Bachelor of Science Thesis

Automated standard compliance testing and visualization for the QLever SPARQL engine

Rico Andris

March 25, 2024

Submitted to the University of Freiburg
IIF – Department of Computer Science
Chair for Algorithms and Data Structures

universität freiburg

University of Freiburg
IIF – Department of Computer Science
Chair for Algorithms and Data Structures

Author	Rico Andris, Matriculation Number: 4943295
Editing Time	January 23, 2024 - March 25, 2024
Examiners	Prof. Dr. Hannah Bast, IIF – Department of Computer Science Chair for Algorithms and Data Structures
Supervisor	M.Sc. Johannes Kalmbach, IIF – Department of Computer Science Chair for Algorithms and Data Structures
Declaration	I hereby declare, that I am the sole author and composer of this Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 25.03.2024
Place, Date

Rico Andris
Signature

Abstract

This thesis addresses the challenge of enhancing the reliability and standard compliance of the QLever SPARQL engine, a tool utilized for querying RDF data. The main focus of this work involves automating the process of executing a set of standardized tests (SPARQL 1.1 test suite) against the engine to identify errors and areas of non-compliance. To make the results easily accessible, a web-based visualization tool was developed, enabling developers to quickly identify and rectify discrepancies. Additionally, this work established the foundation for integrating these tools into the QLever project's GitHub workflow, ensuring that the engine remains compliant with SPARQL 1.1 standards with each update. This enables developers, even those who are not well-versed in SPARQL standards, to promptly identify and resolve compliance issues.

Contents

Chapter 1

Introduction

This chapter outlines the motivation and objective for this thesis.

1.1 Motivation

The motivation behind this thesis stems from the need to enhance the reliability of QLever and to further QLever's support for the SPARQL 1.1 standard. Our goal is to streamline the testing process and make it more efficient by automating the execution of the SPARQL 1.1 test suite. The visualization of the test results makes it easier to identify functionality that is not SPARQL 1.1 compliant or has not yet been implemented by QLever. Integrating this automated testing into the QLever engine's GitHub workflow ensures continuous compliance with the SPARQL 1.1 standard. This facilitates the development of a more robust SPARQL engine.

1.2 Objective

The aim of this thesis is to develop a tool that automates the execution of the SPARQL 1.1 test suite and visually represents the results for easy analysis. Furthermore, this project takes the initial steps to integrate the testing and visualization framework into QLever's GitHub workflow, promoting continuous improvement that maintains the engine's quality and adherence to standards throughout its development.

Chapter 2

Background

This chapter provides the background necessary to understand the thesis. It introduces the SPARQL Protocol and RDF Query Language, the SPARQL 1.1 test suite, QLever and Bootstrap.

2.1 SPARQL

This section explains basic features of the Resource Description Framework (RDF) standard for knowledge bases and the SPARQL query language that are necessary to understand our work.

2.1.1 Semantic Web

The Semantic Web is an extension of the current web. Its goal is to enable machines to understand and interpret data on the web. To achieve this, the Semantic Web relies on standards and technologies that categorize and link data, such as the Resource Description Framework standard. This enables computers to process the data directly, resulting in more intelligent applications that improve tasks like data integration and personalized content delivery. The Semantic Web is a significant advancement in making the internet more accessible for automated processing.

2.1.2 Resource Description Framework

The Resource Description Framework (RDF) is a standard for exchanging data on the semantic web.[1] RDF uses a graph-based model to represent information

[1]Find out more at https://www.w3.org/RDF/

and employs Uniform Resource Identifiers (URIs) to denote relationships between entities. This approach ensures semantic precision and enhances data readability for computers. Additionally, RDF facilitates linking of diverse data sets through semantic relationships. The Resource Description Framework (RDF) is crucial for the operation of the Semantic Web and for improving data interconnectivity and interpretation. An example knowledge base, of the RDF standard, detailing animals and their classification could look like this:

Subject	Predicate	Object
`<Dog>`	`<isA>`	`<Mammal>`
`<Cat>`	`<isA>`	`<Mammal>`
`<Parrot>`	`<isA>`	`<Bird>`

Table 2.1: Example RDF knowledge base

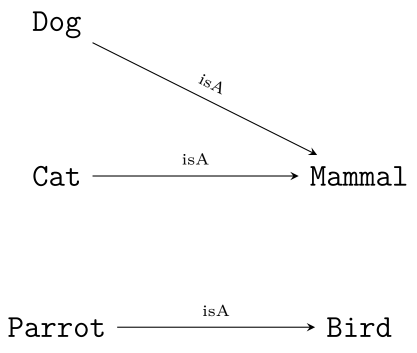


Figure 2.1: Graph representation of the example knowledge base 2.1

Terse RDF Triple Language

The Terse RDF Triple Language (Turtle) format is a textual syntax for expressing data in the Resource Description Framework that is intended to be easier to read and write by humans compared to other RDF serializations like RDF/XML. This example about a golden retriever called Rex, makes the benefits of using the Turtle format very clear:

Listing 2.1: Turtle syntax for describing a dog

```
@prefix ex: <http://example.org/animals#> .

ex:Rex
```

```
    ex:type ex:Dog ;
    ex:name "Rex" ;
    ex:breed "Golden Retriever" ;
    ex:age 5 .
```

Listing 2.2: RDF/XML syntax for describing a dog

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:ex="http://example.org/animals#">

  <rdf:Description rdf:about="http://example.org/animals#Rex">
    <ex:type rdf:resource="http://example.org/animals#Dog"/>
    <ex:name>Rex</ex:name>
    <ex:breed>Golden Retriever</ex:breed>
    <ex:age rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
    5</ex:age>
  </rdf:Description>

</rdf:RDF>
```

2.1.3 SPARQL Protocol and RDF Query Language

The SPARQL Protocol and RDF Query Language (SPARQL) is a cornerstone of the Semantic Web, enabling the querying and manipulation of data stored in the Resource Description Framework.[2] SPARQL was developed to address the need for a standardized query language for RDF data and since then has evolved to include features like update capabilities in SPARQL 1.1. SPARQL is endorsed and standardized by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium (W3C).

Key Features

SPARQL's key features include federated querying, which treats data from multiple sources as a single dataset, and a variety of query forms (SELECT, CONSTRUCT, ASK, DESCRIBE) that allow users to extract, manipulate, and inquire about data in versatile ways. This flexibility supports a broad range of applications. An example query for the knowledge base 2.1 can look like this:

[2]Find out more at https://www.w3.org/TR/sparql11-query/

Listing 2.3: Example SPARQL query to select animals

```
SELECT ?animal
WHERE {
  ?animal isA Mammal .
}
```

?animal
<Dog>
<Cat>

Table 2.2: Example SPARQL query response in the CSV format

2.2 SPARQL Test Suite

2.2.1 Test Suite

A test suite is a collection of test cases that are designed to validate the behavior of a software system. This validation process ensures that the software performs as expected under various conditions and that any changes or updates to the code do not inadvertently introduce errors or regressions.

2.2.2 SPARQL 1.1 Test Suite

To ensure the consistency, compatibility, and reliability of SPARQL implementations across different data stores and applications, the SPARQL 1.1 Test Suite was developed. This comprehensive set of tests verifies that SPARQL query processors adhere to the standards defined by the World Wide Web Consortium (W3C) for SPARQL 1.1.[3]

For better manageability, the tests are split into smaller collections based on the functionality being tested. Each collection has its own directory and manifest file. The manifest file lists the tests in the collection and provides details about them. This is done using the RDF format and in this case is is expressed in Turtle (Section 2.1.2). This makes the test readable for humans and machines. Depending on the test the manifest entry of the test contains different information. An example test called test-01, which specifies the action to run the query *query01.rq*

[3]Find out more at https://www.w3.org/2009/sparql/docs/tests/README.html

on the data *graph01.ttl* and specifies the result in the *result01.srx* , could look like this:

Listing 2.4: How a test is specified in a manifest file

```
:test-01 rdf:type mf:QueryEvaluationTest ;
   mf:name "Test 1" ;
   rdfs:comment "A very useful test" ;
   mf:action
       [ qt:query <query01.rq> ;
         qt:data <graph01.ttl> ] ;
   mf:result <result01.srx> .
```

Components of the SPARQL 1.1 Test Suite

The test suite encompasses several types of tests, each targeting specific features of the SPARQL 1.1 standard:

- Query Evaluation Tests.
- Syntax Tests.
- Result Format Tests.
- Update Evaluation Tests.
- Protocol Tests.
- Service Description Tests.

Query Evaluation Tests

Query Evaluation Tests evaluate the correctness of a query execution and always consist of a correct SPARQL query to be tested and an expected result. SPARQL includes several query result formats:

- SPARQL Query Results XML Format.
- SPARQL Query Results JSON Format.
- SPARQL Query Results TSV Format.
- RDF/XML or Turtle.

In some cases a default graph is given, on which the query is executed. There are special Query Evaluation Tests Entailment Evaluation Tests and Federated Query Tests.

Entailment Evaluation Tests additionally test the entailment regime and entailment profile. Entailment regimes in SPARQL allow users to perform queries that can infer new information from the data that is not explicitly stated.

Federated Query Tests are tests designed to test queries who use the SERVICE keyword. The SERVICE keyword will be executed against an external SPARQL endpoint. The results are then integrated into the overall query results. This allows the user to combine data from multiple sources in a single query, without needing to manually gather and merge data from these sources.

Syntax Tests

Syntax Tests verify that SPARQL query processors correctly interpret and handle the syntax of SPARQL queries according to the specifications of the SPARQL language.

Syntax tests only consist of a query and expect either a positive or negative response to that query from the SPARQL endpoint.

Result Format Tests

SPARQL queries can produce results in various formats, and format tests are used to verify the accuracy and compliance of these output formats. One of these tests is the CSV Result Format Test. The CSV and TSV result formats are lossy formats, meaning they cannot encode all the details of a result. Due to this loss of detail, the format requires special handling. Otherwise, the format Tests work like query evaluation tests.

Listing 2.5: Example result of a CSV test

```
subject,predicate,object
http://example.org/Dog,http://example.org/age,"5"
```

Table 2.3: Possible CSV result matches

Subject	Predicate	Object
`http://example.org/Dog`	`http://example.org/age`	`"5"^^xsd:string`
`http://example.org/Dog`	`http://example.org/age`	`"5"^^xsd:decimal`
`http://example.org/Dog`	`http://example.org/age`	`"5"^^xsd:integer`

Update Evaluation Tests

Update Evaluation Tests verify the implementation of SPARQL Update operations, ensuring correct data manipulation and integrity. Update Evaluation Tests always contain an update query. A update query can contain several graphs, a default graph and any amount of named graphs. The query can make use of all of them.

Protocol Tests

Protocol Tests ensure the correct support of the transmission of queries to the SPARQL service, typically via HTTP GET and POST methods. This includes testing the correct handling of query parameters and content types. Protocol tests consist of an HTTP request and the expected HTTP response.

Special protocol tests are the Graph Store HTTP Protocol Tests. The Graph Store HTTP Protocol defines how clients can manage RDF graph content directly via HTTP.

Service Description Tests

Assess the ability of SPARQL services to accurately describe their capabilities, as per the SPARQL Service Description specification.

2.3 QLever

QLever is a high-performance SPARQL engine developed by the Chair for Algorithms and Data Structures at the University of Freiburg. [1] The SPARQL engine is designed to efficiently execute queries over large RDF datasets. QLever uses a unique indexing strategy that allows it to handle very large datasets more efficiently than many other SPARQL engines. QLever aims to fully support the SPARQL 1.1 query language standard, enabling users to perform a wide range of queries, from simple data retrievals to complex analytical queries. To work with the QLever SPARQL engine we first have to index our graph data. After that we can start the QLever server and execute our SPARQL queries. QLever offers more features that are not relevant for our work.[4] [5]

2.4 Bootstrap

Bootstrap is a free and open-source front-end framework for designing websites

[4]Use QLever at https://qlever.cs.uni-freiburg.de/wikidata
[5]QLever Code https://github.com/ad-freiburg/qlever

and web applications. It provides HTML, CSS, and JavaScript templates for typography, forms, buttons, navigation, and other interface components, as well as optional JavaScript extensions. [6]

[6]Find Bootstrap at https://getbootstrap.com/

Chapter 3

Implementation

This chapter presents an overview of our approach to automating the testing of the SPARQL 1.1 suite for the QLever engine. We also discuss how we visualize differences in test results with the QLever result and the development of the visualization website. Finally, we address the integration of the process into a GitHub workflow.

3.1 Interactions with QLever

This work implements the SPARQL 1.1 test suite for the QLever SPARQL engine and requires a Python interface to interact with the QLever binaries and endpoint.

3.1.1 Running QLever Commands

Some tests include a starting graph for the test. If we want to load a graph into QLever, we have to build an index of that graph, using the QLever *IndexBuilderMain* executable. For graphs with a format not supported by QLever, we use RDFlib to parse it to the turtle format. We also need to remove the index when the next graph needs to be indexed.[1]

Another task is starting and stopping the SPARQL endpoint after indexing. To do this we need to start the Qlever server using the QLever *ServerMain* executable. To achieve these tasks, we use the subprocess python module.[2] This module allows us to execute external commands and interact with other programs in python, as if they were running in the terminal. This is crucial for the tasks mentioned before.

[1]RDFlib at https://rdflib.readthedocs.io/en/stable/
[2]subprocess module at https://docs.python.org/3/library/subprocess.html

The reason we chose the subprocess module, instead of using the os.system module, is the finer control over the subprocess's input, output and error pipes. We use this to capture the output and error messages to give useful information as to why the test encountered an error, for example during indexing. The module also makes it easier to identify and troubleshoot issues with the external commands or processes by providing exceptions.

3.1.2 Sending SPARQL Queries

A crucial part is sending SPARQL queries to the SPARQL endpoint, in this case the QLever server. The SPARQL 1.1 Protocol defines the query and update operation. For the query operation we can either use the HTTP GET method or the HTTP POST method, for the update operation we have to use the HTTP POST method. In our implementation, we use the HTTP POST method because it allows us to directly put the query string, which is read from the given query or update file, into the HTTP Request Message Body, and we do not need different functions for a query operation and update operation. There are two options for sending a POST request. We chose the POST directly and not the URL-encoded version, which is just unnecessary in this case.

An important part is specifying the format to be returned by the QLever server. We do this, using the HTTP header ”Accept”. This header tells the server what the client is able to interpret, using MIME types. We identify the needed MIME type, for a given test, by looking at the format of the result. If a test includes a result file, we use the file extension to indicate the format. For instance, a result file named example.srj indicates the SPARQL 1.1 Query Results JSON format, as denoted by the file extension srj, which in turn specifies the MIME type application/sparql-results+json.

For executing POST requests, our implementation adopts the requests module. This decision is anchored in the module's user-friendliness, compatibility and comprehensive documentation. The module ensures a seamless and consistent interface irrespective of the underlying HTTP library differences. It simplifies HTTP/1.1 requests, eliminating the complexities of manual query string manipulation or POST data form-encoding. It offers several functions like get, post, put and delete, which are all intuitive and make readable code.

3.2 Extracting Tests

To begin, we required a method for extracting the tests outlined in the SPARQL 1.1 test suite. Each test is detailed in a manifest file written in the Terse RDF Triple Language (Turtle) format. There are various methods for extracting data from a Turtle file, but we opted to use SPARQL queries to extract the necessary data. To accomplish this task, a SPARQL engine is required. As we had to implement the QLever SPARQL engine for testing purposes, we will use it to run the queries and extract the tests.

For each test group we wrote a corresponding SPARQL query. Using QLever we parse the manifest Turle file and run the query. Each group of tests then gets saved in a CSV format file, where a row represents the details for one test. The tests are split into groups that correspond with the task of running the tests.

Listing 3.1: Example of a query to extract tests

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mf: ↩
    <http://www.w3.org/2001/sw/DataAccess/tests/test-manifest#>
PREFIX dawgt: ↩
    <http://www.w3.org/2001/sw/DataAccess/tests/test-dawg#>
PREFIX qt: <http://www.w3.org/2001/sw/DataAccess/tests/test-query#>
PREFIX ut: <http://www.w3.org/2009/sparql/tests/test-update#>
PREFIX sd: <http://www.w3.org/ns/sparql-service-description#>
PREFIX ent: <http://www.w3.org/ns/entailment/RDF>
PREFIX rs: <http://www.w3.org/2001/sw/DataAccess/tests/result-set#>

SELECT DISTINCT ?type ?name ?query ?result ?data ?test ↩
    (GROUP_CONCAT(DISTINCT ?feature; SEPARATOR=";") AS ↩
    ?featureList) ?comment ?approval (GROUP_CONCAT(DISTINCT ↩
    ?approvedBy; SEPARATOR=";") AS ?approvedByList) ?regime
WHERE {
    ?test rdf:type mf:QueryEvaluationTest .
    BIND ("QueryEvaluationTest" AS ?type) .
    ?test mf:action ?action .
    ?action qt:query ?query .
    OPTIONAL {?action qt:data ?data .}
    OPTIONAL {?action sd:entailmentRegime ?regime .}
    OPTIONAL {?action qt:graphData ?actionGraphData .}
    ?test mf:result ?result .
    OPTIONAL {?test mf:name ?name .}
    OPTIONAL {?test mf:feature ?feature .}
    OPTIONAL {?test rdfs:comment ?comment .}
    OPTIONAL {?test dawgt:approval ?approval .}
    OPTIONAL {?test dawgt:approvedBy ?approvedBy .}
}
GROUP BY ?type ?name ?query ?result ?data ?test ?comment ↩
    ?approval ?regime
```

3.3 Evaluating Tests

A big part of evaluating tests consists of comparing the given expected result, with the reponse given by the QLever server. These results come in different formats. For each format, we implemented a comparison. Automating the tests of the SPARQL 1.1 test suite for the QLever SPARQL engine, meant a lot of attention was dedicated to the comparison of query results in various formats This section delves into the design choices, unique implementations, and module usage within the comparison functionality for each format.

3.3.1 XML Result Comparison

The core XML handling functionality was developed with a focus on correct results and flexibility of what is considered equal. Before comparing the results we have to parse the XML into use-able data, in our approach we build ElementTrees. A result tree is considered equal, if both trees are empty after deleting every matching element.

ElementTree for XML Parsing

The xml.etree.ElementTree (ET)[3] module was chosen for parsing XML documents due to its balance of efficiency and ease of use. ET provides a straightforward API for navigating and manipulating the XML tree structure, which is essential for detailed comparison of query results. Several alternatives for XML processing

[3]https://docs.python.org/3/library/xml.etree.elementtree.html

exist in the Python ecosystem, such as lxml and minidom. In comparison to the powerful tool lxml, ET was selected for its standard library availability, eliminating external dependencies, and its simplicity for the required tasks. While lxml offers more features and potentially better performance, its additional complexity and dependency requirements were deemed unnecessary for the core comparison needs of this project.

Custom XML Comparison Logic

Instead of leveraging existing XML comparison libraries, a custom comparison logic was implemented. The decision to develop a custom XML comparison mechanism, rather than utilizing an existing library, was based on the unique requirements of SPARQL result comparison. Existing libraries often focus on general-purpose XML comparison, which may not cater to the nuanced differences significant in SPARQL results, such as order-insensitive comparison of certain elements. Another reason was to provide detailed feedback on discrepancies, which will later be used to highlight errors. Additionally, we have full control over what is deemed equal.

The custom logic includes a detailed comparison of XML elements, attributes, and text content. Furthermore, it tracks and compares the use of blank nodes, which are SPARQL specific. When comparing data types it also allows for two different types to be considered equal. This was implemented to differentiate between a failed test and a test that failed because of a QLever design decision. The data types that are considered equal are stored in a config file. Numeric types also need special handling since there might be formatting differences, that we consider equal.

3.3.2 JSON Result Comparison

In our approach, we take advantage of the SPARQL JSON format. The SPARQL standard allows two different structures. A special one for ASK queries, which answer with a boolean, and the general one, which contains the results of a query. The general structure has two relevant lists called *vars* and *bindings*. We consider the lists to be the same if both are empty after deleting matches. The special structure also has the *vars* list and instead of the bindings list it has a boolean, the answer to the ASK query. For the vars list we use the same approach as before and for the boolean we just compare the values.

Module for JSON Parsing

The decision to use Python's built-in json module[4] for reading and writing JSON files stems from its sufficient functionality for the task at hand and its integration into the Python standard library. This choice avoids additional dependencies and leverages familiar syntax and features.

Custom JSON Comparison Logic

While libraries like jsondiff exist for comparing JSON documents, a custom solution was developed to address the specific nuances of SPARQL and of QLever. This includes handling of unordered lists (where list order does not indicate a difference) and specific formatting needs for the test suite's output. The custom approach ensures that comparisons are tailored to the domain-specific requirements of SPARQL, which may not be fully met by general-purpose JSON comparison tools.

3.3.3 CSV/TSV Result Comparison

In the initial phase, we process the results, which are in either CSV or TSV format, by converting them into a Python list of lists, where each nested list corresponds to a row. Disregarding the order, we remove any rows that are identical. Should this operation result in both lists being empty, we determine the results to be equivalent.

Python's Standard CSV Module

The decision to use Python's standard csv module[5] for handling CSV/TSV files was motivated by its robustness, ease of use, and the absence of external dependencies. This choice supports a wide range of CSV-related operations, from file writing to custom parsing, with sufficient flexibility and features for the test suite's requirements.

Custom Comparison Mechanisms

While third-party libraries for CSV/TSV manipulation and comparison exist, custom implementations were preferred to meet the specific needs of SPARQL CSV

[4]https://docs.python.org/3/library/json.html
[5]https://docs.python.org/3/library/csv.html

and TSV data. This approach allows for more granular control over comparison logic, especially in handling special cases unique to SPARQL.

Beyond basic file handling, custom comparison mechanisms were implemented to address the nuances of SPARQL CSV and TSV data. This includes handling numeric data accurately, ensuring that differences in number formatting do not contribute to comparison mismatches and the special handling of the lossy nature of the format and ignoring order or rows and columns when comparing results.

Special care was taken to correctly handle numeric values within CSV/TSV rows by incorporating utility functions, to ensure that numeric comparisons are accurate and not influenced by format variations, such as decimal places or leading zeros. For example 1.234E3 is the equal to 1234.

Because of the lossy nature of the SPARQL CSV/TSV format, we cut of information about data types, only comparing the actual values.

3.3.4 Turtle Result Comparison

Using the RDFlib[6] we turn the given RDF data into a graphs and check if the graphs are isomorphic.

RDFlib for RDF Processing

RDFlib is a powerful and widely used library for working with RDF in Python. The tool's capability to parse and serialize RDF data in various formats is a big reason why we chose this library. In our work, RDFlib facilitates the conversion between different RDF serializations, such as RDF/XML to Turtle, enabling a standardized approach to graph comparison.

RDFlib for Equivalence Checking

The tool employs RDFlib's graph isomorphism features to determine the equivalence of RDF graphs. This method allows for a deep comparison that accounts for RDF's graph nature, ensuring that semantically equivalent graphs are recognized as such, regardless of their serialization differences. This means there is no need for a custom comparison logic.

[6]https://rdflib.readthedocs.io/en/stable/

3.3.5 Query Evaluation Tests and Result Format Tests

A critical component for the QLever engine, is the integration of the SPARQL 1.1 query evaluation tests to validate the engine's compliance with the SPARQL standard. Query evaluation tests and result format tests can be run the same way. So in our approach we don't differentiate between them when running the tests.

A query evaluation test either defines a default graph or the default graph is simply an empty graph. Since these tests do not change the data we do not need to reset the graph after each test. This lets us group the tests using the same default graph. For each group we have to set up the SPARQL endpoint only once. How we setup a SPARQL endpoint with QLever is explained in chapter 3.1.1. Then we send the queries as explained in chapter 3.1.2. The last step is evaluating the results using the comparison functions explained in section 3.3.1 - 3.3.4.

3.3.6 Syntax Tests

Syntax tests are important to test the query processor of the SPARQL engine. Since the given result to a query of a syntax test, does not matter, all syntax tests are run on an empty graph. To check if a syntax tests passes or fails, we just have to check if the QLever engine raises an error. This of course does not take into account why a query raises an error. For example the QLever engine currently does not support ASK queries, so all syntax tests expecting a negative result are considered passed.

3.3.7 Update Evaluation Tests

Our implementation does not fully support the testing of the SPARQL Update evaluation tests. We only compare the resulting default graph with the expected graph not taking into account change made to named graphs. QLever currently does not support update queries and named graphs.

For each update test we set up the SPARQL endpoint. After sending the update query, we send another query with the CONSTRUCT keyword to get the changed default graph. The comparison then is handled as explained in sections 3.3.1 - 3.3.4..

3.3.8 Protocol Tests

The integration of the SPARQL 1.1 protocol tests using telnet addresses the need to validate the QLever engine's adherence to SPARQL protocols. This section provides insight into the protocol testing functionality, emphasizing the use of telnet for simulating client-server interactions.

Telnet for Network Simulation

The decision to use the telnetlib module for protocol testing was driven by the need for a lightweight and flexible means of simulating client-server communication. Telnetlib provides a straightforward interface for sending requests and receiving responses over TCP/IP, closely mirroring real-world SPARQL endpoint interactions.

Request Preparation and Parsing

A significant part of the implementation involved the development of a function , which processes a request-response string to extract and properly format the request part. This function ensures that requests are correctly structured and cleaned before being sent to the SPARQL endpoint.

Expected Response Preparation

Similar to the request, we process the request-response string given and extract the relevant information. Relevant information include the HTTP status code, the Content-Type HTTP header and in some cases the response body, which holds the result to the query of the sent request. The result is either a boolean or a RDF graph.

Response Validation

Upon receiving a response, the tool leverages the pre-proccessed information. We use Pythons String Methods and regular expression operations to confirm if the response contains the correct status code and content-type. To validate the correctness of the response body against expected outcomes, we either use the RDF comparison mentioned in chapter 3.3.4 or we just check if the correct boolean is in the response.

3.3.9 Service Description Tests

This implementation does not support the Service Description Tests. A testing service exists, provided by the W3C.[7]

3.4 Visualizing Differences in Results

The ability to quickly and accurately pinpoint differences in query results aids in the rapid diagnosis and resolution of issues related to query processing and result generation and also enhances the overall development workflow for QLever. By providing a clear visual representation of discrepancies, developers can more easily identify patterns or recurrent issues. We achieve this by generating strings of the expected result and the result given by QLever, which when displayed on the website, highlight differences of the results using the colors red or yellow.

The comparison functions mentioned in sections 3.3.1 - 3.3.4 also return two objects, one object contains the parts of the expected result without a match in the given result and the other object contains the parts of the given result without a match in the expected result. Now we build string representations of the expected and given result. Using the objects containing parts of the results we highlight certain parts of the string representation using the HTML element span.

3.4.1 Challenges

Some problems were encountered using our approach to highlight differences.

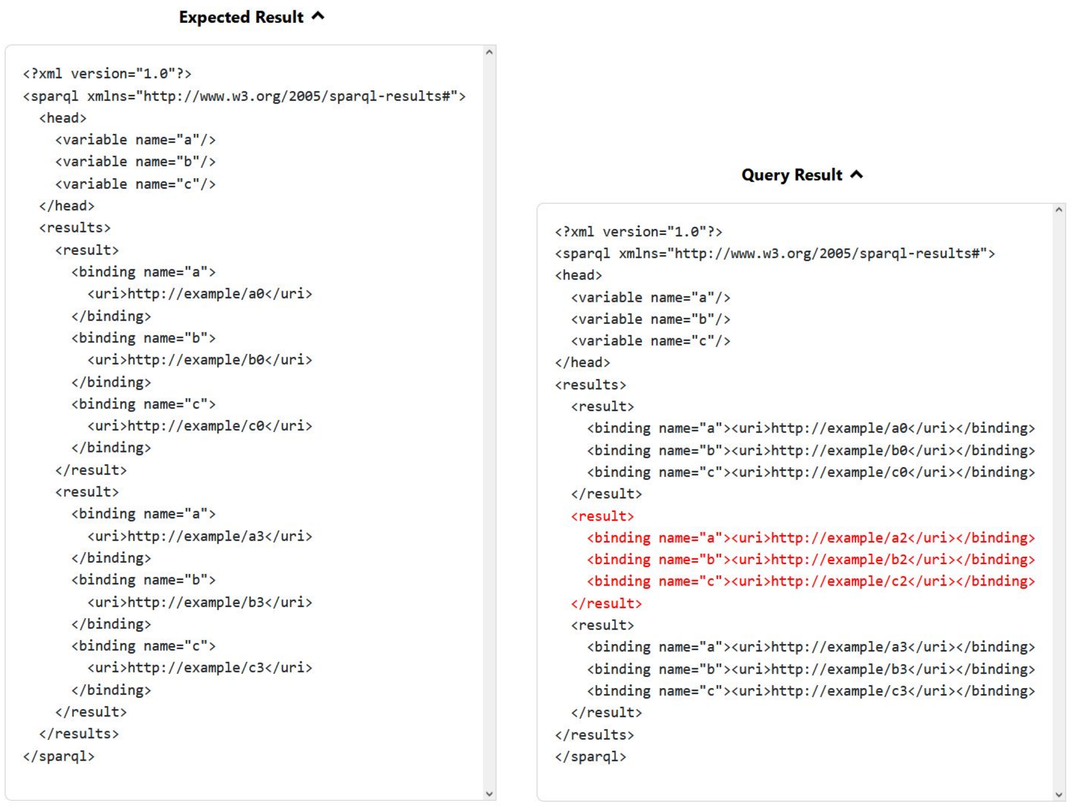
Varying XML Self-Closing Tags

An issue when highlighting differences of two XML elements is the variation in self-closing tags (<tag/>) versus explicitly closed tags (<tag></tag>). To address this, a function was implemented, which utilizes regular expressions to normalize self-closing tags.

Multiple Highlighting

When a response to a query contained a result multiple times, only the first occurrence of the result was highlighted and the other occurrences were not. To

[7]https://www.w3.org/2009/sparql/sdvalidator



```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="a"/>
    <variable name="b"/>
    <variable name="c"/>
  </head>
  <results>
    <result>
      <binding name="a">
        <uri>http://example/a0</uri>
      </binding>
      <binding name="b">
        <uri>http://example/b0</uri>
      </binding>
      <binding name="c">
        <uri>http://example/c0</uri>
      </binding>
    </result>
    <result>
      <binding name="a">
        <uri>http://example/a3</uri>
      </binding>
      <binding name="b">
        <uri>http://example/b3</uri>
      </binding>
      <binding name="c">
        <uri>http://example/c3</uri>
      </binding>
    </result>
  </results>
</sparql>
```

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
<head>
  <variable name="a"/>
  <variable name="b"/>
  <variable name="c"/>
</head>
<results>
  <result>
    <binding name="a"><uri>http://example/a0</uri></binding>
    <binding name="b"><uri>http://example/b0</uri></binding>
    <binding name="c"><uri>http://example/c0</uri></binding>
  </result>
  <result>
    <binding name="a"><uri>http://example/a2</uri></binding>
    <binding name="b"><uri>http://example/b2</uri></binding>
    <binding name="c"><uri>http://example/c2</uri></binding>
  </result>
  <result>
    <binding name="a"><uri>http://example/a3</uri></binding>
    <binding name="b"><uri>http://example/b3</uri></binding>
    <binding name="c"><uri>http://example/c3</uri></binding>
  </result>
</results>
</sparql>
```

Figure 3.1: This is the result of the generated string with HTML elements highlighting the differences

prevent this, we had to add a check to the regular expression to stop a result from being highlighted twice. For example, an XML format containing three empty results would cause the first empty result to be highlighted three times, while the other two would not be highlighted at all.

3.5 Website

The development of the visualization website was guided by ensuring an intuitive, informative, and user-friendly experience, when looking at the test results of a run or when comparing the results of two runs.

The design approach prioritized simplicity and intuitiveness, enabling users to easily interact with the website and access the information they need without extensive guidance. This was achieved through clear labeling, consistent layout, and interactive elements designed for ease of use. Bootstrap's responsive design features were extensively utilized to create a seamless experience on desktops and tablets.

Technology Stack

The core technologies used were HTML, CSS, and JavaScript, providing the foundation for building the website's structure, style, and functionality. Bootstrap was employed to expedite the development process and ensure a responsive design. Its grid system, pre-designed components, and utility classes allowed for rapid layout design and easy implementation.

To run the website locally, Python's HTTP server module (python3 -m http.server) is used to serve the static files (HTML, CSS, JavaScript). This simplifies the deployment and hosting process, reducing the need for complex backend development and maintenance while still efficiently delivering content to users.

Features and Functionality

The website offers essential features and functionalities to facilitate the exploration and analysis of test results Interactive elements and visualizations are implemented using JavaScript, enabling users to dynamically explore test results.

Visual cues and simple graphics help in conveying the status of test outcomes, improving the interpretability of data. JavaScript is also used to provide filtering and search functionalities, allowing users to easily sift through the different runs of the test suite and the test results based on specific criteria such as test names, test groups, test types, test status and error type.

This development approach ensures that the website is not only a valuable tool for visualizing test results but also accessible, easy to use, and responsive, aligning with the project's goals to support the QLever SPARQL engine's development and testing process.

3.6 GitHub Workflow integration

Our work takes the first step towards a seamless GitHub workflow integration. Using GitHub Actions, we created a workflow that builds the current QLever binaries and executes our testing implementation. After running the tests, we push the results to a different GitHub repository. In that repository we host our visualization website with GitHub Pages. The workflow will exit with an error if the current run has tests that passed in a previous run.

Chapter 4

Results

This Chapter presents the outcomes of the project, encompassing the execution of the SPARQL test suite using Python, the development of a web interface for the visualization of the test results, and the integration of these components into the GitHub workflow for the QLever SPARQL engine.

4.1 Test Suite Execution

The Python code developed for executing the SPARQL 1.1 test suite against the QLever engine was successful in automating the testing process. The execution of the test suite resulted in:

4.1.1 Test Coverage

A comprehensive coverage of 600 SPARQL tests consisting of:

- 282 Query Evaluation Tests
- 3 Result Format Tests
- 94 Update Evaluation Tests
- 169 Syntax Tests
- 52 Protocol Tests

4.1.2 Success Rate

The QLever engine successfully processed 23.83% of the test and failed 69.33%. The remaining 6.83% fail because of intended behavior of the QLever engine and are considered passed for the purposes of QLever (Semi-Passed).

As QLever aims to fully support the SPARQL 1.1 standard, let's examine it more closely. The standard is split into the following categories:

Category	**Tests**	**Passed**	**Semi-Passed**	**Failed**	**Pass Rate**
SPARQL 1.1 Query Language	301	97	39	165	32.22% / 45.18%
SPARQL 1.1 Update	157	21	0	136	13.38%
SPARQL 1.1 Query Results, CSV and TSV Formats	6	6	0	0	100%
SPARQL 1.1 Query Results, JSON Format	4	0	2	2	0% / 50%
SPARQL 1.1 Federation Extensions	10	0	0	10	0%
SPARQL 1.1 Entailment Regimes	70	4	0	66	5.71%
SPARQL 1.1 Protocol	34	13	0	21	38.24%
SPARQL 1.1 Graph Store HTTP Protocol	18	2	0	16	11.11%

Table 4.1: SPARQL 1.1 standard tests summary

4.1.3 Error Identification

The tests identified 4 specific areas where the QLever engine's response deviated from expected outcomes, providing clear targets to enhace QLever's support of the SPARQL 1.1 standard.

Of the 416 failed tests 174 are query exceptions, this error indicates that QLever does not support the sent query. For example 18 of the query exceptions fail because ASK queries are not supported, another 13 fail because named graphs are not supported.

72 tests fail because of differences in the expected result and QLever's result to a query, which indicates an error in the implementation of a functionality.

Another 94 tests fail because the HTTP Content-Type header *application/sparql-update* is not supported.

And lastly 60 tests currently fail because the index can not be build. This happened after a current change in the QLever code, which could've been prevented by fully integrating this tool into the QLever GitHub workflow.

These results underscore the effectiveness of the automated testing framework in assessing the QLever engine's compliance, highlighting areas for improvement.

4.2 Visualization Website Development

The development of a website to visualize the SPARQL 1.1 test suite results yielded a user-friendly interface that enables:

4.2.1 Clear Highlighting of Differences

Differences between expected and actual query results are highlighted, employing HTML elements to enhance readability and facilitate error analysis.

4.2.2 Analysis of a single test suite run

Users can interact with the test results, filtering by test type, error type, and other criteria to delve deeper into areas of interest.

4.2.3 Comparison of a two test suite runs

Discrepancies between two runs will be shown to the user, who then can analyze the differences between two runs of the SPARQL 1.1 test suite.

The visualization website serves as a valuable tool for the developers of the QLever engine offering insights into the engine's capabilities and guiding further enhancements.

4.3 GitHub Workflow Integration

The first step of integrating the testing and visualization tools into the QLever engine's GitHub workflow has achieved the automatically execution of the SPARQL 1.1 test suite with each commit to the QLever main branch and the results from the test suite are directly visualized on the project's website, providing immediate feedback to developers.

Chapter 5

Conclusion

This thesis was motivated by the need to enhance the reliability of the QLever SPARQL engine and to expand its compliance to the SPARQL 1.1 standard. Recognizing the challenges inherent in manual testing processes and the difficulty in diagnosing non-compliance or unimplemented functionalities, this work set out to improve the testing for QLever. This was achieved by automating the execution of the SPARQL 1.1 test suite.

Another achievement of this thesis is the development of a visualization tool that intuitively represents differences between expected query results and QLever's query results. This has greatly simplified the task of pinpointing deviations from the SPARQL 1.1 standard, thereby accelerating the process of enhancing QLever's compliance and functionality. Furthermore, taking the first steps to integrate these automated tools into the QLever's GitHub workflow represents a step towards continuous quality assurance during development.

5.1 Limitations

While the project has achieved its objectives, it is not without its limitations. Acknowledging these limitations is crucial for understanding the scope of the study and guiding future directions.

5.1.1 Test Suite Execution and Coverage

One of the foundational aspects of this project was the execution of the SPARQL 1.1 test suite against the QLever engine. The SPARQL 1.1 test suite, while extensive, may not encapsulate all possible query scenarios or edge cases, which poses a limitation. Consequently, there might be aspects of the QLever engine's performance and compliance that remain untested. Another limitation is that our approach does not fully support all Update Evaluation Tests.

5.1.2 Specificity to QLever

The tools and methodologies developed are tailored to the QLever engine, limiting their applicability to other SPARQL engines without modifications.

5.1.3 GitHub Workflow Integration

The GitHub integration of the automated execution and visualization for the SPARQL 1.1 test suite, while a first step forward, presents limitations in its current implementation that may affect its utility for QLever developers. A notable limitation is its application solely to changes in the main branch, excluding pull requests from automated testing. This oversight can lead to scenarios where code that potentially reduces compliance with the SPARQL standard could be merged without undergoing the necessary evaluations. Addressing this limitation is essential to fully harness the benefits of automation in maintaining and enhancing standard compliance throughout the development process.

5.2 Future work

The identification of limitations within the current scope of work lays a foundation for future development. Addressing these limitations not only promises to refine the existing tool but also extends its applicability and effectiveness. This chapter outlines potential directions for future work, informed by the limitations discussed previously.

5.2.1 Enhancing Test Suite Execution and Coverage

Future work could focus on expanding the current SPARQL 1.1 test suite to cover more query scenarios and edge cases, ensuring a more exhaustive evaluation of SPARQL engines.

Another area for improvement is the full support for Update Evaluation Tests within the test suite. Future iterations of the project could aim to integrate these tests seamlessly, ensuring that the SPARQL engine's performance in handling SPARQL update queries is thoroughly assessed.

5.2.2 Broadening Applicability Beyond QLever

To transcend the specificity to QLever, future efforts could be directed towards making the testing and visualization tools more adaptable to other SPARQL engines. This would involve abstracting engine-specific functionalities(Chapters 3.1.1 and 3.1.2) and creating an architecture that allows for easy customization and integration with various SPARQL engines.

5.2.3 Refining GitHub Workflow Integration

A critical area for future improvement is the GitHub workflow integration, specifically extending the automated testing and visualization to include pull requests. Implementing a system that triggers automated tests for every pull request and visualizes the results before merging could significantly mitigate the risk of introducing non-compliant code into the main branch.

Bibliography

[1] Bast and Buchhold. “Qlever: A query engine for efficient sparql+text search.” (2017), [Online]. Available: `https://ad-publications.informatik.uni-freiburg.de/CIKM_qlever_BB_2017.pdf` (visited on 03/24/2023).