessions.

The main outlier in table 7.1 is query *q10**. The reason *q10** has such a low similarity score is mainly because of reason (1). The VKG system implemented does not cover all cases of a successful charging session, and the definition of a failed charging session is simply a session that is not successful. Thus, by not being able to cover all successful cases, automatically, what is not found successful is defined as failed. This results in numerous mismatches when calculating the failed sessions.

In terms of effectiveness, the tool performs well for this use case, as all evaluation queries can be run and return a reasonable result. The majority of the queries score highly on the similarity metrics. Ignoring the outlier of *q10**, the average similarity over the metrics is around 75 to 80%, meaning that the way the VKG system is designed is similar to the calculations of the use case. Effectively, this suggests that a VKG is a valuable tool for data analysis, especially when it comes to calculating faults and events that occur in event-based time series. However, the results are inconclusive on whether a VKG is effective for continuous time series, which is further supported by it being an active field of research.

7.3 Extensibility

The extensibility of the VKG system is measured according to the two dimensions in section 6.5. The first dimension, mapping complexity, is presented in table 7.2. The second dimension, structural quality, is presented in table 7.3.

Mapping complexity, as seen in table 7.2, shows quite clearly that the most complex mapping is charging sessions. This is consistent with the observations we have made during the development of the VKG system as well. The source query in the mapping for charging sessions contains four subqueries, 19 case statements, and 17 functions. This translates to a highly complex mapping, which can also be seen in the results of the scalability metric, section 7.1.

Mapping Name	Score	Mapping Name	Score
charging-session	122.0	control-pilot-error-status	35.0
v2g-comm-fault	34.0	TCP-connection-fault	33.0
oceps-fault	29.0	grid-fault-oceps-reduced-performance	17.0
grid-fault-oceps-critical	11.0	control-pilot-state-e	11.0
CCCM-HVIL-fault	9.0	DCDC-fault	6.0
sensor-value-label	6.0	sysfault13	4.5
sysfault23	4.5	actccsconnectorstatus	4.5
CSU-fault	4.0	EVSE-isolation-fault	4.0
CCCM-CAN-timeout	4.0	sensor	3.5
sensor-has-sensor-value	3.5	vehicle-has-sensor	3.5
ccm-wrong-address	3.0	control-pilot	3.0
evse-current-limit-exceeded	3.0	infeasible-change	3.0
PE-voltage-offset-too-large	3.0	proximity-pilot	3.0
evse-power-limit	3.0	manual-unlock-during-charging	3.0
CCM-comm-issue-EVSE	3.0	charging-terminated-by-EVSE	3.0
inlet-temp-sensor-dc	3.0	vehicle	2.0
sensor-value	2.0		
Average score		11.7	
Median score		4.0	

Table 7.2: Complexity scores of mappings (based on SQL source patterns), sorted and arranged in two columns.

After charging sessions, contributing to high mapping complexity is a set of faults whose source queries have numerous conditions. For example, v2g-comm-fault, which contains 32 conditions. Although a high number of conditions increases mapping complexity, it does not necessarily imply high computational complexity. An exception is control_pilot_state_e, whose mapping definition contains one subquery and three conditions. Despite its relatively low mapping complexity, this suggests a higher computational complexity due to the subquery. The only join identified in the mappings occurs in sensor-value-label, which joins a table of labels to the sensor values. Similarly to control_pilot_state_e, this indicates a potentially high computational complexity while having a low mapping complexity.

Although some mappings have high complexity, the overall picture of the mapping complexity score is positive. The average score is 11.6, driven up by the high scoring mappings such as charging session and control-pilot-error-status. Without a charging session, the average drops to 8.3. However, the median score is just 4, indicating a low overall mapping complexity, which can indicate higher extensibility.

Structural quality is measured through the metrics in table 7.3. The metrics measure between $0 - 1$, apart from inverse multiple inheritance which scores over 1 because there is almost no inverse multiple inheritance in the VKG. It can be seen from the metric *instantiated property ratio* that all properties in the ontology contain instances, while from the metric *instantiated class ratio*, only most classes (also often referred to as concepts) contain instances. This is because certain classes such

as *Fault*, contain only subclasses but not any direct instances by themselves through the mappings. Rather, the mappings are created specifically for the subclasses of *Fault* since that is how the faults are defined in the use-case that the proof of concept aims to mimic.

Metric	Result
Instantiated Class Ratio	0.7895
Instantiated Property Ratio	1.0
Class Instantiation	0.39
Inverse Multiple Inheritance	1.52

Table 7.3: Results of structural quality metrics mentioned in section 6.5.2.

Class instantiation is the lowest scoring metric. This is due to the same fact as mentioned for the instantiated class ratio, where some classes, such as Fault and Vehicle_Fault do not have a direct mapping for instances. Rather, the instances come from the mappings to the subclasses of those classes. This is because the design of the underlying data does not lend itself to defining all faults by a single mapping, instead the faults are divided into their respective subclasses. However, for inference reasons, it still makes sense to have a superclass called *Fault* as it is analytically important to be able to find all faults in the VKG. However, this gives a good case for where extension efforts should be directed to the VKG system as a low score among class instantiation indicates that the classes are not defined in very high detail - something that can be remedied by extending the ontology and making it more detailed.

Table 7.4 contains the results from using the PostgreSQL query planner on every mapping. The results include the operation types which can be found in the PostgreSQL documentation [41]. Additionally, the startup cost and the total cost are present. Both these costs are an arbitrary total sum of the cost of all operations. These costs are what the planner uses to estimate which operations to choose and how expensive they are. The startup cost is a cost that will have to be paid every time such a query is executed, whilst the total cost is the maximum possible cost, achievable only if all values of the query are returned directly.

Cross-referencing the results of table 7.4 with table 7.2, there are some similarities in which mappings are most expensive, and some major differences. The largest difference is with regards to the mappings called sensor-value, sensor-has-sensor-value, and sensor-value-label which are classified as having a low complexity score in table 7.2 but have a high total cost in table 7.4. However, it should be noted that the startup cost of those mappings is quite low, which indicates that if the query is limited to e.g. 100 results, then it would be fast. In contrast, it has a high cost for fetching *all* results. Charging session on the other hand has both a high startup cost and a high total cost, indicating that the query can take a lot of time even if filtered or limited to a certain number of results.

To summarize, the results regarding extensibility show an overall low complexity of the mappings, with some outliers that need to be dealt with, and a high structural

quality. Given this result, if the outliers are rewritten, the VKG system will have high extensibility. Potential extension efforts in the future can possibly sacrifice some structural quality in order to make the VKG system more expressive and detailed.

Mapping Name	**Operation Type(s)**	**Startup Cost**	**Total Cost**
sensor-value-label	Gather, Hash Join, Parallel Seq Scan	1189.55	16469799.35
sensor-has-sensor-value	Seq Scan	0.00	4919307.92
sensor-value	Seq Scan	0.00	4919307.92
charging-session	Subquery Scan, Aggregate, Incremental Sort, Subquery Scan, WindowAgg, Aggregate, Incremental Sort, Subquery Scan, WindowAgg, Gather Merge, Sort, Parallel Seq Scan	124790.06	343445.05
control-pilot-state-e	Subquery Scan, WindowAgg, Gather Merge, Sort, Parallel Seq Scan	126319.92	243097.90
oceps-fault	Gather, Parallel Seq Scan	1000.00	105153.52
control-pilot-error-status	Gather, Parallel Seq Scan	1000.00	109786.78
TCP-connection-fault	Gather, Parallel Seq Scan	1000.00	108231.59
v2g-comm-fault	Gather, Parallel Seq Scan	1000.00	109010.34
grid-fault-oceps-reduced-performance	Gather, Parallel Seq Scan	1000.00	95851.41
grid-fault-oceps-critical	Gather, Parallel Seq Scan	1000.00	91127.86
CCCM-HVIL-fault	Gather, Parallel Seq Scan	1000.00	89572.97
DCDC-fault	Gather, Parallel Seq Scan	1000.00	87265.44
CCCM-CAN-timeout	Gather, Parallel Seq Scan	1000.00	85690.26
CSU-fault	Gather, Parallel Seq Scan	1000.00	85690.26
EVSE-isolation-fault	Gather, Parallel Seq Scan	1000.00	85685.76
proximity-pilot	Gather, Parallel Seq Scan	1000.00	84923.11
infeasible-change	Gather, Parallel Seq Scan	1000.00	84918.21
actccsconnectorstatus	Index Only Scan	0.29	10.63
sysfault13	Index Only Scan	0.29	10.63
vehicle	Seq Scan	0.00	2.34

Table 7.4: Query plans made by the PostgreSQL query planner, ordered by Total Cost (Highest to Lowest)

8

Discussion

This chapter presents a discussion of the trade-offs of the system, highlights some of the advantages VKG offers compared to a RDB, and provides a cross-metric analysis of the result. The chapter ends with a summary and suggestions for further work of the thesis.

8.1 Virtual Knowledge Graph vs RDB

The VKG developed was placed on top of a relational database, meaning that all SPARQL queries of the system are rewritten to SQL. This means that the relational database created for the VKG by definition can produce the same result without the VKG overhead. This section highlights the differences of having a VKG by comparing the evaluation and equivalent SQL queries in terms of interpretability and simplicity of a query. As an example, query *q11* is presented below, written in SPARQL and SQL.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?sensor_name ?time ?value ?explanation WHERE {
   ?vehicle :has_sensor ?sensor ;
            :name ?vehicle_name .
   ?sensor :has_value ?val ;
           :name ?sensor_name .
   ?val :value ?value ;
        :date_time ?time ;
       rdfs:label ?explanation .

   FILTER(?vehicle_name = "vehicle 1"^^xsd:string)
}
```

Listing 13: q11 written in SPARQL.

The main difference between SQL and SPARQL is that in SQL, the focus is on how the tables relate to each other and the keys used to join them, while in SPARQL,

```
SELECT sensors.name, sensor_value.time, sensor_value.value,
sensor_value.explanation FROM vehicles
JOIN sensors ON vehicles.id = sensor.vehicle_id
JOIN sensor_values ON sensor.id = sensor_value.sensor_id
WHERE vehicles.name = 'vehicle 1';
```

Listing 14: q11 written in SQL.

the focus is on the relationships and the concepts. While the query is simpler to write in SQL, it still requires extensive knowledge of the database schema and how everything is connected. For SPARQL, the only thing required of the person writing the query is an understanding of the domain rather than the data schema. If concepts and relationships are clearly defined, a query in SPARQL can be written entirely without knowing the schema or structure of the underlying data source.

8.2 Trade-off in computational placement

There exists a trade-off when designing the underlying data source for a VKG: (1) having complex mappings with general tables or (2) having simpler mappings but with more complex preprocessing, creating more concept specific tables. This is a trade-off of where the complexity should be, either in the pipeline when computing the underlying data source of the VKG or when querying the VKG.

An example of a scientific work that used approach (2) is the Bosch case study [3] where all specific faults were preproccessed to a fault-table. In comparison, the VKG in this thesis uses method (1), where each fault and charging session was computed when queried with the sampled signal data as input.

For a VKG, one can conclude from the result of this thesis that, in terms of the latency approach (2) is better once the VKG system is in place. This is clear, as the most expensive queries in terms of latency involve computing concepts, which can be seen in the scalability result figure 7.2. This can also be confirmed for the extensibility metric, in tables 7.4 and 7.2, where the cost of the source query of the charging session and some of the faults are high.

The decision to choose approach (1) was made in the aspect of generality, keeping the same structure of the data as the data lake, without tailoring the underlying data for the specific use case. For the use case and approach (1), subsampling was necessary to decrease the volume of data the query used for the computations to decrease the latency of the queries while not altering the result of the query. However, the same subsampling technique could be applied to compute the preprocessed tables in approach (2).

In more general terms, this highlights that a VKG framework needs a good structure of the underlying data source to be performant and extensible. Approach (2) therefore better suits the VKG as it puts more emphasis on structuring the data before applying it to the VKG. This comes at the cost of requiring extensive preprocessing, which can be a limiting factor when extending the VKG to more use cases.

8.3 Conclusion

This thesis presents a framework for analysis of high volume, high variety, continuous, and event-based time series data, using a Virtual Knowledge Graph (VKG) approach supported by an underlying PostgreSQL data source. The integration is made with the aim of enriching the data with semantics, enabling more effective and meaningful analysis of the data. Comprising the VKG system are multiple preprocessed PostgreSQL tables derived from a data lake along with the VKG itself. To evaluate the proposed approach, a proof-of-concept system was developed for Volvo Group's Charging Success Rate use case.

The evaluation of the system focused on scalability, effectiveness, and extensibility. Scalability was evaluated on data sets of five different sizes, effectiveness on the largest data set, and extensibility through an analysis of two dimensions of VKG extensibility.

The results for the scalability metric indicate that the VKG can handle most evaluation queries in a reasonable time frame. However, query *q14* is an exception, having a latency of around 140 minutes. This is due to the decision of having a general sampled table for multiple concepts, which negatively affects scalability and extensibility, as discussed in section 8.2. By preprocessing the concepts and ingesting the result into concept tables, the latency of each query can be reduced at the expense of an increased upfront computational cost for these tables.

For the effectiveness, the VKG approach is proven to be well suited for the use case, as it successfully answers all business-related evaluation queries and generates results that are 85% similar using the Dice Coefficient compared to those of Volvo's solution. Furthermore, the introduction of a VKG with an ontology, semantics, and mappings increases the expressiveness and semantic interpretability of the data.

In terms of extensibility, the results indicate that the VKG has a high structural quality and a reasonable average mapping complexity. However, mapping complexity can be improved by addressing complex mappings, such as the charging session mapping.

In conclusion, the proposed VKG framework is shown to effectively enable semantically enriched analysis of complex time series data for a real-world industrial use case. The main takeaway is that when the VKG is integrated with time series data of high volume and variety, it is necessary to scale down the data, as the computational complexity of the analysis is disproportional to the data volume.

8.4 Further Work

This section describes the different areas in which future work can be directed. Both in terms of extending the current system, making it more performant, and also in other areas of research that are closely related to knowledge graphs, where efforts can be directed to make knowledge graphs more appealing for the industry use case.

8.4.1 Increasing performance

Performance of the proof of concept is currently hindered by several factors. These are: High Data Volume, Limitations in VKG system implementations, Underlying data source structure, and Complex mappings.

To remedy the problems of a high data volume and the underlying data source structure, better preprocessing is needed. This preprocessing needs to take into account the underlying data source structure and transform this into something that the VKG system can use, and it also needs to make sure that the high data volume is reduced significantly. This can be done e.g., by removing redundant data points, which is the case with many event-based time series, where keeping only the time when an event occurs is sufficient. For continuous time series, more research has to be done into the storage and querying of such time series before they are ready to be used in a VKG setting, perhaps by incorporating some of the methods for sampling continuous time series mentioned in section 2.2.2. Another way of reducing data volume is to implement sketches, either on the edge close to the data source or in a processing step somewhere in the data pipeline. This can drastically lower the data volume, but comes with the drawback that sketches can only answer approximate results for a set of predefined queries.

Another point of interest with respect to increasing performance is the results of the mapping complexities in table 7.2. As can be seen certain mappings are very complex, and cross referencing this with the result of the scalability metric, it is also specifically the concepts calculated by these mappings that have poor scaling and high latencies. To address this issue and improve query latency, it is suggested that further work focus on ensuring that all mappings are simple mappings where each concept has its pre-computed table.

Finally, there is an issue that VKG systems are currently limited in performance because they require well structured underlying data sources and need the metadata provided by the underlying data source to optimize query rewriting. In this regard, more research is needed, specifically on how a VKG system integrates with the underlying data source.

8.4.2 Materialized and Hybrid Approach

An approach that we investigated early on for the thesis was to either materialize the whole graph into a graph database. This was scoped out of the project early, mainly due to time constraints since there was no implemented VKG for the use case when we started, but also due to the sheer size of the dataset. Now that a tool is implemented, this is a good research topic to evaluate the trade-offs of having a system with more overhead and no data duplication, such as a virtual approach with a fully materialized knowledge graph in a graph database. This could open up more research topics, such as comparing state-of-the-art graph databases or graph models like RDF and property graphs.

Another aspect of this is identifying an efficient hybrid approach by investigating whether certain parts of the graph should be materialized or stay virtual. This can

be done by capturing query trends and materializing the most queried concepts, or identifying and materializing the metadata, for example.

8.4.3 Extending the VKG

Extending the VKG with new, preferably related use cases is essential. This enables even more complex querying, providing deeper insights across multiple domains. Knowledge graphs shine the most in this area, as they allow data analysts to query for the data more easily if the domain expert has done the groundwork and defined the concepts and relationships in the domain.

An example of this related to the use case would be to further extend the KG of areas that could impact the Charging Success Rate. This could be done by adding the configurations of a vehicle, such as the software and hardware of a vehicle. Extending the model in this fashion can lead to new insights for the charging success rate metric, finding new correlations that would be hard to identify without the extension.

Another aspect of extending the tool is by creating an extensive vocabulary for the ontology, which will work as a standard for the company. This is necessary to avoid multiple names for the same concept and enables the data analyst to have an overview of what concepts one can query for.

8.4.4 Knowledge Graphs and AI

There is a lot of research combining AI and *Knowledge Graphs*. AI can leverage semantics through inference and interpretability, which a KG provides, to produce insightful results [43].

Another aspect of combining AI and KG is to find alternative ways to query a KG rather than a graph query. An example of this is question answering from Knowledge Graphs, which uses a neural network that creates an answer based on natural language querying [44].

There is also research regarding *Knowledge Graph Completion* (KGC), which focuses on automating the knowledge graph creation, as it is time-consuming to do manually. One way to do this is by leveraging neural networks to reason its way to new concepts and relationships in the domain given a knowledge graph [45].

Bibliography

[1] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big automotive data: Leveraging large volumes of data for knowledge-driven product development," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 736–741. DOI: 10.1109/BigData.2014.7004298.

[2] G. Xiao, L. Ding, B. Cogrel, and D. Calvanese, "Virtual knowledge graphs: An overview of systems and use cases," *Data Intelligence*, vol. 1, no. 3, pp. 201–223, 2019.

[3] E. G. Kalayc, I. Grangel González, F. Lösch, *et al.*, "Semantic integration of bosch manufacturing data using virtual knowledge graphs," in *The Semantic Web – ISWC 2020*, J. Z. Pan, V. Tamma, C. d'Amato, *et al.*, Eds., Cham: Springer International Publishing, 2020, pp. 464–481, ISBN: 978-3-030-62466-8.

[4] A. Hogan, E. Blomqvist, M. Cochez, *et al.*, "Knowledge graphs," *ACM Comput. Surv.*, vol. 54, no. 4, Jul. 2021, ISSN: 0360-0300. DOI: 10.1145/3447772. [Online]. Available: https://doi.org/10.1145/3447772.

[5] M. Besta, R. Gerstenberger, E. Peter, *et al.*, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *ACM Comput. Surv.*, vol. 56, no. 2, Sep. 2023, ISSN: 0360-0300. DOI: 10.1145/3604932. [Online]. Available: https://doi.org/10.1145/3604932.

[6] N. F. Noy, D. L. McGuinness, *et al.*, *Ontology development 101: A guide to creating your first ontology*, 2001.

[7] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A survey on knowledge graphs: Representation, acquisition, and applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, 2022. DOI: 10.1109/TNNLS.2021.3070843.

[8] B. Mörzinger, "Accessing manufacturing data through virtual knowledge graphs: On the value and semantic peculiarities of time series data," Ph.D. dissertation, Technische Universität Wien, 2019.

[9] M. Ashraf, F. Anowar, J. H. Setu, *et al.*, "A survey on dimensionality reduction techniques for time-series data," *IEEE Access*, vol. 11, pp. 42 909–42 923, 2023. DOI: 10.1109/ACCESS.2023.3269693.

[10] R. Brachman and H. Levesque, *Knowledge representation and reasoning*. Elsevier, 2004.

[11] F. Baader, *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[12] W. W. W. Consortium. "Owl 2 web ontology language: Overview." B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. F. Patel-Schneider, and U. Sattler, Eds., World Wide Web Consortium (W3C). (2012), [Online]. Available: https://www.w3.org/TR/owl2-overview/ (visited on 02/19/2025).

[13] W. W. W. Consortium. "Rdf schema 1.1." R. G. Dan Brickley, Ed., World Wide Web Consortium (W3C). (2014), [Online]. Available: https://www.w3.org/TR/2014/REC-rdf-schema-20140225/ (visited on 05/14/2025).

[14] J. Z. Pan, "Resource description framework," in *Handbook on ontologies*, Springer, 2009, pp. 71–90.

[15] M. Arenas and J. Pérez, "Querying semantic web data with sparql," in *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2011, pp. 305–316.

[16] W. W. W. Consortium. "Sparql query language for rdf." E. Prud'hommeaux and C. Buil-Aranda, Eds., World Wide Web Consortium (W3C). (2013), [Online]. Available: https://www.w3.org/TR/sparql11 (visited on 04/20/2025).

[17] B. Abu-Salih, "Domain-specific knowledge graphs: A survey," *Journal of Network and Computer Applications*, vol. 185, p. 103 076, 2021, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103076. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000990.

[18] A. Khan, "Knowledge graphs querying," *SIGMOD Rec.*, vol. 52, no. 2, pp. 18–29, Aug. 2023, ISSN: 0163-5808. DOI: 10.1145/3615952.3615956. [Online]. Available: https://doi.org/10.1145/3615952.3615956.

[19] A. Hogan, E. Blomqvist, M. Cochez, *et al.*, "Knowledge graphs," *ACM Computing Surveys (Csur)*, vol. 54, no. 4, pp. 1–37, 2021.

[20] J. L. Harrington, *Relational database design and implementation*. Morgan Kaufmann, 2016.

[21] C. J. Date, *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[22] D. Calvanese, A. Gal, D. Lanti, M. Montali, A. Mosca, and R. Shraga, "Conceptually-grounded mapping patterns for virtual knowledge graphs," *Data & Knowledge Engineering*, vol. 145, p. 102 157, 2023.

[23] G. Xiao, D. Lanti, R. Kontchakov, *et al.*, "The virtual knowledge graph system ontop," in *International Semantic Web Conference*, Springer, 2020, pp. 259–277.

[24] *R2rml: Rdb to rdf mapping language*, Sep. 2012. [Online]. Available: https://www.w3.org/TR/r2rml/.

[25] D. Calvanese, B. Cogrel, S. Komla-Ebri, *et al.*, "Ontop: Answering sparql queries over relational databases," *Semantic Web*, vol. 8, no. 3, pp. 471–487, 2017.

[26] E. Kalayci, S. Brandt, D. Calvanese, V. Ryzhikov, G. Xiao, and M. Zakharyaschev, "Ontology-based access to temporal data with ontop: A framework proposal," *International Journal of Applied Mathematics and Computer Science*, vol. 29, no. 1, pp. 17–30, 2019.

[27] J. Sequeda and O. Lassila, *Designing and building enterprise knowledge graphs*. Springer Nature, 2022.

[28] G. Cormode and K. Yi, *Small summaries for big data*. Cambridge University Press, 2020.

[29] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, vol. 15, 2011.

[30] H. Huang, T. Shah, and S. Yoo, "Deep time series sketching and its application on industrial time series clustering," in *2022 IEEE International Conference on Big Data (Big Data)*, IEEE, 2022, pp. 1997–2006.

[31] *Asam mdf*, ASAM, 2009. [Online]. Available: `https://www.asam.net/standards/detail/mdf/wiki/`.

[32] *Parquet file format*, Apache Parquet, 2024. [Online]. Available: `https://parquet.apache.org/docs/file-format/`.

[33] D. Calvanese, C. Okulmus, M. Ortiz, and M. imkus, "On the way to temporal obda systems," in *15th Alberto Mendelzon International Workshop on Foundations of Data Management, AMW 2023, Santiago de Chile, Chile, May 22-26, 2023*, CEUR-WS, 2023.

[34] B. Mörzinger, "Accessing manufacturing data through virtual knowledge graphs: On the value and semantic peculiarities of time series data," Ph.D. dissertation, Technische Universität Wien, 2019.

[35] A. GraSS, C. Beecks, S. A. Chala, C. Lange, and S. Decker, "A knowledge graph for query-induced analyses of hierarchically structured time series information," in *European Conference on Advances in Databases and Information Systems*, Springer, 2023, pp. 174–184.

[36] P. B. Gibbons, "Big data: Scale down, scale up, scale out," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 3–3. DOI: `10.1109/IPDPS.2015.123`.

[37] S. Hoseini, J. Theissen-Lipp, and C. Quix, "Semantic data management in data lakes," *arXiv preprint arXiv:2310.15373*, 2023.

[38] M. Fuller, M. Moser, and M. Traverso, *Trino: The Definitive Guide: SQL at Any Scale, on Any Storage, in Any Environment*. " O'Reilly Media, Inc.", 2022.

[39] M. A. Musen, "The protégé project: A look back and a look forward," *AI matters*, vol. 1, no. 4, pp. 4–12, 2015.

[40] W. B. Frakes and R. Baeza-Yates, *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc., 1992.

[41] *Postgresql 17.5 documentation*, PostgreSQL Global Development Group, 2025. [Online]. Available: `https://www.postgresql.org/docs/current/`.

[42] S. Seo, H. Cheon, H. Kim, and D. Hyun, *Structural quality metrics to evaluate knowledge graphs*, 2022. arXiv: `2211.10011 [cs.AI]`. [Online]. Available: `https://arxiv.org/abs/2211.10011`.

[43] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3580–3599, 2024. DOI: `10.1109/TKDE.2024.3352100`.

[44] M. Yasunaga, H. Ren, A. Bosselut, P. Liang, and J. Leskovec, "Qa-gnn: Reasoning with language models and knowledge graphs for question answering," *arXiv preprint arXiv:2104.06378*, 2021.

[45] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, "A review of relational machine learning for knowledge graphs," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 11–33, 2016. DOI: 10.1109/JPROC.2015.2483592.

A

Appendix 1

A.1 Charging Success Usecase Queries

The following queries are the ones considered for the Charging Success Usecase.

Query q1: *"Return all vehicles and the name of the sensors, order by vehicle"*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle ?name WHERE {
    ?vehicle a :Vehicle .
    ?vehicle :has_sensor ?sensor .
    ?sensor :name ?name .
} ORDER BY ?vehicle
```

Listing 15: SPARQL query for q1.

Query q2: *"Return the name of all sensors that cause the fault 'DCDC Fault'"*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT DISTINCT ?fault_name ?sensor_name WHERE {
  ?fault a :DCDC_Fault .
   ?fault :caused_by ?sensor ;
              rdfs:label ?fault_name .
  ?sensor :name ?sensor_name .
}
```

Listing 16: SPARQL query for q2.

Query q3: *"Return the vehicle name and all valid charging sessions that have occurred after the timestamp '2025-01-28 07:44:14' for a given vehicle"*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?vehicle_name ?charging_session ?start ?end WHERE {
        ?vehicle :has_charging_session ?charging_session ;
                 :name ?vehicle_name .
        ?charging_session :session_start ?start ;
                          :session_end ?end .
        FILTER(?vehicle_name = "CUST-111"^^xsd:string ) .
        FILTER(?start >  "2025-01-28T07:04:14+02:00"^^xsd:dateTime) .
}
```

Listing 17: SPARQL query for q3.

Query q4: *"Return all instances of OCEPS fault, and the vehicles connected to this fault. Filter on the faults that occurred between 2025-01-27 and 2025-02-02"*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?fault ?vehicle WHERE {
  ?fault a :OCEPS_fault ;
           :date_time ?time .
  ?vehicle :has_fault ?fault .

  FILTER(?time > "2025-01-27T00:00:00+01:00"^^xsd:dateTime &&
         ?time < "2025-02-02T00:00:00+01:00"^^xsd:dateTime)
}
```

Listing 18: SPARQL query for q4.

Query q5: *"Return the names of all faults associated with the sensor 'actccsconnectorstatus_ep2_x_cc'"*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT DISTINCT ?sensor ?fault_name WHERE {
    ?sensor a :Actccsconnectorstatus .
    ?fault :caused_by ?sensor ;
       rdfs:label ?fault_name .
}
```

Listing 19: SPARQL query for q5.

Query q6: *"Return the vehicle name, the count of the number of charging sessions and the charging type in that session, group by vehicle and charging type (AC/DC)."*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle_name (COUNT(?cs) AS ?sessions) ?charge_type WHERE {
  ?vehicle :has_charging_session ?cs ;
           :name ?vehicle_name .
  ?cs :charging_type ?charge_type ;
      :isValid ?validity .

  FILTER(?validity = "true"^^xsd:boolean) .
} GROUP BY ?vehicle_name ?charge_type
```

Listing 20: SPARQL query for q6.

Query q7: *"Calculate the aggregated charging session success rate for all charging sessions."*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT (?success / (?success + ?fail) AS ?rate) WHERE {
    {
    SELECT (COUNT(?cs) as ?success) WHERE {
        ?cs a :Charging_Session ;
              :isValid ?validity ;
              :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean && ?isSuccess = "true"^^xsd:boolean)
        }
    }

    {
    SELECT (COUNT(?cs) as ?fail) WHERE {
        ?cs a :Charging_Session ;
              :isValid ?validity ;
              :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean && ?isSuccess = "false"^^xsd:boolean
        }
    }
}
```

Listing 21: SPARQL query for q7.

Query q8: *"Calculate the duration of all valid charging sessions and filter out ses-*

sions whose durations are less than 15 minutes, also return the vehicle. Order by the smallest duration."

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ofn: <http://www.ontotext.com/sparql/functions/>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle ?duration WHERE {
  ?vehicle :has_charging_session ?charging_session .
  ?charging_session :session_start ?start_ts ;
                    :session_end ?end_ts ;
                    :isValid ?validity .

  BIND(ofn:minutesBetween(?start_ts, ?end_ts) as ?duration) .
  FILTER(?validity = "true"^^xsd:boolean && ?duration > 15)

} ORDER BY ?duration
```

Listing 22: SPARQL query for q8.

Query q9: *"Calculate an aggregated charging success rate for every vehicle and filter the results for any rate > 95% or a rate < 50%, group by vehicle, and order by success rate."*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle ?rate WHERE {
  {
  SELECT ?vehicle (COUNT(?cs) as ?success) WHERE {
        ?vehicle :has_charging_session ?cs .
        ?cs  :isValid ?validity ;
             :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean &&
               ?isSuccess = "true"^^xsd:boolean) .
    } GROUP BY ?vehicle
  }
  {
  SELECT ?vehicle (COUNT(?cs) as ?fail) WHERE {
        ?vehicle :has_charging_session ?cs .
        ?cs :isValid ?validity ;
            :isSuccessful ?isSuccess .

        FILTER(?validity = "true"^^xsd:boolean &&
               ?isSuccess = "false"^^xsd:boolean) .
     } GROUP BY ?vehicle
  }

  BIND(?success / (?success + ?fail) AS ?rate).
  FILTER(?rate > 0.9 || ?rate < 0.5)
} order by ?rate
```

Listing 23: SPARQL query for q9.

Query q10: *"Return all vehicle names, charging session start time, and charging session end time where all sessions are valid and successful."*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle_name ?session_start ?session_end  WHERE {
    ?vehicle :has_charging_session ?cs ;
             :name ?vehicle_name .
    ?cs :isValid ?validity ;
        :isSuccessful ?success ;
        :session_start ?session_start ;
        :session_end ?session_end .

  FILTER(?validity = "true"^^xsd:boolean &&
         ?success = "true"^^xsd:boolean) .
}
```

Listing 24: SPARQL query for q10.

Query q11: *"Return sensor names, timestamp, values, and the explanation for the values, filter on vehicle = 'CUST-111'"*

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?sensor_name ?time ?value ?explanation WHERE {
    ?vehicle :has_sensor ?sensor ;
             :name ?vehicle_name .
    ?sensor :has_value ?val ;
            :name ?sensor_name .

    ?val :value ?value ;
         :date_time ?time ;
     rdfs:label ?explanation .

     FILTER(?vehicle_name = "CUST-111"^^xsd:string)
}
```

Listing 25: SPARQL query for q11.

Query q12: *"Count the number of faults of the fault sources; EVSE failure, EVSE or Vehicle failure, User failure, and Vehicle failure, for the vehicle CUST-111 that occurred in valid charging sessions, group by fault source"*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?fault_source (SUM(?fault_count) as ?source_count) WHERE {
    {
    SELECT ?fault_source (COUNT(DISTINCT ?charging_session) as ?fault_count)
    WHERE {
        ?vehicle :has_charging_session ?charging_session ;
                 :has_fault ?fault ;
                 :name ?name .

        ?fault rdfs:label ?fault_type ;
                   :date_time ?fault_ts ;
                rdf:type ?fault_source .

        ?charging_session :session_start ?start_ts ;
                          :session_end ?end_ts ;
                          :isValid ?validity .


        FILTER(?validity = "true"^^xsd:boolean) .
        FILTER(?name = "CUST-111"^^xsd:string) .
        FILTER(?fault_source IN(:Evse_fault, :Vehicle_fault,
                                :User_Fault, :Evse_or_Vehicle_fault)) .
        FILTER(?start_ts <= ?fault_ts && ?fault_ts <= ?end_ts) .

        } GROUP BY ?fault_type ?fault_source
    }
} GROUP BY ?fault_source
```

Listing 26: SPARQL query for q12.

Query q13: *"Count the number of faults in every charging session, group by session type (success/failure) and fault, filter on vehicle = CUST-111"*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?fault (COUNT(DISTINCT ?cs) as ?num_faults) ?isSuccessful WHERE {
    ?vehicle :has_charging_session ?cs ;
             :has_fault ?f ;
             :name ?name .
    ?cs :session_start ?start ;
        :session_end ?end ;
        :isSuccessful ?isSuccessful ;
        :isValid ?validity .

    ?f rdfs:label ?fault ;
           :date_time ?f_ts .

    FILTER(?name = "CUST-111"^^xsd:string &&
           ?validity = "true"^^xsd:boolean &&
           ?start <= ?f_ts && ?end >= ?f_ts) .
} GROUP BY ?fault ?isSuccessful
```

Listing 27: SPARQL query for q13.

Query q14: *"Count the number of occurrences of each fault type that are associated to valid charging sessions, group by fault type, and order by the count."*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?fault_type (COUNT(distinct ?charging_session) AS ?fault_count)
WHERE {
    ?vehicle :has_charging_session ?charging_session ;
             :has_fault ?fault .
    ?charging_session :session_start ?start_ts ;
                      :session_end ?end_ts ;
                      :isValid ?validity .
    ?fault rdfs:label ?fault_type ;
               :date_time ?fault_ts .
    FILTER(?validity = "true"^^xsd:boolean &&
           ?start_ts <= ?fault_ts && ?fault_ts <= ?end_ts)
}
GROUP BY ?fault_type ORDER BY ?fault_count
```

Listing 28: SPARQL query for q14.

A.2 Auxiliary queries

Below are auxiliary queries that are either the same queries as above except that they do not aggregate the result together, or a variation of the original query. The auxiliary queries will be named with a *. These are used for the effectiveness evaluation metric (6.4) in order to compare the result with Volvo's result.

Query q6*: *"Return all charging sessions, type of charger, success or failure, where the charging sessions are valid"*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle ?name WHERE {
    ?vehicle a :Vehicle .
    ?vehicle :has_sensor ?sensor .
    ?sensor :name ?name .
} ORDER BY ?vehicle
```

Listing 29: SPARQL query for q6*. Auxiliary query to q6 that does the same thing except that the result is not aggregated together

Query q10*: *"Return all failed but valid charging sessions"*

```
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT ?vehicle_name ?session_start ?session_end  WHERE {
    ?vehicle :has_charging_session ?cs ;
             :name ?vehicle_name .
    ?cs :isValid ?validity ;
        :isSuccessful ?success ;
        :session_start ?session_start ;
        :session_end ?session_end .

    FILTER(?validity = "true"^^xsd:boolean &&
           ?success = "false"^^xsd:boolean)
}
```

Listing 30: SPARQL query for q10*. Auxiliary query to q10 that does the same thing except that the result is for all failed charging sessions

Query q14*: *"Return the vehicle name, the charging session, the charging session start time and end time, and all faults that have occurred in every charging session"*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.gra.fo/schema/untitled-ekg#>

SELECT DISTINCT ?vehicle_name ?fault_type  ?charging_session ?start_ts ?end_ts
WHERE {
    ?vehicle :has_charging_session ?charging_session  ;
             :name ?vehicle_name ;
             :has_fault ?fault .
    ?charging_session :session_start ?start_ts ;
                      :session_end ?end_ts ;
                      :isValid ?validity .
    ?fault rdfs:label ?fault_type ;
               :date_time ?fault_ts .

    FILTER(?validity = "true"^^xsd:boolean &&
           ?start_ts <= ?fault_ts && ?fault_ts <= ?end_ts)
}
```

Listing 31: SPARQL query for q14*. Auxiliary query to q14 that does the same thing except that the result is not aggregated together