Informatics

Extraktion von SHACL Shapes für Evolving Knowledge Graphs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Eva Pürmayr, BSc
Matrikelnummer 11807199

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.-Ing. Katja Hose

Wien, 2. Dezember 2024

Eva Pürmayr

Katja Hose

Informatics

SHACL Shapes Extraction for Evolving Knowledge Graphs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Eva Pürmayr, BSc
Registration Number 11807199

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.-Ing. Katja Hose

Vienna, December 2, 2024

Eva Pürmayr Katja Hose

Erklärung zur Verfassung der Arbeit

Eva Pürmayr, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Dezember 2024

Eva Pürmayr

Danksagung

Zu Beginn möchte ich meiner Betreuerin, Katja Hose, für ihre Unterstützung und das Feedback während dieser Arbeit danken. Außerdem danke ich Kashif Rabbani, dem Autor von QSE, für seine Hilfe bei meinen Fragen. Zudem möchte ich den Teilnehmer*innen der Interviews für Ihre Zeit und Geduld danken.
Zuletzt bin ich besonders dankbar für die Unterstützung meiner Familie, meiner Freunde und Freundinnen. Eure Beiträge waren sehr hilfreich für den Abschluss dieser Arbeit.

Acknowledgements

I would like to express my gratitude to my advisor, Katja Hose, for her guidance and feedback throughout this project. I also thank Kashif Rabbani, the author of QSE, for his assistance with my questions. Additionally, I want to thank the participants of the expert interview for their time and patience.
Lastly, I am especially grateful to my family and friends for their constant encouragement. Your contributions have been extremely helpful to complete this work.

Kurzfassung

RDF-Wissensgraphen haben ein breites Anwendungsspektrum in der Industrie und in der Wissenschaft. Dabei ist die Datenqualität entscheidend, welche mit der Validierungssprache SHACL überprüft werden kann. Es gibt bereits ein Tool namens QSE (Quality Shapes Extraction), welches automatisch SHACL-Shapes von großen Datensets generiert. Eine webbasierte Erweiterung namens Shactor visualisiert den Extraktionsprozess und zeigt Statistiken an. Wissensgraphen sind allerdings nicht statisch und können mehrere Versionen haben, welche sich eventuell nur minimal unterscheiden. In diesem Bereich gibt es noch keine Tools, mit welchen SHACL-Shapes zwischen verschiedenen Wissensgraphversionen verglichen werden können. Eine weitere offene Frage ist, ob es Möglichkeiten gibt, QSE für Graphen mit mehreren Versionen zu beschleunigen.
Um diese erste Wissenslücke zu schließen, wurde eine webbasierte Anwendung erstellt, mit der SHACL-Shapes verglichen werden können. Ebenso wurden zwei Algorithmen entwickelt. Einer beschäftigt sich mit der Verwendung von Changesets zwischen verschiedenen Graphversionen, um den QSE-Prozess zu beschleunigen, während der andere bestehende SHACL-Shapes in nachfolgenden Versionen eines Wissensgraphen prüft.
Die Forschungsfragen wurden mithilfe des Design-Science-Research-Frameworks beantwortet, wobei als Methoden eine systematische Literaturrecherche, Prototyping, algorithmisches Design, halb-strukturierte Experteninterviews und technische Experimente verwendet wurden.
Die wesentlichen Ergebnisse zeigen, dass ein webbasiertes Tool zum Vergleich von SHACL-Shapes benutzerfreundlicher ist als der Verzicht auf zusätzliche Anwendungen. Der SHACL-Extraktionsprozess von QSE kann durch die Verwendung von Changesets beschleunigt werden, jedoch nur, wenn die Changesets im Vergleich zum tatsächlichen Graphen relativ klein sind. Darüber hinaus ist die Nutzung von SPARQL-Abfragen zur Überprüfung von existierenden SHACL-Shapes in folgenden Graphversionen schneller als QSE mehrfach auszuführen. Diese Version hat allerdings den Nachteil, dass hinzugefügte oder aktualisierte Shapes nicht erkannt werden können.

Abstract

RDF knowledge graphs have a broad range of applications in academia and industry. Ensuring data quality is vital and SHACL can be used as a validation language for this purpose. Previously, an approach called QSE (Quality Shapes Extraction) which automatically extracts SHACL shapes from large datasets, has been released. An extension to this program is called Shactor, a web-based tool that visualizes the extraction process and provides statistics. Since knowledge graphs are not static, there may exist different versions of a graph, maybe with only minimal changes. There is a lack of tools to compare SHACL shapes between these graph versions. Another open issue is determining whether there are methods to accelerate QSE for evolving knowledge graphs.
To address this first gap, the proposed solution involves creating a web-based tool for SHACL shape comparison. Additionally, two algorithms were developed. One focuses on using changesets between graph versions to optimize the QSE extraction process, while the other one is designed to verify existing SHACL shapes in subsequent versions of a knowledge graph.
The research questions were answered by using the Design Science Research framework, utilizing methods such as a Systematic Literature Review, prototyping, algorithm design, semi-structured expert interviews, and technical experiments.
The key findings of the thesis indicate that a web-based tool for SHACL shape comparison is more user-friendly than using no additional tools. The SHACL shapes extraction process used in QSE can be accelerated using graph changesets, but only when the changesets are relatively small in comparison to the actual graph file. Additionally, using SPARQL queries to identify unchanged shapes in a subsequent graph version is faster than executing QSE multiple times, although it has the drawback of not being able to detect added or updated shapes.

Contents

CHAPTER 1

Introduction

Storing complex and interconnected data is quite a challenge for data engineers. Knowledge graphs offer a versatile solution and have a broad range of applications, both in industry and academia. For instance, knowledge graphs can serve as the foundation for machine learning algorithms. A prominent example of a knowledge graph is DBpedia [5], which contains the data behind Wikipedia. This example highlights the potential size of knowledge graphs. Beyond DBpedia, there are many other examples of knowledge graphs across various domains.
Given the wide range of applications and the massive amounts of data involved, maintaining data quality is crucial. Changes to knowledge graphs can occur frequently and may originate from different stakeholders, sometimes introducing errors. For instance, the data in DBpedia changes at least daily, with numerous users from around the globe contributing to these updates. The potential errors in such large knowledge graphs pose a specific problem for applications that rely on the trustworthiness of data within a knowledge graph. Ensuring data quality is therefore vital.
Knowledge graphs can be expressed in the Resource Description Framework (RDF) [20], and data quality can be ensured by using the validation language SHACL [23]. With SHACL, a schema in the form of so-called shapes can be created, which can be used to validate a graph. During this validation, erroneous data can be filtered out, enhancing overall data quality. Creating such schemas manually is impractical - therefore, algorithms have been developed that automatically extract SHACL shapes from large knowledge graphs. One such algorithm is QSE (Quality Shapes Extraction) [84], which uses parameters to create a meaningful set of shapes. This is crucial because an automatically created schema itself can contain errors if it is based on incorrect data. Once a schema is generated, it can be used to validate existing data and filter out incorrect data.
Another important aspect in this area is the evolution of knowledge graphs. Since knowledge graphs are not static, there may exist different versions of a graph, maybe with minimal changes. This evolution is particularly relevant to the topic of data quality.

1.1 Problem Statement

The evolution of knowledge graphs increases the complexity of schema creation to ensure data quality. QSE was written with a focus on the extraction of SHACL shapes for a specific version of a graph, but it does not provide a way to compare these shapes between different graph versions. It would be beneficial for users to identify shapes that remain unchanged across multiple versions of a graph or to detect shapes that have changed. Currently, state of the art is to manually compare the extracted shapes between versions, using a text comparison tool. Using manual methods is the only option, but this is tedious and lacks practicality.
Another problem in this area is the inefficiency of running QSE independently on each version of a graph, even when there are only minimal changes in the data and schema.
In summary, this work aims at shapes extraction and comparison, specifically in the context of evolving knowledge graphs. As there is currently no tool available for this use case, the characteristics of usability and execution time remain unmeasurable.

1.2 Goals and Expected Outcome

To overcome these issues, a user interface will be created, which allows users to conveniently compare SHACL shapes between different versions of a graph. For reusability, graph versions and extracted shapes will be stored in a local database. The tool will allow users to identify which shapes remained the same, have changed, have been added, or have been deleted. Additionally, it will provide information on why shapes were deleted. These features will apply not only to two graph versions but to multiple versions.
The second main goal of this thesis is to explore various methods for making the SHACL shapes extraction more efficient for evolving knowledge graphs. One approach is to check existing SHACL shapes generated by QSE on a subsequent version of a graph using the query language SPARQL. This reduces the execution time for the subsequent version, although it has the drawback of not generating shapes for newly added data. Another approach is to use the changeset between two graph versions and the results from the initial QSE run to build the shapes for the next version. This method also reduces the execution time but requires the changeset between graph versions.
Summing up, after this thesis is finished, users should be able to conveniently and efficiently compare SHACL shapes across various versions of a knowledge graph.

1.3 Research Questions

To specify these goals, the following research questions have been elaborated. There is a graph G with its versions $G_1, G_2, ..., G_n$ and the corresponding SHACL shapes generated by QSE, $S_1, S_2, ..., S_n$. In the following the terms G_1, G_2, S_1, and S_2 will be used, however, they can be replaced by any version of the graph or of the respective shapes. Therefore, the research questions are not limited to two graph versions.

RQ1: What is an appropriate way to compare S_1 and S_2 in a user-friendly way and explain the differences (addition, removal, change)?

RQ2: Given S_1 and the changeset between G_1 and G_2, how can we use the changeset and S_1 to derive S_2?

RQ3: Given SPARQL endpoints hosting the graphs, how can we use SPARQL queries to derive which shapes of S_1 remain unchanged and which were removed for G_2?

Appropriate for RQ1 means user-friendly, useful, and correct. It can be measured during the evaluation of semi-structured interviews with experts. Appropriate for RQ2, and RQ3 means correct and faster in comparison to the method described in the evaluation sections.

CHAPTER 2

Related Work

This chapter delivers a comprehensive overview of knowledge graphs, RDF, SPARQL, and SHACL. Building on this foundation, a dedicated section explores evolving knowledge graphs, which is particularly important to this subject. Additionally, the chapter presents related work retrieved from a systematic literature review, along with a description of the QSE algorithm which forms the basis of this thesis.

2.1 Preliminaries

The following pages cover fundamental aspects of semantic systems. The topics covered include an overview of knowledge graphs, RDF (Resource Description Framework), the query language SPARQL, and the validation language SHACL.

2.1.1 Knowledge Graphs

The focus of this thesis is on semantic systems and knowledge graphs. Semantic systems are systems that make use of explicitly represented knowledge, through conceptual structures such as ontologies, taxonomies, or knowledge graphs [62]. Many formal definitions exist for knowledge graphs, such as the one articulated by professors at TU Wien: "A knowledge graph contains semantically related information as nodes and edges" [62] or "Knowledge Graphs are graph-structured representations intended to capture the semantics about how entities relate to each other" [83]. Knowledge graphs find application across diverse domains, including commercial and academic sectors. Moreover, they serve as the foundation for Data Science, Graph Databases, Machine Learning, Reasoners, and Data Integration/Wrangling. Noteworthy among these knowledge graphs is DBpedia [5], which contains cleaned data from Wikipedia in all languages. In 2023, DBpedia's Databus, the platform hosting its data, provided 4,100 GByte of data [6]. Additionally, other prominent knowledge graphs exist, such as Wikidata [28], compromising approximately

110 million editable records. Wikidata serves as the central repository for Wikimedia projects like Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others. It is a free and open knowledge base. Besides general information, as it is provided by DBpedia, knowledge graphs have found compelling applications in fields such as chemistry and biology. For instance, Linked Life Data [16] offers access to 25 public biomedical databases, facilitating questions like "find all human genes located in Y-chromosome with the known molecular interactions". Around 10 billion RDF statements are provided by Linked Life Data. Finally, PubChem [19], an open chemistry database, is another notable example of a large knowledge graph.

2.1.2 RDF

Knowledge graphs can be expressed as property graphs or by the Resource Description Framework (RDF) [62]. The primary distinction lies in the fact that property graphs allow edges to have attributes. However, this thesis focuses on RDF, which was developed in the 1990s and is a W3C recommendation [20]. The primary strength of RDF lies in its ability to associate metadata with resources. An RDF graph contains numerous statements, known as triples, each consisting of a subject, a predicate, and an object. An example is illustrated in Figure 2.1.

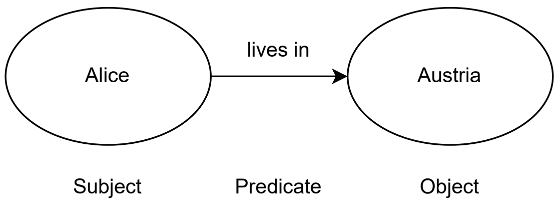


Figure 2.1: Example of RDF

An important term in knowledge graphs is the URI, which stands for Uniform Resource Identifier. In RDF, anything with a URI is considered a resource. URIs are strings that identify objects and resources typically found on the web. However, it is important to note that resources do not have to be on the web. A further important property of URIs is their uniqueness. A URI can be a Uniform Resource Name (URN), a Uniform Resource Locator (URL), or a combination of both. An Internationalized Resource Identifier (IRI) is a URI. An example of defining a resource using a URI could be "http://semantics.id/ns/example#Alice". To build RDF triples, the RDF data model was defined which includes four sets [90]:

- The set of resources $\mathcal{R}$ contains all entities for which RDF statements can be made, and these are identified by URIs.
- The set of properties $\mathcal{P}$ lists all features with values that can be attached to resources. Properties, which are also RDF resources identified by URIs, define the

relationships between subjects and objects in an RDF triple and are referred to as predicates. These properties form a subset of all resources.

- The set of literals $\mathcal{L}$ includes other values such as character sequences, integers, decimals, dates, or booleans.
- Lastly, the set of statements $E : \langle s, p, o \rangle$ contains all triples in the graph. As previously mentioned, a triple contains a subject, a predicate, and an object. The subject of an RDF statement is an RDF resource or a blank resource, the predicate is an RDF property. The object can be either an RDF resource, a literal, or a blank resource. For instance, an adapted example from the one above could have the predicate "age" and the object "25", where the object is a literal, unlike the original example where the object is a resource.

As blank resources or blank nodes were already mentioned, they exist besides RDF resources and literals. They are used to improve the information structure and are not globally unique. For example, a list cannot be modeled as an RDF triple without using blank nodes. The definition of RDF graphs can be formally articulated as:

Definition 1 *(RDF graph [84]). Given the sets of resources $\mathcal{R}$, literals $\mathcal{L}$ and blank nodes $\mathcal{B}$, an RDF graph $\mathcal{G} : \langle N, E \rangle$ is a graph with nodes $N \subset (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})$ and edges $E \subset \{\langle s, p, o \rangle \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})\}$*

A small example of a full graph can be seen in Listing 2.1. In this example, two people are defined, who both have a name (Bob and Alice) and mutually know each other. Prefixes are defined, which can be used in RDF to shorten URIs. For instance, "foaf:name" abbreviates "<http://xmlns.com/foaf/0.1/name>". A dot always indicates the end of an RDF statement.

```
 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<http://example.org/alice> a foaf:Person .
<http://example.org/bob> a foaf:Person .
<http://example.org/alice> foaf:name "Alice" .
<http://example.org/bob> foaf:name "Bob" .
<http://example.org/alice> foaf:knows
    <http://example.org/bob> .
<http://example.org/bob> foaf:knows
    <http://example.org/alice> .
```

Listing 2.1: RDF graph in Turtle syntax

Furthermore, the letter "a" indicates, that the resource "Alice" is a Person. This predicate a, which is an element of $\mathcal{P}$, abbreviates the property rdf:type [20]. It links all instances

of a class to the node representing the class. Formally, all classes, which form a subset of $\mathcal{R}$, can be defined as $C : \{c \in \mathcal{R} \mid \exists s \in \mathcal{R} \text{ so that } \langle s, a, c\rangle \in \mathcal{E}\}$.
RDF graphs can be represented in various formats, such as RDF/XML, N3/Turtle, or in N-Triples format, which is shortened as "nt" or generally as a labeled directed graph. An example of an RDF graph in N-Triples format based on Listing 2.1 in a shortened version is shown in Listing 2.2.

```
<http://example.org/alice>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
<http://xmlns.com/foaf/0.1/name> "Alice" .
<http://example.org/alice>
<http://xmlns.com/foaf/0.1/knows> <http://example.org/bob> .
```

Listing 2.2: RDF graph in N-Triples Format

Knowledge graphs can be stored by different graph stores, for instance, GraphDB [11]. GraphDB is provided by Ontotext, available as a free or commercial version. It offers a visual interface for the interaction with graphs but also supports integration with various programming languages. An alternative is OpenLink Virtuoso [17] which can also be used in Microsoft Azure or Amazon AWS. To work with knowledge graphs, there exist different frameworks, such as Apache Jena [2] or Eclipse RDF4J [9] which are both solely compatible with Java.

2.1.3 SPARQL

Knowledge graphs can be queried by the query language SPARQL [25]. SPARQL shares similarities with SQL, as it incorporates familiar keywords like "select", "where", "group by", and "order by". An example query, as shown in Listing 2.3, retrieves the names of people along with the number of their friends.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name (COUNT(?friend) AS ?count)
WHERE {
    ?person foaf:name ?name .
    ?person foaf:knows ?friend .
} GROUP BY ?person ?name
```

Listing 2.3: SPARQL example

The WHERE clause specifies a pattern to be matched in the graph. The variable "?person" represents an RDF resource in the graph and is further constrained by the statements, that the person must have a name and a friend. SPARQL offers various other functionalities, including insert, update, and delete statements, filtering, ordering, and subqueries. SPARQL will be used in this thesis to query data in triple-stores, especially in RQ3.

2.1.4 SHACL

Since data quality is an important topic in all areas, SHACL [23], which stands for Shapes Constraint Language, has evolved as a language to validate RDF graphs against a certain set of conditions. SHACL can be expressed as an RDF graph itself, it can also serve as the description of the data graph. This representation is termed a "shapes graph" S, whereas the actual dataset is called the "data graph" G. Shapes graphs contain different shapes, which describe certain classes and their properties.
A key concept within SHACL is the node shape, which is often used to describe a class in the RDF graph and may contain multiple property shapes. A property shape is usually used to describe the objects of a property of a certain class. Mathematically, this can be defined as:

Definition 2 *(Shape graph [84]). A shape graph S contains node shapes N, with $\langle s, \tau_n, \Phi_n \rangle \in N$ where s is the subject IRI of the node shape (or the name), $\tau_n \in C$ is the target class, and Φ_n are the property shapes, in the form $\phi_n : \langle \tau_p, T_p, C_p \rangle$. τ_p is the target property, defined with "sh:path" and $T_p \subset R$ contains, in case of the node kind of τ_p "sh:Literal", an IRI describing the literal type e.g. "xsd:string". In case of the node kind of τ_p "sh:IRI", a set of IRIs is provided. C_p is a pair $(n, m) \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$ with $n \leq m$, which describes the min and max cardinality constraints.*

An illustrative example is presented in Listing 2.4, where the validation of the class "ex:Person" is specified. Every person must have at least one name.

```
ex:PersonShape a sh:NodeShape ;
sh:targetClass ex:Person ;
sh:property [
    a sh:PropertyShape;
    sh:path ex:name;
    sh:minCount 1
] .
```

Listing 2.4: SHACL example

SHACL shapes can have different targets. In this example, the target is the class ex:Person, specified using the predicate "sh:targetClass". The prefix "sh" abbreviates SHACL i.e. the URI "http://www.w3.org/ns/shacl#". However, targets of shapes can be defined in different ways, not necessarily limited to classes. For instance, a shape might exclusively target several nodes, denoted by "sh:targetNode". Alternatively, it is possible to define a predicate, e.g. "ex:knows" and target subjects or objects of this predicate, where the IRIs "sh:targetSubjectsOf" and "sh:targetObjectsOf" would be used. However, in this thesis, targets of node shapes will be only identified with "sh:targetClass". The IRI

of the node shape shown in Listing 2.4 is “ex:PersonShape”, which can also be considered as the name of the node shape. Property shapes can also have IRIs that identify them; in this example, it could be “ex:namePersonShapeProperty”.
Besides various SHACL keywords used for property shapes, some important IRIs include “sh:NodeKind”. This IRI defines whether the object of the specified predicate is “sh:Literal” or “sh:IRI”, among other possible values. Another constraint is “sh:datatype” which states the data type of the target objects, such as “xsd:integer” or “rdf:langString”. The “sh:class” predicate is used to ensure that the target object is an instance of the specified class. Each property shape mandatorily requires a predicate “sh:path”, whose object specifies an IRI declaring the focus nodes of property shapes reachable from the node shape via this path. Property shapes can also include cardinality constraints like “sh:minCount” or “sh:maxCount”. Additionally, SHACL supports lists with “sh:in” and logical constraints with “sh:or” to express more complex shapes. SHACL provides many more options than those listed here, enabling a detailed description of a graph’s schema. When a data graph is validated against a shapes graph, it is verified that all target entities fulfill all property shapes in the given node shape. This can be expressed in the following way:

Definition 3 *(Validating Shapes Semantics [84]). All entities e, which are instances of τ_n, in a graph G are validated by a node shape n, if the following conditions are fulfilled for all property shapes $\phi_n : \langle \tau_p, T_p, C_p \rangle$ in n:*

- *If the node kind of ϕ_n is “sh:Literal”, then for every triple $(e, \tau_p, l) \in G$, l is a literal of type T_p*
- *If the node kind of ϕ_n is “sh:IRI”, then for every triple $(e, \tau_p, o) \in G$, o is an instance of type C_p, or of one of its subclasses*
- *$n \leq |\{(s, p, o) \in G : s = e \land p = \tau_p\}| \geq m$, where $C_p = (n, m)$*

There also exist other logic-based languages, which can describe the schema of a graph, namely the Web Ontology Language (OWL) [18], or Shape Expressions (ShEx) [24]. However, SHACL is particularly important in this thesis because QSE outputs SHACL shapes, which form the foundation for all research questions.

2.2 State of the Art

After understanding the fundamentals of semantic systems, specifically RDF and SHACL, the subsequent section dives into the central themes of this thesis: evolving knowledge graphs and the QSE algorithm. This section also covers the findings from the literature review, including topics such as data quality in knowledge graphs and other algorithms for extracting shapes from existing graphs.

2.2.1 Evolving Knowledge Graphs

As one might expect, knowledge graphs are not static; they evolve over time just like the reality they present. DBpedia [5], for instance, is updated regularly as new knowledge is generated continuously.
Time can be represented in evolving knowledge graphs in different ways [83]. It can be saved as data, for example, the construction year of a building can be recorded in the knowledge graph. Contrastingly, time can also be saved as metadata. This includes the creation time of an entry, the time an entry was deleted, and the existence of different versions of an entry over time. Consequently, when time is saved as metadata, a knowledge graph can be dynamic – providing access to all observable atomic changes over time – or versioned, offering static snapshots of the knowledge graph at specific points in time. In this thesis, only versioned knowledge graphs will be considered.
When discussing evolution, several new terms must be considered. First, structural evolution can be measured, where different descriptive statistics, such as centrality or connectedness are measured over time within a graph. Dynamics in an evolving knowledge graph can be assessed by examining growth and change frequencies, which can be interesting to compare across different areas of the graph. Timeliness refers to the freshness of the data, indicating if data is out-of-date or out-of-sync. Monotonicity describes whether data is only added to the graph or if there are also updates and deletions. Semantic drift happens when there is a change in the meaning of a concept over time.
It is also important to consider who makes changes in a knowledge graph. These changes can be made by anonymous users, registered users, authoritative users, or bots. Understanding who is making the changes is crucial, as it can impact data quality in various ways.
Often, when changes in knowledge graphs occur, they happen at the level of object literals [83]. Changes can be atomic, focusing on operations at the resource level, or they can be local or global. Changes can also be monitored at the schema level. In a study of the DyLOD dataset (Dynamic Linked Data Observatory) over one year, between 20% and 90% of schema structures changed between two versions. This indicates that while some nodes retained the same schema structure, new schema structures were added, and some were no longer used. This observation is particularly important for this thesis.
Pelgrin et al. [80] provide an example of how characteristics can be used to study the evolution of a dataset. They analyzed certain areas of DBpedia from 2010 onwards, focusing on major versions up to 2019. For instance, change ratios between these versions indicate that until 2015, the growth rate was steady, but after that, the deletion ratio exceeded the insertion ratio, signaling a major design shift. Subsequently, a negative growth rate might suggest more curated data in the later versions. Additionally, characteristics of the vocabulary dynamicity show that during the major design shift, noisy data was removed, as this parameter remained unchanged. Other notable characteristics of these DBpedia versions include entity changes, the triple-to-entity change score, and object updates.
However, the dimension of time opens new challenges in knowledge graphs. For instance,

there is a need for new data models and storage methods tailored to evolving knowledge graphs. Additionally, there will be novel approaches to query processing, reasoning, and learning that incorporate the temporal aspect in evolving knowledge graphs.
Based on this knowledge, various projects have been developed, and extensive research has been conducted in this area. For instance, EvolveKG [70] is a framework designed to reveal cross-time knowledge interactions. Another approach [91] involves creating a summary graph for different versions of objects, which refers to the dynamic option of evolving knowledge graphs, as discussed earlier. Additionally, KGdiff [67] has been developed to track changes in both the schema and the individual data points.

2.2.2 Data Quality in Knowledge Graphs

Numerous approaches have also been developed in the area of data quality which is a vital topic for knowledge graphs due to the huge amount of data involved and the schema flexibility. Data quality is related to this thesis, as the primary purpose of the generated SHACL shapes is to improve the quality of knowledge graphs by applying these shapes to the knowledge graph to filter out spurious triples. This method is one approach to increasing data quality, however, there are also other approaches. GraphGuard [47] introduces a framework aimed at improving data quality in knowledge graph pipelines for both humans and machines. Additionally, there has been a systematic review [95] of quality management in knowledge graphs, as well as a general survey [65] on knowledge graphs, including the temporal aspects of them. Another paper [63] aims to enhance existing quality assessment frameworks by incorporating additional quality dimensions and metrics.
Also in the area of data quality, a method [74] has been developed to judge whether an incoming graph change is correct or not with classifiers based on topographical features of a graph. PAC (property assertion constraints) [46] has the goal of checking data before it gets added to the knowledge graph. With this approach, errors can be prevented and the quality of knowledge graphs can be enhanced. PAC works by restricting the range of properties using SPARQL. Additionally, a dissertation [45] has focused on incorporating data transformations in knowledge graphs to clean the data and to complete the graphs by calculating derived data.

2.2.3 Quality Shapes Extraction

QSE (Quality Shapes Extraction) [84] is an algorithm designed to extract SHACL shapes from knowledge graphs to ensure data quality. It is particularly designed for very large knowledge graphs and it is implemented in Java. The source code is publicly available on GitHub [7]. A main advantage of QSE is that it eliminates the need for manual steps during shape generation as this would be unmanageable for huge datasets such as Wikidata, where there are approximately two million property shapes to identify. Furthermore, a key objective is to eliminate spurious shapes that can be generated during automatic shapes extraction. For instance, in DBpedia, some entities that represent musical bands are incorrectly assigned to the class "dbo:City". Without filtering, these

errors would be replicated in the extracted shapes.
To address this issue, QSE calculates two parameters: support and confidence. The support parameter determines the number of entities associated with a certain class for a node shape while for a property shape, it defines the cardinality of entities conforming to that class. The confidence parameter measures the ratio between the number of entities that conform to a property shape and the total number of entities that are instances of the target class. Formally, this can be described in the following way:

Definition 4 *(Support [84]). Given a node shape* $\langle s, \tau_n, \Phi_n \rangle \in N$ *and a property shape* $\phi_n : \langle \tau_p, T_p, C_p \rangle \in \Phi_n$*, the support is defined as the number of entities e which satisfy* ϕ_n

$$supp(\phi_n) = |\{e \in R,\ which\ satisfy\ \phi_n\}|$$

Definition 5 *(Confidence [84]). Given a node shape* $\langle s, \tau_n, \Phi_n \rangle \in N$ *and a property shape* $\phi_n : \langle \tau_p, T_p, C_p \rangle \in \Phi_n$*, the confidence is defined as the proportion of entities which fulfill* ϕ_n *among the entities that are instances of the target class* τ_n *of s*

$$conf(\phi_n) = \frac{supp(\phi_n)}{|\{e | \langle e, a, \tau_n \rangle \in \mathcal{E}\}|}$$

When these parameters are configured, QSE generates shapes only if they exceed the specified thresholds for support and confidence values. If these thresholds are not set, QSE generates all shapes, which are referred to as the default shapes.
QSE is available as a command-line tool, primarily designed to extract SHACL shapes from graphs provided as a file, which must be formatted in the N-Triples syntax. Moreover, QSE offers a query-based option for graphs that are stored on graph stores such as GraphDB. For this approach, QSE uses a set of SPARQL queries to extract information from the graph.
To generate shapes from a graph, it is necessary to analyze all nodes and their types in a graph, as nodes can be used as subjects or objects. Furthermore, it is necessary to count how often a property connects nodes of given types. This is done in four steps, where the QSE algorithm involves two iterations through all triples. During the initial phase (entity extraction), all instances based on their types are counted, hence only triples containing a type declaration (e.g. “rdf:type”) are considered. The algorithm stores the entity types associated with each entity and calculates the total count for each class. Subsequently, in the second run (entity constraints extraction), the algorithm gathers metadata for property shapes. Here, all triples without a type declaration are analyzed. The subject and the object types for each predicate are saved. The results, just as the previous data, are stored in maps. Following this, in the third step, support and confidence metrics are computed from the previously saved data. Based on these metrics, spurious shapes can be filtered out. Finally, in the last step, the algorithm generates SHACL shapes. The names of the shapes are declared and the information gathered so far including support and confidence values is brought to the node and the corresponding property shapes. To

output the shapes, they are stored locally in a triple store and exported as Turtle file. Additionally, Java objects representing the node shapes (NS) and property shapes (PS), as well as the constraints defined within the property shapes (referred to as ShaclOrListItem) exist, which can be used in subsequent Java programs, but are not visible to the end user. A class diagram of these Java objects is shown in Figure 2.2. These objects are created at the end of the QSE algorithm when the SHACL shapes are created. Prior to this, only maps with encoded values for class names and other elements are used. The Java shapes objects are hierarchically organized, meaning that each node shape includes a list of property shapes, and each property shape contains a list of constraints. All other details, such as the shape name and target class, are also stored within these Java objects. It is important to note, that these objects are not available in the "main" branch of QSE but are only created in the "shactor version" [8]. More details on these different implementations are provided in Section 4.2.

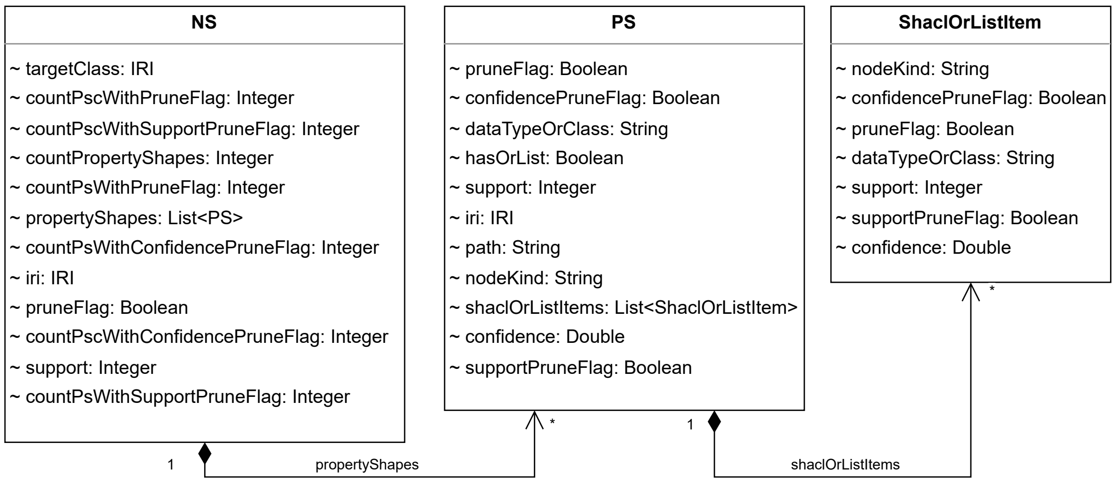

NS
~ targetClass: IRI
~ countPscWithPruneFlag: Integer
~ countPscWithSupportPruneFlag: Integer
~ countPropertyShapes: Integer
~ countPsWithPruneFlag: Integer
~ propertyShapes: List<PS>
~ countPsWithConfidencePruneFlag: Integer
~ iri: IRI
~ pruneFlag: Boolean
~ countPscWithConfidencePruneFlag: Integer
~ support: Integer
~ countPsWithSupportPruneFlag: Integer
PS
~ pruneFlag: Boolean
~ confidencePruneFlag: Boolean
~ dataTypeOrClass: String
~ hasOrList: Boolean
~ support: Integer
~ iri: IRI
~ path: String
~ nodeKind: String
~ shaclOrListItems: List<ShaclOrListItem>
~ confidence: Double
~ supportPruneFlag: Boolean
ShaclOrListItem
~ nodeKind: String
~ confidencePruneFlag: Boolean
~ pruneFlag: Boolean
~ dataTypeOrClass: String
~ support: Integer
~ supportPruneFlag: Boolean
~ confidence: Double
1
propertyShapes
*
1
shaclOrListItems
*

Figure 2.2: Class diagram of the Java objects created by QSE

Due to the potential size of knowledge graphs, QSE is offered in two versions: the exact version, where the steps are described in detail above, and the approximate version. The exact version keeps type and property information in memory which can lead to a high memory consumption. QSE-Approximate was developed to enable shapes extraction on commodity machines with limited memory. This algorithm is based on a multi-tiered dynamic reservoir sampling algorithm which replaces the first phase of QSE-Exact where all triples are read.
Another configuration possibility for QSE is to list the names of the classes C on which QSE should be executed. This feature allows the extraction of shapes for specific classes only, rather than running QSE on the entire graph.

QSE Explained by an Example

To better grasp the theory behind QSE, a simple example with Alice and Bob is provided. A snippet from this knowledge graph is presented in Listing 2.2, however, the complete RDF graph is available in Listing B.1. A third person, Jenny, is added to the graph. After executing QSE-Exact without parameters for support and confidence, the full output is shown in Listing B.4. Listing 2.5 shows the node shape generated for the class "Person" in Turtle syntax. This can be seen in the last triple of the snippet, by the property "targetClass". The node shape has a support of three because there are three people in the knowledge graph. Furthermore, there are three property shapes in the node shape, the first one describing the "rdf:type" property for the instances of the "Person" class, and the second one is about the "knows" property, where each person knows another person. Lastly, the property shape described in Listing 2.6, is about the "name" property. It shows that the property is set for Alice, Bob, and Jenny and therefore the confidence is 100%, which is written as "1E0". The object of these triples must be a literal of the type "xsd:string" and because the property is set for all instances, "minCount=1" is added to the shapes, marking it as obligatory. The "path" describes the property, namely "<http://xmlns.com/foaf/0.1/name>".

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://shaclshapes.org/PersonShape> rdf:type
    <http://www.w3.org/ns/shacl#NodeShape> ;
  <http://shaclshapes.org/support> "3"^^xsd:int ;
  <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/instanceTypePersonShapeProperty> ;
  <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/knowsPersonShapeProperty> ;
  <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/namePersonShapeProperty> ;
  <http://www.w3.org/ns/shacl#targetClass>
    <http://xmlns.com/foaf/0.1/Person> .
```

Listing 2.5: Result of QSE-Exact on the People Knowledge Graph, node shape for the class "Person"

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/namePersonShapeProperty> rdf:type
    <http://www.w3.org/ns/shacl#PropertyShape> ;
  <http://shaclshapes.org/confidence> 1E0 ;
  <http://shaclshapes.org/support> "3"^^xsd:int ;
  <http://www.w3.org/ns/shacl#NodeKind>
    <http://www.w3.org/ns/shacl#Literal> ;
```

```
<http://www.w3.org/ns/shacl#datatype> xsd:string ;
<http://www.w3.org/ns/shacl#minCount> 1 ;
<http://www.w3.org/ns/shacl#path>
  <http://xmlns.com/foaf/0.1/name> .
```

Listing 2.6: Result of QSE-Exact on the People Knowledge Graph, property shape for the property “knows” from the class “Person”

In addition to the Turtle file generated, QSE creates Java objects that represent the node and property shapes. These generated objects provide the same information as the SHACL shapes in the Turtle file. The resulting objects for this example are illustrated in Figure 2.3.
During the calculation, different maps are generated. Instances for these maps are illustrated in Figure 2.4. First, a translation table is created and stored in the “stringEncoder”, which assigns a unique number to each IRI to reduce memory consumption. Next, the parser - a Java class responsible for the triple parsing in QSE - creates the “classEntityCount” map, which records the number of instances for each class. In this case, there are three people in the graph, therefore the key representing the class “Person” (encoded as 0) has a value of three. The “entityDataHashMap”, which stores the types for each resource along with their associated properties, is generated. For instance, in the case of Alice, who is representative of all people in the graph, her “classType” includes only the class "Person" (encoded as 0). Alice is associated with three properties: type, name, and knows, encoded as 1, 3, and 5 respectively. The “objTypes” map further specifies the expected class for the object of each property: the type property points to “undefined” (encoded as 2), the name property to a string (encoded as 4), and the knows

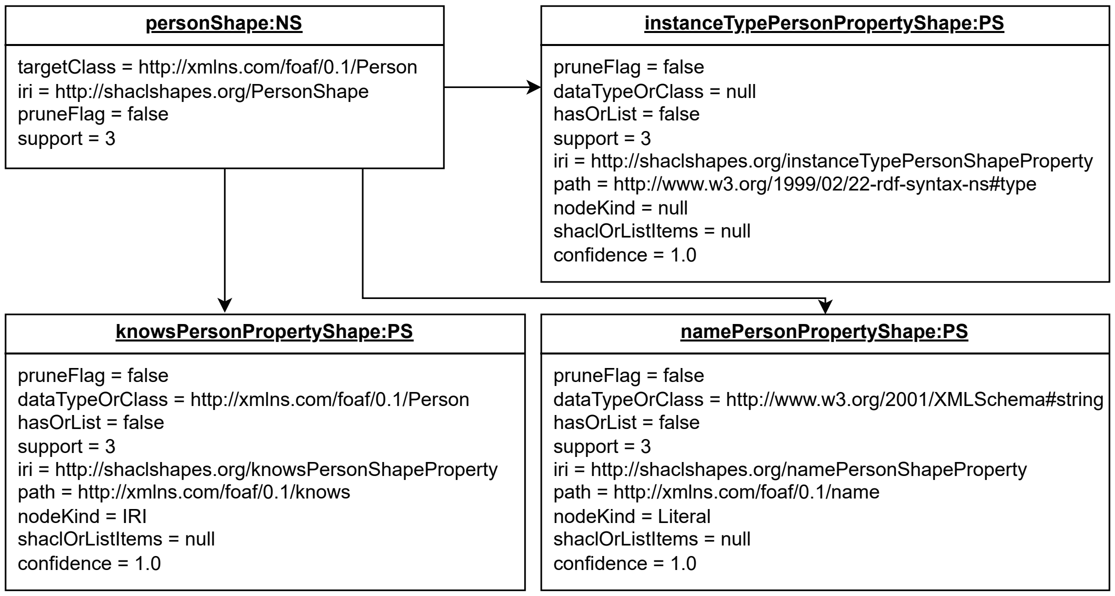

personShape:NS
targetClass = http://xmlns.com/foaf/0.1/Person
iri = http://shaclshapes.org/PersonShape
pruneFlag = false
support = 3
instanceTypePersonPropertyShape:PS
pruneFlag = false
dataTypeOrClass = null
hasOrList = false
support = 3
iri = http://shaclshapes.org/instanceTypePersonShapeProperty
path = http://www.w3.org/1999/02/22-rdf-syntax-ns#type
nodeKind = null
shaclOrListItems = null
confidence = 1.0
knowsPersonPropertyShape:PS
pruneFlag = false
dataTypeOrClass = http://xmlns.com/foaf/0.1/Person
hasOrList = false
support = 3
iri = http://shaclshapes.org/knowsPersonShapeProperty
path = http://xmlns.com/foaf/0.1/knows
nodeKind = IRI
shaclOrListItems = null
confidence = 1.0
namePersonPropertyShape:PS
pruneFlag = false
dataTypeOrClass = http://www.w3.org/2001/XMLSchema#string
hasOrList = false
support = 3
iri = http://shaclshapes.org/namePersonShapeProperty
path = http://xmlns.com/foaf/0.1/name
nodeKind = Literal
shaclOrListItems = null
confidence = 1.0

Figure 2.3: Object diagram of shapes created by QSE

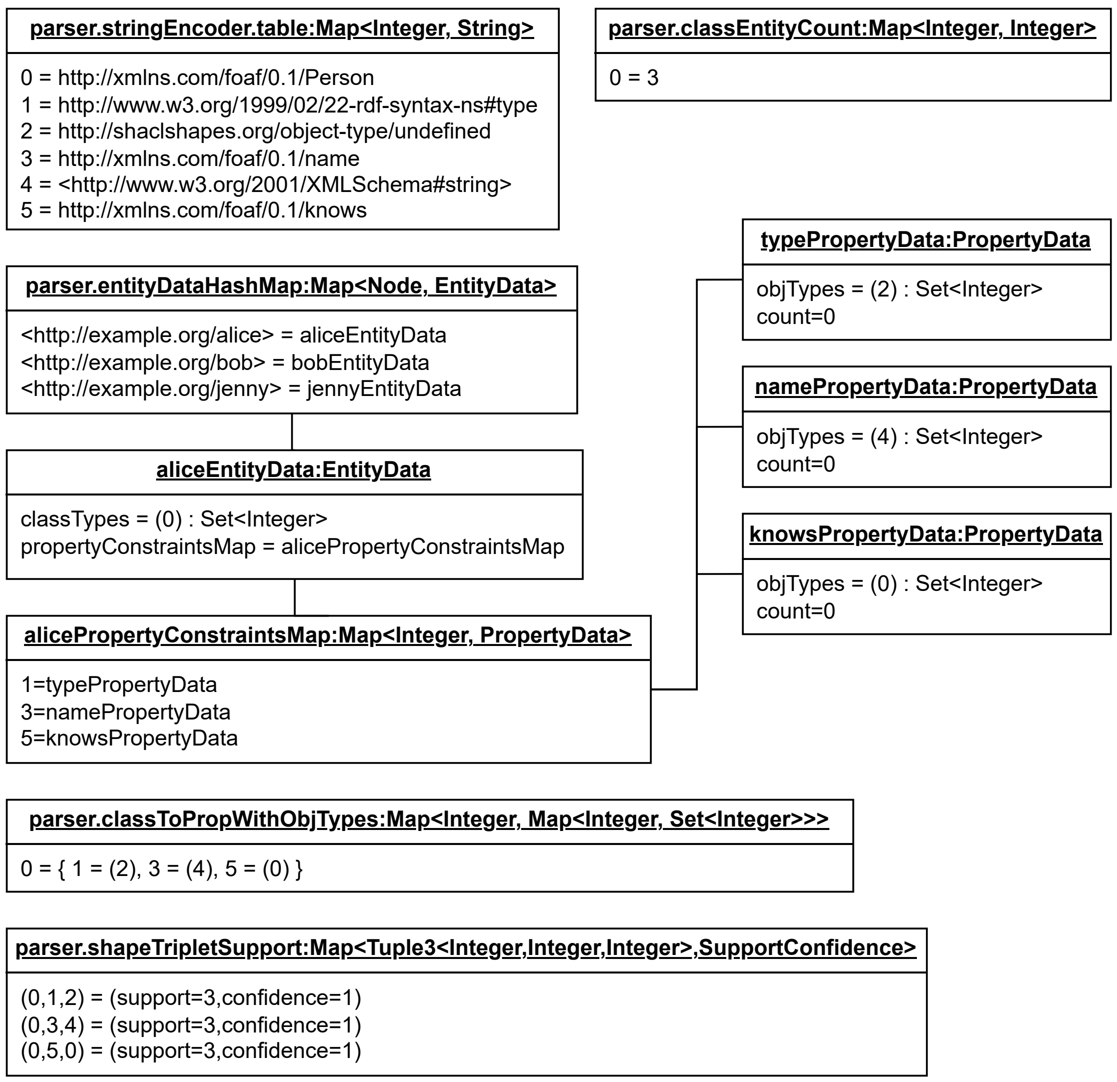
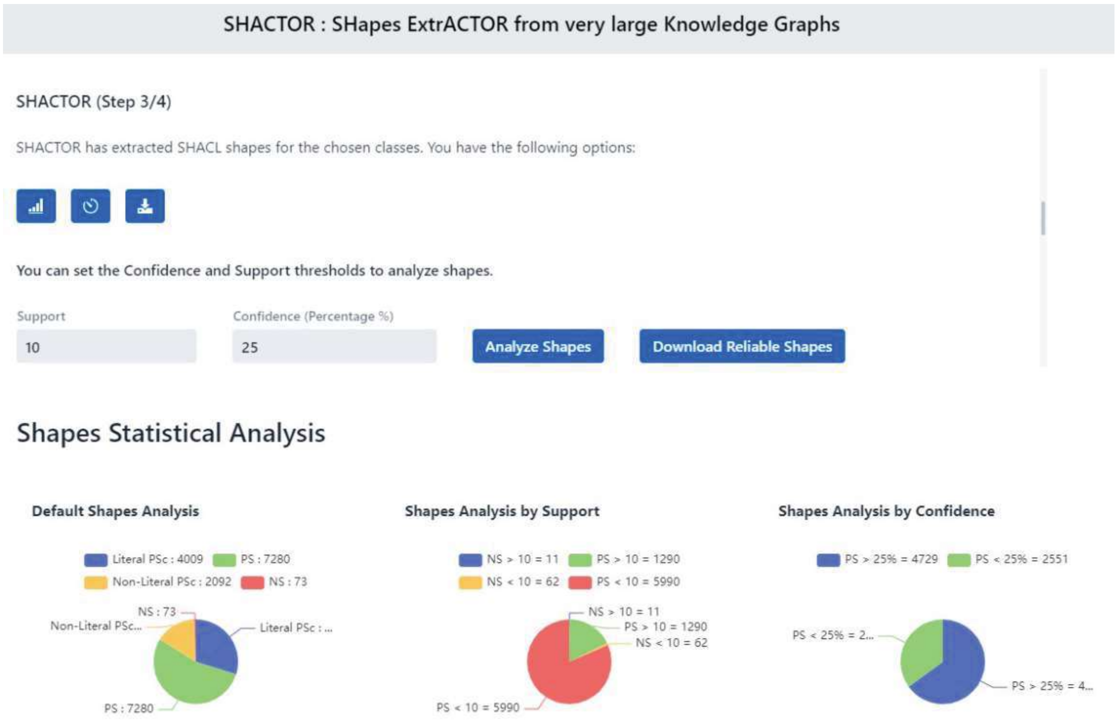

parser.stringEncoder.table:Map<Integer, String>
0 = http://xmlns.com/foaf/0.1/Person
1 = http://www.w3.org/1999/02/22-rdf-syntax-ns#type
2 = http://shaclshapes.org/object-type/undefined
3 = http://xmlns.com/foaf/0.1/name
4 = <http://www.w3.org/2001/XMLSchema#string>
5 = http://xmlns.com/foaf/0.1/knows
parser.classEntityCount:Map<Integer, Integer>
0 = 3
parser.entityDataHashMap:Map<Node, EntityData>
<http://example.org/alice> = aliceEntityData
<http://example.org/bob> = bobEntityData
<http://example.org/jenny> = jennyEntityData
aliceEntityData:EntityData
classTypes = (0) : Set<Integer>
propertyConstraintsMap = alicePropertyConstraintsMap
alicePropertyConstraintsMap:Map<Integer, PropertyData>
1=typePropertyData
3=namePropertyData
5=knowsPropertyData
typePropertyData:PropertyData
objTypes = (2) : Set<Integer>
count=0
namePropertyData:PropertyData
objTypes = (4) : Set<Integer>
count=0
knowsPropertyData:PropertyData
objTypes = (0) : Set<Integer>
count=0
parser.classToPropWithObjTypes:Map<Integer, Map<Integer, Set<Integer>>>
0 = { 1 = (2), 3 = (4), 5 = (0) }
parser.shapeTripletSupport:Map<Tuple3<Integer,Integer,Integer>,SupportConfidence>
(0,1,2) = (support=3,confidence=1)
(0,3,4) = (support=3,confidence=1)
(0,5,0) = (support=3,confidence=1)

Figure 2.4: Object diagram of the maps created by QSE during shapes calculation

property to a Person (encoded as 0). Another key map, "classToPropWithObjTypes", is the foundation for generating SHACL shapes later on. This nested map summarizes data from the "entityDataHashMap" across all entities. The first key in this map represents the node shape, the key in the second level corresponds to the property shapes, and the third level identifies the target classes for each property shape. Additionally, the "shapeTripletSupport" map is crucial as it stores the support and confidence values for all shapes. In this view, the information from the "classToPropWithObjTypes" map is presented in a flattened form.

Shactor

The visual extension to QSE is called Shactor [85]. It is a web-based tool that visualizes the extraction process and provides valuable statistics. Shactor was developed by using Vaadin [27] and supports only the file-based version of QSE. An example screenshot after the extraction process is shown in Figure 2.5


SHACTOR : SHapes ExtrACTOR from very large Knowledge Graphs

SHACTOR (Step 3/4)

SHACTOR has extracted SHACL shapes for the chosen classes. You have the following options:

You can set the Confidence and Support thresholds to analyze shapes.

Support: 10

Confidence (Percentage %): 25

Shapes Statistical Analysis

Figure 2.5: Statistics provided by Shactor after analyzing version one of the BEAR-B dataset

Limitations in the QSE algorithm

QSE is a powerful tool with many great features, but there are some important limitations to be aware of. One significant issue arises when resources share the same name but have different IRIs, such as http://dbpedia.org/property/name versus http://dbpedia.org/ontology/name. In such cases, QSE should generate two distinct shapes, but instead, only one shape is created. This occurs because QSE only takes the name of the resource to generate a shape name. It leads to various confusing problems such as multiple support statements, multiple confidence statements, multiple classes in a node shape, or even multiple or-items lists in a single property shape which are meant for different shapes. This behavior also affects the Java objects. Multiple property shapes are generated if they share the same name. Since each node shape can only have one target class, this similarly leads to the creation of multiple Java node shapes with the same name. However,

when the SHACL shapes are eventually written to a file, they are merged back into a single shape. This inconsistency complicates handling such cases, as the information represented in the Java objects differs from what is outputted in the Turtle file. The behavior was not intentionally designed, there should be a mechanism to prevent the issue. Additionally, when a node shape contains multiple property shapes with the same name, the suffix "_1" is added to one of the shapes, which can further contribute to confusion.
A simple solution to this problem would be to incorporate the full or more parts of the IRI in the SHACL shape name. This approach would significantly reduce the likelihood of a shape targeting multiple IRIs which would resolve all of the above-mentioned problems. However, a drawback of this solution is that the longer names would make the shape names less readable for end users.
Besides this problem, there are some minor limitations. QSE is unable to handle comments, so they must be removed from the graph file before QSE can parse it. There are also inconsistencies between the query-based and file-based versions of QSE. For the file-based version, the "instanceType" property shapes are generated while in the query-based option, they are filtered out. This kind of property shape is generated when a triple describes the class of a resource, i.e. the predicate of the triple equals "rdf:type". Furthermore, there are some encoding issues for literal objects of a certain type. For instance, the triple "<http://dbpedia.org/resource/2015_US_Open_(tennis)> <http://dbpedia.org/ontology/budget> "4.22534E7"^^ <http://dbpedia.org/datatype/usDollar>" results in a property shape with the class $< \%3Chttp://dbpedia.org/datatype/usDollar \%3E >$, where the angle brackets could not be resolved correctly. Another minor issue arises with the pruning mechanism that filters out all shapes with values less than or equal to the threshold. This behavior might be unintuitive for some users as they might expect the pruning to exclude only shapes with values strictly less than the threshold.

Other Shapes Extraction Algorithms

Many algorithms follow the idea of extracting SHACL shapes or other schema information from existing data sources to improve data quality and eliminate the need for manual schema creation. Examples include SHACLGEN [22] or ShapeDesigner [38], which follow a similar approach as QSE-Exact but lack the ability to process large knowledge graphs [84]. Additional examples include Shape Induction [73] and the approach described by Spahiu et al. [89] which both extract SHACL shapes from existing data. SCOOP [48] similarly extracts SHACL shapes from existing knowledge graphs but uses ontologies and data schemas primarily rather than individual entities. IOP [75, 76] is a predicate logic formalism that identifies specific shapes over connected entities in the knowledge graph, with its corresponding learning method known as SHACLearner. SheXer [52], using a Python library, produces SHACL shapes similar to QSE. A logic called ABSTAT [89] which automatically extracts SHACL shapes from knowledge graphs has already been considered in QSE. Another approach, the Ontology Design Pattern [77], uses ontologies to automatically generate SHACL shapes. Astrea [43] employs Astrea-KG, which provides mappings between ontology constraint patterns and SHACL constraint patterns. While

not generating SHACL shapes itself, Trav-SHACL [54] plans the execution and traversal of a shapes graph to detect invalid entries early.
Also not directly related to shapes extraction, visualizing SHACL shapes has also been explored. A master's thesis [33] was written that generates a 3D model of interconnected SHACL shapes.

CHAPTER 3

Methods

This chapter describes the scientific methods that are used in this work. First, all methods are described individually, and then the process of answering the research questions using the mentioned methods is explained.

3.1 Research Methods

This section explores the scientific methods used in this thesis, namely the Design Science Research framework, systematic literature review, prototyping, semi-structured expert interviews, qualitative content analysis, algorithmic design, and technical experiments.

3.1.1 Design Science Research

Design Science [61], rooted in engineering, focuses on problem-solving through the creation of useful artifacts. In Design Science Research design is both a process and a product and the evaluation of both is crucial. This leads to a continuous cycle where artifacts undergo regular evaluation and subsequent refinement, often referred to as the design cycle [60].
Two important terms in design science are relevance and rigor. The artifact's relevance to its environment (people, organizations, technology) is crucial. This relationship forms a cycle: the artifact's development is shaped by the environmental needs and, in turn, influences that environment through its application, termed the relevance cycle. Simultaneously, ensuring rigorous development involves addressing existing knowledge. Here, the knowledge base influences the design cycle, while the finished product contributes back to the knowledge base in what's known as the rigor cycle.
The heart of Design Science is the iterative design cycle, with relevance and rigor cycle as its inputs. These cycles depend on each other, although during the actual research execution, the design cycle remains relatively independent.
Hevner proposed seven guidelines for the design science framework:

1. Design as Artifact: an artifact (construct, model, method, instantiation) must be produced
2. Problem Relevance: ensuring the solution is relevant to business problems
3. Design Evaluation: evaluation methods must be executed to ensure the utility, quality, and efficacy
4. Research Contributions: contributions in the area of design artifact, design foundations, and/or design methodologies must be provided
5. Research Rigor: employing rigorous methods during construction and evaluation
6. Design as a Search Process: searching for an artifact requires available means to reach desired ends while satisfying laws
7. Communication of Research: the results must be presented to a technology-oriented and management-oriented audience

An especially important topic within the design cycle revolves around evaluation [79]. This involves establishing specific criteria, such as functionality, performance, or usability as benchmarks for assessment. The study has evaluated several artifact types (e.g. algorithm, instantiation, construct) and evaluation method types (e.g. expert evaluation, technical experiment, prototype). The choice of evaluation method depends on the type of artifact; technical experiments are predominantly used for algorithms, instantiations, methods, and models.

3.1.2 Systematic Literature Review

Both rigor and relevance cycle require an understanding of the current state of the art. Particularly, the relevance cycle relies on identifying existing research gaps, a task that will be addressed through a partial systematic literature review (SLR) [68]. Typically, a systematic literature review includes the following steps:

1. Planning: This phase begins by identifying the need for an SLR, (optionally) commissioning the SLR, and defining research questions. After that, the review protocol must be developed and evaluated, which includes search terms, resources where to search, study selection criteria, and a quality checklist. A pilot phase helps refine these elements.
2. Conducting: Firstly, the research must be identified and primary studies should be selected. After that, the study quality must be assessed, followed by data extraction, monitoring, and synthesis.
   It is also important to consider synonyms, abbreviations, and combinations in the search terms. Documentation throughout ensures transparency and replicability. After all primary papers have been obtained, the papers must be filtered. Instead

of reading all the papers, some papers can be dropped based on the titles and then on the abstracts (study selection criteria). Then the remaining papers can be read and further reduced. In the end, data needs to be extracted with a pre-defined form, if possible, by multiple reviewers. During data synthesis, data is summarized. This can be done descriptively, qualitatively, or quantitatively.

3. Reporting: The last step involves specifying the dissemination mechanism (how and where the results should be published, e.g. in a magazine, journal articles, or on a website), formatting, and evaluating the report. During the evaluation, peer reviewing can be used e.g. for journal articles. Quality checklists and other prepared documents from the planning phase can be utilized here.

3.1.3 Prototyping

Within the design cycle of Design Science Research, an artifact is developed. Prototyping [37] has parallels to software engineering, which often uses agile development cycles with frequent feedback. The primary aim is to construct the Minimum Viable Product before completing the development process, aligning with the Plan-Do-Check-Act methodology. Camburn et al. [39] describe prototypes as a pre-production representation of some aspect of the final design. Prototypes can be used for refinement, communication, exploration of new concepts, and learning. Noteworthy guidelines include testing, timing, ideation, fixation, feedback, usability, and fidelity. Various prototyping techniques exist, such as iterative, parallel, requirement relaxation, or subsystem isolation, each offering distinct advantages for particular objectives. Despite the diverse techniques, detailed guidance on prototyping remains limited, except for the iterative aspect.

3.1.4 Semi-Structured Expert Interviews

Semi-structured interviews (SSI) [32] are a qualitative evaluation method within the design cycle, allowing feedback integration into the construct phase. Unlike traditional surveys, SSI engage fewer participants, employing open questions that focus on "how" and "why", and the structure is not fixed. The disadvantages of SSI are that they are time-consuming, labor-intensive, and require interviewer sophistication. SSI prove beneficial when seeking open responses and individual perspectives, suitable for groups like program recipients or interested stakeholders.
The methodological steps include selecting respondents and arranging interviews. The target group should be identified, and researchers should choose a manageable random sample. A brief introduction letter detailing the interview's time frame is crucial before sending invitations. This should be piloted before sending. Next, the questions should be drafted and an interview guide should be created. There should not be too many issues on the agenda. Closed questions can be an ideal start, open questions can follow these questions. Attention to translations and avoiding questions that evoke pressure to give socially acceptable answers is essential. Flexibility during the interview allows agenda adaptation, keeping easy questions at the start, challenging ones at the end, and

demographic inquiries at the interview's conclusion.
During the interview, the interviewer should establish a positive first impression and ask for permission to record the interview. This allows the interviewer to participate more actively. Preparation ensures clear prioritization of questions. Interviewers should maintain calm, listen actively, and seek clarification when necessary.
The analysis involves visualizing fixed choice answers in tables/graphs and employing qualitative content analysis to categorize open ones.

3.1.5 Qualitative Content Analysis

Qualitative content analysis [72] is an approach of systematic, rule-guided qualitative text analysis that can be used in the design cycle of DSR. It is used for fitting text material into a model of communication. The aspects of text interpretation will be put into categories, that follow the research question. There are two ways for category development, namely inductive and deductive category development. Inductive category development forms the categories out of the text, whereas deductive formulates the categories on theory.
The detailed steps of inductive category development involve, based on the research question, the determination of the category definition. Then, categories are formulated based on the material. After 10-50% of the material, the categories are revised and the process starts again. After working through all the material, everything should be checked for reliability and the results can be interpreted, which can trigger another iteration.

3.1.6 Algorithmic Design

During the design cycle, another method applicable is Algorithmic Design [88], also known as Algorithm Engineering, a scientific approach to developing efficient algorithms. The method involves iterative steps: design, analysis, implementation, and experiments. A key focus lies in crafting algorithms that are simple, implementable, and reusable, with performance being a crucial factor.
It is crucial in the analysis phase that the design algorithm is easy to analyze, to close the gap between theory and practice. Implementing the algorithm demands attention to nuances across programming languages and hardware, as minor details can yield significant differences. The last step is experiments, which can influence the analysis again. Many experiments can be done with relatively little effort. However, they also require nontrivial planning, evaluation, archiving, and interpretation of the results. The starting point should be a falsifiable hypothesis, which can then be used for the next iteration of the cycle.
Realistic models serve as inputs during the design phase, representing abstractions of application problems. During the experimentation phase, real inputs are required. Most of these steps are closely related to applications.

3.1.7 Technical Experiments

Design of Experiments [35] is a statistical tool that can be used for planning and conducting experiments. A practical methodology can be split into the planning, designing, conducting, and analyzing phases [49]. The planning phase requires a clear, objective, specific, and measurable statement of the problem. A meaningful response characteristic should be carefully chosen, there can also be multiple. Furthermore, process variables and design parameters must be selected, mostly they are set from knowledge of historical data. The next steps are classification and determining the level of process variables, the interaction between all variables must also be known.
The design phase is highly individualized, depending on many factors. Key principles to consider here include randomization, replication, and blocking [34].
In the conducting phase, the experiment is carried out and the results are evaluated. It is important to document everything during the experiment, even unusual occurrences.
The analysis phase involves interpreting results and drawing conclusions based on the collected data.

3.2 Methodological Steps

To answer the research questions mentioned in Section 1.3, the framework of DSR is used. As previously discussed, Design Science Research uses three cycles. In this thesis, a Systematic Literature Review serves as the methodology for the relevance and rigor cycle. Therefore, needs can be derived from the literature, and knowledge from the rigor cycle can influence the thesis. Ultimately, this process contributes to the knowledge base and aligns with evolving business needs by the cycle's conclusion. Prototyping and algorithmic design are used in the design cycle. Methods during the evaluation are semi-structured expert interviews along with qualitative content analysis and technical experiments.

3.2.1 Systematic Literature Review

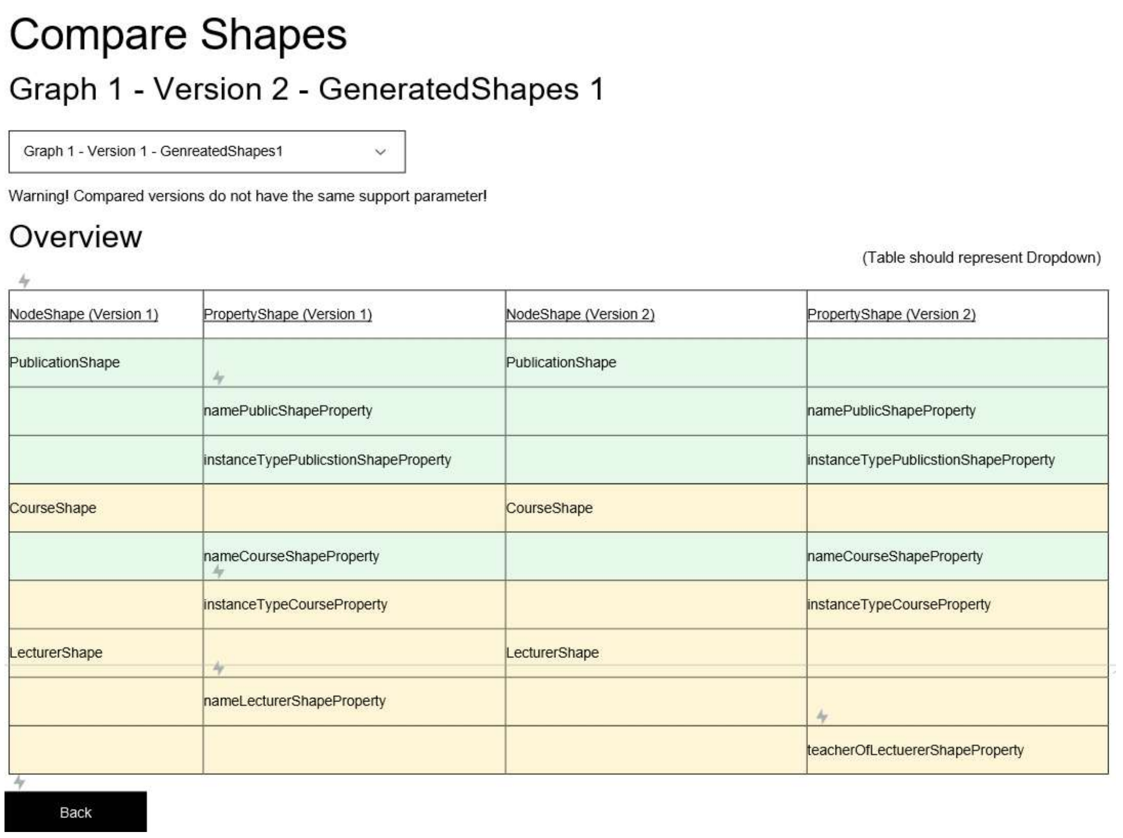
Given the time-consuming nature of a Systematic Literature Review, a partial review was chosen. During the planning phase, a review protocol was established and only one online library was defined, where strict selection criteria were applied, to limit the number of papers. In the conducting phase, qualitative data extraction was done, aiming to provide a broad overview of the knowledge base and business needs rather than addressing a specific research question. Reporting was done in a simplified way. The literature review was conducted in the mid of May 2024.
The review protocol contained the following items:

- Objectives: Establish a broad understanding of the current knowledge base in the area of evolving knowledge graphs, automatic extraction of data quality constraints from knowledge graphs, and comparing of SHACL shapes and changesets between knowledge graph versions.

- Search Keywords: “evolving knowledge graphs”, “data quality in knowledge graphs”, “shacl extraction from knowledge graphs”, “comparing shacl shapes”, “differences between knowledge graphs versions”.
- Study selection criteria: published after 2000, written in English or German, accessible with TU-Vienna Account for free.
- Library: Google Scholar.
- Study Exclusion Criteria: Has nothing to do with research questions or objectives.
- Procedure: In the conduct phase, the first ten results for each search keyword were evaluated. The outcomes of this phase are detailed in Table A.1. After reviewing the titles and applying the exclusion criteria, the abstracts of the papers were read and each paper was rated for relevance on a scale from 1 (very relevant) to 3 (not relevant). The results of this phase are presented in Table A.2. Papers with a high relevance (rated 1) were further assessed by examining parts of the paper or the entire paper.
- Data Extraction: Based on their abstracts and in some cases the full paper, key sections of the content have been summarized.
- Data Synthesis: The content of the papers was descriptively synthesized.
- Data Reporting: The results of the Systematic Literature Review are presented in Section 2.2.

3.2.2 Prototyping

During the design cycle, RQ1 was answered using the prototyping method [37] within the construction phase. JustInMind [10], a design and prototyping tool for web and mobile apps, was chosen for this purpose. The free version was adequate for this task, as the offered features were sufficient. The initial stages involved low-fidelity prototypes to determine the most suitable design that satisfies the end user’s requirements. Although only one prototype was developed, this one was already pretty similar to the final web app. Examples of this prototype are showcased in Figure 3.1 and Figure 3.2. Iteratively, the minimum viable product was developed, incorporating feedback from the evaluation phase.


Compare Shapes

Graph 1 - Version 2 - GeneratedShapes 1

Graph 1 - Version 1 - GenreatedShapes1

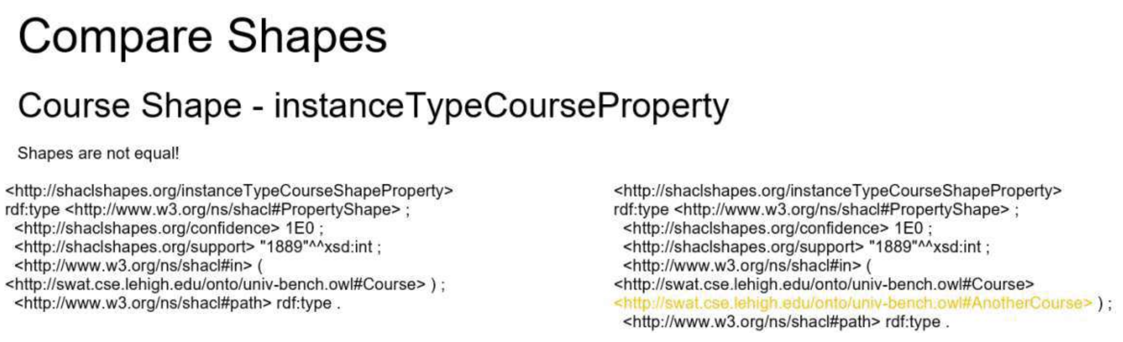
Warning! Compared versions do not have the same support parameter!

Overview

(Table should represent Dropdown)

NodeShape (Version 1)	PropertyShape (Version 1)	NodeShape (Version 2)	PropertyShape (Version 2)
PublicationShape		PublicationShape	
	namePublicShapeProperty		namePublicShapeProperty
	instanceTypePublicstionShapeProperty		instanceTypePublicstionShapeProperty
CourseShape		CourseShape	
	nameCourseShapeProperty		nameCourseShapeProperty
	instanceTypeCourseProperty		instanceTypeCourseProperty
LecturerShape		LecturerShape	
	nameLecturerShapeProperty		
			teacherOfLectuererShapeProperty

Figure 3.1: Prototype of a comparison


Compare Shapes

Course Shape - instanceTypeCourseProperty

Shapes are not equal!

```
<http://shaclshapes.org/instanceTypeCourseShapeProperty>
rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
  <http://shaclshapes.org/confidence> 1E0 ;
  <http://shaclshapes.org/support> "1889"^^xsd:int ;
  <http://www.w3.org/ns/shacl#in> (
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#Course> ) ;
  <http://www.w3.org/ns/shacl#path> rdf:type .
```

```
<http://shaclshapes.org/instanceTypeCourseShapeProperty>
rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
  <http://shaclshapes.org/confidence> 1E0 ;
  <http://shaclshapes.org/support> "1889"^^xsd:int ;
  <http://www.w3.org/ns/shacl#in> (
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#Course>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#AnotherCourse> ) ;
  <http://www.w3.org/ns/shacl#path> rdf:type .
```

Figure 3.2: Prototype of comparing a shape in detail

3.2.3 Algorithmic Design

Contrastingly, RQ2, and RQ3 used algorithmic design [88] in the construction phase of the design cycle. The approach entails maintaining algorithm simplicity, reusability, and utilizing libraries. Iterative development through small experiments refined the algorithms. Unit tests with synthesized data and small knowledge graphs were used in this process. As realistic input, graph snapshots from the BEAR datasets [4] were used to test the applications. The BEAR, **BE**nchmark of RDF **AR**chives, datasets are specifically designed for testing evolving semantic web data. They contain three real-world datasets and provide a standard for benchmarking RDF archives. The BEAR datasets will be described in more detail in Section 6.4.1.

3.2.4 Evaluation

For the evaluation part of the design cycle for RQ1, semi-structured expert interviews [32] with students, who have already participated in the course "Introduction To Semantic Systems" [1] were conducted. The analysis of interview content employed the inductive approach of qualitative content analysis [72]. The important part here was to measure the usability of the user interface in comparison to finding differences across the shapes without the ShapeComparator. The detailed procedure and the results are provided in Chapter 5.3.
For the evaluation phase in the design cycle for RQ2, and RQ3, technical experiments [35] were conducted, aligning with common practices in Design Science Research projects for instantiations [49]. The experiments ran on a virtual machine since the data size of the test data was big. After it was ensured that the algorithm worked correctly, the execution time became the measurable metric. The execution times of the developed methods were compared to the execution times of the baseline. More details on the baseline and on the definition of the execution times are provided in Chapter 6.4.2 and 7.4.

CHAPTER 4

Shared Components

This chapter provides an overview of key components essential to all algorithms of this diploma thesis. It starts with the basic concept of comparing SHACL shapes. Next, the chapter covers the technical details of the implementation as well as a description of the contributions made to the QSE algorithm, which forms the basis of all implementations.

4.1 Comparing SHACL Shapes

A key task for this thesis is to enable the comparison of generated SHACL shapes. The information sources for this comparison include the SHACL file generated by QSE and the internal Java objects after the QSE execution. Both of these information sources contain all node and property shapes. QSE structures SHACL files always in the same way using the TurtleFormatter [3]. This tool offers various configurations for structuring Turtle files and the resulting SHACL shapes are mostly consistent, with a few exceptions. However, comparing entire SHACL shape files directly is impractical, and also the Java objects are not easily comparable or interpretable by users.
A primary challenge is to present a single SHACL shape in a readable format that is user-friendly and also easily comparable by algorithms. To achieve this goal, it was decided to present the SHACL shapes as Turtle text snippets with some modifications, which will be described later in this section. This format is both readable and familiar to users, and easy to compare for algorithms. An example is given in Listing 4.1, which represents the raw SHACL snippet that can be found among many other shapes in the SHACL file generated by QSE, and Listing 4.2 which contains the modified Turtle text snippet for a shape.

```
<http://shaclshapes.org/agePersonShapeProperty> rdf:type
  <http://www.w3.org/ns/shacl#PropertyShape> ;
  <http://www.w3.org/ns/shacl#or> ( [
    <http://shaclshapes.org/confidence> 3,3333E-2 ;
```

```
      <http://shaclshapes.org/support> "2"^^xsd:int ;
      <http://www.w3.org/ns/shacl#NodeKind>
        <http://www.w3.org/ns/shacl#Literal> ;
      <http://www.w3.org/ns/shacl#datatype> xsd:integer ;
    ] [
      <http://shaclshapes.org/confidence> 3,3333E-2 ;
      <http://shaclshapes.org/support> "2"^^xsd:int ;
      <http://www.w3.org/ns/shacl#NodeKind>
        <http://www.w3.org/ns/shacl#Literal> ;
      <http://www.w3.org/ns/shacl#datatype> xsd:double ;
    ] ) ;
    <http://www.w3.org/ns/shacl#path>
      <http://example.org/age> .
```

Listing 4.1: Example output from QSE for a SHACL shape

```
<http://shaclshapes.org/agePersonShapeProperty> rdf:type
  <http://www.w3.org/ns/shacl#PropertyShape> ;
    <http://www.w3.org/ns/shacl#or> ( [
        <http://www.w3.org/ns/shacl#NodeKind>
          <http://www.w3.org/ns/shacl#Literal> ;
          <http://www.w3.org/ns/shacl#datatype> xsd:double ;
    ] [
        <http://www.w3.org/ns/shacl#NodeKind>
          <http://www.w3.org/ns/shacl#Literal> ;
        <http://www.w3.org/ns/shacl#datatype> xsd:integer ;
    ] ) ;
<http://www.w3.org/ns/shacl#path>
  <http://example.org/age> .
```

Listing 4.2: Modified Turtle text snippet for a SHACL shape

Several approaches were considered on how to use the information provided by QSE to get these comparable SHACL shape text snippets. In the final version, a combination of the two artifacts was used. The Java objects served as the primary source of information, providing an overview of all existing shapes, the attributes stored within each shape, and how they relate to each other. The SHACL file was used to obtain the Turtle text snippets, which are used for shape comparison.
A crucial decision in this context is the assumption that the most effective method for identifying a shape is by using its name. The name of a shape is not defined by the SHACL standard, it is up to the user to give a shape a meaningful name. However, QSE structures the shape names during shape generation always in the same way. Node shapes are named after their target class and the suffix “Shape”. The name of a property shape begins with the path of the property, then uses the name of the corresponding node shape, and ends with the suffix “Property”. Examples are the node shape “PersonShape”

and its corresponding property shape “agePersonShapeProperty” for the property age. After some experiments, it became clear that the fastest way to get a single SHACL shape as text by name is to use regular expressions (regex) on the SHACL file and obtain the SHACL text snippet of a shape in this way. With regular expressions, texts can be efficiently searched by using keywords, wildcards, and other rules. Since the SHACL file is formatted with the TurtleFormatter, the regex is relatively simple “`\n<%s>.*? \.`”. It searches for new lines (`\n`) that start with the shape name enclosed in angle brackets. The shape name is provided as a parameter in the string, which is stated as `%s`. Then any characters are accepted (`.*?`) until there is a space followed by a dot, which marks the end of a shape. The dot is a special character in regular expressions and therefore has to be escaped with a backslash. The result of this regex for the shape name “http://shaclshapes.org/agePersonShapeProperty” is given in Listing 4.1.
After extracting the unmodified text for a shape with regex, it still needs to be made comparable to other shapes. First, the support and confidence triples need to be removed, as these values typically vary between different graph versions. Support and confidence values are not part of the standard SHACL terminology, they were added by the QSE algorithm during shape generation. That is also why support and confidence use the namespace “http://shaclshapes.org” and not the SHACL namespace “http://www.w3.org/ns/shacl”. The removal was done by converting the text segment into a Jena graph and then filtering the support and confidence statements. Jena and RDF4J are Java libraries that allow the creation of so-called models from a Turtle file, which can then be managed as a graph in Java. Additionally, issues arose with SHACL-In and SHACL-Or items, as these are not considered by the TurtleFormatter. Some adjustments to the text were needed to reorder these items alphabetically. This task was solved using regex and Java string operations. Finally, the shape is now comparable to shapes with an identical name from other graph versions. This procedure is also illustrated in Algorithm 4.1. As given in the examples in Listing 4.1 and Listing 4.2, the modified Turtle snippet does not contain triples for support and confidence anymore, and the constraints in the or-list were ordered alphabetically.

Algorithm 4.1: Modify a SHACL shape to make it comparable

```
Input  : $S_f$: SHACL shapes in Turtle format, shapeName
Output : comparableShape
1 $comparableShape$ ← use regex on $S_f$ to find SHACL shape with the name
    $shapeName$
2 Remove all support and confidence triples in $comparableShape$
3 Run TurtleFormatter [3] on $comparableShape$
4 Reorder items in SHACL-In lists in $comparableShape$ alphabetically
5 Reorder items in SHACL-Or lists in $comparableShape$ alphabetically
```

An alternative attempt involved reading the whole SHACL file provided by QSE into an RDF4J or Jena model, querying it using the shape’s name, and recreating the Turtle text snippet to avoid using regex and string operations on the SHACL file. Jena or

RDF4J models can be queried using SPARQL or provided methods to retrieve only the relevant parts for each shape. This was achieved by searching for triples where the subject equals the name of the shape. However, more complex shapes can include SHACL-Or items or SHACL-In lists that are internally stored with blank nodes. To retrieve the information stored in these blank nodes, the triples had to be loaded recursively from the model again. For instance, querying a small node shape by name was relatively straightforward. However, querying a more complex property shape from a large model led to complications. When a property shape includes a list, as illustrated in Listing 4.1, the content of this list is stored as a blank node. To retrieve the content of this blank node, it is necessary to query the entire model again for the blank node's identifier. RDF lists involve multiple blank nodes: the first element is referenced by one blank node, and the remaining elements, also called the rest, are also referenced in a blank node. Consequently, retrieving the content of these triples requires multiple iterations over the entire model. The complexity increases with longer lists or if an item in the list contains a SHACL-In list. Therefore, this approach, in general, proved to be inefficient, which is why the final solution was to use regex to extract the shape segments directly from the SHACL file. Although this approach has several disadvantages, such as potential inconsistencies, it is notably faster.

4.2 Technical Details of the Implementation

The implementation includes three tools to address all research questions: the ShapeComparator, the SHACL-DiffExtractor, and the SPARQL-ShapeComparator. The ShapeComparator is a web application, while the other two tools are command-line applications. To implement this project Java (Version 17) was used. A project was created that includes a Java package for each tool.
QSE is available in different versions, namely the "main" branch [7], which is used for applying QSE on a single graph in the console efficiently, and the "shactor version". Shactor utilizes a variant of the QSE algorithm that differs from the main branch. This version generates Java objects for each node shape, property shape, and similar elements, in addition to producing SHACL shapes as a Turtle file. These objects are needed to display the SHACL shapes in Shactor's graphical interface. Consequently, this thesis employs the "shactor version" of QSE [8]. It is used for the graphical part, the ShapeComparator, but also for the algorithms since the performance impact for the Java shape object generation in QSE is minimal.
Some modifications in the QSE algorithm were needed for the successful implementation of all functionalities described in this thesis. These modifications were done on the "shactor"-branch and are described in detail in Section 4.3. Of course, these modifications are included in the QSE version, which is used by all parts of this project.
The code used in this project is open source and available on GitHub at the following link: https://github.com/dbai-tuw/QseEvolvingKg.

4.3 Extensions of the QSE Algorithm

The QSE algorithm in the “shactor”-branch already offers many capabilities, such as providing Java objects for node and property shapes. However, some adjustments were necessary to enable the ShapeComparator, the SHACL-DiffExtractor, and the SPARQL-ShapeChecker to work correctly. The extensions described were developed as part of this thesis. However, the code was directly contributed to the official QSE repository to allow other developers to benefit from the enhancements as well.
The main adjustments involved updating the code to make internal maps, methods, and results accessible to external programs. For example, the SHACL-DiffExtractor needs access to the “shapeTripletSupport”-map to recalculate the confidence values, as described in Algorithm 6.1. Another modification was made to the “PropertyData” class in the QSE algorithm. Originally, QSE only recorded which object types were referenced by a given property. However, for the SHACL-DiffExtractor to work properly, it was necessary to save also the count for each object type. This change enabled parsing the changesets to obtain the correct support for a shape. The feature can be enabled or disabled in the QSE configuration file.
Additional updates included upgrading the reference to GraphDB [11] from version 9.3 to 10.3, along with fixing several minor bugs. In the query-based version of QSE, the output shapes were duplicated after each run, so a method was added to clear the output directory to prevent this issue. Additionally, the query-based option of QSE produced incomplete shapes for literal property shapes, which was resolved by fixing a parameter. The QSE algorithm uses a SAIL (Storage And Inference Layer) repository [21] to export SHACL shapes to a file. Previously, the SAIL repository was not closed after each run, as the program was typically used only once at a time. This led to an issue where QSE could not extract shapes more than once, which was resolved by ensuring the SAIL repository was properly closed. Another issue involved an exception being thrown if the runtime logs file was missing, which occurred when QSE had not been run for a specific graph yet. The exception was resolved by creating a default file to minimize error messages. Lastly, an improvement was made to allow setting pruning thresholds directly from the program, rather than only through a file.
The updated code is available in the “shactor” branch of the QSE repository at https://github.com/dkw-aau/qse/tree/shactor.

CHAPTER 5

ShapeComparator

This chapter presents an in-depth overview of the ShapeComparator, which is the result of research question 1. The ShapeComparator is a web application designed to compare SHACL shapes generated by QSE [84]. The chapter starts with exploring the tool's requirements and functionality, followed by a demonstration with screenshots, along with implementation details. Finally, it concludes with an evaluation based on semi-structured interviews.

5.1 Requirements and Functionality

To answer research question 1, the ShapeComparator needs to enable a user to compare SHACL shapes in a user-friendly way using a visual interface. The ShapeComparator allows users to upload different versions of a graph, for example, G_1 and G_2. Next, the tool facilitates the extraction of shape graphs, like S_1 and S_2, using QSE. Finally, users can compare these generated shapes with each other.
A brief overview of the overall architecture for this tool is shown in Figure 5.1. The diagram illustrates the involvement of a database that stores graph versions provided by the user and the generated shapes. Additionally, the application uses the QSE algorithm [84] to generate SHACL shapes. The application is accessible via a web browser. The user workflow is provided in Figure 5.2. While this diagram illustrates the default workflow, users have the flexibility to return to previous steps at any time. The subsequent sections describe the functionalities of these steps in more detail.

5.1.1 Storage of Different Graph Versions

To make the comparison of graph versions easier, the ShapeComparator offers the possibility to save graphs and their respective versions. Each graph can have multiple versions associated with it. For instance, the graph BEAR-B, which is described in more

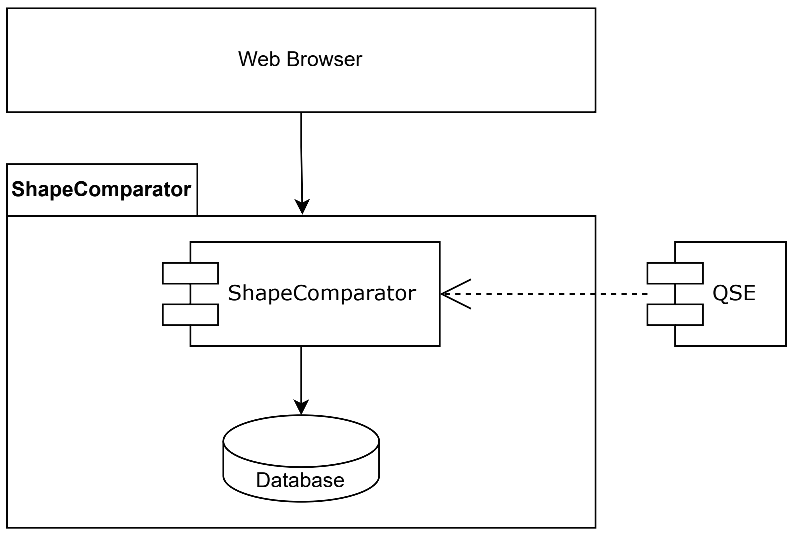


Figure 5.1: ShapeComparator - Architecture

detail in Section 6.4.1, has 89 versions.
In compliance with QSE, the ShapeComparator requires that only graphs or versions in N-Triples format can be uploaded. Users have the option to either upload a file or use a pre-loaded graph.
The capability to store graphs and their versions improves the user experience significantly. Without this feature, users would have to upload files repeatedly for comparisons, which would be very inconvenient and consume additional time and effort. Shactor [85] does not include this time-saving feature. This is specifically convenient as there may arise scenarios where running QSE multiple times on the same graph file is necessary, for instance, with varying support or confidence parameters.

5.1.2 Extract SHACL Shapes

When QSE should be run, the user chooses a graph and the desired version. Similar to Shactor, all classes C mentioned in the graph G and how often they occur in G are listed. The user can then choose the target classes on which QSE should be run. Additionally, the user can either extract default shapes (all shapes) or set the support and confidence parameters for pruning. Notably, a node shape does not have a confidence parameter.

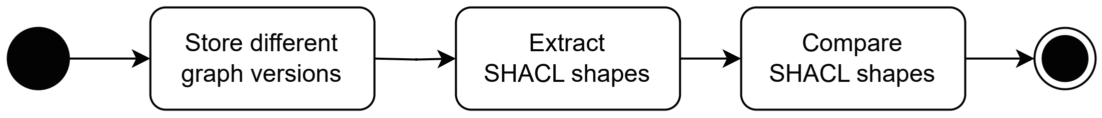


Figure 5.2: ShapeComparator - Activity Diagram

The program also stores the extracted SHACL shapes in the database, allowing them to be selected for comparison later.

5.1.3 Compare SHACL Shapes

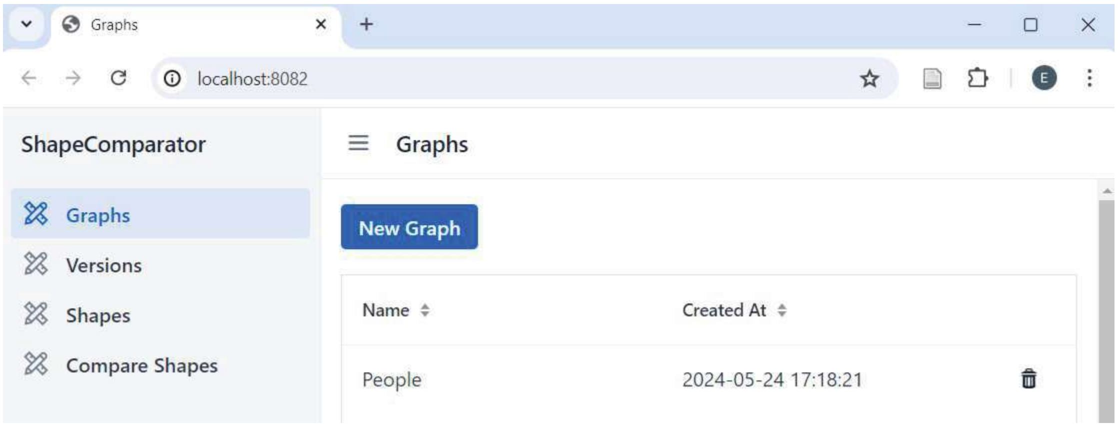
Finally, the extracted shapes S are available for comparison. Users can choose one or multiple QSE runs along with their extracted shapes for comparison. In general, there is the possibility to compare extracted shapes with all combinations of different parameters such as support, confidence, or chosen classes. However, there will be warnings, if extracted shapes with different support or confidence parameters are compared or if extracted shapes with different classes are selected. The order in which the objects are compared can also be chosen. For instance, if there are results from two QSE runs (S_1 and S_2), then S_1 can be compared with S_2, or vice versa, S_2 with S_1.
For an overview of the comparison, node and property shapes are organized in a tree-view format. Objects from different QSE runs are displayed side by side, with shapes mapped by their name across different QSE runs. A key assumption of this project is that shapes can be identified by their name as QSE structures shape names following a consistent naming convention. However, more details on this topic are provided in Section 4.1. Shapes, which are added, deleted, or do not have the same content across different versions are highlighted so that the user can easily find differences. Users can search for specific shapes per name or filter by identical node shapes, identical property shapes, or different shapes in general.
There is a detailed view for every node or property shape in the comparison overview, where the SHACL shapes of the QSE runs are displayed as text. It was decided to display SHACL shapes as Turtle text snippets, as this format is both readable and familiar to users as the intended users of this tool have experience with SHACL in any way.
In this detailed view, discrepancies are also visually highlighted in red or green, indicating additions or removals in the text. If multiple QSE runs were selected for comparison, there is the possibility of selecting the desired runs, as only two shapes can be directly compared at a time. Additionally, to the text comparison, all shapes are listed along with their corresponding support and confidence values, as these were omitted from the original SHACL shape.
When a user selects a row in the overview where a shape was deleted, a prompt will appear, explaining why this shape was excluded. This information is retrieved from the default shapes which are also preserved during shapes extraction.

5.2 Implementation

In reference to Shactor [85], the application was developed with Vaadin (Version 24.2.4) [27], and Spring [26]. Vaadin is a full-stack framework that allows front-end development with Java. For this project, the free version of Vaadin was selected as the features provided were sufficient. Spring is a popular Java framework that simplifies the development of different applications, such as microservices, web apps, cloud-based-, or serverless applications. In this context, it is used alongside Vaadin and offers the advantage of reducing boilerplate code for web application development and simplifying configuration tasks. Additionally, the Spring framework facilitates an easy integration of a local database. Another tool involved was the JavaScript library jsdiff [15] to compare the text segments of SHACL shapes visually. This tool is used to highlight the differences automatically in the details view of ShapeComparator.

5.2.1 Illustration with Screenshots

An extended version of the example from Section 2.1.2 was chosen to showcase the functionalities of ShapeComparator. The complete RDF graphs are available in Appendix B. The ShapeComparator is designed for laptop screens, a screenshot is illustrated in Figure 5.3 The initial screen of the ShapeComparator application shows a list of all graphs



Name	Created At
People	2024-05-24 17:18:21

Figure 5.3: Full screenshot of the ShapeComparator application

saved in the application. Each graph can have multiple versions associated with it. In this demonstration, the “People” knowledge graph has three versions. These versions of the knowledge graph are fully printed in Listings B.1, B.2 and B.3.
In the first version of the knowledge graph, there are three people, Alice, Bob, and Jenny, who have a name attribute and an attribute that indicates whether they know each other. A snippet of this knowledge graph is shown in Listing 5.1.

```
<http://example.org/alice>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/name> "Alice" .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/bob> .
```

Listing 5.1: Snippet of the People Knowledge Graph (Version 1)

In the second version of this knowledge graph three cats, each with an associated color, are added. A snippet of this knowledge graph can be seen in Listing 5.2.

```
<http://example.org/orangeCat>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Cat> .
<http://example.org/orangeCat>
    <http://example.org/color> "orange" .
```

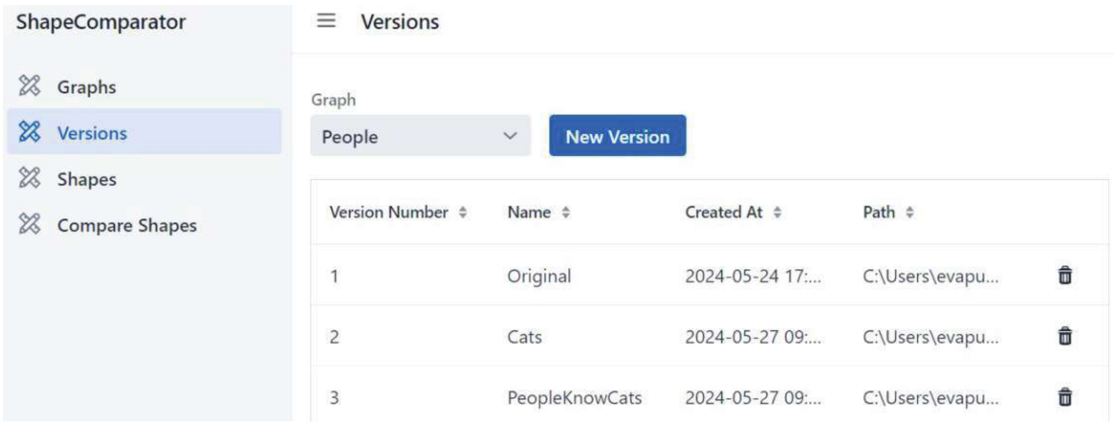
Listing 5.2: Snippet of the People Knowledge Graph (Version 2)

In the third version, the color attribute is removed from two of the three cats, and instead of knowing each other, Alice, Bob, and Jenny each now know one cat, as shown in Listing 5.3.

```
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/orangeCat> .
```

Listing 5.3: Snippet of the People Knowledge Graph (Version 3)

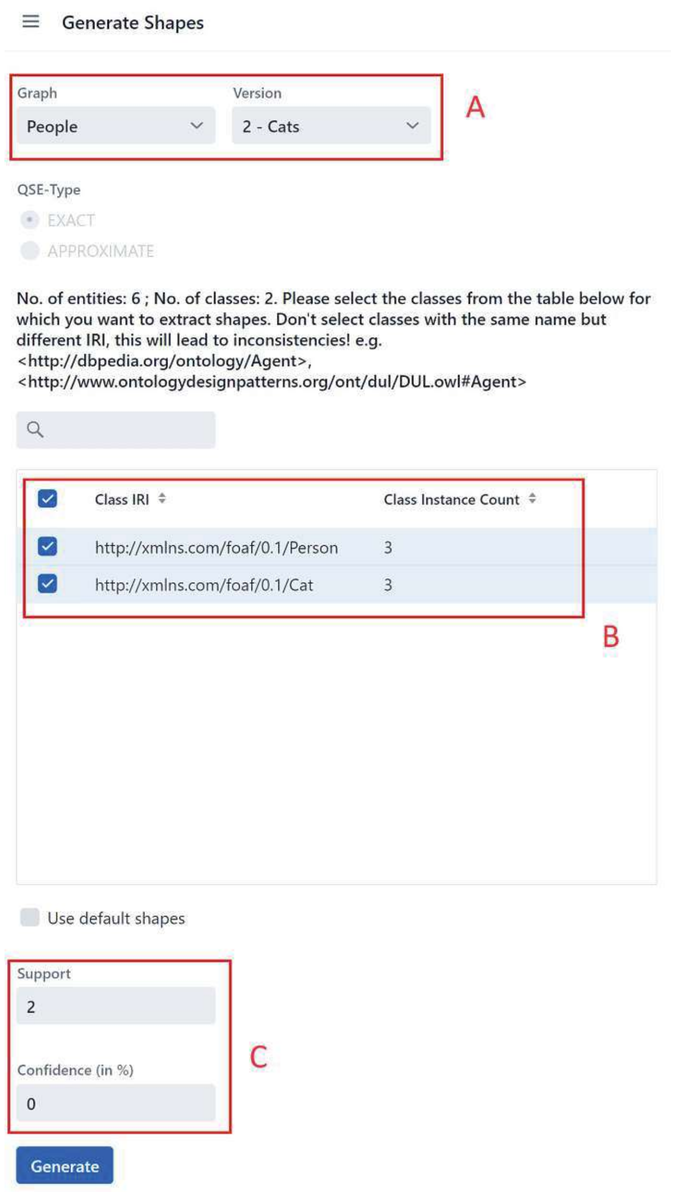
The user interface for the overview of the versions of a graph is illustrated in Figure 5.4. While the version number is generated automatically, the user has the option to assign a custom name to each version. When the graph is created, the initial version is automatically named “Original”. Users can edit the name of a graph version by double-clicking it in the list. However, the order of the graph versions cannot be modified.



Version Number	Name	Created At	Path
1	Original	2024-05-24 17:...	C:\Users\evapu...
2	Cats	2024-05-27 09:...	C:\Users\evapu...
3	PeopleKnowCats	2024-05-27 09:...	C:\Users\evapu...

Figure 5.4: Overview of all versions of a graph

After the knowledge graph versions are saved, to avoid uploading them repeatedly, the user can continue to extract SHACL shapes. Figure 5.5 illustrates this process for the second version of the graph (Box A). QSE automatically lists all classes along with their instance counts for selection (Box B). As previously described, this version includes three people and three cats. The support parameter is set to two and the confidence parameter is set to zero (Box C). This means that all shapes, with a support of two or lower, are not considered for the resulting shapes. The confidence parameter is set to zero percent, ensuring no triples are dropped because of too low confidence values. The detailed definitions of the support and confidence parameters are provided in Section 2.2.3.



Graph: People
Version: 2 - Cats

QSE-Type
[✓ EXACT]
APPROXIMATE

No. of entities: 6 ; No. of classes: 2. Please select the classes from the table below for which you want to extract shapes. Don't select classes with the same name but different IRI, this will lead to inconsistencies! e.g. <http://dbpedia.org/ontology/Agent>, <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Agent>

✓	Class IRI	Class Instance Count
✓	http://xmlns.com/foaf/0.1/Person	3
✓	http://xmlns.com/foaf/0.1/Cat	3

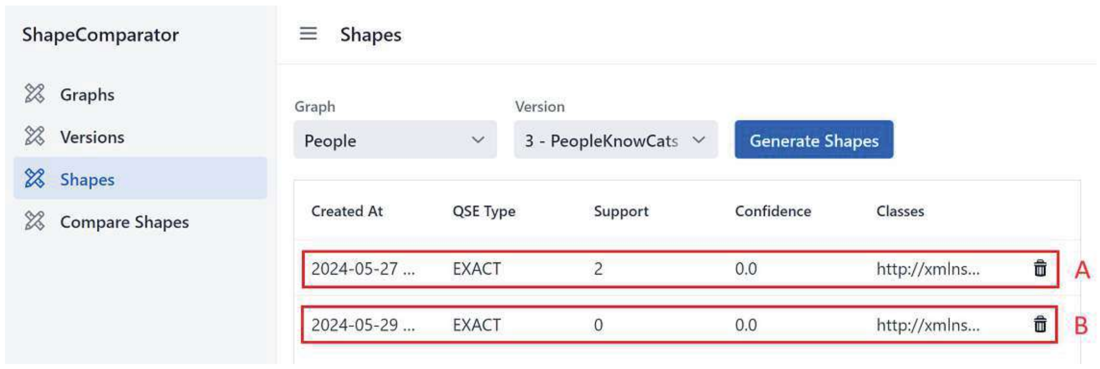
Use default shapes

Support: 2

Confidence (in %): 0

Figure 5.5: SHACL shapes can be extracted for a version of a graph

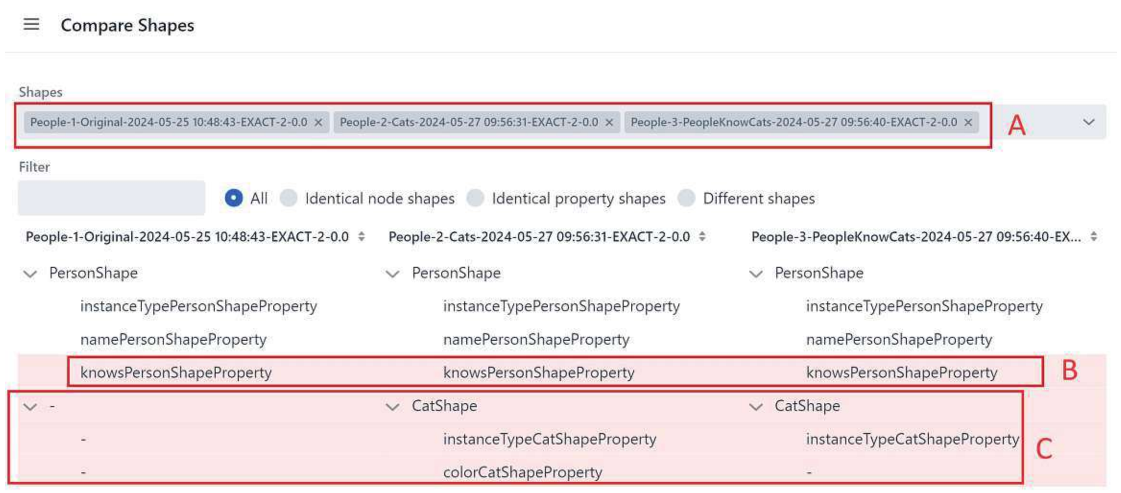
The overview of all extracted shapes for the third version is illustrated in Figure 5.6. In this example, shapes were extracted once with a support threshold of two (Box A), and once without any pruning thresholds (Box B). The resulting shapes of a QSE run without pruning thresholds are also called default shapes, meaning that all shapes are created and none are discarded, as indicated by the support and confidence parameter thresholds of zero. A more detailed description of this topic is provided in Section 2.2.3. The system is able to store the resulting shapes for multiple QSE runs for a version separately. These shapes are saved in the database.



Created At	QSE Type	Support	Confidence	Classes
2024-05-27 ...	EXACT	2	0.0	http://xmlns...
2024-05-29 ...	EXACT	0	0.0	http://xmlns...

Figure 5.6: Overview of all extracted shapes for a version of a graph

Finally, the extracted shapes are available for comparison in Figure 5.7. At the top of the page, three QSE runs along with their extracted shapes are chosen (Box A). To identify QSE runs in this view, they are named after the following naming convention: <graph name>-<version number>-<version name>-<creation timestamp>-<QSE type>-<support threshold>-<confidence threshold>. The overview indicates changes between versions with a light red background for these lines. In this example the “knows” property in the class Person has changed, as this property points to cats in the third version of the graph (Box B). Additionally, there are changes for the cat class (Box C).
Clicking on the row “knowsPersonShapeProperty” in the comparison table (Figure 5.9) redirects to a more detailed comparison view, where the discrepancies are visually highlighted in red and green, indicating additions or removals in the text. “Cat” is written in green, as the targeted class is now a “Cat” and not “Person” anymore, which is written in red (Box A). Below the comparison, the SHACL shapes for all versions are listed along with their corresponding support and confidence values (Box B). In this example, the text for the first version was removed, since nothing has changed between the first and the second version for the property “knowsPersonShapeProperty”.
To showcase the removal of a shape due to low support or confidence values, the detailed comparison of the “colorCatShapeProperty” is chosen. In this example, the SHACL shape no longer exists in the third version of the graph because the “color” attributes of the black and grey cats were deleted. As a result, the support for this property is only one, which is below the defined threshold of two when the shapes were created. This explanation is displayed in Figure 5.8 which can be opened by clicking on the row in the overview table, which is displayed in Figure 5.7.



Shapes

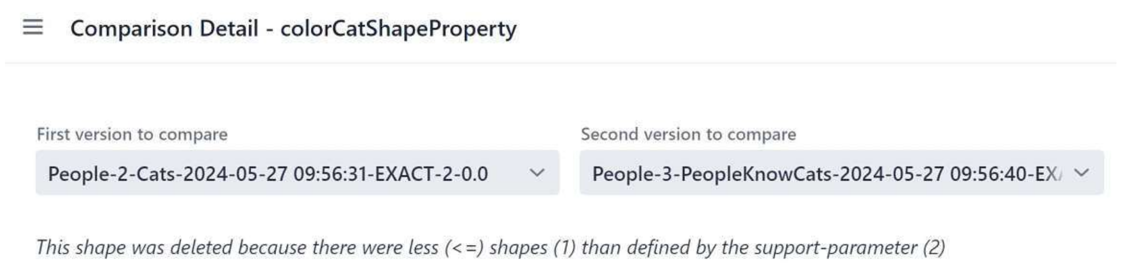
People-1-Original-2024-05-25 10:48:43-EXACT-2-0.0 × People-2-Cats-2024-05-27 09:56:31-EXACT-2-0.0 × People-3-PeopleKnowCats-2024-05-27 09:56:40-EXACT-2-0.0 ×

Filter

(● All) Identical node shapes Identical property shapes Different shapes

People-1-Original-2024-05-25 10:48:43-EXACT-2-0.0	People-2-Cats-2024-05-27 09:56:31-EXACT-2-0.0	People-3-PeopleKnowCats-2024-05-27 09:56:40-EX...
PersonShape	PersonShape	PersonShape
instanceTypePersonShapeProperty	instanceTypePersonShapeProperty	instanceTypePersonShapeProperty
namePersonShapeProperty	namePersonShapeProperty	namePersonShapeProperty
knowsPersonShapeProperty	knowsPersonShapeProperty	knowsPersonShapeProperty
-	CatShape	CatShape
-	instanceTypeCatShapeProperty	instanceTypeCatShapeProperty
-	colorCatShapeProperty	-

Figure 5.7: Comparison between extracted shapes



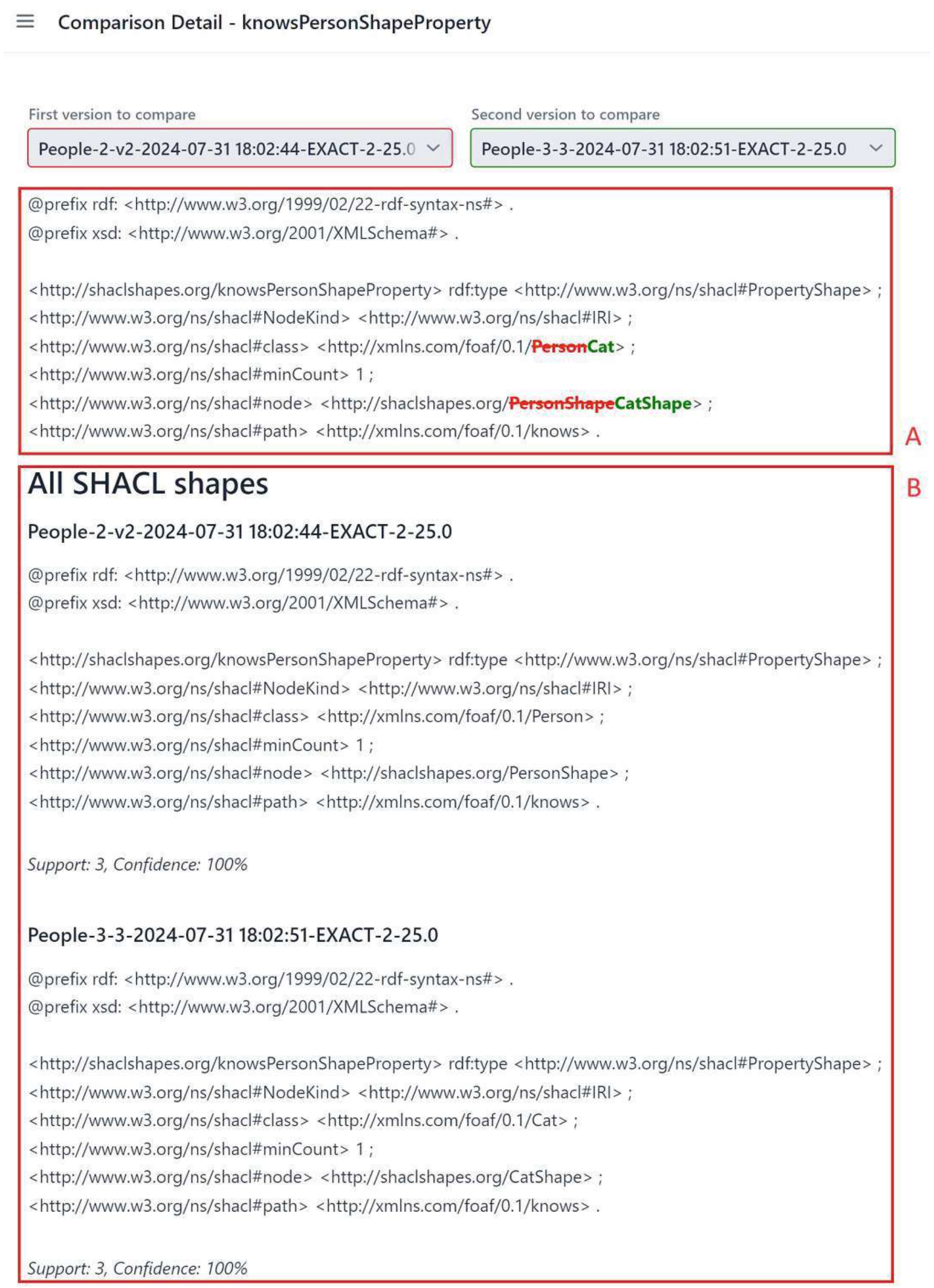
Comparison Detail - colorCatShapeProperty

First version to compare: People-2-Cats-2024-05-27 09:56:31-EXACT-2-0.0

Second version to compare: People-3-PeopleKnowCats-2024-05-27 09:56:40-EX/

This shape was deleted because there were less (<=) shapes (1) than defined by the support-parameter (2)

Figure 5.8: Detailed comparison view in case a shape was deleted



Comparison Detail - knowsPersonShapeProperty
First version to compare
People-2-v2-2024-07-31 18:02:44-EXACT-2-25.0
Second version to compare
People-3-3-2024-07-31 18:02:51-EXACT-2-25.0
A
B

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/knowsPersonShapeProperty> rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
<http://www.w3.org/ns/shacl#NodeKind> <http://www.w3.org/ns/shacl#IRI> ;
<http://www.w3.org/ns/shacl#class> <http://xmlns.com/foaf/0.1/~~Person~~Cat> ;
<http://www.w3.org/ns/shacl#minCount> 1 ;
<http://www.w3.org/ns/shacl#node> <http://shaclshapes.org/~~PersonShape~~CatShape> ;
<http://www.w3.org/ns/shacl#path> <http://xmlns.com/foaf/0.1/knows> .
```

All SHACL shapes

People-2-v2-2024-07-31 18:02:44-EXACT-2-25.0

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/knowsPersonShapeProperty> rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
<http://www.w3.org/ns/shacl#NodeKind> <http://www.w3.org/ns/shacl#IRI> ;
<http://www.w3.org/ns/shacl#class> <http://xmlns.com/foaf/0.1/Person> ;
<http://www.w3.org/ns/shacl#minCount> 1 ;
<http://www.w3.org/ns/shacl#node> <http://shaclshapes.org/PersonShape> ;
<http://www.w3.org/ns/shacl#path> <http://xmlns.com/foaf/0.1/knows> .
```

Support: 3, Confidence: 100%

People-3-3-2024-07-31 18:02:51-EXACT-2-25.0

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/knowsPersonShapeProperty> rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
<http://www.w3.org/ns/shacl#NodeKind> <http://www.w3.org/ns/shacl#IRI> ;
<http://www.w3.org/ns/shacl#class> <http://xmlns.com/foaf/0.1/Cat> ;
<http://www.w3.org/ns/shacl#minCount> 1 ;
<http://www.w3.org/ns/shacl#node> <http://shaclshapes.org/CatShape> ;
<http://www.w3.org/ns/shacl#path> <http://xmlns.com/foaf/0.1/knows> .
```

Support: 3, Confidence: 100%

Figure 5.9: Detailed comparison view

5.2.2 Database Model

As a database, a local H2 instance [13] is used as it is lightweight and easy to integrate with the Spring framework [26]. The H2 database engine is a relational database system written in Java which makes it ideal for the integration with Java applications. Its advantages include being very fast, open source, and having a small memory footprint. The database is needed to store the graphs, versions, QSE runs, and the node and property shapes generated. The object-relational mapping framework Hibernate [29] is used to define Java classes which are then automatically mapped to relational database tables. This feature is also utilized in this context. The database tables for this application, which were automatically created, are displayed in Figure 5.10.

A graph can have multiple versions and those can have different extracted shapes. For each QSE run (i.e., extracted shapes), classes C can be defined before the extraction process. This feature, described in Section 2.2.3, allows QSE to be executed only on specific classes in a graph, rather than processing all triples in a graph. The IRI of a class is saved in the database for this use case. An extracted-shapes object, representing a QSE run, contains all node shapes generated by QSE. Consequently, any number of node shapes can be associated with an extracted-shapes object, depending on the graph. The user has the option to prune the shapes via the support and confidence parameter during the extraction process or to generate all shapes without pruning, which are also referred to as the default shapes, as defined in Section 2.2.3. Depending on this choice, either the pruned shapes and the default shapes are saved (both are needed to explain, why a shape was removed), or only the default shapes are generated. If the user does not define the support and confidence parameter, then all shapes are generated and only these are saved in the database. In this case, the default shapes are saved in the list of the node shapes for an extracted-shapes object and the list where the default shapes would be saved in the pruning scenario is left empty. Each node shape has multiple property shapes. The concept of node and property shapes is also modeled in the Java objects of QSE, as defined in Section 2.2.3, however, the tables saved in the database are slightly different to accommodate customization to the use case of the ShapeComparator application.

The mapping tables, (i.e. EXTRACTED_SHAPES_NODE_SHAPES), do not have primary keys because they are automatically created by the Spring framework to represent lists in the Java classes. Adding primary keys to these tables would require the creation of additional classes, which would make the code less readable.

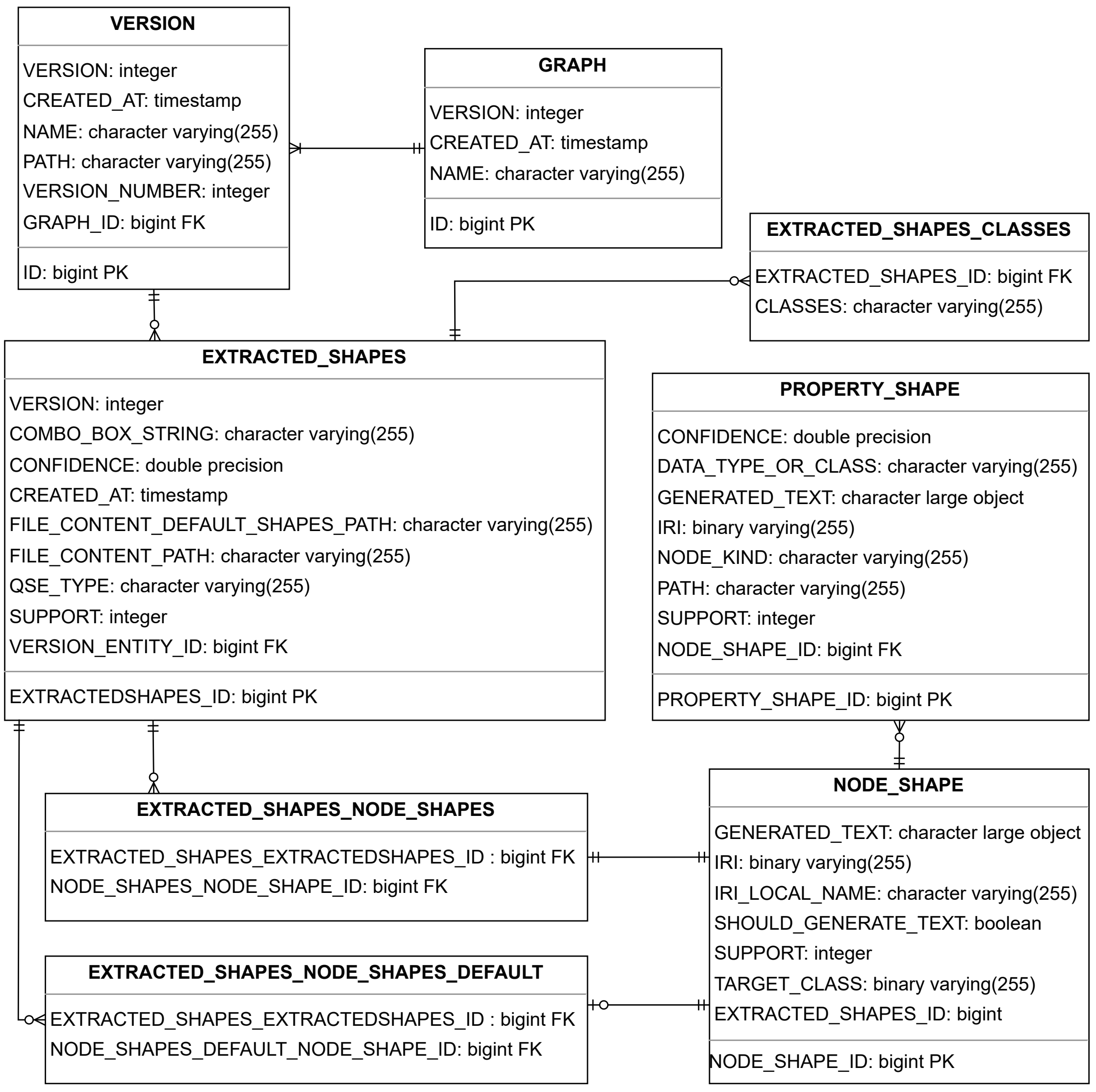

VERSION
VERSION: integer
CREATED_AT: timestamp
NAME: character varying(255)
PATH: character varying(255)
VERSION_NUMBER: integer
GRAPH_ID: bigint FK
ID: bigint PK
GRAPH
VERSION: integer
CREATED_AT: timestamp
NAME: character varying(255)
ID: bigint PK
EXTRACTED_SHAPES_CLASSES
EXTRACTED_SHAPES_ID: bigint FK
CLASSES: character varying(255)
EXTRACTED_SHAPES
VERSION: integer
COMBO_BOX_STRING: character varying(255)
CONFIDENCE: double precision
CREATED_AT: timestamp
FILE_CONTENT_DEFAULT_SHAPES_PATH: character varying(255)
FILE_CONTENT_PATH: character varying(255)
QSE_TYPE: character varying(255)
SUPPORT: integer
VERSION_ENTITY_ID: bigint FK
EXTRACTEDSHAPES_ID: bigint PK
PROPERTY_SHAPE
CONFIDENCE: double precision
DATA_TYPE_OR_CLASS: character varying(255)
GENERATED_TEXT: character large object
IRI: binary varying(255)
NODE_KIND: character varying(255)
PATH: character varying(255)
SUPPORT: integer
NODE_SHAPE_ID: bigint FK
PROPERTY_SHAPE_ID: bigint PK
EXTRACTED_SHAPES_NODE_SHAPES
EXTRACTED_SHAPES_EXTRACTEDSHAPES_ID : bigint FK
NODE_SHAPES_NODE_SHAPE_ID: bigint FK
NODE_SHAPE
GENERATED_TEXT: character large object
IRI: binary varying(255)
IRI_LOCAL_NAME: character varying(255)
SHOULD_GENERATE_TEXT: boolean
SUPPORT: integer
TARGET_CLASS: binary varying(255)
EXTRACTED_SHAPES_ID: bigint
NODE_SHAPE_ID: bigint PK
EXTRACTED_SHAPES_NODE_SHAPES_DEFAULT
EXTRACTED_SHAPES_EXTRACTEDSHAPES_ID : bigint FK
NODE_SHAPES_DEFAULT_NODE_SHAPE_ID: bigint FK

Figure 5.10: ERD of the database used for saving graphs and shapes in the ShapeComparator

5.2.3 Algorithm for Comparing SHACL Shapes

Building on the general concept of the shape comparison described in Section 4.1, an approach was developed to efficiently compare the SHACL shapes within the ShapeComparator. On the comparison overview page, users can select which extracted shapes (i.e. QSE runs) should be compared with each other. The overview page table includes a column for each extracted shapes object, which varies based on the number of extracted shapes the user selects in the combo box. The table rows represent the node and property shapes across the selected extracted shapes objects, with a hierarchical structure where node shapes are parent objects to property shapes as illustrated in Figure 5.7
When a user selects an extracted shapes object in the combo box, the algorithm processes the nodes and property shapes generated for that QSE run. For each row in the overview table, an instance of the "ComparisonTreeViewItem" class is created. This class contains the name of the shape and a map of the shapes across the extracted shapes objects. The ID of the extracted shapes object is used as a key in the map and the value contains either a node or a property shape because each row in the table can represent a node or a property shape comparison. Additionally, a boolean in the comparison object indicates whether the shapes are identical across the QSE runs.
These "ComparisonTreeViewItem" objects form the basis for both the comparison overview and the comparison detail view. Whenever a user selects a new extracted shapes object in the combo box, the list of comparison objects is updated accordingly. This may involve

Algorithm 5.1: Creation of the comparison table for the ShapeComparator

```
Input  : selectedItems: chosen items in the combo box of extracted shapes
Output : nodeShapesToShow

1  nodeShapesToShow ← new List<ComparisonTreeViewItem>
2  foreach extractedShape ∈ selectedItems do
3      Check if extractedShape is cached - otherwise fetch from database
4      foreach nodeShape ∈ extractedShape.nodeShapes do
5          if getName(nodeShape) ∈ getNames(nodeShapesToShow) then
6              Add nodeShape to the corresponding item in nodeShapesToShow
7          end
8          else
9              Add a new ComparisonTreeViewItem to nodeShapesToShow with the
                 name getName(nodeShape) and add nodeShape to the list of shapes
10         end
11     end
12 end
13 foreach item ∈ nodeShapesToShow do
14     Set item.shapesEqual to true if all Turtle text snippets of the node shapes in
         item are identical - otherwise false
15 end
```

```
//To add child elements to the Vaadin TreeView, all ComparisonTreeViewItems
  must be reiterated
foreach $item \in treeView.items$ do
    $propertyShapesToShow \leftarrow$ new List<ComparisonTreeViewItem>
    if $item$ represents a comparison of node shapes then
        foreach $extractedShape \in selectedItems$ do
            $nodeShape \leftarrow$ find getName($item$) in $extractedShape.nodeShapes$
            foreach $propertyShape \in nodeShape.propertyShapes$ do
                if $getName(propertyShape) \in$
                 $getNames(propertyShapesToShow)$ then
                    Add $propertyShape$ to the corresponding item in
                     $propertyShapesToShow$
                end
                else
                    Add a new ComparisonTreeViewItem to
                     $propertyShapesToShow$ with the name
                     getName($propertyShape$) and add $propertyShape$ to the list
                     of shapes
                end
            end
        end
    end
    foreach $item \in propertyShapesToShow$ do
        Set $item$.shapesEqual to true if all Turtle text snippets of the property
         shapes in $item$ are identical - otherwise false
    end
    Add $propertyShapesToShow$ as child items to $nodeShapeToShow$
end
```

creating new rows or modifying the maps in existing comparison objects. The boolean indicating if all rows contain the same Turtle snippet is also updated.
Algorithm 5.1 illustrates this process. The list "nodeShapesToShow", consisting of "ComparisonTreeViewItems" represents all rows in the comparison tree view. The algorithm begins by iterating through all QSE runs and their corresponding node shapes. Based on the name of the node shape, it determines whether to create a new row in the table or update an existing one by adding the node shape to the appropriate "ComparisonTreeViewItem". Finally, the Turtle text snippets in each row are checked for differences, and the boolean value in the "ComparisonTreeViewItem" is updated accordingly.
Since the tree view on the overview page is hierarchically structured, the algorithm must iterate through all rows again to add or update the corresponding child items. However, as this use case involves only a single level of hierarchy, only rows describing node shapes

are considered. The algorithm iterates through the list of QSE runs and identifies the corresponding node shape for each run. Then, all property shapes associated with the node shape are processed. A new row represented by a “ComparisonTreeViewItem” is created if no row exists for a given property shape name. If a “ComparisonTreeViewItem” with the same shape name already exists, the property shape is added to the existing item. Finally, the algorithm checks each row to determine whether all Turtle text snippets are identical. The list of “ComparisonTreeViewItems ” is added to the node shape row, completing the table’s preparation. This algorithm is triggered whenever changes occur in the combo box of the selected QSE runs.

5.2.4 Performance Enhancements

After completing the initial version of the ShapeComparator, it was tested with very small datasets to uncover potential edge cases. However, as testing progressed with larger datasets, it became clear that certain features needed to be adjusted to enhance the speed of the web application.
Originally, it was planned to store the extracted SHACL files directly in the database as text. However, this approach quickly led to performance issues while querying data because SHACL files can get quite large. Therefore, the extracted shapes and graph versions are stored in the project directory as Turtle files and the file paths pointing to the files are saved in the database as a reference.
A further performance improvement involved creating the generated Turtle text snippets for the detail view for each node and property shape only once and saving it to the database, which notably accelerated the comparison process. Details on this topic are provided in Section 4.1.
Additionally, performance improvements were implemented for loading data from the database into the main memory, particularly for node and property shape objects. When fetching an extracted shapes object (also known as a QSE run) from the database, there are two options for handling referenced node and property shapes. Those can either be loaded immediately when the extracted shapes object is needed (eager loading) or loaded only when required (lazy loading). For this project, lazy loading for the node and property shapes was implemented by default. When only a small number of objects are referenced, the difference between eager and lazy loading is negligible for the end user. However, when hundreds of shapes with their associated Turtle text are loaded, the performance difference becomes significant.
In the overview of all extracted shapes, only meta information such as creation date, support, and confidence parameters are needed, therefore lazy loading is the preferred solution here. This approach is also applied when generating options in the combo box on the comparison overview page, where only meta information is required. However, when creating the table in the comparison overview, all shapes and their details are fetched from the database.
Section 5.2.3 explains how the comparison of SHACL shapes works for the ShapeComparator. Users can select multiple extracted shapes (i.e. QSE runs) at the top of the comparison overview page. This selection generates a table that compares all shapes

across the different extracted shapes. Although the computation of this table is straightforward, users will notice a delay when there are many shapes. Therefore, optimizing this part is crucial for performance improvements. A performance improvement in this area is to cache the comparison objects in the overview, to prevent recreating all comparison objects when the user returns from the comparison details page.

5.3 Evaluation

As described in Chapter 3.2.4, to evaluate the ShapeComparator, a qualitative approach was employed through semi-structured interviews. Due to the time-intensive nature of interviews, a convenience sample of six participants was selected. All participants completed the course “Introduction to Semantic Systems” at TU Wien [1] within the past two years. This prerequisite ensured that all participants shared a foundational understanding of semantic systems and knowledge graphs.

5.3.1 Procedure

The interviews were conducted virtually via Zoom [30] between late September and early October 2024, with each session lasting approximately 20 minutes. Before the interviews, the participants received an introductory letter outlining the interview’s purpose. They were also informed about their rights, and informed consent was obtained for the research participation and the processing of personal data. This process followed the official recommendations for informed consent at TU Wien [12]. All interviews were recorded; due to technical difficulties with an external recording tool during two sessions, the recording was later switched to Zoom’s built-in recording feature.
The interview began with introductory questions about when the participants took the course at TU Wien and whether they remembered what RDF and SHACL were. After that, a brief example of RDF, SHACL, and QSE was presented to ensure all participants had a shared understanding of the topic. Next, the demonstration knowledge graph for the interview was introduced. This graph included two versions, where the first version consisted of three people and three cats. The graph was visually displayed as shown in Figure 5.11. To illustrate the evolution of the knowledge graph, in the second version of the graph, the “foaf:knows” property no longer pointed to people but instead pointed to the cats (e.g., “Alice” knows the “orangeCat”, “Bob” knows the “blackCat”, and “Jenny” knows the “greyCat”). Following this introduction, the evaluation of the tools began. First, the QSE extraction process was demonstrated using Shactor [85]. The two versions of the knowledge graph had been pre-loaded into the program so that QSE could be executed on them. In this way, the participants could be familiarized with QSE. Next, the two resulting Turtle files containing the SHACL shapes for both graph versions were displayed side by side. Participants were then asked to identify the differences between the SHACL shapes and to estimate the number of the SHACL shapes they believed it would be practicable to compare them in this way or by using a text comparison tool.
In the final step, the ShapeComparator was introduced to the participants. They were

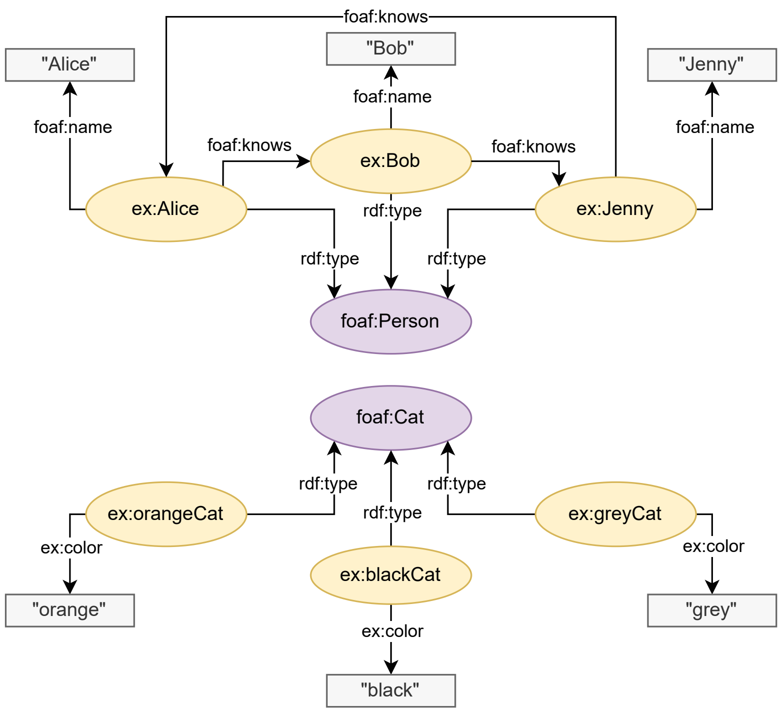


Figure 5.11: Demonstration knowledge graph - Version 1

guided through the tool to demonstrate its main features. The SHACL shapes for the test knowledge graph versions had been pre-computed. Then, the SHACL shapes were compared between the two versions of the test knowledge graph. At the end of the interview, the participants were invited to ask any questions and to share their preferences regarding the methods used (Shactor with manual comparison versus the ShapeComparator). They were also asked whether they believed the ShapeComparator would provide additional value to a knowledge graph engineer. Finally, participants were asked to rate the tool's usability and to give any other feedback. Open-ended questions were employed in this part to get comprehensive insights.

5.3.2 Results

The results were obtained using the qualitative content analysis by Mayring [72]. The inductive approach for category development was selected, as it allows categories to emerge directly from the content. These categories were then continuously revised and adapted throughout the analysis process. All results are summarized in Table 5.1.

Statements	Mentions
Semester of course Introduction into Semantic Systems	
Winter semester 2022	4
Winter semester 2023	2

Continuation of Table 5.1	
Statements	Mentions
Familiarity with RDF knowledge graphs	
Yes	6
Familiarity with SHACL	
No, since it was not lectured in winter semester 2022	3
Yes	2
Partial	1
Difficulties comparing SHACL shapes by using only text files or a text comparison tool	
It is difficult to see differences using only text files	6
In the text files, there are a lot of lines, the shapes look similar and there is a lot of unneeded information	4
A text-comparison tool helps in this process	3
Syntax highlighting would be great	2
The process of comparing shapes manually is cumbersome and error-prone	1
Until which size of a graph is comparing SHACL shapes manageable without the ShapeComparator	
It can get complicated quickly	5
It is already difficult for a graph with seven shapes without additional tools	3
Ten shapes	1
Hundred lines of text	1
Thousand or even ten thousand lines	1
Small knowledge graphs only, definitely not a production knowledge graph	1
Depends on the kind and the number of changes	1
Little changes are easier to find than larger changes	1
It is tedious for larger files if you cannot use additional tools	1
The graph can be bigger if a text comparison tool can be used	1
It is even difficult with a text comparison tool to see what has already been compared	1
The references in the files can be in different positions, the user might have to jump back and forth	1
Tool comparison between Shactor and the ShapeComparator	
I would prefer using the ShapeComparator over text-comparison tools	6
Usefulness of the ShapeComparator for a knowledge graph engineer	
The ShapeComparator brings additional value to a knowledge graph engineer	6
The tool can motivate a knowledge graph engineer	1
Intuitiveness of the ShapeComparator	

Continuation of Table 5.1	
Statements	Mentions
The shape that included changes was clearly visible to the interviewee in the overview of the ShapeComparator	4
It was not clear which shape had differences only from the overview menu	2
Usability	
It is easy to spot differences between shapes	6
Good usability/the tool looks good/it is user-friendly	4
The program is clear for the user, it is straightforward, understandable	3
The tool hides unnecessary details	3
The interface is not too complicated and clean	3
The changes are easier to scan with the overview page	2
It is nice that the ShapeComparator can compare multiple extracted shapes	2
The tool saves time/effort	2
The tool makes the process less tedious and frustrating	1
The feature for filtering is helpful	1
It's okay, not amazing, not bad, needs polishing	1
The tool is specifically designed for comparing shapes	1
Suggested improvements	
It would be nice to see the graph versions and the extracted SHACL shapes	1
It would be nice if the ShapeComparator had a graphical interface in a bubble-like manner	1
In case more versions are compared it is tricky to see the changes between two single versions on the overview page, it would be beneficial to see this information	1
A combination of Shactor and the ShapeComparator would be nice	1
There could be a more aggregated view on the overview page, in case the list of shapes is very long, collapsing the node shapes automatically in this case would help	1
Shapes could be marked with a dot if something has changed	1
On the overview page, the shapes should be automatically filtered for changed shapes only	1
Criticism	
Unsure, if the tool brings additional value if there are a lot of changes	1
It can get confusing if the IRI that is replaced in the SHACL shape is very long and gets printed twice (once red, once green)	1
Feedback for the interview	
It would have been great to use the ShapeComparator myself to test the usability	2

Continuation of Table 5.1	
Statements	Mentions
An example with the ShapeComparator and a big knowledge graph would have been good to see	1
Other comments	
The tool somehow looks similar to a version control tool	1

Table 5.1: Results of the qualitative content analysis of the interview for the ShapeComparator

All participants were familiar with RDF knowledge graphs, though only some had prior knowledge of SHACL. In the first part of the interview, which focused on the challenges of comparing SHACL shapes manually or using a text comparison tool, all participants agreed that identifying differences between SHACL shapes using only text files is difficult. Although text comparison tools helped in this process, four participants noted that the text often contains many lines and unneeded information so that the shapes appear quite similar. Additionally, one participant outlined that this method is cumbersome and error-prone.

The responses regarding the manageable size of a SHACL shapes file for manual comparison varied. Five participants said that the process can get complicated quite fast. Three participants mentioned that comparing the test knowledge graph, which consists of seven shapes, was already challenging without additional tools. For a maximum size limit, participants suggested thresholds of ten shapes, one hundred lines of text, or even up to ten thousand lines. However, one participant pointed out that the complexity also depends on the kind and the number of changes. Another one added that, even with a text comparison tool, it could be difficult to track what has already been compared and that users may need to jump back and forth across different parts of the document to find references.

After the ShapeComparator demonstration, all participants agreed that they would prefer using the ShapeComparator over manual comparison methods and that the tool would bring additional value to a knowledge graph engineer.

Regarding the intuitiveness of the ShapeComparator, when participants were shown the overview page with one shape containing a difference, four participants immediately recognized which shape had changed, while two needed assistance. However, all participants agreed that the ShapeComparator makes it easy to spot differences between shapes. Four participants also commented positively on the tool's usability, noting that it looks good and is user-friendly. Three of them highlighted that the process is straightforward and the tool hides unnecessary details. The overview page received positive feedback, as well as the feature that allows the comparison of multiple versions of extracted shapes. Additionally, participants mentioned that the tool saves time and effort, though one participant noted that the interface looks okay but needs some polishing.

Suggested improvements include adding a feature that allows users to view the Turtle file of the graph version and the extracted SHACL shapes, as well as providing a graphical interface for the shapes, similar to the illustration in Figure 5.11. In the case of comparing more than two extracted shapes, it would be helpful to indicate on the overview page which specific versions have differences, as the overview page currently displays only that a change exists, without specifying between which versions. One participant also suggested that integrating Shactor with the ShapeComparator would be beneficial. For larger sets of shapes, a more aggregated view on the overview page was recommended, including collapsing node shapes by default and changing the default filter to changed shapes only.

One negative point raised was that a participant was unsure whether the ShapeComparator would bring additional value if there were a lot of changes. Regarding the interview process, two participants noted that they would need to use the ShapeComparator themselves to assess the usability and that an example with a large knowledge graph would have been interesting.

CHAPTER 6

SHACL-DiffExtractor

This chapter provides an overview of the requirements and the functionalities, along with the algorithm and concepts to answer RQ2. The resulting tool is called the SHACL-DiffExtractor. Furthermore, this chapter presents the implementation and the evaluation, along with the BEAR [4] datasets and the system specifications for the evaluation.

6.1 Requirements and Functionality

As outlined in RQ2 in Section 1.3, the idea behind this research question is to generate SHACL shapes for a subsequent version of a graph by using the changesets between two versions of a graph instead of parsing all triples from the newer version. Given a version of a graph G_1, which is used to generate the SHACL shapes S_1, the objective is to derive the SHACL shapes S_{2*} by utilizing the updates between G_1 and G_2. These changes are represented by two sets: $G^+_{1,2}$, containing all triples added in G_2 that were not present in G_1, and $G^-_{1,2}$, which includes all removed triples, which were part of G_1 but are absent in G_2. The output of the SHACL-DiffExtractor is denoted as S_{2*}, while the actual output from QSE applied to G_2 is denoted as S_2. The goal is for S_{2*} to match S_2.
The advantage of this approach is that it eliminates the need to run QSE on G_2, thereby avoiding reading all triples twice. This method operates under the assumption that the changeset between two graph versions is significantly smaller than the overall graph size. Consequently, iterating over the changed triples should be much faster than iterating over G_2. However, this assumption heavily depends on how much changed between G_1 and G_2. For instance, if G_1 and G_2 are large but only very few triples have changed, the approach is efficient. Conversely, if nearly all triples from G_1 are deleted and many new triples are added in G_2, the method would be impractical. Thus, the efficacy of the SHACL-DiffExtractor is highly dependent on the specific use case. The prerequisites are the changesets $G^+_{1,2}$ and $G^-_{1,2}$. Often, the computation of $G^+_{1,2}$ and $G^-_{1,2}$ is resource-intensive and does not pay off. However, if the changeset is accessible, the graph size

is large, and the changeset is assumed to be relatively small, this approach provides an excellent alternative instead of running QSE on G_2.
As displayed in the architecture diagram in Figure 6.1, the SHACL-DiffExtractor uses the file-based version of QSE. It is crucial to understand that the SHACL-DiffExtractor requires not only the final output from QSE - S_1 as Turtle file and in the form of Java objects - but also the intermediate data generated by QSE. As described in Section 2.2.3, QSE produces various maps during the SHACL shapes generation process, and these maps are reutilized when the SHACL-DiffExtractor is executed. Additionally, the sets $G^+_{1,2}$ and $G^-_{1,2}$ provided as files in N-Triples format are needed for a successful execution.

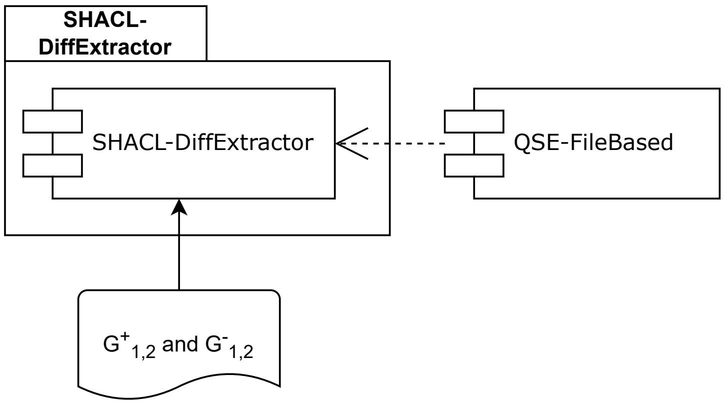


Figure 6.1: SHACL-DiffExtractor - Architecture

To understand the steps involved in the execution process, Figure 6.2 provides an overview. However, it is important to note that the SHACL-DiffExtractor is not restricted to comparing two versions only - it can be applied iteratively across multiple versions as needed. This figure serves as a high-level overview, while the detailed algorithm including an example is discussed in Section 6.2.
It is assumed that QSE has already been executed on a graph version G_1. During this execution, not only the final output in the form of the Turtle file and the Java objects describing the SHACL shapes S_1 have been produced, but also different maps are generated, as described in Section 2.2.3.
All these objects from the initial run are utilized and adapted in the following steps. The progress begins with the first phase, where the entity extraction from QSE is performed for the added triples $G^+_{1,2}$. Next, an algorithm is executed, which at first parses all added triples, followed by parsing the deleted triples. During these steps, the previously mentioned maps are updated accordingly. Afterward, the algorithm runs once more over the deleted triples to check their type predicates. Finally, the support and confidence values are updated.
These internal QSE maps are then used in the second phase of the algorithm. They are saved before any modifications occur, allowing a comparison before and after the adaptations. With this information, deleted shapes can be computed easily. Additionally, an “updates”-map between the old and the new version, with the same structure as the

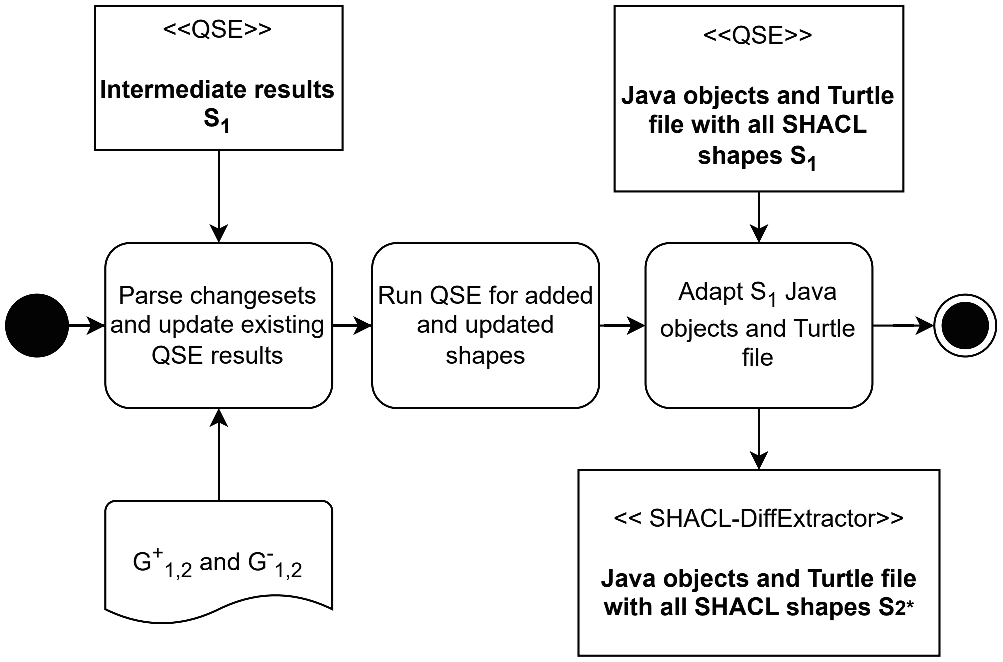

<<QSE>>
Intermediate results S1
<<QSE>>
Java objects and Turtle file with all SHACL shapes S1
Parse changesets and update existing QSE results
Run QSE for added and updated shapes
Adapt S1 Java objects and Turtle file
G+1,2 and G-1,2
<< SHACL-DiffExtractor>>
Java objects and Turtle file with all SHACL shapes S2*

Figure 6.2: SHACL-DiffExtractor - Activity Diagram

"classToPropWithObjTypes" map, is generated. This map only contains information for added and modified shapes. Using this "updates"-map, the file-based version of QSE is executed once again, this time only for added and modified shapes.
In the final step, the original input SHACL shapes S_1, both in the form of Java objects and as Turtle file, are updated so that they become S_{2*}. The algorithm iterates through the newly generated SHACL shapes from QSE in the second phase, adding them to S_1 or replacing modified shapes in S_1. Additionally, any deleted shapes are removed from the file and the Java objects. The updated Java objects and the Turtle file for S_{2*} can then be used for subsequent execution of the SHACL-DiffExtractor, for instance, on the graph version G_3.
Pruning shapes based on support and confidence thresholds can also be applied during this process. The program is available as a command-line tool.

6.2 Algorithms and Concepts

The central concept behind RQ2 is to use the changesets $G^+_{1,2}$ and $G^-_{1,2}$ between two graphs to generate SHACL shapes using QSE. As outlined in Section 6.1, QSE must first be executed on G_1 to produce the SHACL shapes S_1, which are output in the form of a Turtle file and the corresponding Java shape objects. Additionally, this execution generates the intermediate maps Ψ_{CEC} ("classEntityCount"), Ψ_{CTP} ("classToPropWithObjTypes"),Ψ_{ETD} ("entityDataHashMap"), and Ψ_{Support} ("shapeTripletSupport") [84] as mentioned in Section 2.2.3. The task can be broken down into three sub-tasks.

6.2.1 Parse Changesets

First, the changesets in the form of Turtle files are parsed twice and the QSE maps from the first execution are updated. These steps are outlined in Algorithm 6.1. The objective of this algorithm is to modify the QSE maps as if G_2 had been parsed directly. With this approach, QSE can later be run again to generate added or adapted shapes.
The algorithm starts by running the step “entity extraction” from QSE on the added triples, requiring no code modifications for this part. Then, the added triples are processed again. For each triple, the types of the object are recorded (a literal can have only one type, while an IRI may have multiple types). Using this information, the corresponding entry in the $\Psi_{\text{ETD}*}$ map is updated. The objects from QSE denoted with an asterisk (*) indicate that the map has been adapted to differentiate them from the original QSE output. A key modification in QSE for this step is saving a count for each object type in the information for an entity. More details on this topic can be found in Section 4.3. Although QSE does not require this information in a single run, it is essential for correctly handling deletions later on. Finally, the maps $\Psi_{\text{CTP}*}$ and $\Psi_{\text{Support}*}$ can be updated accordingly. During this step, the map “editedShapesMap” is also populated, which will be used in Algorithm 6.2.
A similar algorithm is applied to handle deleted statements. In this task, all triples in $G^-_{1,2}$ are parsed. As in the algorithm for the added triples, the object types for each triple are identified. However, instead of increasing the counter for these object types, the counter is decreased. If the counter reaches zero, the object type is removed entirely. Following this, the maps $\Psi_{\text{CTP}*}$ and $\Psi_{\text{Support}*}$ are updated accordingly. Support values are adjusted and entries are removed from $\Psi_{\text{CTP}*}$ if they are no longer needed. This algorithm mirrors the steps taken for the added triples but includes additional conditions to remove unnecessary entries from the maps when they are no longer required.
As a last step, the algorithm loops through the deleted triples once again, focusing on those with a type predicate, such as “rfd:type”. It then performs any necessary cleanup steps on the maps $\Psi_{\text{ETD}*}$, $\Psi_{\text{CEC}*}$, $\Psi_{\text{CTP}*}$, and $\Psi_{\text{Support}*}$ if needed. Ultimately, the confidence is calculated by running the QSE step “Support and Confidence Computation” by QSE [84]. Only the confidence will be recalculated since the support values have already been updated. At this point, all maps are identical to what they would be if QSE had been run on G_2.

Algorithm 6.1: Parse graph changesets and update QSE maps

```
Input  : $S_{1j}$: Java objects from $S_1$, $G^+_{1,2}$: added triples from $G_1$ to $G_2$, $G^-_{1,2}$:
         deleted triples from $G_1$ to $G_2$, $\Psi_{\text{CEC}}$: “classEntityCount” map from $S_1$,
         $\Psi_{\text{ETD}}$: “entityDataHashMap” from $S_1$, $\Psi_{\text{CTP}}$:
         “classToPropWithObjTypes” map from $S_1$, $\Psi_{\text{Support}}$:
         “shapeTripletSupport” from $S_1$
Output : $\Psi_{\text{CEC}*}$, $\Psi_{\text{ETD}*}$, $\Psi_{\text{CTP}*}$, $\Psi_{\text{Support}*}$, editedShapesMap
1  Function getObjTypes(triple t, Set objTypes):
2      if $t.o$ is Literal then
3          $objTypes.add(getType(t.o))$
4      else
5          foreach $objType \in \Psi_{ETD*}.get(t.o)$ do
6              $objTypes.add(objType)$
7          end
8      end
9  $\Psi_{\text{CEC}*} \leftarrow \Psi_{\text{CEC}}$; $\Psi_{\text{ETD}*} \leftarrow \Psi_{\text{ETD}}$; $\Psi_{\text{CTP}*} \leftarrow \Psi_{\text{CTP}}$; $\Psi_{\text{Support}*} \leftarrow \Psi_{\text{Support}}$
10 Run QSE step “entity extraction” [84] on $G^+_{1,2}$ to update $\Psi_{\text{ETD}*}$ and $\Psi_{\text{CEC}*}$
11 foreach $t \in G^+_{1,2}$ do
12     objTypes $\leftarrow$ new Set<Integer>
13     currentEntityData $\leftarrow$ $\Psi_{\text{ETD}*}$.get(t.s)
14     getObjTypes($t$, $objTypes$)
15     //Add objTypes to currentEntityData
16     currentPropertyData $\leftarrow$ currentEntityData.propertyConstraintsMap.get(t.p)
17     foreach $objType \in objTypes$ do
18         Add $objType$ to currentPropertyData
19         Increase the counter for this $objType$ by 1
20     end
21     //Update $\Psi_{\text{CTP}*}$ and $\Psi_{\text{Support}*}$
22     foreach $objType \in \Psi_{ETD*}.get(t.o)$ do
23         Update editedShapesMap with $objType$ and t.p to keep track of changes
24         Add $objTypes$ to $\Psi_{\text{CTP}*}.get(objType)$ for the key t.p
25         foreach $classObjType \in objTypes$ do
26             if Count for $classObjType$ in $currentPropertyData = 1$ then
27                 Add 1 to the triple $\langle objType, t.p, classObjType \rangle$ in $\Psi_{\text{Support}*}$
28             end
29         end
30     end
31 end
```

```
foreach $t \in G^{-}_{1,2}$ do
    objTypes ← new Set<Integer>; currentEntityData ← $\Psi_{\text{ETD}^*}$.get(t.s)
    getObjTypes($t$, $objTypes$)
    currentPropertyData ← currentEntityData.propertyConstraintsMap.get(t.p)
    foreach $objType \in objTypes$ do
        Decrease the counter for this $objType$ by 1
        if $Counter\ for\ objType = 0$ then
            Remove $objType$ from currentPropertyData
        end
    end
    //Update $\Psi_{\text{CTP}^*}$ and $\Psi_{\text{Support}^*}$
    foreach $objType \in \Psi_{ETD^*}.get(t.o)$ do
        Update editedShapesMap with $objType$ and t.p to keep track of changes
        objTypesToDelete ← new Set<Integer>
        foreach $classObjType \in objTypes$ do
            if $count\ for\ classObjType\ in\ currentPropertyData = 0$ then
                objTypesToDelete.add(classObjType);
            end
        end
        foreach $classObjType \in objTypes$ do
            currentSupportTriple ← $\langle objType, t.p, classObjType \rangle$ in $\Psi_{\text{Support}^*}$
            if $currentPropertyData\ does\ not\ contain\ entry\ for\ classObjType$ then
                Subtract 1 for the triple $currentSupportTriple$
            end
            if $objTypesToDelete.contains(classObjType) \wedge support\ for\ currentSupportTriple = 0$ then
                Delete $currentSupportTriple$
                Remove $objTypesToDelete$ from $\Psi_{\text{CTP}^*}.get(objType)$ for t.p
            end
        end
        if $\Psi_{Support^*}\ does\ not\ contain\ a\ triple\ like\ \langle objType, t.p, \_ \rangle$ then
            $\Psi_{\text{CTP}^*}.get(objType)$.remove(t.p)
            if $\Psi_{CTP^*}.get(objType)\ is\ empty$ then
                $\Psi_{\text{CTP}^*}.remove(objType)$
            end
        end
        Remove entries from $currentEntityData.propertyConstraintsMap$ that have no objTypes left
    end
end
```

```
foreach $t \in G^{-}_{1,2}$ do
    if $t.p = TypePredicate$ then
        currentEntityData ← $\Psi_{\text{ETD}*}$.get(t.s)
        currentEntityData.classTypes.remove(t.o)
        if currentEntityData.classTypes.isEmpty() then
            $\Psi_{\text{ETD}*}.remove(t.s)$
        end
        Decrease $\Psi_{\text{CEC}*}.get(t.o)$ by 1
        if $\Psi_{CEC*}.get(t.o) = 0$ then
            $\Psi_{\text{CEC}*}.remove(t.o)$
            Remove all entries from $\Psi_{\text{Support}*}$ with a triple like $\langle t.o, \_, \_ \rangle$
            $\Psi_{\text{CTP}*}.remove(t.o)$
        end
    end
end
Run QSE step "Support And Confidence Computation" [84] to update $\Psi_{\text{Support}*}$
```

6.2.2 Generate Updated Shapes with QSE

This step aims to utilize the previously adapted maps, denoted with an asterisk (*) to distinguish them from the original maps, to rerun QSE. This procedure is outlined in Algorithm 6.2. It is divided into three sub-tasks. First, the deleted shapes need to be identified by comparing the maps Ψ_{CTP} and $\Psi_{\text{CTP}*}$. The algorithm iterates over the node shapes to detect any deletions. For each node shape, the corresponding property shapes are examined. This process also considers the pruning thresholds for support and confidence. The output is a map, which contains all deleted shapes.
Next, a map is created to capture all added and modified shapes. This map is passed on to QSE later on, to avoid regenerating all shapes. The algorithm achieves this by iterating through $\Psi_{\text{CTP}*}$ and identifying shapes that are not part of Ψ_{CTP}. This process is first applied to node shapes and then to property shapes within each node shape. To accurately capture changes, the algorithm utilizes the "editedShapesMap" which was generated during the first task where added and deleted triples were parsed. This ensures that no shapes are missed, even if triples were added and then deleted. While looping through the "editedShapesMap" all node and property shapes are added to the "updatedShapes" map. It is important to check whether a shape has been deleted in the meantime by using the "deletedShapes" map. After this step, the "updates"-map is complete, with all content finalized.
Now, QSE is run again, but this time only for the "updatedShapes" map. It shares the same structure as Ψ_{CTP} and thus it replaces this map in QSE. As output, QSE generates a Turtle file $S_{1\_2}f$ and the Java node objects $S_{1\_2}j$ for the updated shapes.

Algorithm 6.2: Generate maps to run QSE

```
Input  : $\Psi_{\text{CTP}}$, $\Psi_{\text{CTP*}}$, $\Psi_{\text{CEC*}}$, $\Psi_{\text{Support*}}$, $\omega$: support-threshold, $\epsilon$:
         confidence-threshold, editedShapesMap
Output : deletedShapes, $S_{1\_2}j$, $S_{1\_2}f$
1  deletedShapes ← new HashMap<Integer, Set<Integer>>()
2  foreach nodeShapeEntry $\in \Psi_{CTP}$ do
3      nodeShapeKey ← nodeShapeEntry.getKey()
4      if not $\Psi_{CTP*}$.containsKey(nodeShapeKey) or not
        $\Psi_{CEC*}$.containsKey(nodeShapeKey) or $\Psi_{CEC*}$.get(nodeShapeKey) $< \omega$
        then
5          Add nodeShapeEntry to deletedShapes
6      else
7          foreach propertyShapeId $\in$ nodeShapeEntry.getValue().keySet() do
8              supportKeys ← $\Psi_{\text{Support*}}$ with all triples like
                $\langle nodeShapeKey, propertyShapeId, \_\rangle$
9              maxConfidenceItem ← item from supportKeys in $\Psi_{\text{Support*}}$ with
                the highest confidence
10             if not $\Psi_{CTP*}$.get(nodeShapeKey).containsKey(propertyShapeId) or
                maxConfidenceItem.support $\leq \omega$ or maxConfidenceItem.confidence
                $\leq \epsilon$ then
11                 Add propertyShapeId to deletedShapes.get(nodeShapeKey)
12             end
13         end
14     end
15 end
```

```
$updatedShapes \leftarrow$ new HashMap<Integer, Map<Integer, Set<Integer>>>()
//Add added shapes
foreach $nodeShapeEntry \in \Psi_{CTP^*}$ do
    nodeShapeKey $\leftarrow$ nodeShapeEntry.getKey()
    if not $\Psi_{CTP}.containsKey(nodeShapeKey)$ then
        Add $nodeShapeEntry$ to updatedShapes
    end
    else
        foreach $propertyShapeEntry \in nodeShapeEntry$ do
            if not
             $\Psi_{CTP}.get(nodeShapeKey).containsKey(propertyShapeEntry.getKey())$
             then
                if not $updatedShapes.containsKey(nodeShapeKey)$ then
                    Add a new entry in updatedShapes for nodeShapeKey
                end
                Add propertyShapeEntry to updatedShapes.get(nodeShapeKey)
            end
        end
    end
end
//Add modified shapes
foreach $nodeShapeEntry \in editedShapesMap$ do
    nodeShapeKey $\leftarrow$ nodeShapeEntry.getKey()
    if not $updatedShapes.containsKey(nodeShapeKey)$ then
        Add $nodeShapeEntry$ to updatedShapes
    end
    foreach $propertyShapeId \in nodeShapeEntry.getValue()$ do
        if not $updatedShapes$ contains $propertyShapeId$ and not
         $deletedShapes$ contains $propertyShapeId$ then
            Add $propertyShapeId$ with constraints from $\Psi_{\mathrm{CTP}^*}$ to updatedShapes
        end
    end
    if $updatedShapes.get(nodeShapeKey).isEmpty()$ then
        updatedShapes.remove(nodeShapeKey)
    end
end
Run QSE with updatedShapes instead of the original $\Psi_{\mathrm{CTP}}$
```

6.2.3 Merging Original and Updated Shapes

This last step aims to merge the updated shapes S_{1_2} with the original shapes S_1, resulting in S_{2*}. The output S_{2*} should be identical to S_2, which represents the shapes that would have been generated if QSE had been run directly on G_2. This merging process involves updating the Java shape objects S_{1j} and the Turtle file S_{1f}. Specifically, the added shapes from S_{1_2} must be added to S_1. Also, adapted shapes, for example when the data type of a property shape changes, need to be replaced in S_1. Additionally, deleted shapes must be handled correctly. These steps are outlined in Algorithm 6.3.
The process begins by iterating through the Java shape objects generated by QSE for the updated shapes S_{1_2j}. The algorithm first checks each node shape to determine whether it is an added or edited shape by verifying if a node with the same IRI exists in S_{1j}. If a matching node shape is found, it indicates that changes have occurred, for example, a property shape has been added to this node shape. The program then examines all property shapes within the edited node shape. Again, the algorithm verifies for each property shape, if it was already present in S_{1j}. If this is the case, the old version of the property shape is removed from S_{2j*} and replaced with the updated version generated by QSE. The same procedure is applied to the Turtle file: the old property shape is removed from S_{2f*}, and the newly generated SHACL shape is added instead.
Here, a special condition needs to be addressed. QSE includes a special class, the “PostConstraintsAnnotator”, which annotates shapes with a triple denoted as “node” after all shapes have been generated. This “node” triple assigns each property shape the class it is associated with. For example, if a property shape “knowsPersonShape-Property” points to the class “Person”, the property shape would include a triple “<http://shaclshapes.org/knowsPersonShapeProperty>
<http://www.w3.org/ns/shacl#node> <http://shaclshapes.org/PersonShape>”. However, QSE cannot always add this triple during the generation of the updated shapes, as the corresponding node shape might not have been generated in Algorithm 6.2 if no changes were detected. To handle this, the merging algorithm mimics this behavior by looping through S_{1j} to ensure these “node” triples are added when necessary.
If the property shape was not found in S_{1j} then the Java node object is added to S_{2j*} and the corresponding Turtle text snippet is added to S_{2f*}. Additionally, a reference must be included in the text of the node shape. Once all property shapes have been processed, the Turtle text snippet of the node shape is added to S_{2f*}, replacing the existing one.
Newly added node shapes are handled similarly. The Java shape object from S_{1_2j} is added to S_{2j*}, and the corresponding text snippet from S_{1_2f} gets included to S_{2f*}. Then, all property shapes in this newly added node shape are incorporated into the Turtle file S_{2f*}. Finally, the TurtleFormatter [3] is run on S_{2f*} to ensure that the output is the same as if QSE had been executed. Since the algorithm cannot guarantee the correct order and formatting of SHACL shapes during the merging process, the TurtleFormatter automatically handles this task.
As a result of this process, the generated updated shapes are now added to the original shapes S_1 and adapted shapes have been replaced accordingly. The only task left is to handle deleted shapes. In Algorithm 6.2, these deleted shapes were identified by

comparing Ψ_{CTP} and $\Psi_{\text{CTP}*}$, resulting in the "deletedShapes" map. This map is now used to remove the corresponding Java node shape objects and the Turtle text snippets from S_{2f*} and S_{2j*}. To accomplish this, the algorithm loops through the "deletedShapes" map. A key point to note is that "deletedShapes" only contains encoded numbers rather than fully defined node or property shapes. Therefore, the procedure relies on the "stringEncoder" from QSE, which provides a dictionary to translate the encoded numbers back into IRIs. The first step is to find the actual node shape object in S_{1j} by matching the target class of the node shape with the decoded value from the "stringEncoder". Similarly to the other algorithms, once the node shape is identified, all associated property shape keys are checked. It then locates the according property shape as Java object in S_{1j} by comparing the "path" value. Then the Java object can be deleted from S_{2j*} and the Turtle text snippet can be removed from S_{2f*}. Additionally, the reference in the node shape for the property shape needs to be deleted from the Turtle file. Finally, for each node shape, the algorithm verifies if any property shapes remain. If none are left, the entire node shape can be deleted from both S_{2j*} and S_{2f*}. By completing these steps, the final output - the Java objects as well as the SHACL file in Turtle format - should be identical to what would have been produced if QSE had been run directly on G_2.

Algorithm 6.3: Merge updated shapes with original shapes

```
Input  : $S_{1j}$, $S_{1f}$, $S_{1\_2j}$, $S_{1\_2f}$, deletedShapes, stringEncoder
Output : $S_{2j*}$, $S_{2f*}$
1  $S_{2f*} \leftarrow S_{1f}$
2  $S_{2j*} \leftarrow S_{1j}$
3  //merge shapes
4  foreach $updatedNS \in S_{1\_2j}$ do
5      existingNS ← find node shape in $S_{1j}$ based on IRI
6      if $existingNS$ is not null then
7          $existingNS.support \leftarrow updatedNS.support$
8          $nodeShapeText \leftarrow$ find text for node shape in $S_{1f}$
9          foreach $updatedPS \in updatedNS$ do
10             existingPS ← find property shape in $S_{1j}$ based on IRI
11             if $existingPS$ is not null then
12                 $existingNS.propertyShapes.remove(existingPS)$
13                 $existingNS.propertyShapes.add(updatedPS)$
14                 Delete property shape from $S_{2f*}$
15                 $newPSText \leftarrow$ property shape in Turtle format from $S_{1\_2f}$
16                 $addNodeTriple(newPSText)$
17                 Add $newPSText$ to $S_{2f*}$
18             else
19                 $existingNS.propertyShapes.add(updatedPS)$
20                 $newPSText \leftarrow$ property shape in Turtle format from $S_{1\_2f}$
21                 $addNodeTriple(newPSText)$
22                 Add $newPSText$ to $S_{2f*}$
23                 Add a reference for $updatedPS$ in $nodeShapeText$
24             end
25         end
26         Delete node shape from $S_{2f*}$
27         Add $nodeShapeText$ to $S_{2f*}$
28     else
29         $S_{2j*}.add(updatedNS)$
30         Add node shape in $S_{1\_2f}$ to $S_{2f*}$
31         foreach $updatedPS \in updatedNS$ do
32             Add property shape in $S_{1\_2f}$ to $S_{2f*}$
33         end
34     end
35 end
36 Run TurtleFormatter [3] on $S_{2f*}$
```

```
//Delete shapes
foreach $nodeShapeEntry \in deletedShapes$ do
    $nodeShape \leftarrow$ find node shape in $S_{1j}$ based on target class encoded with the
      stringEncoder from $nodeShapeEntry.getKey()$
    foreach $propertyShapeKey \in nodeShapeKey.getValue()$ do
        $propertyShape \leftarrow$ find property shape in $S_{1j}$ based on path encoded with
          the stringEncoder from $propertyShapeKey$
        Delete $propertyShape$ from $S_{2f*}$
        Delete the reference for $propertyShape$ in node shape from $S_{2f*}$
        $nodeShape.remove(propertyShape)$
    end
    if $nodeShape.propertyShapes.isEmpty()$ then
        Delete $nodeShape$ from $S_{2f*}$
        Remove $nodeShape$ from $S_{2j*}$
    end
end
```

To better grasp the previously discussed algorithms, an example is provided using the People Knowledge Graph. The complete knowledge graph versions can be found in Listing B.2 and Listing B.3. This example focuses on the second and the third version of the knowledge graph to demonstrate the addition and removal of triples. The initial knowledge graph features three people, Alice, Bob, and Jenny, who all know each other. Additionally, the knowledge graph includes three cats. In the subsequent version of the knowledge graph, however, the people no longer know each other but instead know the cats. These changes can be followed in Listing 6.1.

```
Added triples:
<http://example.org/alice> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/orangeCat> .
<http://example.org/bob> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/blackCat> .
<http://example.org/jenny> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/greyCat> .

Deleted triples:
<http://example.org/alice> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/bob> .
<http://example.org/bob> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/alice> .
<http://example.org/jenny> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/alice> .
<http://example.org/blackCat> <http://example.org/color> "black" .
```

```
<http://example.org/greyCat> <http://example.org/color> "grey" .
```

Listing 6.1: Changeset for version 2 and version 3 of the People Knowledge Graph

After running the SHACL-DiffExtractor without any pruning thresholds, the resulting QSE maps are visualized in Figure 6.3, reflecting the steps outlined in Algorithm 6.1. One key change is in the "objTypes" for the "knowsPropertyData" of Alice, Bob, and Jenny. Previously referencing a person (encoded with 0), it now points to the class "Cat" (encoded with 1). Additionally, a new map "objTypeCount" has been introduced in the "PropertyData" class, which tracks the count of each referenced class, as described in Section 4.3. Furthermore, in the "classToPropWithObjTypes" map, the same changes are visible for the "Person" class. The "knows" property (encoded as 6) now references the class "Cat". The "shapeTripletSupport" map has also been updated: the entry for the triple (0,6,0) was removed as it is no longer needed, and instead, a new entry for the triple (0,6,1) was added. Moreover, the support and confidence values for the "color" property of the class "Cat" changed, since two corresponding triples were deleted.
After executing Algorithm 6.2, two maps are generated: the "deletedShapes" map and the "updatedShapes" map. In this case, however, the "deletedShapes" map remains empty, as no shapes were deleted. On the other hand, the "updatedShapes" map, which mirrors the structure as the QSE map "classToPropWithObjTypes", contains two entries: $0 = \{6 = (1)\}$ and $1 = \{7 = (5)\}$. The first entry corresponds to the "knows" property in the "Person" class, while the second refers to the "color" shape in the "Cat" class. Although only the support and confidence values for the "color" shape changed, it is still recreated, since there were changes for this shape. The node shapes "PersonShape" and "CatShape" are recreated as well, as the parent node shape of property shapes are always generated.
In the third phase of the algorithm, outlined in Algorithm 6.3, the updated shapes are merged with the original shapes. Since all shapes already existed, their corresponding Turtle text snippets and the Java objects are replaced. The final result matches exactly the result of QSE if it had been executed on the third version of the People Knowledge Graph, which is given in Listing B.3.

6.3 Implementation

The requirements described in Section 6.1, were implemented by using the algorithms presented in Section 6.2 as a Java command line application in order to answer RQ2. The program first runs QSE on the initial version of the knowledge graph and then executes the above-mentioned algorithms. These contain parsing all added and deleted triples and updating the previously generated internal maps from QSE. Using this updated information, QSE is executed again, but this time, it generates shapes only for the elements where changes or additions occurred. In the final step of the algorithm, these shapes are merged with the shapes generated in the first run. This approach avoids the need to parse all triples from G_2, leading to improved performance. The algorithm also supports handling multiple versions, using the output from the previous run as input for

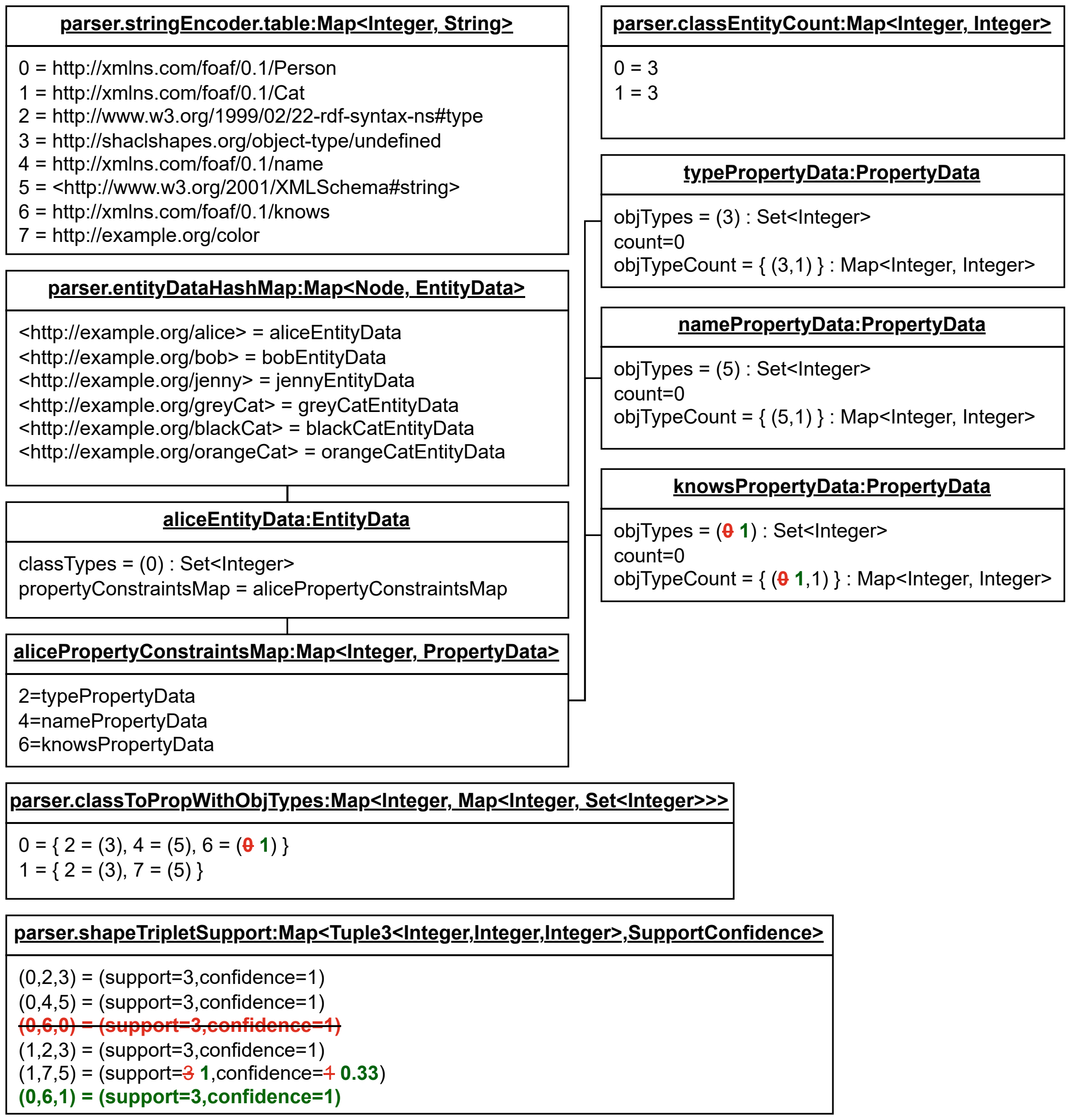

parser.stringEncoder.table:Map<Integer, String>
0 = http://xmlns.com/foaf/0.1/Person
1 = http://xmlns.com/foaf/0.1/Cat
2 = http://www.w3.org/1999/02/22-rdf-syntax-ns#type
3 = http://shaclshapes.org/object-type/undefined
4 = http://xmlns.com/foaf/0.1/name
5 = <http://www.w3.org/2001/XMLSchema#string>
6 = http://xmlns.com/foaf/0.1/knows
7 = http://example.org/color
parser.classEntityCount:Map<Integer, Integer>
0 = 3
1 = 3
typePropertyData:PropertyData
objTypes = (3) : Set<Integer>
count=0
objTypeCount = { (3,1) } : Map<Integer, Integer>
namePropertyData:PropertyData
objTypes = (5) : Set<Integer>
count=0
objTypeCount = { (5,1) } : Map<Integer, Integer>
knowsPropertyData:PropertyData
objTypes = (~~0~~ 1) : Set<Integer>
count=0
objTypeCount = { (~~0~~ 1,1) } : Map<Integer, Integer>
parser.entityDataHashMap:Map<Node, EntityData>
<http://example.org/alice> = aliceEntityData
<http://example.org/bob> = bobEntityData
<http://example.org/jenny> = jennyEntityData
<http://example.org/greyCat> = greyCatEntityData
<http://example.org/blackCat> = blackCatEntityData
<http://example.org/orangeCat> = orangeCatEntityData
aliceEntityData:EntityData
classTypes = (0) : Set<Integer>
propertyConstraintsMap = alicePropertyConstraintsMap
alicePropertyConstraintsMap:Map<Integer, PropertyData>
2=typePropertyData
4=namePropertyData
6=knowsPropertyData
parser.classToPropWithObjTypes:Map<Integer, Map<Integer, Set<Integer>>>
0 = { 2 = (3), 4 = (5), 6 = (~~0~~ 1) }
1 = { 2 = (3), 7 = (5) }
parser.shapeTripletSupport:Map<Tuple3<Integer,Integer,Integer>,SupportConfidence>
(0,2,3) = (support=3,confidence=1)
(0,4,5) = (support=3,confidence=1)
~~(0,6,0) = (support=3,confidence=1)~~
(1,2,3) = (support=3,confidence=1)
(1,7,5) = (support=~~3~~ 1,confidence=~~1~~ 0.33)
(0,6,1) = (support=3,confidence=1)

Figure 6.3: Object diagram - QSE maps after running Algorithm 6.1 on Version 3 of the People Knowledge Graph with Version 2 as basis

the next one.
For this task, all graph versions must be available as files in N-Triples format. Additionally, the changesets between the versions, containing the added and deleted triples, must also be available as files in N-Triples format. The SHACL-DiffExtractor modifies both the Java objects and the resulting SHACL file from the initial QSE run, which can then be used in subsequent executions.
The program can be executed by running the class “shacldiffextractor.ShaclDiffExtractor”

from the command line.
A configuration file named “ConfigShaclDiffExtractor.properties” is available to supply parameters for the execution of the program. The parameters and their descriptions are provided below:

- filePathInitialVersion: This parameter specifies the full file path for the first version, on which QSE will be run. The file must be available in N-Triples format.
- initialVersionName: This parameter defines the name of the dataset for the first version. This allows the graph version to have a different name than the file name.
- pruningThresholds: The pruning thresholds for QSE can be defined here in the format $\{(confidence, support)\}$. These will be considered during the first execution of QSE but also during parsing the changesets. The confidence value must be a decimal number between 0 and 1, while the support value can be any non-negative number (with zero as the default value).
- doMetaComparison: A boolean value (“true” or “false”) is used to determine if the algorithm’s results should be compared to the baseline. The baseline runs QSE on all versions provided in the configuration file, ignoring the changesets. If set to “true”, QSE will be executed in parallel, producing results from both the SHACL-DiffExtractor and QSE, along with a comparison of the two methods. If the parameter is set to “false”, only the SHACL-DiffExtractor will be run.
- A list of knowledge graph versions, which the SHACL-DiffExtractor will check, is specified. Since there are multiple file paths needed for each version, the following configuration properties must be prefixed with the version name to group the corresponding file paths. The SHACL-DiffExtractor will be run on each of these versions, using the QSE results from the previous execution as a basis. All files must be available in N-Triples format. For each version, the following parameters must be provided:
  - <versionName>.filePathAdded: This parameter specifies the full file path for the added triples between the previous version and this version.
  - <versionName>.filePathDeleted: This parameter contains the full file path for the deleted triples between the previous version and this version.
  - <versionName>.filePathFullVersion: This parameter is optional, it is only required when the parameter “doMetaComparison” is set to “true”. It provides the file path for the full knowledge graph version.

A key implementation detail involves adapting QSE, as discussed in Section 4.3. The implemented version of the “shactor” branch had to be updated, so that the maps, the internal methods, and variables could be used by the SHACL-DiffExtractor. Additionally, the structure of the “PropertyData” class was revised, to fulfill the requirements for

handling added and deleted triples.
The results generated by the SHACL-DiffExtractor can be found in the project's output directory. Comparison results are saved as text files, while SHACL shapes are stored in Turtle format. Depending on the configuration, the SHACL-DiffExtractor saves various outputs, including the QSE results for all provided versions, the QSE results specifically for added and updated shapes between versions, and the final resulting SHACL shapes from the SHACL-DiffExtractor. For the meta-comparison, the SHACL-DiffExtractor generates its own comparison output, detailing added, deleted, and modified shapes. The same comparison is generated for the baseline. The meta-comparison is then derived by comparing these two sets of results.
A further improvement for this tool includes an integration with the ShapeComparator. In this case, the database mentioned in Section 5.2.2 could also be reused to store generated shapes and graph versions.

6.4 Evaluation

This section begins by describing the test data and system information for the machine used during the evaluation. This information applies to the evaluations of both the SHACL-DiffExtractor and the SPARQL-ShapeChecker, which is detailed in Section 7.4. Lastly, the evaluation results of the SHACL-DiffExtractor are presented.

6.4.1 System Information and Test Data

The SHACL-DiffExtractor and the SPARQL-ShapeChecker were tested on a Linux Virtual machine with Debian (Version 11), 8 CPU cores, a speed of 2200 MHz, 128 GB of RAM, and 780 GB of disk space available.

Dataset	BEAR-A	BEAR-B (daily)	BEAR-C
Number of versions	58	89	32
Triples in first version	30 million	33,502	485,179
Triples in last version	66 million	43,907	563,738
Growth	101%	100.744%	100.478%
Change ratio	31%	1.778%	67.617%
Change ratio - adds	33%	1.252%	33.671%
Change ratio - deletes	27%	0.526%	33.946%
Static core	3.5 million	32,448	178,484
Version-oblivious triples	376 million	83,134	9,403,540

Table 6.1: Statistics of the BEAR datasets

The data used during the evaluation is sourced from BEAR [4] which was designed for testing archiving and querying of evolving semantic web data. In this thesis, these datasets are used in the evaluation section of the SHACL-DiffExtractor, as discussed in Section 6.4.2, and for assessing the SPARQL-ShapeChecker, as detailed in Section 7.4. The BEAR datasets, collectively known as the **BE**nchmark of RDF **AR**chives (BEAR) consist of three real-world datasets. The BEAR data offers valuable statistics and provides various methods for accessing the data, such as having one N-Triples file per version.
Table 6.1 provides basic statistics for the BEAR datasets. The BEAR-A dataset, which contains snapshots from the Dynamics Linked Data Observator is very large and homogeneous, with one file reaching 5 GB. As Figure 6.4 shows, the data growth between the versions is marginal, except for the latest versions, where a lot of data is added.
The BEAR-B data has been compiled from DBpedia Live Changesets over three months (August to October 2015), including the 100 most volatile resources along with their updates. For BEAR-B, there is an instant, an hourly, and a daily version available. The instant version contains a new version for every change that has been made, but for this project, the daily changesets were used since there are already 89 different versions available. Unfortunately, there is no graph with the number of statements available for this dataset, however, the dataset grows continuously.
The data from BEAR-C (Europeans Open Data portals) is very homogeneous. It contains a version for each week, therefore the dataset describes 32 weeks of data. The number of statements per version is printed in Figure 6.5. The growth is very limited since most of the updates are modifications on the metadata with similar numbers for additions and deletions.
For testing, the data from BEAR-B and BEAR-C was used. While the datasets from BEAR-A were too large, the data from BEAR-B has the disadvantage that it includes multiple classes with the same name but a different IRI. QSE has limitations in this area, as described in Section 2.2.3 which can lead to confusing outputs. The data from BEAR-C is very homogeneous, therefore not many differences can be seen.

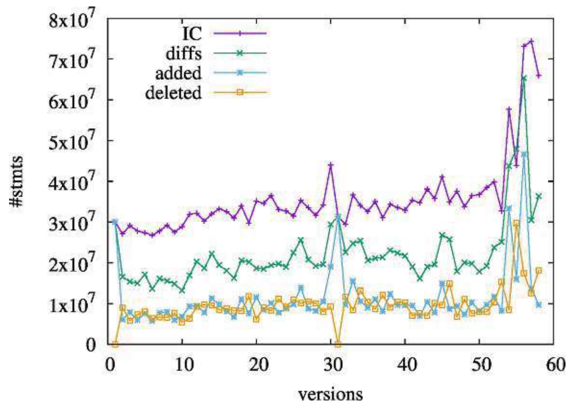


Figure 6.4: BEAR-A: Number of statements per version [4]

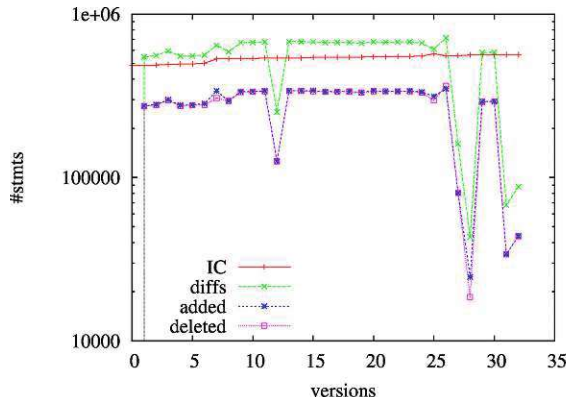


Figure 6.5: BEAR-C: Number of statements per version [4]

6.4.2 Evaluation of the SHACL-DiffExtractor

The evaluation of the SHACL-DiffExtractor was conducted by comparing its performance to running QSE for the specified versions of the graph which is the baseline. The measurable characteristic used for this comparison is the execution time, which is measured using the "Instant" class from the "java.time" package [14]. This class allows capturing an instantaneous point in time. A timestamp is recorded and saved into a variable before starting the execution of the code to be measured, and another timestamp is recorded afterward. The duration between these two timestamps represents the execution time for a specific step.
The execution time for the baseline includes the time taken by QSE to generate SHACL shapes for the initial version and the following version, as well as the time used to compare the resulting SHACL shapes of the two QSE runs, as illustrated in Equation 6.1. For the SHACL-DiffExtractor, the execution time includes the time to run QSE on the initial version, the time needed to execute the SHACL-DiffExtractor, and the time spent comparing the initial run's results with those produced by the SHACL-DiffExtractor. This calculation is detailed in Equation 6.2. The comparison process of the resulting SHACL shapes considers added, edited, and deleted shapes.

$$\Delta t_{\text{Baseline}} = t_{\text{QSE\_n}} + t_{\text{QSE\_n+1}} + t_{\text{Comparison}} \tag{6.1}$$

$$\Delta t_{\text{RQ2}} = t_{\text{QSE\_n}} + t_{\text{SHACL-DiffExtractor\_n+1}} + t_{\text{Comparison}} \tag{6.2}$$

Based on the comparison between the results, a meta-comparison can be performed to evaluate the content changes between the baseline and the SHACL-DiffExtractor. Comparing SHACL shapes was handled by the procedure described in Section 4.1. The identification of added and deleted SHACL shapes is achieved by comparing the resulting shapes. For edited shapes, the algorithm described in Algorithm 6.3 additionally monitors the names of shapes that have been modified. The results of this meta-comparison are saved in a text file located in the project directory, with an example provided in Listing 6.2. In this example, the added and deleted shapes from the baseline and the SHACL-DiffExtractor match. However, one property shape was modified in the second QSE run but not by the SHACL-DiffExtractor, while two property shapes were edited by the SHACL-DiffExtractor but not by QSE. This approach makes it easy to detect discrepancies between the baseline and the SHACL-DiffExtractor, which helps to verify the correctness of the SHACL-DiffExtractor.

```
===== Comparison of Compare-Methods =====
==== Added Node Shapes ====
==== Added Property Shapes ====
==== Deleted Node Shapes ====
==== Deleted Property Shapes ====
==== Edited Node Shape Names ====
==== Edited Property Shape Names ====
    == Unique in QSE-Comparison (Count = 1) ==
        http://shaclshapes.org/someProperty
    == Unique in SHACL-DiffExtractor-Comparison (Count = 2) ==
        http://shaclshapes.org/anotherProperty
        http://shaclshapes.org/differentProperty
Execution Time QSE Total: 68 seconds
Execution Time SHACL-DiffExtractor Total: 32 seconds
```

Listing 6.2: Result of a meta comparison between the SHACL-DiffExtractor and QSE

The evaluation of the execution times was carried out on the virtual machine and with the test data described in Section 6.4.1. The algorithm was primarily assessed with the BEAR-C dataset, as well as the BEAR-B dataset. However, the dataset BEAR-B is less ideal for this evaluation due to its numerous shapes, with the same name but different IRIs, as described in Section 2.2.3. The evaluation involved comparing the initial seven versions sequentially for both datasets (e.g. V1 - V2, V2 - V3...). The changesets between the versions are provided by the BEAR datasets. For each version, two files are available which contain the added triples and the deleted triples between two consecutive versions. The execution times for this evaluation are presented in Table 6.2. To ensure the accuracy of the SHACL-DiffExtractor, the changes in shapes were analyzed for each run by the meta-comparison. The results, shown in Table 6.2, reveal that the SHACL-DiffExtractor is only sometimes faster than the baseline and, in some cases, is actually slower. This can be explained with the statistics for the datasets described in Table 6.1. While the versions of the BEAR-C dataset are larger and have a higher change ratio of around 68%, the BEAR-B dataset is smaller with a change ratio of only about 2%. This suggests that the algorithm performs better with lower change ratios. However, significant performance improvements are evident when the change ratio is small. For the BEAR-B dataset, a performance increase of 37% is observed.
As Table 6.3 shows, there are some cases, where the SHACL-DiffExtractor and the baseline do not produce the same results. In the BEAR-C dataset, the issue consistently arises with the same shape. In a node shape, there are two classes referenced with the same name but a different IRI. This leads to discrepancies in one of the property shapes. While the changes are correctly calculated, a part of the property shape is deleted during

the merge process. Internally, QSE generates two node shape objects in this specific case, even though there is only one node shape in the Turtle file, which causes the differences. For the BEAR-B dataset, multiple issues lead to the changes. Between versions BEAR-B-1 and BEAR-B-2, the differences in the EPB column (edited property shapes unique in QSE) can be explained by an error in the dataset. There is an entity (http://dbpedia.org/resource/Revival_(Selena_Gomez_album)), which is associated with nine types (hence, nine shapes are affected). In version BEAR-B-1, this entity includes a statement with the property "http://dbpedia.org/ontology/wikiPageID", but this statement is missing in version BEAR-B-2. Unfortunately, this change is not recorded in the changeset for the deleted triples. As a result, QSE detects the discrepancies, while the SHACL-DiffExtractor does not. The change is that the support for the property shape no longer matches the support for the node shape, causing the "minCount=1" requirement to be removed.
The differences between BEAR-B-1 and BEAR-B-2 in the EPS column (edited property shapes for the SHACL-DiffExtractor) can be explained by the fact that all the property shapes showing changes have another property shape with the same name but a different IRI. In this specific case, the affected property shapes target the property "name" but for each of these property shapes, there is an additional property shape denoted as "name_1" in the node shape. One property shape targets the IRI "http://xmlns.com/foaf/0.1/name", while the other shape targets the IRI "http://dbpedia.org/property/name". When QSE is executed again, it does not always assign the same name to a shape for a given IRI, which can lead to swapping the shapes. For example, if a shape with the "_1" suffix changes but the one without the suffix remains the same, and QSE recalculates it during Algorithm 6.2, the updated name may no longer include the suffix. As a result, the original shape without the suffix may be overwritten. Alternatively, the shapes may be swapped between the SHACL-DiffExtractor and the QSE version, causing them to appear as changes. This issue also explains the differences between versions BEAR-B-2 and BEAR-B-3.
As discussed in Section 2.2.3, a potential solution to this problem is to revise the method of generating SHACL shape names to ensure that no shape addresses multiple IRIs.

Version 1	Version 2	Baseline [seconds]	SHACL-DiffExtractor [seconds]
BEAR-C-1	BEAR-C-2	11	11
BEAR-C-2	BEAR-C-3	8	8
BEAR-C-3	BEAR-C-4	7	7
BEAR-C-4	BEAR-C-5	6	7
BEAR-C-5	BEAR-C-6	7	7
BEAR-C-6	BEAR-C-7	6	6
BEAR-B-1	BEAR-B-2	68	34
BEAR-B-2	BEAR-B-3	55	19
BEAR-B-3	BEAR-B-4	54	21
BEAR-B-4	BEAR-B-5	53	17
BEAR-B-5	BEAR-B-6	54	18
BEAR-B-6	BEAR-B-7	54	18

Table 6.2: Execution times for the baseline and the SHACL-DiffExtractor

Version 1	Version 2	ANB	ANS	APB	APS	DNB	DNS	DPB	DPS	ENB	ENS	EPB	EPS
BEAR-C-1	BEAR-C-2	0	0	0	0	0	0	0	0	0	0	0	1
BEAR-C-2	BEAR-C-3	0	0	0	0	0	0	0	0	0	0	0	0
BEAR-C-3	BEAR-C-4	0	0	0	0	0	0	0	0	0	0	0	1
BEAR-C-4	BEAR-C-5	0	0	0	0	0	0	0	0	0	0	0	1
BEAR-C-5	BEAR-C-6	0	0	0	0	0	0	0	0	0	0	0	1
BEAR-C-6	BEAR-C-7	0	0	0	0	0	0	0	0	0	0	0	1
BEAR-B-1	BEAR-B-2	0	0	0	0	0	0	0	0	0	0	9	7
BEAR-B-2	BEAR-B-3	0	0	0	0	0	0	0	0	0	0	34	12
BEAR-B-3	BEAR-B-4	0	0	0	0	0	0	0	0	0	0	0	0
BEAR-B-4	BEAR-B-5	0	0	0	0	0	0	0	0	0	0	0	0
BEAR-B-5	BEAR-B-6	0	0	0	0	0	0	0	0	0	0	0	0
BEAR-B-6	BEAR-B-7	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.3: Changes in shapes between the baseline and the SHACL-DiffExtractor. Legend: A (added), D (deleted), E (edited), N (node shape), P (property shape), B (baseline), S (SHACL-DiffExtractor) e.g. DNB (deleted node shapes in the baseline)

CHAPTER 7

SPARQL-ShapeChecker

This chapter outlines the requirements and functionalities, algorithms, implementation, and evaluation of the SPARQL-ShapeChecker, addressing research question 3.

7.1 Requirements and Functionality

As described in RQ3 in Section 1.3, the primary goal of the described algorithm is to enhance the speed of QSE for evolving knowledge graphs. In this context, QSE may have already been executed on a previous version of a graph G_1, resulting in the SHACL shapes S_1, and the goal is to obtain S_2 for a subsequent version G_2 of a graph.
As noted in Section 2.2.3, QSE can be applied to graphs either available as files or stored in a graph database such as GraphDB. For this algorithm, the query-based option is utilized, where the graph is saved in a graph store and QSE processes the graph using SPARQL queries.
Given that changes in the schema between graph versions are often minimal, it is more efficient to validate the existing shapes S_1 rather than to reprocess all triples in the graph G_2, as QSE does. The reason for this is that the shapes graph, which defines the schema, is usually significantly smaller compared to the data graph. For example, a graph containing information about hundreds or thousands of people and their names can be considered. In contrast, the shapes graph that describes the schema of the individuals might only include a few shapes. In this example, the shapes graph would include a node shape for the class "Person" and a property shape for the "name" attribute. Thus, while the data graph grows substantially in size with the addition of more individuals, the shapes graph would remain the same. Although there are exceptions, such as small data graphs with only a few entries, processing the shapes graph S_1 is generally more efficient than iterating over the whole graph G_2 which can be computationally expensive. This approach leverages the fact that changes in the schema are infrequent, so reevaluating the shapes directly can quickly determine whether the graph G_2 adheres to the existing

schema constraints without redundant reprocessing.
The validated shapes produced by the SPARQL-ShapeChecker are denoted as S_{2*} in this context whereas the shapes generated if QSE would be run on G_2 are denoted as S_2. The main advantage of this approach is that it is faster than running QSE on G_2. However, the drawback of this solution is that the shape recognition is incomplete because new shapes that were not present in S_1 cannot be identified which means that $S_{2*} \neq S_2$. Therefore, the goal of this method is not to generate S_2, it rather aims to balance runtime efficiency with the accuracy of the results. The SPARQL-ShapeChecker is therefore mainly useful for large graphs where shapes extraction performance is particularly important and where the detection of new shapes is secondary.
The software architecture for this solution is shown in Figure 7.1. The SPARQL-ShapeChecker needs the query-based version of QSE to obtain S_1 and it also needs access to G_2 which must be stored on a triplestore like GraphDB.

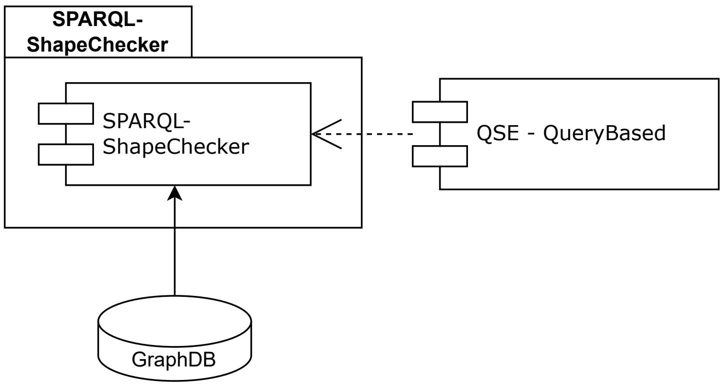


Figure 7.1: SPARQL-ShapeChecker - Architecture

The main steps in this algorithm are illustrated in Figure 7.2. The process uses the Java objects representing all shapes from S_1 as input. These objects are simultaneously created with the SHACL file in Turtle format by QSE, as described in Section 2.2.3. Each shape in S_1 is checked for its relevance in G_2 by using SPARQL queries. Specifically, the algorithm verifies whether all node and property shapes, along with the property shapes' constraints, are still satisfied in G_2 by updating the corresponding support and confidence values. Users can specify the support and confidence parameter thresholds, and the algorithm checks if these thresholds continue to be met in G_2. If the support and confidence parameters are not specified, the algorithm checks if any instances of the graph still meet the shapes, which corresponds to a support value of zero. Based on this verification, i.e. the updated support and confidence values, the algorithm copies and modifies the Turtle file containing all SHACL shapes S_1, removing any parts that are no longer valid. This resulting file is denoted as S_{2*f}. For the modification of S_{2*f}, regular expressions are used. The resulting output is a new Turtle file with the shapes S_{2*}. The detailed algorithm for this process is described in Section 7.2.
There is also the option to use the SPARQL-ShapeChecker with multiple versions. In this scenario, the user can specify the initial version of the graph on which QSE should

be executed, along with a list of versions that the SPARQL-ShapeChecker validates.

7.2 Algorithms

The task of checking SHACL shapes in a subsequent version of a graph as described in Section 7.1 can be divided into two main steps. In the first step, the generated SHACL shapes in the form of Java objects from the initial QSE run are used as input. The algorithm loops through these objects and checks the constraints in the second version of the graph. As outlined in Section 2.2.3, the objects contain node and property shapes, as well as the constraints saved within each property shape. At first, a SPARQL query is created, which returns the class count for all classes mentioned in the node shapes in the Java objects. With this result, the support values of the node shapes can be updated. The algorithm then iterates through the property shapes for each node shape, constructing a SPARQL query depending on the property shapes' attributes. For instance, the query differs slightly if the property shape targets a literal versus an IRI. If the property shape contains multiple constraints, the algorithm also loops through these and updates the support and confidence values accordingly. The output from this step consists of the same objects as those provided by QSE in the first run but with updated support and confidence values. The pseudo-code for this algorithm is provided in Algorithm 7.1.
In the second step, which is illustrated in Algorithm 7.2, the adapted objects from the first step are iterated through. The input for this step is the SHACL shapes file from the first QSE run, and the output is also saved as a Turtle file, with the updated SHACL shapes. During this process, any objects (including node shapes, property shapes, and constraints in property shapes) that do not meet the thresholds for support or confidence

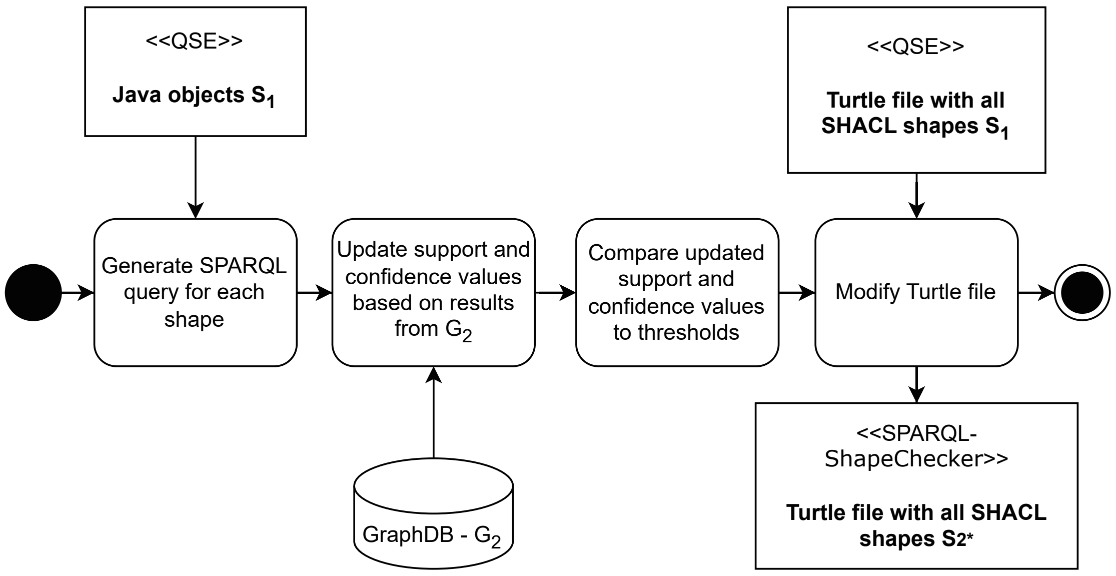

<<QSE>>
Java objects S1
<<QSE>>
Turtle file with all SHACL shapes S1
Generate SPARQL query for each shape
Update support and confidence values based on results from G2
Compare updated support and confidence values to thresholds
Modify Turtle file
GraphDB - G2
<<SPARQL-ShapeChecker>>
Turtle file with all SHACL shapes S2*

Figure 7.2: SPARQL-ShapeChecker - Activity Diagram

Algorithm 7.1: Validate existing shapes with SPARQL

```
Input   : $S_{1j}$: Java objects from $S_1$, $G_2$: second version of graph
Output  : $S_{2*j}$: Adapted Java objects for $G_2$
1 $S_{2*j} \leftarrow S_{1j}$
2 Run SPARQL query on $G_2$ which returns all counts for all classes mentioned in
    $S_{1j}$
3 Update the support values for the node shapes in $S_{2*j}$
4 foreach node shape $\langle s, \tau_n, \Phi_n \rangle \in S_{2*j}$ do
5     foreach property shape $\phi_n : \langle \tau_p, T_p, C_p \rangle \in \Phi_n$ do
6         Run SPARQL query on $G_2$ with given nodeKind and $\tau_p$ for all $T_p$ which
            returns the count for the specific constraint and update the support and
            confidence values for $\phi_n$ in $S_{2*j}$
7     end
8 end
9 return $S_{2*j}$
```

are deleted. If these thresholds are not specified, then all shapes with a support of zero are removed. The algorithm begins by examining the node shapes. If a node shape does not meet the thresholds, it is deleted along with all property shapes that belong to it. The deletion, just as the selection of shapes which was described in Section 4.1, works per name and is completed with regex. Next, the algorithm processes the property shapes associated with each node shape. If a property shape does not meet the thresholds, it is removed from the file just as the triple in the node shape that references the property shape. This part of the algorithm utilizes regular expressions as well. If a property shape contains multiple constraints, the algorithm iterates through these. It checks their support and confidence values, removing any that do not meet the thresholds. This step also uses regular expressions. However, since the constraints in a property shape lack names like those of node or property shapes, the regex searches for constraints using keywords such as “NodeKind”, “class”, or “datatype”, which values are stored in the Java objects. This process is adjusted based on the constraint’s attributes, as different SHACL terms are used depending on whether the constraint targets an IRI or a literal. There are specific edge cases to consider for the deletion of shapes. One such case occurs when there is a SHACL-Or list with multiple constraints in the first version of a property shape but there is only a single constraint in the second version left because the other constraints did not meet the thresholds and were therefore deleted. In this scenario, the list must be removed. For this task, the shape is converted to a Jena model, and the algorithm then queries the Jena model to find the list and its sole remaining entry. Next, it creates RDF triples for each triple within the list item but uses the property shape as a subject, rather than the list itself. Subsequently, the list and all associated triples are removed from the Jena model. The algorithm also generates new triples for support and confidence in the property shape, using the values specified in the constraint. Finally, the updated Jena model is converted back into a string using the TurtleFormatter library [3].

In this way, it can be ensured that the created shape has the same format as all other property shapes created by QSE in the Turtle file. Another edge case arises when the support values of a node shape and a property shape are no longer equal. In this case, it is necessary to remove the SHACL-minCount triple if it was present in the initial version of the shape. The reason why this value is removed rather than adapted is that QSE only creates SHACL-minCount triples with a value of one and disregards all other cases. The result of this second step is a Turtle file containing all validated SHACL shapes. This file serves as the basis for comparison during the evaluation phase. It is essential that the shapes in the file match exactly the format produced by QSE. The comparison algorithm compares the Turtle snippets of each shape as described in Section 4.1. If these snippets differ in formatting or text but have the same semantic meaning, the comparison algorithm may incorrectly determine that the shapes are not identical. Therefore, the formatting of the SHACL shapes in the Turtle file is crucial.
An example is provided for the graphs shown in Listings 7.1 and 7.2. In the first version of the graph, there is a person named Alice, aged 25. However, in the second version of the graph Alice's age is no longer mentioned. After running QSE on version one, the resulting Java objects, are visualized in Figure 7.3. A node shape for the class "Person" is created, along with two property shapes: one for the property "name" and one for the property "age". Only relevant properties are displayed in this figure. Additionally, the SHACL shapes are generated by QSE and saved as a file, as shown in Listing 7.3.

```
<http://example.org/alice>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
<http://xmlns.com/foaf/0.1/name> "Alice" .
<http://example.org/alice> <http://xmlns.com/foaf/0.1/age>
  "25"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Listing 7.1: Example graph - Version 1

```
<http://example.org/alice>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
<http://xmlns.com/foaf/0.1/name> "Alice" .
```

Listing 7.2: Example graph - Version 2

Algorithm 7.2: Delete unvalidated SHACL shapes from file

```
Input  : $S_{2*j}$: Adapted Java objects from $S_1$ for $G_2$, $S_{1f}$: Turtle file with all
         SHACL shapes $S_1$, $\omega$: support-threshold, $\epsilon$: confidence-threshold
Output : Turtle file with updated SHACL shapes $S_{2*f}$
1  $S_{2*f} \leftarrow S_{1f}$
2  foreach node shape $\langle s, \tau_n, \Phi_n \rangle \in S_{2*j}$ do
3      if support of node shape $\leq \omega$ then
4          delete node shape from $S_{2*f}$ with regex based on $s$
5          delete all associated property shapes $\Phi_n$ from $S_{2*f}$ with regex
6      else
7          foreach property shape $\phi_n : \langle \tau_p, T_p, C_p \rangle \in \Phi_n$ do
8              if support of $\phi_n \leq \omega$ or confidence of $\phi_n \leq \epsilon$ then
9                  delete $\phi_n$ from $S_{2*f}$ with regex
10                 delete triple in node shape that makes $\phi_n$ part of the node shape
11             else
12                 if $|T_p| > 1$ then
13                     foreach constraint in $\phi_n$ do
14                         if support of constraint $\leq \omega$ or confidence of constraint $\leq \epsilon$
                            then
15                             Delete constraint from the property shape from $S_{2*f}$
                                with regex
16                         end
17                     end
18                     if only one constraint is left after deletion then
19                         Remove the SHACL-Or list from the property shape and
                            connect the constraint directly to the property shape
20                     end
21                 end
22                 if support of node shape $\neq$ support of $\phi_n$ then
23                     Remove the SHACL-MinCount triple from property shape from
                        $S_{2*f}$ with regex
24                 end
25             end
26         end
27     end
28 end
29 return $S_{2*f}$
```

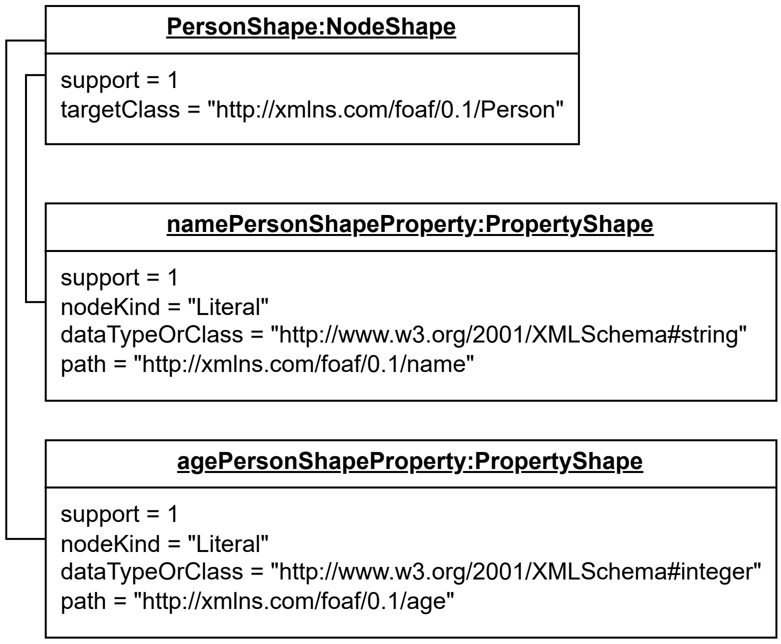

PersonShape:NodeShape
support = 1
targetClass = "http://xmlns.com/foaf/0.1/Person"
namePersonShapeProperty:PropertyShape
support = 1
nodeKind = "Literal"
dataTypeOrClass = "http://www.w3.org/2001/XMLSchema#string"
path = "http://xmlns.com/foaf/0.1/name"
agePersonShapeProperty:PropertyShape
support = 1
nodeKind = "Literal"
dataTypeOrClass = "http://www.w3.org/2001/XMLSchema#integer"
path = "http://xmlns.com/foaf/0.1/age"

Figure 7.3: Object diagram - QSE Java objects

```
<http://shaclshapes.org/PersonShape> rdf:type
  <http://www.w3.org/ns/shacl#NodeShape> ;
    <http://shaclshapes.org/support> "1"^^xsd:int ;
    <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/agePersonShapeProperty> ;
    <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/namePersonShapeProperty> ;
    <http://www.w3.org/ns/shacl#targetClass>
    <http://xmlns.com/foaf/0.1/Person> .

<http://shaclshapes.org/agePersonShapeProperty> rdf:type
  <http://www.w3.org/ns/shacl#PropertyShape> ;
    <http://shaclshapes.org/confidence> 1E0 ;
    <http://shaclshapes.org/support> "1"^^xsd:int ;
    <http://www.w3.org/ns/shacl#NodeKind>
    <http://www.w3.org/ns/shacl#Literal> ;
    <http://www.w3.org/ns/shacl#datatype> xsd:integer ;
    <http://www.w3.org/ns/shacl#minCount> 1 ;
    <http://www.w3.org/ns/shacl#path>
    <http://xmlns.com/foaf/0.1/age> .

<http://shaclshapes.org/namePersonShapeProperty> rdf:type
```

```
    <http://www.w3.org/ns/shacl#PropertyShape> ;
      <http://shaclshapes.org/confidence> 1E0 ;
      <http://shaclshapes.org/support> "1"^^xsd:int ;
      <http://www.w3.org/ns/shacl#NodeKind>
      <http://www.w3.org/ns/shacl#Literal> ;
      <http://www.w3.org/ns/shacl#datatype> xsd:string ;
      <http://www.w3.org/ns/shacl#minCount> 1 ;
      <http://www.w3.org/ns/shacl#path>
      <http://xmlns.com/foaf/0.1/name> .
```

Listing 7.3: SHACL shapes as result from QSE, deleted parts are indicated in italics

Using the SPARQL-ShapeChecker, Algorithm 7.1 is run on version two of the graph, using the Java objects as input. After this step, the support value of the property shape “agePersonShapeProperty” is set to zero, since Alice’s age is deleted in the second version of the graph. With these updated Java objects Algorithm 7.2 is then executed. Since there is an object with a support of zero, the algorithm takes the SHACL file as input (Listing 7.3) and uses regex to search for the shape “agePersonShapeProperty”, subsequently deleting it from the SHACL file. Furthermore, the algorithm removes the reference in the PersonShape with regex as indicated by the italic text in Listing 7.3.

7.3 Implementation

The requirements of the SPARQL-ShapeChecker, outlined in Section 7.1, were implemented using the algorithms detailed in Section 7.2 in Java as a command line application to answer RQ3. The program executes QSE on the first version of the graph and then performs the validation and deletion steps as described in detail in Section 7.2. For this task, all graph versions must be available on a triplestore like GraphDB. As input for the SPARQL-ShapeChecker, both the Java objects and the resulting SHACL file from the initial QSE run are used. The Java objects represent the node and property shapes as objects, while the SHACL file provides the same information in Turtle format. In the first step of the algorithm, the validation phase, the support and confidence values in the Java objects are updated. These new support and confidence values are calculated using SPARQL queries on the consecutive versions of the graph. Next, the algorithm adapts the SHACL shapes in the given Turtle file by using regular expressions, removing shapes that no longer meet the thresholds for support or confidence.
When the program is run from the command line, a config file named “ConfigSparql-ShapeChecker.properties” is available to provide parameters for the execution of the SPARQL-ShapeChecker. The parameters and their description are outlined below:

- graphDbUrl: The URL through which the graph is accessible on a triple store. For example, if GraphDB is hosted locally, it could be something like http://localhost:7200/.

- dataSetNameQSE: The name of the dataset for the first version. QSE is executed on this version, and the results form the basis for the SPARQL-ShapeChecker. The algorithm checks the URL provided via the graphDBUrl parameter and searches for a dataset with this name.

- dataSetsToCheck: A list of version names available via the graphDbUrl parameter should be given here. The SPARQL-ShapeChecker will be executed on each of these versions, with the QSE results from the version of the parameter dataSetNameQSE as a basis. There can be one or multiple dataset names, separated by commas.

- pruningThresholds: As used by QSE, the pruning thresholds can be defined here. They must be provided in the format $\{(confidence, support)\}$. The confidence value should be a decimal number between 0 and 1, while the support value can be any non-negative number. Zero is the default value.

- doMetaComparison: A boolean value ("true" or "false") is used for comparing the algorithm to QSE. If it is set to "true", QSE will be run in parallel and output the results from both the SPARQL-ShapeChecker and QSE, along with a comparison of the two methods. If the parameter is set to "false", only the output from the SPARQL-ShapeChecker will be generated.

The program can be executed by running the class "sparqlshapechecker. SparqlShapeChecker" from the command line.
Similar to the SHACL-DiffExtractor the results produced by the SPARQL-ShapeChecker are located in the project's output directory. Based on the configuration, the SPARQL-ShapeChecker saves the QSE results for all provided versions, the final SHACL shapes generated by the SPARQL-ShapeChecker, as well as comparisons for the SPARQL-ShapeChecker itself, the baseline, and the meta-comparison.
A further improvement for this tool includes an integration with the ShapeComparator. In this case, the database mentioned in Section 5.2.2 could also be reused to store generated shapes and graph versions. Using the example provided in Section 7.2, a successful demonstration of the SPARQL-ShapeChecker is shown. In this example, a triple was removed in the second version of the graph which led to the removal of a property shape from the SHACL file without the need to iterate over all triples in the second version of the graph. This leads to a significant reduction in execution time as the algorithm only needs to process the generated SHACL shapes.
Using the example provided in Subsection 5.2.1, a more challenging use case can be discussed. When version one is used as the basis and the SPARQL-ShapeChecker is executed on the second version of the graph, the limitations of not considering newly added triples and shapes become apparent. In this scenario, only new entities (e.g. cats and their colors) are added to the second version of the graph. Consequently, the SHACL shapes produced by the SPARQL-ShapeChecker miss the CatShape and the colorCatShapeProperty, causing the output to look the same as for the first version of the graph. This limitation is a known, significant drawback of this algorithm.

When the SPARQL-ShapeChecker is executed on the third version of the graph, using the second version as the basis, the colorCatShapeProperty is removed successfully using a support threshold of two. In this case, the knowsPersonShapeProperty is also deleted by the SPARQL-ShapeChecker because the target objects of the “knows” property in the third version are cats instead of people. Consequently, the SPARQL-ShapeChecker cannot find this property anymore. The output of the SPARQL-ShapeChecker for version three would therefore look like the shape provided in Listing 7.4.
However, the SPARQL-ShapeChecker demonstrates a performance advantage, making it considerably faster than running QSE twice. The experiments to prove this observation will be discussed in the evaluation in Section 7.4.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/CatShape> rdf:type
    <http://www.w3.org/ns/shacl#NodeShape> ;
  <http://www.w3.org/ns/shacl#targetClass>
    <http://xmlns.com/foaf/0.1/Cat> .

<http://shaclshapes.org/PersonShape> rdf:type
    <http://www.w3.org/ns/shacl#NodeShape> ;
  <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/namePersonShapeProperty> ;
  <http://www.w3.org/ns/shacl#targetClass>
    <http://xmlns.com/foaf/0.1/Person> .

<http://shaclshapes.org/namePersonShapeProperty> rdf:type
     <http://www.w3.org/ns/shacl#PropertyShape> ;
  <http://shaclshapes.org/confidence> 1E0 ;
  <http://shaclshapes.org/support> "3"^^xsd:int ;
  <http://www.w3.org/ns/shacl#NodeKind>
    <http://www.w3.org/ns/shacl#Literal> ;
  <http://www.w3.org/ns/shacl#datatype> xsd:string ;
  <http://www.w3.org/ns/shacl#minCount> 1 ;
  <http://www.w3.org/ns/shacl#path>
    <http://xmlns.com/foaf/0.1/name> .
```

Listing 7.4: Result of the SPARQL-ShapeChecker on the People Knowledge Graph (Version 3) with the People Knowledge Graph (Version 2) as basis

7.4 Evaluation

Similar to the procedure described in Section 6.4.2, the evaluation of the SPARQL-ShapeChecker was done by comparing its performance to the baseline, which consists of running QSE for the specified versions of the graph. The measurable characteristic used for this comparison is the execution time.

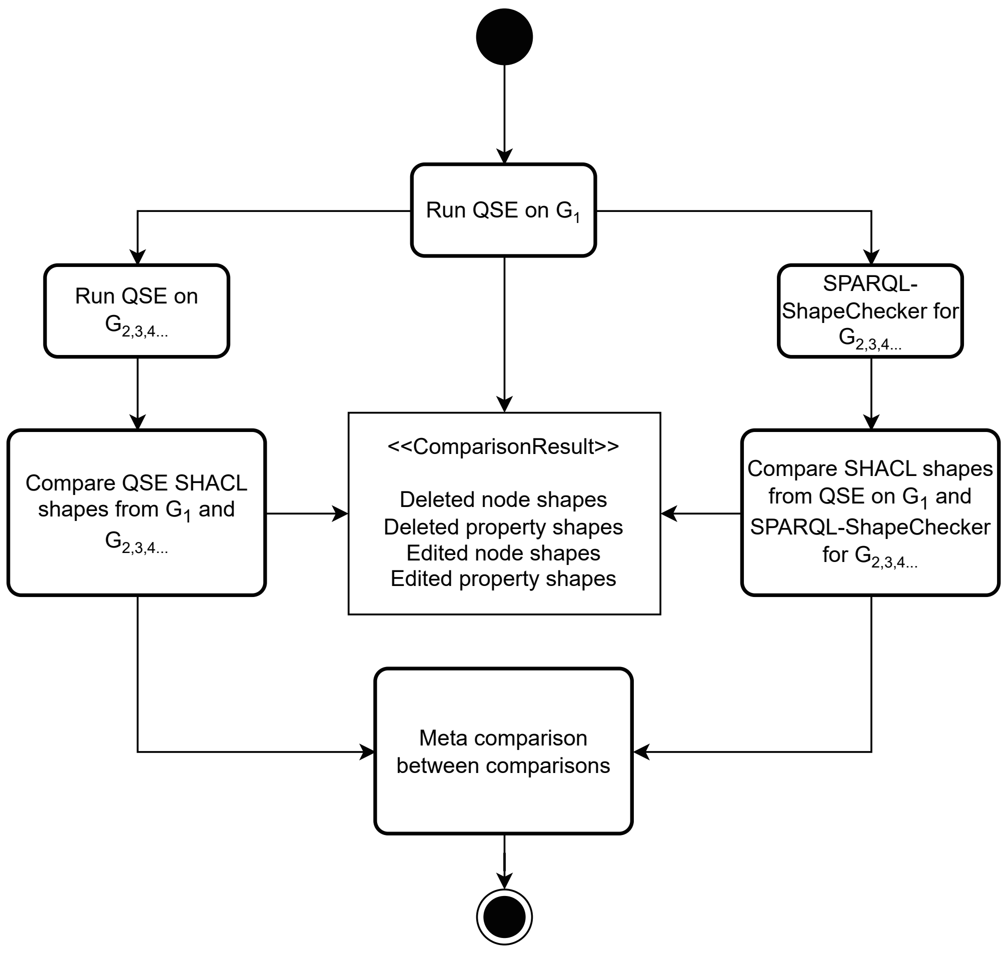

Run QSE on G_1
Run QSE on $G_{2,3,4...}$
SPARQL-ShapeChecker for $G_{2,3,4...}$
Compare QSE SHACL shapes from G_1 and $G_{2,3,4...}$
<<ComparisonResult>>
Deleted node shapes
Deleted property shapes
Edited node shapes
Edited property shapes
Compare SHACL shapes from QSE on G_1 and SPARQL-ShapeChecker for $G_{2,3,4...}$
Meta comparison between comparisons

Figure 7.4: Activity Diagram for the evaluation of the SPARQL-ShapeChecker

The process and the composition of the execution times are visualized in Figure 7.4. The left path presents the baseline, and the right one shows the process for the SPARQL-ShapeChecker. For the baseline, the execution time includes the time taken by QSE to generate SHACL shapes for the first version of the graph, which is the same for both the baseline and the SPARQL-ShapeChecker since QSE is run once for both methods. Additionally, the execution time for the baseline includes the time required for QSE to be run on the second version of the graph. For the SPARQL-ShapeChecker, this part represents the execution time of the algorithm itself. The comparison time is included in both execution times. This refers to the time required to compare all shapes from the first version with those from the second version, using the algorithm described in Section 4.1. The main focus in this use case lies on deleted node and property shapes and edited node and property shapes, as the SPARQL-ShapeChecker cannot detect added shapes.

The equations for the execution times are formally expressed in Equation 7.1 for the baseline and in Equation 7.2 for the SPARQL-ShapeChecker.

$$\Delta t_{\text{Baseline}} = t_{\text{QSE\_n}} + t_{\text{QSE\_n+1}} + t_{\text{Comparison}} \tag{7.1}$$

$$\Delta t_{\text{RQ3}} = t_{\text{QSE\_n}} + t_{\text{SPARQL-ShapeChecker\_n+1}} + t_{\text{Comparison}} \tag{7.2}$$

To compare the results of the baseline with the SPARQL-ShapeChecker content-wise, a meta comparison was developed which compares the differences found by the baseline and the SPARQL-ShapeChecker with each other. The results of this meta-comparison are output in a text file in the project directory. For the People Knowledge Graph, used in Section 7.3, the comparison between the second and the third version is provided in Listing 7.5. As it was previously mentioned, the "knowsPersonShapeProperty" does not point to the same class anymore which is why the property shape is deleted in the version of the SPARQL-ShapeChecker and edited in the QSE comparison. Also, the node shape of the SPARQL-ShapeChecker is adapted, since the reference for the "knowsPersonShapeProperty" is removed.

```
===== Comparison of Compare-Methods =====
==== Added Node Shapes ====
==== Added Property Shapes ====
==== Deleted Node Shapes ====
==== Deleted Property Shapes ====
    === Unique in SPARQL-ShapeChecker-Comparison (Count = 1) ===
        http://shaclshapes.org/knowsPersonShapeProperty
==== Edited Node Shape Names ====
    === Unique in SPARQL-ShapeChecker-Comparison (Count = 1) ===
        http://shaclshapes.org/PersonShape
==== Edited Property Shape Names ====
    === Unique in QSE-Comparison (Count = 1) ===
        http://shaclshapes.org/knowsPersonShapeProperty
Execution Time QSE Total: 5 seconds
Execution Time SPARQL-ShapeChecker Total: 4 seconds
```

Listing 7.5: Result of the meta comparison between the SPARQL-ShapeChecker on the People Knowledge Graph (Version 3) with the People Knowledge Graph (Version 2) as basis

The evaluation of the execution times was conducted on the virtual machine and with the test data as described in Section 6.4.1. The algorithm was primarily evaluated with the BEAR-C dataset. The evaluation was conducted for three scenarios. First, the initial seven versions were compared sequentially (e.g. V1 - V2, V2 - V3...). Second, multiple versions were compared against a common base version (V1). Finally, versions at intervals of ten were compared sequentially. The same evaluation process was applied

to the BEAR-B dataset. However, BEAR-B is less suited for this use case, as it exposes the limitations of QSE. This dataset contains multiple classes or resources with identical names but different IRIs, leading to discrepancies in the results produced by the SPARQL-ShapeChecker when compared to the baseline. Further details for these scenarios will be described later in this section.
The execution times for this evaluation are presented in Table 7.1. It is evident that the SPARQL-ShapeChecker is consistently faster, with an average execution time reduction of 27%. To validate the correctness of the data, the updates in shapes were analyzed for each run. The results, shown in Table 7.2, indicate that no modifications occurred in the BEAR-C dataset. However, there were many additional shapes in the BEAR-B dataset, but no removal of node shapes.
As already addressed before, the results in the shapes between the baseline and the SPARQL-ShapeChecker are not always the same. The following list provides an overview of scenarios where the results of the meta-comparison differ:

- Node or property shapes were added in the new version. Similarly, for existing node shapes, if property shapes are added, the node shape changes accordingly.
- If the type of a property shape changes (e.g., from rdf:langString to xsd:string), it will be deleted by the algorithm because no triples can be found for the original type anymore.
- If a property shape has multiple constraints with different types which are listed in a SHACL-Or list and one type changes, the property shape will be modified by both algorithms. However, the modifications will differ, as the SPARQL-ShapeChecker deletes the constraint. Similarly, if a new constraint with a new type is added to the property shape, QSE will add it to the property shape, while the SPARQL-ShapeChecker will leave the shape unchanged.
- Property shapes associated with node shapes that have multiple target classes lead to ambiguity. It is unclear which class the property shape refers to. In this case, the node shapes also have multiple support entries.
  There also arise problems with the $minCount = 1$ constraint since this is only set when the support of the property shapes matches the support of the node shape. In such cases of ambiguity, it becomes challenging to differentiate whether this condition holds true or not. Therefore, sometimes $minCount = 1$ is added, and sometimes it is not. In this scenario, multiple SHACL-OR lists may also arise within the property shape. This limitation is also listed in Section 2.2.3.
- When property shapes of the same node shape have identical names, but different IRIs, two different shapes are created, distinguished by suffixes. An example are the shapes producerTelevisionShowShapeProperty and producer_1TelevisionShowShapeProperty. In this scenario, the name of the property shape does not identify a shape correctly anymore. When QSE is run several times and only one of these shapes is edited or deleted, it becomes uncertain which

shape was originally meant. Because the comparison algorithm only compares the names of shapes, differences can arise here. Even if both property shapes are still present, it cannot be reliably predicted which one will receive the suffix "_1". Therefore, discrepancies can occur.
There are also cases where multiple path triples are created for a property shape (similar to the case with different suffixes). This results in discrepancies again. An example is provided in Listing 7.6.

- If the support of a property shape changes so that it equals the support of the node shape in the second version, QSE adds the requirement *minCount* = 1. This constraint is not added by the SPARQL-ShapeChecker. However, SPARQL-ShapeChecker deletes the constraint if the condition is not met anymore.

```
<http://shaclshapes.org/residencePersonShapeProperty> rdf:type
  <http://www.w3.org/ns/shacl#PropertyShape> ;
<http://shaclshapes.org/confidence> 2E-1 ;
<http://shaclshapes.org/confidence> 2,2222E-1 ;
<http://shaclshapes.org/support> "1"^^xsd:int ;
<http://shaclshapes.org/support> "2"^^xsd:int ;
<http://www.w3.org/ns/shacl#NodeKind>
    <http://www.w3.org/ns/shacl#IRI> ;
<http://www.w3.org/ns/shacl#class>
    <http://shaclshapes.org/undefined> ;
<http://www.w3.org/ns/shacl#path>
    <http://dbpedia.org/ontology/residence> ;
<http://www.w3.org/ns/shacl#path>
    <http://dbpedia.org/property/residence> .
```

Listing 7.6: Property shape with different paths

Version 1	Version 2	Baseline [seconds]	SPARQL-ShapeChecker [seconds]
BEAR-C-1	BEAR-C-2	41	30
BEAR-C-2	BEAR-C-3	30	19
BEAR-C-3	BEAR-C-4	29	18
BEAR-C-4	BEAR-C-5	28	19
BEAR-C-5	BEAR-C-6	29	19
BEAR-C-6	BEAR-C-7	29	19
BEAR-C-1	BEAR-C-2	25	19
BEAR-C-1	BEAR-C-3	25	19
BEAR-C-1	BEAR-C-4	25	19
BEAR-C-1	BEAR-C-5	25	19
BEAR-C-1	BEAR-C-6	25	19
BEAR-C-1	BEAR-C-7	25	18
BEAR-C-1	BEAR-C-10	30	19
BEAR-C-10	BEAR-C-20	31	20
BEAR-C-20	BEAR-C-30	30	20
BEAR-B-1	BEAR-B-2	253	192
BEAR-B-2	BEAR-B-3	235	175
BEAR-B-3	BEAR-B-4	242	178
BEAR-B-4	BEAR-B-5	240	177
BEAR-B-5	BEAR-B-6	239	179
BEAR-B-6	BEAR-B-7	241	180
BEAR-B-1	BEAR-B-2	220	168
BEAR-B-1	BEAR-B-3	216	168
BEAR-B-1	BEAR-B-4	217	167
BEAR-B-1	BEAR-B-5	218	167
BEAR-B-1	BEAR-B-6	216	166
BEAR-B-1	BEAR-B-7	217	167
BEAR-B- 1	BEAR-B-10	260	197
BEAR-B- 10	BEAR-B-20	243	180
BEAR-B- 20	BEAR-B-30	247	183
BEAR-B- 30	BEAR-B-40	258	188
BEAR-B- 40	BEAR-B-50	265	195
BEAR-B- 50	BEAR-B-60	273	208
BEAR-B- 60	BEAR-B-70	282	214
BEAR-B- 70	BEAR-B-80	290	224

Table 7.1: Execution times for the baseline and the SPARQL-ShapeChecker

Version 1	Version 2	A	A	D	D	D	D	E	E	E	E
		N	P	N	N	P	P	N	N	P	P
				B	S	B	S	B	S	B	S
BEAR-C-1	BEAR-C-2	0	0	0	0	0	0	0	0	0	0
BEAR-C-2	BEAR-C-3	0	0	0	0	0	0	0	0	0	0
BEAR-C-3	BEAR-C-4	0	0	0	0	0	0	0	0	0	0
BEAR-C-4	BEAR-C-5	0	0	0	0	0	0	0	0	0	0
BEAR-C-5	BEAR-C-6	0	0	0	0	0	0	0	0	0	0
BEAR-C-6	BEAR-C-7	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-2	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-3	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-4	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-5	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-6	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-7	0	0	0	0	0	0	0	0	0	0
BEAR-C-1	BEAR-C-10	0	0	0	0	0	0	0	0	0	0
BEAR-C-10	BEAR-C-20	0	0	0	0	0	0	0	0	0	0
BEAR-C-20	BEAR-C-30	0	0	0	0	0	0	0	0	0	0
BEAR-B-1	BEAR-B-2	0	0	0	0	0	0	0	0	0	3
BEAR-B-2	BEAR-B-3	0	0	0	0	0	0	0	0	7	3
BEAR-B-3	BEAR-B-4	0	104	0	0	0	0	13	0	112	3
BEAR-B-4	BEAR-B-5	0	0	0	0	0	0	0	0	0	3
BEAR-B-5	BEAR-B-6	0	0	0	0	0	0	0	0	0	3
BEAR-B-6	BEAR-B-7	0	0	0	0	0	0	0	0	0	3
BEAR-B-1	BEAR-B-2	0	0	0	0	0	0	0	0	0	3
BEAR-B-1	BEAR-B-3	0	0	0	0	0	0	0	0	7	3
BEAR-B-1	BEAR-B-4	0	104	0	0	0	0	13	0	119	3
BEAR-B-1	BEAR-B-5	0	104	0	0	0	0	13	0	119	3
BEAR-B-1	BEAR-B-6	0	104	0	0	0	0	13	0	119	3
BEAR-B-1	BEAR-B-7	0	104	0	0	0	0	13	0	119	3
BEAR-B-1	BEAR-B-10	0	104	0	0	0	0	13	0	124	3
BEAR-B-10	BEAR-B-20	3	554	0	0	4	73	55	36	609	110
BEAR-B-20	BEAR-B-30	2	470	0	0	4	16	52	26	642	18
BEAR-B-30	BEAR-B-40	0	448	0	0	1	28	61	28	650	12
BEAR-B-40	BEAR-B-50	0	525	0	0	4	4	58	21	407	23
BEAR-B-50	BEAR-B-60	2	337	0	0	14	24	25	16	345	14
BEAR-B-60	BEAR-B-70	4	345	0	0	0	2	34	17	446	21
BEAR-B-70	BEAR-B-80	0	105	0	0	4	4	37	4	179	2

Table 7.2: Changes in shapes between the baseline and the SPARQL-ShapeChecker. Legend: AN (added node shapes), AP (added property shapes), D (deleted), E (edited), N (node shape), P (property shape), B (baseline), S (SPARQL-ShapeChecker) e.g. DNB (deleted node shapes in the baseline)

CHAPTER 8

Discussion

The results from the three previously discussed tools show that significant progress has been made toward improving data quality through the extraction of SHACL shapes in evolving knowledge graphs. Although the three research questions target different approaches in this area, QSE serves as the foundational basis for each of them.
The aim of RQ1 was to develop a method for comparing extracted SHACL shapes across different versions of a knowledge graph. There has already been a graphical user interface in the form of a web application developed, which is called Shactor. While this tool visualized the SHACL shapes extraction process using QSE, it did not offer a method to compare SHACL shapes. Although manual comparison methods or text comparison tools could be used for this task, these procedures proved to be impractical for larger knowledge graphs. RQ1 was solved by developing the ShapeComparator and based on the results from the semi-structured expert interviews discussed in Section 5.3, the feedback is mainly positive. The initial challenge of providing a tool to efficiently compare SHACL shapes across multiple graph versions has been solved successfully. All participants indicated that they would prefer using this tool over manual or text-based comparison methods.
Certainly, there is also room for improvement for the ShapeComparator. For example, usability enhancements could be made when users compare SHACL shapes from many different graph versions at once. Additionally, if a large number of shapes are analyzed, further testing and refinements in usability would be beneficial. Nevertheless, the primary research question has been successfully answered which delivers significant benefits to users.
RQ2 and RQ3 focused on improving the execution time of QSE for evolving knowledge graphs. Specifically, the aim of RQ2 was to optimize the performance by using changesets between different versions of a knowledge graph. The resulting tool is called the SHACL-DiffExtractor and it parses the changeset to produce SHACL shapes for a subsequent version of a graph instead of using the actual graph file. QSE must have been run on

the initial version of the graph in order to produce SHACL shapes and the intermediate results. The SHACL-DiffExtractor then parses the changeset to obtain the SHACL shapes for the following version, avoiding the need to rerun QSE from scratch on the following version.
A key assumption for this approach is that the changeset size is relatively small compared to the overall size of the graph. However, this assumption does not apply universally to all knowledge graphs and therefore the usage of the SHACL-DiffExtractor should be evaluated depending on the specific use case. As demonstrated in Section 6.4.2, the assumption did not hold for the dataset BEAR-C. In this case, the change ratio was around two-thirds of the graph size, resulting in roughly equal execution times for running QSE multiple times and using the SHACL-DiffExtractor. In contrast, the BEAR-B dataset has a change ratio of only 2%. The SHACL-DiffExtractor did result in a performance improvement for this knowledge graph. This demonstrates that the research question was answered successfully, however, the tool is only effective when the changeset is relatively small compared to the overall size of the knowledge graph.
Another limitation of the SHACL-DiffExtractor is the requirement to have changesets between knowledge graph versions available. If a use case does not automatically generate changesets and only the graph versions are accessible, calculating the changesets would be computationally expensive. In such scenarios, it is better to rely solely on QSE. However, there might be use cases where only changesets are available, and in these situations, the SHACL-DiffExtractor demonstrates significant strengths.
Another limitation arises from QSE itself, as it produces ambiguous results when resources in a graph share the same name but have different IRIs, as described in Section 2.2.3. This issue became evident during the evaluation of the SHACL-DiffExtractor and the SPARQL-ShapeChecker with real-world datasets, which led to inconsistent results.
The final part of the thesis focused, similarly to the SHACL-DiffExtractor, on improving the execution time of QSE for evolving knowledge graphs. To answer RQ3, a tool called SPARQL-ShapeChecker has been developed, which utilizes the query-based version of QSE. For this approach, the initially generated SHACL shapes by QSE were re-evaluated on a subsequent version of a graph. While this method offers a significant performance improvement, the trade-off is that added or updated shapes cannot be detected. As a result, the tool is only suitable for specific use cases. For example, a huge knowledge graph where QSE has already been executed and users do not anticipate significant schema changes would be an ideal scenario. However, if a greater schema evolution is expected, the SPARQL-ShapeChecker might not be the most suitable option.

CHAPTER 9

Conclusion & Future Work

This thesis examined three distinct approaches for QSE in combination with evolving knowledge graphs. A graphical interface for comparing SHACL shapes and two algorithmic extensions have been developed. The algorithms aimed at enhancing the execution time of QSE across different versions of a knowledge graph, where one approach involved utilizing changesets between versions, while the other one re-evaluated existing SHACL shapes on a subsequent graph version. With these tools, open research questions could be resolved, however, there is still potential for further work in this area.
There are many integration possibilities for the ShapeComparator. It could be connected with Shactor so that users could benefit from the valuable statistics. Furthermore, the ShapeComparator could support the approximate and query-based version of QSE in future releases. Additionally, an integration with the two algorithmic approaches - the SHACL-DiffExtractor and the SPARQL-ShapeChecker - would be convenient to provide a visual interface. Another enhancement would be to allow users to upload any SHACL file, regardless of whether it was generated by QSE or not. Implementing this feature could widen the area of application, however, the implementation could be challenging, as raw Turtle files would need to be mapped to the internal structure of QSE.
For the SHACL-DiffExtractor, an integration with the approximate and the query-based option of QSE would be convenient. Similarly for the SPARQL-ShapeChecker the integration of the approximate and the file-based version of QSE would be beneficial. However, another interesting aspect is whether there is a way to modify QSE so that the execution time can be reduced for a subsequent run by only using the internal information and the originally generated shapes. This would eliminate the need for changesets and the trade-off from the SPARQL-ShapeChecker that added shapes cannot be discovered. However, this research question presents a greater challenge since a subsequent version of a graph must still be completely parsed to find all updates. Furthermore, future work could include minor adaptions in QSE itself. One example includes the handling of resources with the same name but different IRIs in knowledge graphs.

APPENDIX A

Systematic Literature Review

	Title, Reference	Exclusion Reason
	"evolving knowledge graphs"	
1	Evolving Knowledge Graphs [70]	
2	Know-Evolve: Deep Temporal Reasoning for Dynamic Knowledge Graphs [92]	Title: Reasoning is not in line with the thesis topic
3	Summarizing Entity Temporal Evolution in Knowledge Graphs [91]	
4	How does knowledge evolve in open knowledge graphs? [83]	
5	Analysing the Evolution of Knowledge Graphs for the Purpose of Change Verification [74]	
6	EvolveKG: a general framework to learn evolving knowledge graphs [69]	
7	KGdiff: Tracking the Evolution of Knowledge Graphs [67]	
8	Predicting the co-evolution of event and Knowledge Graphs [50]	
9	Knowledge Graphs Evolution and Preservation – A Technical Report from ISWS 2019 [31]	
10	Representing Scientific Literature Evolution via Temporal Knowledge Graphs [86]	Title: Scientific Literature Evolution not in line with the thesis topic
	"data quality in knowledge graphs"	
11	GraphGuard: Enhancing Data Quality in Knowledge Graph Pipelines [47]	
12	Improving and Assessing Data Quality of Knowledge Graphs [45]	
13	Knowledge Graph Quality Management: A Comprehensive Survey [95]	

Continuation of Table A.1		
	Title, Reference	Exclusion Reason
14	Knowledge Graph Completeness: A Systematic Literature Review [64]	
15	Steps to Knowledge Graphs Quality Assessment [63]	
16	What Are Links in Linked Open Data? A Characterization and Evaluation of Links between Knowledge Graphs on the Web [58]	Title: Links in Linked Open Data are not in line with the thesis topic
17	Knowledge Graphs 2021: A Data Odyssey [94]	
18	Knowledge graphs [51]	
19	A Practical Framework for Evaluating the Quality of Knowledge Graph [40]	
20	Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL [89]	
	“shacl extraction from knowledge graphs”	
21	A Library for Visualizing SHACL over Knowledge Graphs [33]	
22	SCOOP all the Constraints’ Flavours for your Knowledge Graph [48]	
23	Learning SHACL shapes from knowledge graphs [76]	
24	Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL [89]	Duplicate
25	Extraction of Validating Shapes from Very Large Knowledge Graphs [84]	
26	Using Knowledge Graph Technologies to Contextualize and Validate Declarations in the Social Security Domain [41]	
27	Automatic extraction of shapes using sheXer [52]	
28	Property assertion constraints for an informed, error-preventing expansion of knowledge graphs [46]	
29	Automatic Construction of SHACL Schemas for RDF Knowledge Graphs Generated by Direct Mappings [42]	Language: Korean, not available with TU-Wien Account
30	Trav-SHACL: Efficiently Validating Networks of SHACL Constraints [54]	
	“comparing shacl shapes”	
31	Comparing ShEx and SHACL [55]	Not available with TU-Wien account
32	Using Ontology Design Patterns To Define SHACL Shapes [77]	
33	Semantic rule checking of cross-domain building data in information containers for linked document delivery using the shapes constraint language [57]	

Continuation of Table A.1		
	Title, Reference	Exclusion Reason
34	Astrea: automatic generation of SHACL shapes from ontologies [43]	
35	Trav-SHACL: Efficiently validating networks of SHACL constraints [54]	Duplicate
36	Formalizing Property Constraints in Wikidata [53]	
37	Semantics and Validation of Recursive SHACL [44]	
38	An Argument for Generating SHACL Shapes from ODPs [78]	
39	Comparison of the eye's wave-front aberration measured psychophysically and with the Shack–Hartmann wave-front sensor [87]	Title: not relevant for knowledge graphs
40	Absolute sphericity measurement: a comparative study of the use of interferometry and a Shack–Hartmann sensor [81]	Title: not relevant for knowledge graphs
	"differences between knowledge graphs versions"	
41	Summarizing entity temporal evolution in knowledge graphs [91]	Duplicate
42	KGDiff: Tracking the evolution of knowledge graphs [67]	Duplicate
43	Knowledge Graphs on the Web-An Overview. [59]	
44	Knowledge graphs: A practical review of the research landscape [66]	
45	Bias in Knowledge Graphs–an Empirical Study with Movie Recommendation and Different Language Editions of DBpedia [93]	Title: Bias not relevant for thesis topic
46	Explaining and suggesting relatedness in knowledge graphs [82]	
47	A survey on knowledge graphs: Representation, acquisition, and applications [65]	
48	Measuring accuracy of triples in knowledge graphs [71]	Title: Accuracy not relevant for thesis topic
49	AYNEC: all you need for evaluating completion techniques in knowledge graphs [36]	
50	Modelling dynamics in semantic web knowledge graphs with formal concept analysis [56]	Title: Modelling dynamics not relevant for thesis topic

Table A.1: All initial search results for all search terms

	Title, Reference	Comments	Re-levance
1	Evolving Knowledge Graphs [70]	Theoretical description of EvolveKG [69] - a framework that reveals cross-time knowledge interaction with desirable performance. This is partly relevant for this thesis since an extra framework is created with a Derivative Graph, allowing knowledge prediction. In this thesis, only snapshots of evolving graphs are considered.	2
3	Summarizing Entity Temporal Evolution in Knowledge Graphs [91]	Envisions an approach where a summary graph is created to catch the evolution of all entities across all versions.	2
4	How does knowledge evolve in open knowledge graphs? [83]	Explains the basics of evolving knowledge graphs.	1
5	Analysing the Evolution of Knowledge Graphs for the Purpose of Change Verification [74]	This paper deals with topological features of a graph to generate classifiers that can judge whether an incoming graph change is correct or incorrect.	2
6	EvolveKG: a general framework to learn evolving knowledge graphs [69]	Already mentioned in [70].	3
7	KGdiff: Tracking the Evolution of Knowledge Graphs [67]	Proposes a software that tracks changes in the schema and for the individuals.	2
8	Predicting the co-evolution of event and Knowledge Graphs [50]	The goal of this paper is to predict unobserved facts by using previous temporal information from knowledge graphs and static information. This is not relevant since prediction is not part of this thesis.	3
9	Knowledge Graphs Evolution and Preservation – A Technical Report from ISWS 2019 [31]	Collection of ten papers: Papers 5,6,10 are not relevant. Paper 1 differentiates between different types of evolution and checks if machine learning can capture this evolution. Paper 2 focuses on the characteristics of an evolving knowledge graph, in this case, DBpedia. The third paper discusses changes between two versions of a knowledge graph. Paper 4 is about changes in ontologies and how they can be characterized. Paper 7 deals with the integration of data in knowledge graphs. Versioned knowledge graphs are targeted in Chapter 8. Finally, paper 9 deals with the support of interactive updates in knowledge graphs.	2

Continuation of Table A.2

	Title, Reference	Comments	Relevance
11	GraphGuard: Enhancing Data Quality in Knowledge Graph Pipelines [47]	Introduces a framework for better data quality in knowledge graph pipelines for humans and machines. This paper is partly relevant to this thesis because the goal of QSE and SHACL is not to check data quality during data ingestion.	2
12	Improving and Assessing Data Quality of Knowledge Graphs [45]	Dissertation which focuses on including data transformations in knowledge graphs that can help to clean the data and complete knowledge graphs by calculating derived data. Furthermore, this paper is about validating knowledge graphs. Validation is done by a reasoning solution called Validatrr. Cleaning, completing and reasoning are not directly addressed therefore this paper is not directly relevant.	2
13	Knowledge Graph Quality Management: A Comprehensive Survey [95]	This paper provides a systematic review of quality management in knowledge graphs, also including quality management processes, such as quality assessment, error detection, error correction, and completion.	2
14	Knowledge Graph Completeness: A Systematic Literature Review [64]	Different quality dimensions exist - such as accuracy or completeness. This paper summarizes terminologies related to completeness. Therefore it is not relevant for this thesis, since completeness is not a topic.	3
15	Steps to Knowledge Graphs Quality Assessment [63]	The goal of this paper is to extend existing quality assessment frameworks by adding quality dimensions and quality metrics. It is only partly relevant for this thesis since quality dimensions are evaluated with SHACL.	2
17	Knowledge Graphs 2021: A Data Odyssey [94]	This paper discusses advances and lessons learned in the history of knowledge graphs. The abstract is too general that it could be relevant for this thesis.	3
18	Knowledge graphs [51]	Entire book on knowledge graphs that includes a general introduction, how to build and use knowledge graphs, and specific use cases.	2
19	A Practical Framework for Evaluating the Quality of Knowledge Graph [40]	In this paper, existing frameworks for quality in knowledge graphs are assessed and a practical framework is proposed which determines if a knowledge graph is suitable for an intended purpose.	3

Continuation of Table A.2			
	Title, Reference	Comments	Relevance
20	Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL. [89]	The approach of this paper is used in QSE. It proposes a semantic profiling tool that helps to enhance the understanding of the data by using SHACL.	1
21	A Library for Visualizing SHACL over Knowledge Graphs [33]	This master thesis created a library called SHACLViewer which illustrates SHACL shapes in a 3D context.	1
22	SCOOP all the Constraints' Flavours for your Knowledge Graph [48]	The goal of the SCOOP framework is to extract SHACL shapes from already existing knowledge graphs (similar to QSE). However, it mainly uses ontologies and data schemas instead of the entities itself.	2
23	Learning SHACL shapes from knowledge graphs [76]	This paper introduces Inverse Open Path (IOP), a predicate logic formalism that presents specific shapes over connected entities from a knowledge graph. The corresponding learning method is called SHACLearner.	2
25	Extraction of Validating Shapes from Very Large Knowledge Graphs [84]	The thesis builds on this paper, therefore it is excluded.	3
26	Using Knowledge Graph Technologies to Contextualize and Validate Declarations in the Social Security Domain [41]	A master thesis, which generates SHACL shapes for a specific use case namely forms which describe the work of employees in a quarter. Since SHACL shapes are generated manually, the thesis is not relevant.	3
27	Automatic extraction of shapes using sheXer [52]	sheXer produces SHACL shapes similar to QSE using a python library. This paper is also mentioned in the related work of QSE.	2
28	Property assertion constraints for an informed, error-preventing expansion of knowledge graphs [46]	PAC (property assertion constraints) have the goal of checking data before it is added to a knowledge graph. With this approach, errors can be prevented and the quality of knowledge graphs can be enhanced. PAC works by restricting the range of properties using SPARQL.	2

Continuation of Table A.2

	Title, Reference	Comments	Relevance
30	Trav-SHACL: Efficiently Validating Networks of SHACL Constraints [54]	Trav-SHACL plans the execution and the traversal of a shapes graph so that invalid entries are detected early. For this task, the shapes graph is reordered.	2
32	Using Ontology Design Patterns To Define SHACL Shapes [77]	Similar to [48], this paper reuses the Ontology Design Pattern (ODP) and contexts to automatically generate SHACL shapes.	2
33	Semantic rule checking of cross-domain building data in information containers for linked document delivery using the shapes constraint language [57]	This case study uses SHACL in the domain of Information Container for Linked Document Delivery (ICDD). Since this is a specific case study, it is not relevant to this thesis.	3
34	Astrea: automatic generation of SHACL shapes from ontologies [43]	Similar to [48] this approach also uses ontologies to automatically generate SHACL shapes. Astrea uses Astrea-KG which provides mappings between ontology constraint patterns and SHACL constraint patterns.	2
36	Formalizing Property Constraints in Wikidata [53]	This paper compares constraints in Wikidata, which uses its own RDF data model for constraints. These can be created with SPARQL and SHACL. This paper is not relevant, since Wikidata will not be used in this thesis.	3
37	Semantics and Validation of Recursive SHACL [44]	As the title suggests, this paper explores recursion for SHACL constraints. It proposes concise formal semantics of the core elements of SHACL and validates recursion for these elements. This paper is not relevant since the recursion of SHACL elements is not a topic for this thesis.	3
38	An Argument for Generating SHACL Shapes from ODPs [78]	This book chapter extends the idea of [77].	3
43	Knowledge Graphs on the Web-An Overview. [59]	This book chapter provides an overview and a comparison of publicly available knowledge graphs (DBpedia, Wikidata) and gives insights into their sizes and contents. This chapter is not relevant to this thesis since it is too general.	3

Continuation of Table A.2			
	Title, Reference	Comments	Re-levance
44	Knowledge graphs: A practical review of the research landscape [66]	The cross-disciplinary nature of knowledge graphs is very important. This paper gives an overview of the major strands in the research landscape and their different communities. This paper is too general for this thesis.	3
46	Explaining and suggesting relatedness in knowledge graphs [82]	It provides a tool called RECAP, which explains the relatedness of a pair of entities in a knowledge graph. Relatedness is not a topic of this thesis, therefore it is not relevant.	3
47	A survey on knowledge graphs: Representation, acquisition, and applications [65]	This survey covers a broad range of topics regarding knowledge graphs. However, also temporal knowledge graphs are discussed, which makes the paper partly relevant.	2
49	AYNEC: all you need for evaluating completion techniques in knowledge graphs [36]	The tool AYNEC provides a suite for the evaluation of knowledge graph completion techniques. Since knowledge graph completion is not a topic of this thesis, this paper is not relevant.	3

Table A.2: Comments and possible exclusion reasons for all papers after the first round based on the Abstract. Relevance ranges from 1 (absolutely relevant) to 3 (not relevant)

APPENDIX B

Demonstration Knowledge Graphs

```
<http://example.org/alice>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/bob>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/jenny>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/name> "Alice" .
<http://example.org/bob>
    <http://xmlns.com/foaf/0.1/name> "Bob" .
<http://example.org/jenny>
    <http://xmlns.com/foaf/0.1/name> "Jenny" .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/bob> .
<http://example.org/bob>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/alice> .
<http://example.org/jenny>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/alice> .
```

Listing B.1: People Knowledge Graph (Version 1)

```
<http://example.org/alice>
```

```
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Person> .
    <http://example.org/bob>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Person> .
    <http://example.org/jenny>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Person> .
    <http://example.org/alice>
        <http://xmlns.com/foaf/0.1/name> "Alice" .
    <http://example.org/bob>
        <http://xmlns.com/foaf/0.1/name> "Bob" .
    <http://example.org/jenny>
        <http://xmlns.com/foaf/0.1/name> "Jenny" .
    <http://example.org/alice>
        <http://xmlns.com/foaf/0.1/knows>
        <http://example.org/bob> .
    <http://example.org/bob>
        <http://xmlns.com/foaf/0.1/knows>
        <http://example.org/alice> .
    <http://example.org/jenny>
        <http://xmlns.com/foaf/0.1/knows>
        <http://example.org/alice> .
    <http://example.org/orangeCat>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Cat> .
    <http://example.org/blackCat>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Cat> .
    <http://example.org/greyCat>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Cat> .
    <http://example.org/orangeCat>
        <http://example.org/color> "orange" .
    <http://example.org/blackCat>
        <http://example.org/color> "black" .
    <http://example.org/greyCat>
        <http://example.org/color> "grey" .
```

Listing B.2: People Knowledge Graph (Version 2)

```
    <http://example.org/alice>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Person> .
```

```
<http://example.org/bob>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/jenny>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/name> "Alice" .
<http://example.org/bob>
    <http://xmlns.com/foaf/0.1/name> "Bob" .
<http://example.org/jenny>
    <http://xmlns.com/foaf/0.1/name> "Jenny" .
<http://example.org/alice>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/orangeCat> .
<http://example.org/bob>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/blackCat> .
<http://example.org/jenny>
    <http://xmlns.com/foaf/0.1/knows>
    <http://example.org/greyCat> .
<http://example.org/orangeCat>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Cat> .
<http://example.org/blackCat>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Cat> .
<http://example.org/greyCat>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Cat> .
<http://example.org/orangeCat>
    <http://example.org/color> "orange" .
```

Listing B.3: People Knowledge Graph (Version 3)

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://shaclshapes.org/PersonShape> rdf:type
    <http://www.w3.org/ns/shacl#NodeShape> ;
  <http://shaclshapes.org/support> "3"^^xsd:int ;
  <http://www.w3.org/ns/shacl#property>
    <http://shaclshapes.org/instanceTypePersonShapeProperty> ;
  <http://www.w3.org/ns/shacl#property>
```

```
        <http://shaclshapes.org/knowsPersonShapeProperty> ;
      <http://www.w3.org/ns/shacl#property>
        <http://shaclshapes.org/namePersonShapeProperty> ;
      <http://www.w3.org/ns/shacl#targetClass>
        <http://xmlns.com/foaf/0.1/Person> .

<http://shaclshapes.org/instanceTypePersonShapeProperty>
        rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
      <http://shaclshapes.org/confidence> 1E0 ;
      <http://shaclshapes.org/support> "3"^^xsd:int ;
      <http://www.w3.org/ns/shacl#in>
        (<http://xmlns.com/foaf/0.1/Person> ) ;
      <http://www.w3.org/ns/shacl#path> rdf:type .

<http://shaclshapes.org/knowsPersonShapeProperty>
        rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
      <http://shaclshapes.org/confidence> 1E0 ;
      <http://shaclshapes.org/support> "3"^^xsd:int ;
      <http://www.w3.org/ns/shacl#NodeKind>
        <http://www.w3.org/ns/shacl#IRI> ;
      <http://www.w3.org/ns/shacl#class>
        <http://xmlns.com/foaf/0.1/Person> ;
      <http://www.w3.org/ns/shacl#minCount> 1 ;
      <http://www.w3.org/ns/shacl#node>
        <http://shaclshapes.org/PersonShape> ;
      <http://www.w3.org/ns/shacl#path>
        <http://xmlns.com/foaf/0.1/knows> .

<http://shaclshapes.org/namePersonShapeProperty>
        rdf:type <http://www.w3.org/ns/shacl#PropertyShape> ;
      <http://shaclshapes.org/confidence> 1E0 ;
      <http://shaclshapes.org/support> "3"^^xsd:int ;
      <http://www.w3.org/ns/shacl#NodeKind>
        <http://www.w3.org/ns/shacl#Literal> ;
      <http://www.w3.org/ns/shacl#datatype> xsd:string ;
      <http://www.w3.org/ns/shacl#minCount> 1 ;
      <http://www.w3.org/ns/shacl#path>
        <http://xmlns.com/foaf/0.1/name> .
```

Listing B.4: SHACL shapes for the People Knowledge Graph (Version 1) from QSE-Exact with no parameters for support or confidence

Overview of Generative AI Tools Used

The primary AI tool used in this thesis was ChatGPT. It was used instead or in combination with Google for general questions such as finding out how to make text bold in LaTeX. Additionally, ChatGPT was used to generate small code snippets, like comparing two maps in Java. The most significant use of ChatGPT however, was for reformulating and correcting self-written English sentences during the writing process. In addition, the free version of Grammarly was utilized to identify grammatical issues in the text. Of course, all outputs from AI tools were reviewed and revised as necessary.

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

BEAR BEnchmark of RDF ARchives. 18, 27, 28, 35, 71–76, 88, 89, 91, 92, 94, 111, 113

DSR Design Science Research. xiii, 21, 23–25, 28

IRI Internationalized Resource Identifier. 6, 9, 10, 16, 18, 19, 44, 52, 58, 64–66, 72, 74, 75, 79, 80, 89, 94, 95

NS Node Shape. 14

PS Property Shape. 14

QSE Quality Shapes Extraction. vii, ix, xi, xiii, xv, 1, 2, 5, 10, 12–19, 29–33, 35–37, 39, 41, 44, 46–49, 55–59, 61–65, 68–70, 72–75, 77–79, 81, 83–90, 93–95, 101, 102, 108, 111, 115, 117

RDF Resource Description Framework. xi, xiii, 1, 5–10, 15, 28, 32, 38, 49, 51, 53, 72, 80, 98, 103, 111, 117

regex Regular Expressions. 31, 32, 80, 82, 84

RQ Research Question. 3, 9, 26–28, 55, 57, 68, 77, 84, 93, 94

SHACL Shapes Constraint Language. xi, xiii, xv, 1, 2, 5, 9, 10, 12–14, 16, 17, 19, 20, 25, 29–33, 35–41, 46, 48–53, 55–57, 64, 65, 69, 73, 77–82, 84, 85, 87, 89, 93–95, 101–103, 108, 111, 115, 117

SLR Systematic Literature Review. 22

SPARQL SPARQL Protocol And RDF Query Language. xi, xiii, 2, 3, 5, 8, 9, 12, 13, 32, 77–80, 84, 102, 103, 115, 117

SSI Semi-structured interviews. 23

URI Uniform Resource Identifier. 6, 7, 9

URL Uniform Resource Locator. 6, 84, 85

URN Uniform Resource Name. 6

Bibliography

[1] 188.399 Introduction to Semantic Systems | TU Wien. https://tiss.tuwien.ac.at/course/educationDetails.xhtml?dswid=9706&dsrid=233&semester=2023W&courseNr=188399. Accessed: 2023-11-28.

[2] Apache Jena. https://jena.apache.org/. Accessed: 2024-05-14.

[3] atextor/turtle-formatter. https://github.com/atextor/turtle-formatter. Accessed: 2024-04-12.

[4] BEAR | BEnchmark of RDF ARchives. https://aic.ai.wu.ac.at/qadlod/bear.html. Accessed: 2023-11-14.

[5] DBpedia. https://www.dbpedia.org/. Accessed: 2024-06-22.

[6] DBpedia: Dataset releases Archives. https://www.dbpedia.org/dataset-releases/. Accessed: 2024-05-14.

[7] dkw-aau/qse . https://github.com/dkw-aau/qse. Accessed: 2024-07-10.

[8] dkw-aau/qse at shactor. https://github.com/dkw-aau/qse/tree/shactor. Accessed: 2024-07-12.

[9] Eclipse RDF4J. https://rdf4j.org/. Accessed: 2024-05-14.

[10] Free prototyping tool for web & mobile apps - Justinmind. https://www.justinmind.com. Accessed: 2024-04-12.

[11] GraphDB Downloads and Resources. https://graphdb.ontotext.com/. Accessed: 2024-05-11.

[12] Guidance Document Informed Consent. https://www.tuwien.at/index.php?eID=dms&s=4&path=Documents/GDPR%20Guidelines%20and%20FAQs/Research_Projects_Guidance_Document_Informed_Consent.pdf. Accessed: 2024-10-01.

[13] H2 Database Engine. https://www.h2database.com/html/main.html. Accessed: 2024-07-30.

[14] java.time (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html. Accessed: 2024-08-07.

[15] kpdecker/jsdiff. https://github.com/kpdecker/jsdiff. Accessed: 2024-04-12.

[16] Linked Life Data - A Semantic Data Integration Platform for the Biomedical Domain. http://linkedlifedata.com/. Accessed: 2024-05-14.

[17] OpenLink Software: Virtuoso Homepage. https://virtuoso.openlinksw.com/. Accessed: 2024-06-24.

[18] OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition). https://www.w3.org/TR/owl2-rdf-based-semantics/. Accessed: 2024-05-17.

[19] PubChem. https://pubchem.ncbi.nlm.nih.gov/docs/about. 2024-05-14.

[20] RDF Schema 1.1. https://www.w3.org/TR/rdf-schema. Accessed: 2024-06-25.

[21] SailRepository (Eclipse RDF4J 4.3.0). https://rdf4j.org/javadoc/4.3.0/org/eclipse/rdf4j/repository/sail/SailRepository.html. Accessed: 2024-10-16.

[22] shaclgen: Shacl graph generator. https://github.com/uwlib-cams/shaclgen. Accessed: 2023-11-29.

[23] Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/. Accessed: 2024-05-11.

[24] ShEx - Shape Expressions. https://shex.io/. Accessed: 2024-05-17.

[25] SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/. Accessed: 2024-05-11.

[26] Spring. https://spring.io/. Accessed: 2024-07-30.

[27] Vaadin Docs. https://vaadin.com/docs/latest/. Accessed: 2024-04-12.

[28] Wikidata. https://www.wikidata.org. Accessed: 2024-05-14.

[29] Your relational data. Objectively. - Hibernate ORM. https://hibernate.org/orm/. Accessed: 2024-07-30.

[30] Zentrale Verbindungsplattform | Zoom. https://zoom.us/de. Accessed: 2024-10-01.

[31] Abbas, N., Alghamdi, K., Alinam, M., Alloatti, F., Amaral, G. C. M., d'Amato, C., Asprino, L., Beno, M., Bensmann, F., Biswas, R., Cai, L., Capshaw, R., Carriero, V. A., Celino, I., Dadoun, A., Giorgis, S. D., Delva, H., Domingue, J., Dumontier, M., Emonet, V., van Erp, M., Espinoza-Arias, P., Fallatah, O., Ferrada, S., Ocaña, M. G., Georgiou, M., Gesese, G. A., Gillis-Webber, F., Giovannetti, F., Buey, M. G., Harrando, I., Heibi, I., Horta, V. A. C., Huber, L., Igne, F., Jaradeh, M. Y., Keshan, N., Koleva, A., Koteich, B., Kurniawan, K., Liu, M., Ma, C., Maas, L., Mansfield, M., Mariani, F., Marzi, E., Mesbah, S., Mistry, M., Tirado, A. C. M., Nguyen, A., Nguyen, V. B., Oelen, A., Pasqual, V., Paulheim, H., Polleres, A., Porena, M., Portisch, J., Presutti, V., Pustu-Iren, K., Mendez, A. R., Roshankish, S., Rudolph, S., Sack, H., Sakor, A., Salas, J., Schleider, T., Shi, M., Spinaci, G., Sun, C., Tietz, T., Dhouib, M. T., Umbrico, A., van den Berg, W., and Xu, W. Knowledge Graphs Evolution and Preservation - A Technical Report from ISWS 2019. *CoRR abs/2012.11936* (2020).

[32] Adams, W. Conducting Semi-Structured Interviews. In *Handbook of Practical Program Evaluation*, J. Wholey, H. Hatry, and K. Newcomer, Eds. Jossey-Bass, 2015, pp. 492–505.

[33] Alom, H. A Library for Visualizing SHACL over Knowledge Graphs. Master's thesis, Hannover: Gottfried Wilhelm Leibniz Universität Hannover, Mar. 2022.

[34] Antony, J. 2 - Fundamentals of Design of Experiments. In *Design of Experiments for Engineers and Scientists (Second Edition)*, J. Antony, Ed., second edition ed. Elsevier, Oxford, 2014, pp. 7–17.

[35] Antony, J. 4 - A Systematic Methodology for Design of Experiments. In *Design of Experiments for Engineers and Scientists (Second Edition)*, J. Antony, Ed., second edition ed. Elsevier, Oxford, 2014, pp. 33–50.

[36] Ayala, D., Borrego, A., Hernández, I., Rivero, C. R., and Ruiz, D. AYNEC: All You Need for Evaluating Completion Techniques in Knowledge Graphs. In *ESWC* (2019), vol. 11503 of *Lecture Notes in Computer Science*, Springer, pp. 397–411.

[37] Bello, M. J. G. Scientific Prototyping: A Novel Approach to Conduct Research and Engineer Products. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)* (Nov. 2018), pp. 32–35.

[38] Boneva, I., Dusart, J., Fernández-Álvarez, D., and Gayo, J. E. L. Shape Designer for ShEx and SHACL constraints. In *ISWC (Satellites)* (2019), vol. 2456 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 269–272.

[39] Camburn, B., Viswanathan, V., Linsey, J., Anderson, D., Jensen, D., Crawford, R., Otto, K., and Wood, K. Design prototyping methods: state of the art in strategies, techniques, and guidelines. *Design Science 3* (2017), 1–33.

[40] Chen, H., Cao, G., Chen, J., and Ding, J. A Practical Framework for Evaluating the Quality of Knowledge Graph. In *Knowledge Graph and Semantic Computing: Knowledge Computing and Language Understanding* (2019), X. Zhu, B. Qin, X. Zhu, M. Liu, and L. Qian, Eds., vol. 1134 of *Communications in Computer and Information Science*, Springer, pp. 111–122.

[41] Chiem Dao, D. Using Knowledge Graph Technologies to Contextualize and Validate Declarations in the Social Security Domain. Master's thesis, Université de Liège, Liège, Belgique, June 2023.

[42] Choi, J.-W. Automatic Construction of SHACL Schemas for RDF Knowledge Graphs Generated by Direct Mappings. *25*, 10 (2020), 23–34.

[43] Cimmino, A., Fernández-Izquierdo, A., and García-Castro, R. Astrea: Automatic Generation of SHACL Shapes from Ontologies. In *The Semantic Web* (2020), A. Harth, S. Kirrane, A.-C. Ngonga Ngomo, H. Paulheim, A. Rula, A. L. Gentile, P. Haase, and M. Cochez, Eds., vol. 12123 of *Lecture Notes in Computer Science*, Springer, pp. 497–513.

[44] Corman, J., Reutter, J. L., and Savkovic, O. Semantics and Validation of Recursive SHACL. In *The Semantic Web – ISWC 2018* (2018), D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, and E. Simperl, Eds., vol. 11136 of *Lecture Notes in Computer Science*, Springer, pp. 318–336.

[45] De Meester, B. *Improving and assessing data quality of knowledge graphs.* PhD Thesis, Ghent University, 2020.

[46] Dibowski, H. Property Assertion Constraints for an Informed, Error-Preventing Expansion of Knowledge Graphs. In *Knowledge Graphs and Semantic Web* (2021), vol. 1459 of *Communications in Computer and Information Science*, Springer, pp. 234–248.

[47] Dorsch, R., Freund, M., Fries, J., and Harth, A. Graphguard: Enhancing Data Quality in Knowledge Graph Pipelines. In *SemIIM* (2023), vol. 3647 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[48] Duan, X., Chaves-Fraga, D., Derom, O., and Dimou, A. SCOOP All the Constraints' Flavours for Your Knowledge Graph. In *ESWC (2)* (2024), vol. 14665 of *Lecture Notes in Computer Science*, Springer, pp. 217–234.

[49] Durakovic, B. Design of experiments application, concepts, examples: State of the art. *Periodicals of Engineering and Natural Sciences 5* (Dec. 2017), 421–439.

[50] Esteban, C., Tresp, V., Yang, Y., Baier, S., and Krompaß, D. Predicting the co-evolution of event and Knowledge Graphs. In *2016 19th International Conference on Information Fusion (FUSION)* (July 2016), IEEE, pp. 98–105.

[51] Fensel, D., Simsek, U., Angele, K., Huaman, E., Kärle, E., Panasiuk, O., Toma, I., Umbrich, J., and Wahler, A. *Knowledge Graphs - Methodology, Tools and Selected Use Cases.* Springer, 2020.

[52] Fernandez-Álvarez, D., Labra-Gayo, J. E., and Gayo-Avello, D. Automatic extraction of shapes using sheXer. *Knowledge-Based Systems 238* (Feb. 2022), 107975.

[53] Ferranti, N., Polleres, A., de Souza, J. F., and Ahmetaj, S. Formalizing Property Constraints in Wikidata. In *Wikidata@ISWC* (2022), vol. 3262 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[54] Figuera, M., Rohde, P. D., and Vidal, M.-E. Trav-SHACL: Efficiently Validating Networks of SHACL Constraints. In *Proceedings of the Web Conference 2021* (June 2021), WWW '21, Association for Computing Machinery, pp. 3337–3348.

[55] Gayo, J. E. L., Prud'hommeaux, E., Boneva, I., and Kontokostas, D. Comparing ShEx and SHACL. In *Validating RDF Data*, J. E. L. Gayo, E. Prud'hommeaux, I. Boneva, and D. Kontokostas, Eds. Springer International Publishing, 2018, pp. 233–266.

[56] González, L., and Hogan, A. Modelling Dynamics in Semantic Web Knowledge Graphs with Formal Concept Analysis. In *Proceedings of the 2018 World Wide Web Conference* (Republic and Canton of Geneva, CHE, Apr. 2018), WWW '18, International World Wide Web Conferences Steering Committee, pp. 1175–1184.

[57] Hagedorn, P., Pauwels, P., and König, M. Semantic rule checking of cross-domain building data in information containers for linked document delivery using the shapes constraint language. *Automation in Construction 156* (Dec. 2023), 105106.

[58] Haller, A., Fernández, J. D., Kamdar, M. R., and Polleres, A. What Are Links in Linked Open Data? A Characterization and Evaluation of Links between Knowledge Graphs on the Web. *Journal of Data and Information Quality 12*, 2 (May 2020), 9:1–9:34.

[59] Heist, N., Hertling, S., Ringler, D., and Paulheim, H. Knowledge Graphs on the Web-An Overview. In *Knowledge Graphs for eXplainable Artificial Intelligence*, vol. 47 of *Studies on the Semantic Web*. IOS Press, 2020, pp. 3–22.

[60] Hevner, A. R. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems 19*, 2 (Jan. 2007). Article 4.

[61] Hevner, A. R., March, S. T., Park, J., and Ram, S. Design Science in Information Systems Research. *Management Information Systems Quarterly 28*, 1 (Mar. 2004), 75–105.

[62] Hose, K., and Sallinger, E. ISS-1. Available in TUWEL, Nov. 2023. Lecture notes for the course 188.399 Introduction to Semantic Systems at TU Wien.

[63] Huaman, E. Steps to Knowledge Graphs Quality Assessment. *arXiv.org abs/2208.07779* (2022).

[64] Issa, S., Adekunle, O., Hamdi, F., Cherfi, S. S.-S., Dumontier, M., and Zaveri, A. Knowledge Graph Completeness: A Systematic Literature Review. *IEEE Access 9* (2021), 31322–31339.

[65] Ji, S., Pan, S., Cambria, E., Marttinen, P., and Yu, P. S. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Transactions on Neural Networks and Learning Systems 33*, 2 (Feb. 2022), 494–514.

[66] Kejriwal, M. Knowledge Graphs: A Practical Review of the Research Landscape. *Information 13*, 4 (Apr. 2022), 161.

[67] Keshavarzi, A., and Kochut, K. J. KGdiff: Tracking the Evolution of Knowledge Graphs. In *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)* (Aug. 2020), pp. 279–286.

[68] Kitchenham, B. A., and Charters, S. Guidelines for performing Systematic Literature Reviews in Software Engineering. Tech. Rep. EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[69] Liu, J., Yu, Z., Guo, B., Deng, C., Fu, L., Wang, X., and Zhou, C. EvolveKG: a general framework to learn evolving knowledge graphs. *Frontiers Comput. Sci. 18*, 3 (Jan. 2024), 183309.

[70] Liu, J., Zhang, Q., Fu, L., Wang, X., and Lu, S. Evolving Knowledge Graphs. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* (Paris, France, Apr. 2019), IEEE, pp. 2260–2268.

[71] Liu, S., d'Aquin, M., and Motta, E. Measuring Accuracy of Triples in Knowledge Graphs. In *Language, Data, and Knowledge* (Cham, 2017), J. Gracia, F. Bond, J. P. McCrae, P. Buitelaar, C. Chiarcos, and S. Hellmann, Eds., vol. 10318, Springer International Publishing, pp. 343–357.

[72] Mayring, P. Qualitative Content Analysis. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research 1*, 2 (June 2000). Art. 20.

[73] Mihindukulasooriya, N., Rashid, M. R. A., Rizzo, G., García-Castro, R., Corcho, O., and Torchiano, M. RDF shape induction using knowledge base profiling. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Apr. 2018), SAC '18, ACM, pp. 1952–1959.

[74] Nishioka, C., and Scherp, A. Analysing the Evolution of Knowledge Graphs for the Purpose of Change Verification. In *2018 IEEE 12th International Conference on Semantic Computing (ICSC)* (Jan. 2018), IEEE Computer Society, pp. 25–32.

[75] Omran, P. G., Taylor, K., Méndez, S. J. R., and Haller, A. Towards SHACL Learning from Knowledge Graphs. In *ISWC (Demos/Industry)* (2020), vol. 2721 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 94–99.

[76] Omran, P. G., Taylor, K., Méndez, S. J. R., and Haller, A. Learning SHACL shapes from knowledge graphs. *Semantic Web 14*, 1 (2023), 101–121.

[77] Pandit, H. J., O'Sullivan, D., and Lewis, D. Using Ontology Design Patterns To Define SHACL Shapes. In *WOP@ISWC* (2018), vol. 2195 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 67–71.

[78] Pandit, H. J., O'Sullivan, D., and Lewis, D. An Argument for Generating SHACL Shapes from ODPs. In *Advances in Pattern-Based Ontology Engineering*, vol. 51 of *Studies on the Semantic Web*. IOS Press, 2021, pp. 134–141.

[79] Peffers, K., Rothenberger, M., Tuunanen, T., and Vaezi, R. Design Science Research Evaluation. In *Design Science Research in Information Systems. Advances in Theory and Practice* (2012), K. Peffers, M. Rothenberger, and B. Kuechler, Eds., vol. 7286 of *Lecture Notes in Computer Science*, Springer, pp. 398–410.

[80] Pelgrin, O., Galárraga, L., and Hose, K. Towards fully-fledged archiving for RDF datasets. *Semantic Web 12*, 6 (2021), 903–925. Publisher: IOS Press.

[81] Pfund, J., Lindlein, N., Schwider, J., Burow, R., Blümel, T., and Elssner, K.-E. Absolute Sphericity Measurement: a Comparative Study of the Use of Interferometry and a Shack–Hartmann Sensor. *Opt. Lett. 23*, 10 (May 1998), 742–744. Publisher: Optica Publishing Group.

[82] Pirrò, G. Explaining and Suggesting Relatedness in Knowledge Graphs. In *The Semantic Web - ISWC 2015* (Cham, 2015), M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, K. Thirunarayan, and S. Staab, Eds., vol. 9366 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 622–639.

[83] Polleres, A., Pernisch, R., Bonifati, A., Dell'Aglio, D., Dobriy, D., Dumbrava, S., Etcheverry, L., Ferranti, N., Hose, K., Jiménez-Ruiz, E., Lissandrini, M., Scherp, A., Tommasini, R., and Wachs, J. How Does Knowledge Evolve in Open Knowledge Graphs? *Transactions on Graph Data and Knowledge 1*, 1 (Dec. 2023), 11:1–11:59. Publisher: Dagstuhl.

[84] Rabbani, K., Lissandrini, M., and Hose, K. Extraction of Validating Shapes from Very Large Knowledge Graphs. *Proceedings of the VLDB Endowment 16*, 5 (Jan. 2023), 1023–1032.

[85] Rabbani, K., Lissandrini, M., and Hose, K. SHACTOR: Improving the Quality of Large-Scale Knowledge Graphs with Validating Shapes. In *Companion of the 2023 International Conference on Management of Data* (New York, NY, USA, 2023), SIGMOD '23, Association for Computing Machinery, pp. 151–154.

[86] Rossanez, A., dos Reis, J. C., and da Silva Torres, R. Representing Scientific Literature Evolution via Temporal Knowledge Graphs. In *MEPDaW@ISWC* (2020), vol. 2821 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 33–42.

[87] Salmon, T. O., Thibos, L. N., and Bradley, A. Comparison of the eye's wave-front aberration measured psychophysically and with the Shack–Hartmann wave-front sensor. *J. Opt. Soc. Am. A 15*, 9 (Sept. 1998), 2457–2465. Publisher: Optica Publishing Group.

[88] Sanders, P. Algorithm Engineering – An Attempt at a Definition. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, S. Albers, H. Alt, and S. Näher, Eds., vol. 5760 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 321–340.

[89] Spahiu, B., Maurino, A., and Palmonari, M. Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL. In *WOP@ISWC* (2018), vol. 2195 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 52–66.

[90] Szeredi, P., Lukácsy, G., Benkő, T., and Nagy, Z. The Semantic Web and the RDF language. In *The Semantic Web Explained: The Technology and Mathematics behind Web 3.0*. Cambridge University Press, 2014, pp. 52–121.

[91] Tasnim, M., Collarana, D., Graux, D., Orlandi, F., and Vidal, M.-E. Summarizing Entity Temporal Evolution in Knowledge Graphs. In *Companion Proceedings of The 2019 World Wide Web Conference* (May 2019), WWW '19, Association for Computing Machinery, pp. 961–965.

[92] Trivedi, R., Dai, H., Wang, Y., and Song, L. Know-Evolve: Deep Temporal Reasoning for Dynamic Knowledge Graphs. In *Proceedings of the 34th International Conference on Machine Learning* (July 2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 3462–3471.

[93] Voit, M. M., and Paulheim, H. Bias in Knowledge Graphs – an Empirical Study with Movie Recommendation and Different Language Editions of DBpedia. *CoRR abs/2105.00674* (May 2021). Publisher: arXiv.

[94] Weikum, G. Knowledge Graphs 2021: A Data Odyssey. *Proceedings of the VLDB Endowment 14*, 12 (July 2021), 3233–3238.

[95] Xue, B., and Zou, L. Knowledge Graph Quality Management: A Comprehensive Survey. *IEEE Transactions on Knowledge and Data Engineering 35*, 5 (May 2023), 4969–4988.