Bachelor of Science Thesis

Efficient and Correct Federated Queries for the QLever SPARQL engine

Moritz Dom

March 4, 2025

Submitted to the University of Freiburg
Department of Computer Science
Chair for Algorithms and Datastructures

universität freiburg

University of Freiburg
Department of Computer Science
Chair for Algorithms and Datastructures

Author Moritz Dom,
Matriculation Number: 5306205

Editing Time December 04, 2024 - March 04, 2025

Examiners Prof. Dr. Hannah Bast,
Department of Computer Science
Chair for Algorithms and Datastructures

Supervisor Johannes Kalmbach,
Department of Computer Science
Chair for Algorithms and Datastructures

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, 04.03.2025

Place, Date

Signature

Abstract

The QLever SPARQL+Text query engine is an efficient application used for querying both large-scale knowledge graphs and text corpus. Federated query is a feature of the SPARQL query language that allows querying multiple data sources in a single query. In this thesis we present an improved implementation of federated query processing in QLever, increasing both their efficiency and robustness. By constraining the result size of federated queries based on the query context, we were able to decrease their RAM usage in QLever significantly.

Contents

1 Introduction **1**

1.1 Problem definition . . . 1

2 Background **2**

2.1 Resource Description Framework . . . 2

2.2 SPARQL . . . 3

2.3 QLever . . . 3

2.4 SPARQL Federated Queries . . . 5

3 Approach **7**

3.1 Result format . . . 7

3.2 Lazy processing of JSON results . . . 9

3.3 Lazy export of JSON results . . . 12

3.4 Efficient Federated Queries . . . 12

3.5 Runtime added blank nodes . . . 16

3.6 Runtime Information for Federated Queries . . . 19

4 Benchmarks **21**

4.1 Setup . . . 21

4.2 LazyJsonParser . . . 21

4.3 Efficient Federated queries . . . 23

5 Conclusion **25**

5.1 Future Work . . . 25

5.1.1 SERVICE variables . . . 25

5.1.2 WebSocket client . . . 25

6 Acknowledgments **26**

7 Bibliography **A**

1 Introduction

Knowledge bases are a popular way to represent structured data in a graph-like format. They are used in various applications, such as search engines, question answering systems, and recommendation systems.
Query engines such as QLever allow users to query these knowledge bases using the SPARQL query language. SPARQL (SPARQL Protocol and RDF Query Language) is a powerful and expressive query language designed for querying RDF (Resource Description Framework) knowledge bases. However, while the SPARQL query language is powerful and well defined, the query performance depends on the implementation of the query engine.
Federated queries are a key feature of SPARQL, enabling users to address multiple knowledge bases with a single query.
In this thesis, we present several improvements to the implementation of federated queries in the QLever query engine. We focused on enhancing their efficiency and correctness, while also improving their usability to make it easier for users to query multiple knowledge graphs in a single query.

1.1 Problem definition

Originally, the implementation of federated queries for the QLever SPARQL query engine was in a work-in-progress state. While providing the base functionality, it did not work reliably in some more specific cases. For example, QLever did not support all RDF types such as blank nodes in the result of federated queries. Additionally, the implementation was not efficient in terms of memory usage and performance.
To address these limitations, we focused our improvements on the following aspects:

- Support for all RDF Types: Ensuring support for blank nodes in order to make the use of federated queries seamless.
- Optimized memory usage: Reduced memory usage by importing/exporting results lazily.
- Improved performance: Reduced computation time by improving federated queries based on the query context.
- Enhanced usability: Allow users to debug federated queries more easily by providing detailed runtime information.

2 Background

In this chapter, we provide the background information necessary to understand the topics discussed in this thesis.

2.1 Resource Description Framework

The Resource Description Framework (RDF) is a model for data interchange on the web[4]. It is a standard by the World Wide Web Consortium (W3C) and is used to represent information in a machine-readable format. RDF data is stored as triples of subject, predicate, and object. Each subject and object represents an information entity, and the predicate represents the relationship between the two entities. Triples form a directed, labeled graph, where the subject and object are nodes and the predicate is the arc between them:

Figure 2.1: RDF triple as directed graph

Both subjects and predicates are denoted by Uniform Resource Identifier (URI), while objects can be either URIs or literal values. A URI has the same format as a URL, so e.g. the URI of the city of Freiburg im Breisgau in Wikidata is `https://www.wikidata.org/wiki/Q2833`. Unlike in this case, a URI does not necessarily have to point to an existing web resource. Their main purpose is to uniquely identify a resource, allowing for interoperability of different RDF datasets. Once someone wants to create a new resource identifier, they should create a new URI, using a domain they own. To ensure readability, we will use a simplified notation for URIs in this thesis. Instead of the full URI, we will use the name, such as `<Freiburg im Breisgau>`. For example, key information about the city of Freiburg im Breisgau can be stored in a knowledge base as follows:

subject	**predicate**	**object**
<Freiburg im Breisgau>	<is-a>	<city> .
<Freiburg im Breisgau>	<population>	"237,244" .
<Freiburg im Breisgau>	<country>	<Germany> .
<Germany>	<part-of>	<Europe> .

Table 2.1: Example: RDF knowledge base

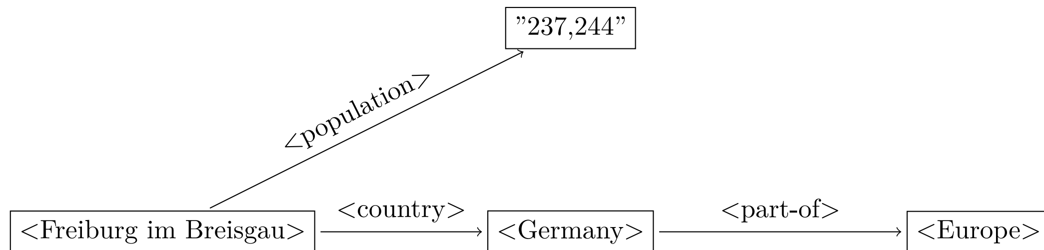

Figure 2.2: Graph of the knowledge base example

2.2 SPARQL

To retrieve information from a knowledge base, we need a language to query the data. SPARQL (SPARQL Protocol and RDF Query Language) is the most common query language for RDF knowledge bases. Using the previous example (Listing 2.1), we can query the knowledge base for the population and country of Freiburg using the following SPARQL query:

```
SELECT ?population ?country WHERE{
  <Freiburg im Breisgau> <population> ?population .
  <Freiburg im Breisgau> <country> ?country .
}
```

Listing 2.1: Simple SPARQL query

The query consists of a `SELECT` and `WHERE` clause. The `SELECT` clause specifies the variables to retrieve, with each of them corresponding to a column in the result set. In the `WHERE` clause we define the pattern to search for in the knowledge base. Here, we are looking for triples where the subject is `<Freiburg im Breisgau>` and the predicate is `<population>`. By setting the object to the variable `?population`, we request all nodes that occur as an object in relation with the given subject and predicate in the knowledge base. The result of a query consists of all combinations of these variables that match the defined pattern in the knowledge base. A query like this can be computed by a SPARQL query engine. In this case, the query results in a single row:

?population	**?country**
"237,244"	<Germany>

Table 2.2: Result of query 2.1

2.3 QLever

QLever is an efficient SPARQL+Text query engine developed at the Chair of Algorithms and Datastructures at the University of Freiburg since 2017 [1]. It is capable of handling large datasets such as Wikidata (≈ 20 billion triples) efficiently. By allowing combined search on both RDF datasets and text corpora, QLever can be used to answer queries on both RDF knowledge bases and unstructured text data within the same query.

Query Processing

QLever computes the result of a SPARQL query in three steps:

- Query Planning
- Computation
- Export of the result

In query planning, a query execution tree is created from the SPARQL query. Each node of the graph is a basic operation, such as e.g. SCAN, SORT or SERVICE. Two subtrees with common variables can be merged using a JOIN operation. The optimal query execution tree is determined by comparing the cost estimate of different possible execution trees.

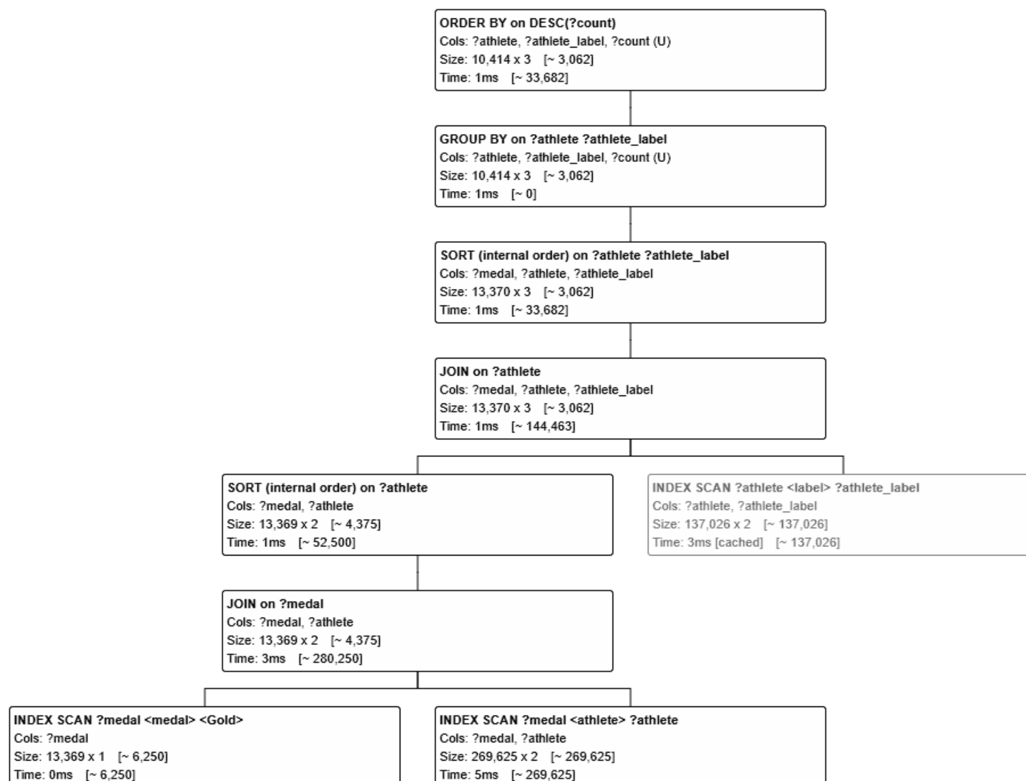

Figure 2.3: Query execution tree

In query execution, the operations given in the tree are computed in a bottom-up order. A more detailed explanation of the query processing can be found in the QLever paper [1]. In the final export step, the result of the query is sent to the requesting client using HTTP in the requested format. QLever supports multiple output formats, such as TSV, JSON, and CSV.

In order to reduce memory usage and increase the speed of the query computation, QLever recently started using a lazy computation approach. This means that operations may support the import and computation of partial results. That also applies to the export of the final result, which is streamed to the client in chunks. More information on QLevers transition to lazy computation can be found in [2].

Technical details

The QLever engine is implemented in modern C++ and supports both Linux and macOS 64-bit operating systems. For convenience, a regularly updated Docker image is available. The free and open-source code can be obtained at `https://github.com/ad-freiburg/qlever` under the Apache License 2.0. To get a quick first impression, users can try queries on a couple of common knowledge bases like Wikidata at `https://qlever.cs.uni-freiburg.de`. The web interface is called QLever UI and is also available on GitHub (`https://github.com/ad-freiburg/qlever-ui`).

2.4 SPARQL Federated Queries

In the SPARQL 1.1 standard, a *SPARQL Protocol service* is defined as a server that processes incoming SPARQL Protocol operations (such as queries) and returns their result[1]. A *SPARQL endpoint* is further defined as the URI at which such a Protocol service listens for requests.
With a simple SPARQL query, we can retrieve data from one RDF dataset by sending it to a SPARQL endpoint. If we want to extend our query with information from a dataset provided by a different SPARQL endpoint, we can use a `SERVICE` clause to address that endpoint with a federated query. A `SERVICE` clause consists of the URL of the SPARQL endpoint, the body of the queries `WHERE` clause and an optional `SILENT` keyword.

```
SERVICE <SILENT(optional)> <URL> {
  SPARQL pattern
}
```

Listing 2.2: SERVICE clause pattern

Consider the following example query, computing all TV series with over 100,000 votes on IMDb ordered by their rating. It contains a `SERVICE` clause retrieving the IMDb id, votes and ranking of all tv series from the IMDb database. Note that the variable `?imdb_id` is common in both the local- and Service query. It will be used to join the intermediate results of the two. It is therefore crucial to make sure that the variable bindings are of matching format in both datasets.

```
SELECT ?movie ?imdb_id ?title (MIN(YEAR(?start_date)) AS ?year) ?imdb_votes ↩
    ?imdb_rating WHERE {
  ?movie <IMDb ID> ?imdb_id .
  ?movie <start-time> ?start_date .
  ?movie rdfs:label ?title FILTER (LANG(?title) = "en") .
  SERVICE <https://qlever.cs.uni-freiburg.de/api/imdb> {
    ?movie_imdb imdb:id ?imdb_id .
    ?movie_imdb imdb:type "tvSeries" .
    ?movie_imdb imdb:numVotes ?imdb_votes .
    ?movie_imdb imdb:averageRating ?imdb_rating .
  }
}
```

[1] `https://www.w3.org/TR/sparql11-protocol/#terminology`

```
GROUP BY ?movie ?title ?imdb_id ?imdb_votes ?imdb_rating
HAVING (?imdb_votes > 100000)
ORDER BY DESC(?imdb_rating)
```

Listing 2.3: Example query: TV series ordered by ImDb rating

To compute the result of the `SERVICE` clause, the engine creates a `SELECT` query from the body of the `SERVICE` clause, capturing all included variables as columns using the `*` operator. The result of this query is then requested from the SPARQL endpoint, by sending the query to the URL specified in the `SERVICE` clause using a HTTP connection.

```
SELECT ?imdb_id ?movie_imdb ?imdb_votes ?imdb_rating WHERE{
   ?movie_imdb imdb:id ?imdb_id .
   ?movie_imdb imdb:type "tvSeries" .
   ?movie_imdb imdb:numVotes ?imdb_votes .
   ?movie_imdb imdb:averageRating ?imdb_rating .
}
```

Listing 2.4: Query sent to the service endpoint

If a SPARQL endpoint is not accessible, the execution of the `SERVICE` clause and, consequently, the entire query will fail. We can ignore that by using the `SILENT` keyword. It will let the `SERVICE` clause result in one result row with no bindings, preserving the intermediate result of the subtree it is merged with.
`SERVICE` clauses can be nested, such that the result of one is used as input for another. This requires that the service endpoint of the outer `SERVICE` clause supports basic federated queries.

3 Approach

In this chapter, the approach of this thesis is presented in detail.

3.1 Result format

To initiate the computation of a `SERVICE` query, the QLever Service operation sends a HTTP request to the SPARQL endpoint. It mainly consists of

- the query itself
- and the format (mime-type) the result shall be responded in.

For the transmission of a SPARQL query result, multiple formats are defined by the W3C consortium such as JSON, XML, TSV, and CSV. The previous implementation of the Service operation used the TSV (tab-separated-values) format. As the previous implementation using TSV as a result format led to incorrect representation of RDF data, we chose to replace it with the more robust JSON format.

TSV

The TSV format represents data as a list of tab-separated values. This makes it one of the simplest and human-readable formats for table-structured data. The SPARQL results TSV format is specified with the first line to list the results variables, followed by one line for each result row [5]. Each binding is encoded using the TURTLE (Terse RDF Triple Language) language [3].

```
?athlete ?athlete_label ?count
<http://wallscope.co.uk/.../HeikeFriedrich> "Heike Friedrich"@en 4
<http://wallscope.co.uk/.../LauraLudwig> "Laura Ludwig"@en 2
<http://wallscope.co.uk/.../KarlNeukirch> "Karl Neukirch"@en 2
```

Listing 3.1: Example of a TSV result

JSON

JSON (JavaScript Object Notation) is another format frequently used for data-exchange on the web. Unlike TSV, JSON allows for an intuitive and human-readable representation of nested data structures. It also offers support for basic data types such as strings, numbers, booleans, arrays, and objects.

The SPARQL 1.1 Query Results JSON Format specifies the structure of the JSON response [6]. It consists of a `head`- and a `results` object. The `head` object contains metadata such as a list of the results column names. Unlike in the TSV-format, the header may contain additional

optional metadata such as links refering to further information. The `results` object contains a `bindings` array, with each element representing one result row.

```
{
  "head": {
    "vars": ["athlete", "athlete_label", "count"]
  },
  "results": {
    "bindings": [
      {
        "athlete": {
          "type": "uri",
          "value": "http://wallscope.co.uk/.../HeikeFriedrich"
        },
        "athlete_label": {
          "type": "literal",
          "value": "Heike Friedrich",
          "xml:lang": "en"
        },
        "count": {
          "datatype": "http://www.w3.org/2001/XMLSchema#integer",
          "type": "literal",
          "value": "4"
        }
      },

      ⋮
    ]
  }
}
```

Listing 3.2: SPARQL 1.1 Query Results JSON Format example

Conclusion

While the TSV format is a valid choice for the correct representation of RDF data, the implementation of the QLever Turtle parser was not capable of parsing it correctly. We handled the issue by requesting results in the more robust JSON format. Once the turtle parser is fixed, returning to the TSV format is an option to be considered, as it is the less verbose format.

3.2 Lazy processing of JSON results

Motivation

We have described in Chapter 2 that many operations in QLever use a lazy computation approach. With its input being a data stream, the service operation is well-suited to use lazy computation as well. The main processing done in the service operation involves writing each result binding into a `idTable` data structure, which is used to handle (intermediate-) results within the QLever engine. Before that the result has to be parsed from the incoming data stream. With the previous result format TSV (Section 3.1), lazy computation was possible by breaking the incoming data into rows at the most recently read newline character. In Section 3.1, we have introduced SPARQL 1.1 Query Results JSON format[6] as the new result format requested by the service operation. Because the JSON format structures data with more syntactical complexity, we had to implement a parser allowing us to work with partial JSON results.
Given the example in Listing 3.2, we can see that the JSON format is structured in such a way that each result row is represented as an element of an array. Therefore this array grows linearly with the number of result rows, making it the part of the result object that we want to split into smaller chunks for lazy computation.

Implementation

For this thesis, we have implemented the following single-pass parser named `LazyJsonParser`.
First of all, we define a few terms that are used in the following explanation:

- `key-path`: A list of keys that lead to a specific element in a JSON object.
- `arrayPath`: A key-path that leads to the main array of the JSON object.

The parser is initialized with the `arrayPath` expected in the input, for the SPARQL JSON result format it is thus set to `["results", "bindings"]`.
In the implementation we distinguish the input in three sequential sections:

1. **before**,
2. **within**,
3. and **after** the `arrayPath`

The parser receives sequential parts of a raw JSON string cut at arbitrary positions. For each part of the input, the `parseChunk` method (Listing 3.3) is called. At first, the input string is appended to the remaining input from previous calls. Then, the parsing process is resumed in the currently active section, with the option to fall through to the next one, once a section has been finished. Finally, the parser tries to construct a partial result from the current input.

```
std::optional<nlohmann::json> LazyJsonParser::parseChunk(
    std::string_view inStr) {
  size_t idx = input_.size();
  absl::StrAppend(&input_, inStr);

  // End-index (exclusive) of the current `input_` to construct a result.
  size_t materializeEnd = 0;

  // If the previous chunk ended within a Literal, finish parsing it.
  if (inLiteral_) {
    parseLiteral(idx);
    ++idx;
  }

  // Resume parsing the current section.
  if (std::holds_alternative<BeforeArrayPath>(state_)) {
    parseBeforeArrayPath(idx);
  }
  if (std::holds_alternative<InArrayPath>(state_)) {
    materializeEnd = parseInArrayPath(idx);
  }
  if (std::holds_alternative<AfterArrayPath>(state_)) {
    std::optional<size_t> optEnd = parseAfterArrayPath(idx);
    if (optEnd) {
      materializeEnd = optEnd.value();
    }
  }

  return constructResultFromParsedChunk(materializeEnd);
}
```

Listing 3.3: parseChunk method

Sections

Literals

Independent of the section, literals are parsed separately to ignore syntactic elements such as curly braces or brackets within them. Once a quotation mark is read, the `inLiteral_` flag is set. Until reading the next unescaped quotation mark, the parser ignores all syntactic elements specific to the JSON format.

Before

Initially starting in the “before“ section, the main objective of the parser is to determine if the `arrayPath` has been reached. In order to do so, the parser maintains a stack of the current path.

Once the current path is equal to the `arrayPath`, the section is changed to "within the arrayPath".

Within

Within the `arrayPath`, the current path no longer has to be maintained. In this section, the parser only has to remember the last position of the most recently encountered element. JSON elements, regardless of their type, have one thing in common: The number of opening and closing curly braces (with syntactical meaning) within them is equal. The same is true for opening and closing brackets. Assuming that the input is valid JSON, we can maintain a combined counter of the open brackets and curly braces. As elements in a JSON array are separated by a comma, the parser checks if the counter is equal to zero at each comma. If it is, the position can be saved as position after the latest element read. By also checking the counter at each closing bracket, we can check if the end of the `arrayPath` has been reached. If so, the parser switches to the "after the arrayPath" section.

After

After parsing of the `arrayPath` is completed, the final task is to find the end of the JSON object. This was implemented using a counter of the remaining curly braces to be closed. It is initialized with the size of the `arrayPath`. Once the parser reaches the end, we can yield the last partial result and return from the coroutine.

Invalid input

As the parser does not check whether the input is valid JSON before constructing a partial result, we had to cover the case of invalid JSON input. Given that the input is invalid JSON, the parser might read a string of arbitrary size without noticing. To prevent that, we have limited the maximum input string size that can yield a partial result to 1,000,000 characters. If this threshold is exceeded, the parser will stop parsing and return an error message. This is sufficient for our use case, as both the header and each element in the `arrayPath` are relatively small.

Partial result construction

The `parseChunk` method returns a partial result if one of the following two conditions is met:

- The end of an element in the `arrayPath`
- or the end of the entire JSON object has been reached

Returning a partial result means that the input string (since yielding the last partial result) until the `materializeEnd` index is returned. Additionally, the prefix and suffix of the input string have to be reconstructed in order to form a valid JSON object. Both are precomputed during the initialization of the parser, based on the `arrayPath`. Given e.g. the SPARQL JSON format, the prefix would be `{"results": { "bindings": [` and the suffix `]}}`.

When yielding the first partial result, the prefix is already contained in the input string. Subsequent results have to be prepended with the precomputed prefix of the `arrayPath`. Analogue to that, the suffix has to be appended to the result string for each but the final partial result.
To ensure the correctness of the partial result, we additionally parse the result string using the `nlohmann/json` library[7].

3.3 Lazy export of JSON results

In Section 3.2, we introduced the `LazyJsonParser`, which enables the lazy import of JSON results for the service operation. Similar to it, we have also implemented the lazy export of results for both the SPARQL JSON and QLever JSON (used by QLever UI) formats for this thesis. Previously, the entire result set was materialized in JSON format in memory before being sent to the client, leading to high RAM usage during the export step of query evaluation. Now, QLever transmits the result to the client in small chunks.
With lazy import and export, federated queries between QLever instances can now be executed in a fully lazy manner.

3.4 Efficient Federated Queries

Motivation

Given a `SERVICE` clause, we have previously described that its query is sent to and computed by the given SPARQL endpoint.
When creating a query, users tend to target the SPARQL endpoint with the dataset that can provide the most information. Another common use case would be to extend an existing query with information missing in the current dataset. This sometimes results in a large result of the service query, while the result of the rest of the query is more concise. Depending on the operation used to merge the two, most of the service queries result might not be relevant for the end result.
To give an example, suppose we want to know which movies Ethan Coen has directed. The movies, their `imdb_id`, and the director, Ethan Coen, can be found in the Wikidata dataset. Additionally, we want the votes and rating in the IMDb database for each of the movies. We can add them by joining a `SERVICE` clause on the `?imdb_id` variable, which is present in both datasets.

```
SELECT ?movie ?title ?year ?imdb_id ?imdb_votes ?imdb_rating WHERE {
  { SELECT ?movie (YEAR(MIN(?date)) AS ?year) WHERE {
    ?movie <instance-of> <film>; <publication-date> ?date } GROUP BY ?movie
  }
  ?movie <IMDb ID> ?imdb_id .
  ?movie <director> <Ethan Coen> .
  ?movie <label> ?title FILTER (LANG(?title) = "en") .

  SERVICE <https://qlever.cs.uni-freiburg.de/api/imdb> {
    ?movie_imdb imdb:id ?imdb_id .
    ?movie_imdb imdb:type "movie" .
    ?movie_imdb imdb:numVotes ?imdb_votes .
    ?movie_imdb imdb:averageRating ?imdb_rating .
  }
}
ORDER BY DESC(?imdb_votes)
```

Listing 3.4: Query for the movies directed by Ethan Coen

The query returns the expected result, but in the runtime information we can see, that the intermediate result of the service query consists of $\approx 315,000$ rows (Figure 3.1). This is due to the fact that the query sent to the service endpoint asked for the rating and votes of every movie in the database. As the result of the service query is joined with the rest of the query in a 1:1 relation (implied by the id), only 21 rows of the service result are actually used for the end result. The unused rows only waste computation time and memory on both SPARQL endpoint and QLever instance. The serialization and deserialization of the result for network transfer, as well as the transfer itself, take a lot of time.

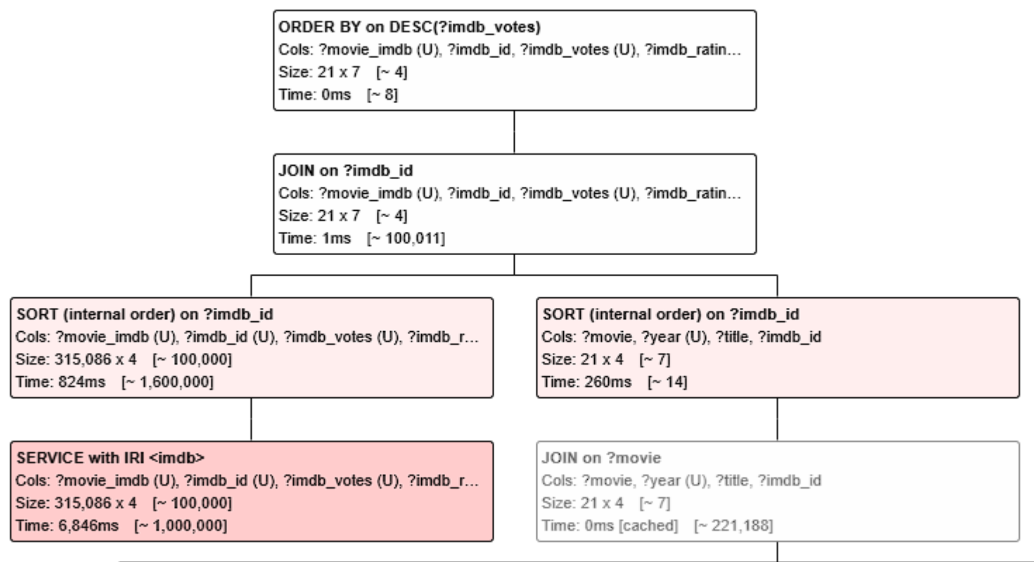

Figure 3.1: Runtime information of the query in 3.4

Workaround

For queries with only one `SERVICE` clause, there is a workaround. As the query consists of two parts depending on different datasets, we can choose the one we expect the smaller result from as our service query. In the example query 3.4, we could have chosen the IMDb dataset as the main dataset and Wikidata as the service dataset. While that reduces the runtime of the service- and thus the entire query, the unnecessary computation is still performed on the main endpoint. However, there are several reasons why this is not always practical:

- More complex queries might contain nested `SERVICE` clauses
- The user might not know which part of the query will result in the smaller result
- Reduced convenience for the user
- New users might not be aware of the problem

Improvement

The intuitive way to handle this problem is to provide the SPARQL endpoint with more information, such that it is able to compute only the relevant part of its intermediate result. In the Query execution tree, we define the operation merging a service operation as `"parent"`, and the operation it is merged with as its `"sibling"`. Sort operations in between service and parent or sibling and parent operation are ignored, as they only change the order of the result rows, not the result set itself.

Parent operations

Given a parent operation with two operands, A and B (left and right), we define a result row in either one of the operands as *relevant* if they influence the computation, and thus the result of the parent operation. The SPARQL standard defines four operations that can merge two subtrees: Join, Optional, Minus and Union.

- The **Join** operation combines two result sets on their common variables. The result consists of all combinations of rows from A and B, where the values of the common variables are equal. This means that rows in A are relevant if they have matching values for the common variables with at least one row in B and vice versa.
- The **Optional** join operation (also known as left join) combines two result sets by including all rows from the left operand A. For each row in A, a row with matching common variables from B is included in the result. If no matching row in B is found, the variables from B are unbound (null) in the result. Therefore, rows from B are only relevant if they have matching values with rows in A.
- The **Minus** operation subtracts all rows from A that have matching values in the common variables with a row in B. This results in all rows from A that do not have any matching rows in B. Thus, rows in B are only relevant for the computation of the parent operation, if there is a matching row in A subtracted due to it.
- The **Union** operation combines two result sets by including all rows from both A and B without any conditions on the common variables. It results in the union of the the two result sets. Therefore, all rows of A and B are relevant and part of the result.

In summary, this means that we can reduce the result of an operand to the relevant rows, if it is either a child of a Join operation or the right child of a Optional- or Minus operation. This can be done by joining it with the common variables of the other operands result.

Application

If we precompute the result of the sibling operation, we can use the bindings of the common variables to directly constrain the service query sent to the SPARQL endpoint. As suggested by the W3C consortium, we can use a `VALUES` clause to join them with the pattern of the service query[1].
For the example query 3.4, this means that we add a `VALUES` clause with the 21 `imdb_id`s to the service query. This results in the following query sent to the SPARQL endpoint:

```
SELECT ?imdb_id ?imdb_votes ?imdb_rating {
  VALUES (?imdb_id) {("tt0477348") ("tt0118715") ... } .
  ?movie_imdb imdb:id ?imdb_id .
  ?movie_imdb imdb:type "movie" .
  ?movie_imdb imdb:numVotes ?imdb_votes .
  ?movie_imdb imdb:averageRating ?imdb_rating .
}
```

Listing 3.5: Resulting service query (`VALUES` clause truncated for readability)

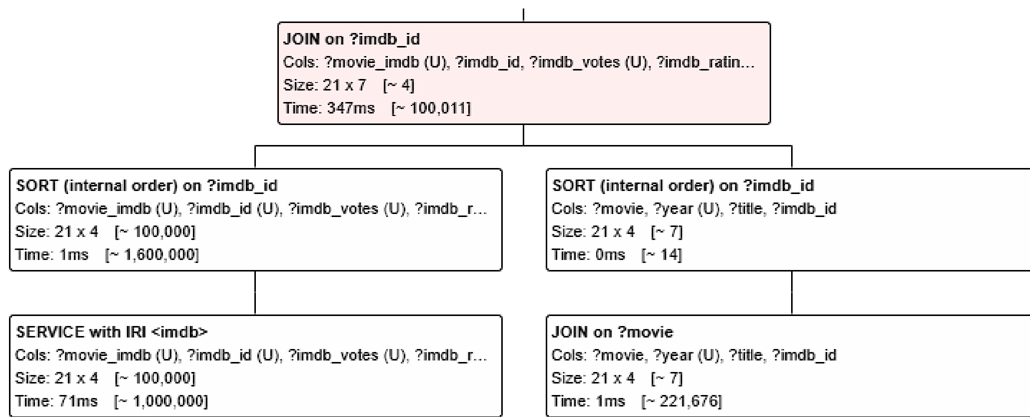

Figure 3.2: Runtime information for query 3.4, using the improvement

Comparing the Runtime information before (3.1) and after (3.2) adding the presented improvement, we can see a drastic reduced time consumption of the Service operation. While it previously took around 6,800ms to compute and transfer $\approx 315,000$ rows, we were able to reduce the result to the relevant 21 rows taking only 71ms.
A more detailed evaluation can be found in the Benchmarks chapter.

[1] https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/#values

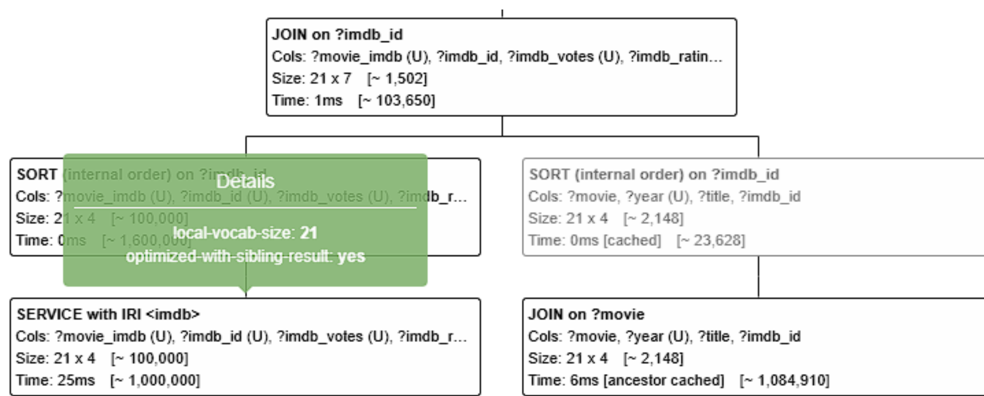

Figure 3.3: Runtime information of an optimized Service operation

final implementation

While constraining the service query with the precomputed sibling result worked well in the presented example, there are some limitations to be considered for a general implementation.
As the sibling operation has to be precomputed, the improvement is only applicable if the sibling result is smaller than the service result. If the service result is smaller, the improvement would not have any effect. However, we can not estimate the service operations result size ahead of time. Therefore we have to define a threshold for the maximum size of the sibling result to be considered. By default, it is set to 10,000 rows.
With all that in mind, we have modified the computation of the mentioned parent operations as follows:

- **Join**: If exactly one of the operands is a Service operation, we precompute its sibling. Precomputing a Service operation as sibling would not be feasible, as we can not estimate the size of its result.
- **Optional**: If the right operand is a Service operation, we precompute its sibling.
- **Minus**: If the right operand is a Service operation, we compute the left operand first.

If the sibling operation supports lazy computation, we stop the precomputation once its size exceeds the threshold. This way, we can avoid unnecessary delay for the computation of the Service operation.
Otherwise, we have to precompute the entire sibling operation.
Additionally, we have added a runtime information detail to both the service and its sibling operation (Figure 3.3). This way, we can visualize the usage of the presented optimization to the user.

3.5 Runtime added blank nodes

Blank nodes

As described in the introduction, the interoperability of different RDF datasets is possible, due to the definition of nodes as URIs. A blank node, however, is a node for which neither a URI

or a literal is given. Both subjects and objects can be blank nodes. Each blank node is an identifier for a particular thing, such that they can be understood as anonymous resources. The scope of a blank node identifier is limited to the serialization of a given RDF graph, such that for example, a node `_b23` does not represent the same resource when appearing in datasets of different knowledge graphs.

Blank nodes in QLever

Internally, QLever keeps an index of all blank nodes. The Indexbuilder creates this index for a given RDF dataset. It is then loaded during startup of the QLever query engine. Apart from blank nodes introduced by the dataset the query engine operates on, queries might introduce additional blank nodes at runtime. As the result of a service query is based on a different dataset, the labels of its blank nodes are too. Therefore, the blank nodes of the service query result cannot be immediately represented using the existing index.

Implementation

To support the usage of blank nodes added at runtime, we have implemented the `BlankNodeManager`. It allows to represent both internal and external blank nodes using the same index. It is initialized during the startup of QLever provided with the number of blank nodes contained in the local dataset. As it is populated in sequential order, we can determine the range of available indices. Given that the dataset contains a total of k blank nodes, the remaining range of blank node indices available at runtime can be computed as $[k, \text{maxIndex}]$.
Since blank nodes introduced by a query can only be used for its own computation, we have to manage them on a per-query basis. Each query context has its own `LocalBlankNodeManager` instance. Operations of the query can retrieve new blank node indices from it. Because a query likely introduces more than one blank node at a time, we manage their indices in blocks. The size of these blocks is constant and is set during the initialization of the `BlankNodeManager`.

LocalBlankNodeManager

New blank node indices can be requested from the `LocalBlankNodeManager` using the `getId()` function (Listing 3.6). It returns the next free index of the last allocated block. If no block has been allocated yet, or all indices of the current block are in use already, a new block is allocated first.

```
1 uint64_t BlankNodeManager::LocalBlankNodeManager::getId() {
2   if (blocks_->empty() || blocks_->back().nextIdx_ == idxAfterCurrentBlock_) {
3     blocks_->emplace_back(blankNodeManager_->allocateBlock());
4     idxAfterCurrentBlock_ = blocks_->back().nextIdx_ + blockSize_;
5   }
6   return blocks_->back().nextIdx_++;
7 }
```

Listing 3.6: LocalBlankNodeManager getId() function

Using this approach, the (temporary) fragmentation a single `LocalBlankNodeManager` can cause is limited to a single block with only one index actually in use. Once the query is removed, the allocated blocks are freed from the `BlankNodeManager`, allowing them to be used by other queries.

BlankNodeManager

Given that a blank node index is represented as a 64-bit unsigned integer, the number of blocks the BlankNodeManager can manage is limited to $\frac{2^{64}-k}{\text{blockSize}}$. We also define a block index B, where B_i represents the block starting at blank node index $k + i \cdot \text{blockSize}$.
For the allocation of a new block, we decided to use a random selection algorithm in order to ensure constant time complexity. We generate a random block index and check if it is already in use. If not, we add it to the set of used blocks and return the block. Otherwise we retry until we are successful. Because multiple instances of the `LocalBlankNodeManager` might call the function at the same time, we had to guarantee a thread-safe access to the `usedBlocksSet_`.

```
BlankNodeManager::Block BlankNodeManager::allocateBlock() {
  // The Random-Generation Algorithm's performance is reduced once the number of
  // used blocks exceeds a limit.
  auto numBlocks = usedBlocksSet_.rlock()->size();
  AD_CORRECTNESS_CHECK(
      numBlocks < totalAvailableBlocks_ / 256,
      absl::StrCat("Critical high number of blank node blocks in use: ",
                   numBlocks, " blocks"));

  auto usedBlocksSetPtr = usedBlocksSet_.wlock();
  while (true) {
    auto blockIdx = randBlockIndex_();
    if (!usedBlocksSetPtr->contains(blockIdx)) {
      usedBlocksSetPtr->insert(blockIdx);
      return Block(blockIdx, minIndex_ + blockIdx * blockSize_);
    }
  }
}
```

Listing 3.7: Block allocation

complexity

The time complexity of the allocation function depends on the number of already allocated blocks as follows: Given that n blocks are already allocated and we have a total of t available blocks. Then the probability of choosing the index of a free block is $P = \frac{t-n}{t}$. As each trial is indepent with the same probability, we have a *Bernoulli process* following a geometric distribution. The expected number of trials therefore amounts to $E[trials] = \frac{1}{P}$. Therefore the

average time complexity of the algorithm is:

$$O(n) = \frac{1}{P} = \frac{t}{t-n}$$

In order to prevent reaching the worst case scenario where most blocks are in use and the complexity approaches $O(n)$, we have added a check limiting the number of blocks in use. By setting it to $\frac{1}{256}$ of the total available blocks, we can ensure that the worst case of the algorithm is $O(\frac{256}{255}) \approx O(1)$.
The blank node index is implemented as 64-bit unsigned integer, theoretically allowing for 2^{64} blank nodes. Ignoring the indices reserved for internal blank nodes and assuming a block size of 1024, this would result in $\frac{2^{64}}{1024} = 2^{54}$ blocks, and therefore $\frac{2^{54}}{256} = 2^{46}$ blocks useable at the same time. This is more than sufficient, as storing only one index for each of these blocks would result in an unfeasible memory cost of 8 bytes $\cdot 2^{46} \approx 563$ TB. This means, that the allocation of new Blocks will always happen in constant time.

3.6 Runtime Information for Federated Queries

Runtime Information in QLever

During computation of a query, QLever collects detailed information about the query execution, referred to as *Runtime Information*.
For each operation of the query execution tree, the following information is collected (including but not limited to):

- execution time
- status of the operation
- result size
- whether the result was computed or read from cache
- additional operation-dependent details

A QLever endpoint assigns a UUID (Universally Unique Identifier) to each incoming query before computing its result. It can be set by the client and passed with the initial query request as HTTP header `Query-Id`, otherwise it is randomly generated by the QLever engine.
To obtain the runtime information, QLever provides a websocket endpoint, which can be queried using the UUID of the query. Given e.g. the QLever Wikidata endpoint at `https://qlever.cs.uni-freiburg.de/api/wikidata`, our websocket client has to be connected to `https://qlever.cs.uni-freiburg.de/api/wikidata/watch/<query-id>` to receive updates on the runtime information of the query with id `<query-id>`.
The QLever UI webinterface uses this functionality to display the runtime information to the user. It allows users to identify possible bottlenecks in their queries, enabling them to optimize execution time and general efficiency.
However, previously a detailed runtime information of federated queries was not available for the QLever engine, as the service query runs on a different endpoint than the main query. The runtime inflicted by the query execution on the remote endpoint was therefore implicitly added to the Service operation.

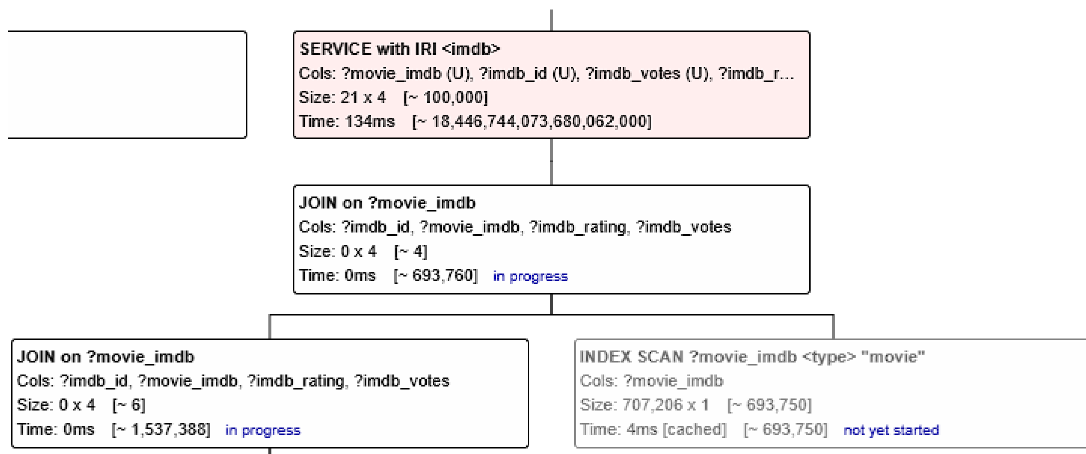

Figure 3.4: Runtime information of a service query, visualized in the QLever UI

Implementation

For this thesis, we have implemented a WebSocket client allowing us to fetch the runtime information of federated queries run on remote QLever endpoints. Because the runtime information is not standardized for SPARQL endpoints, the implementation is specific to the QLever engine. The WebSocket connection, or at least the retrieval of runtime information will fail and be silently ignored, if the remote endpoint is not a QLever instance.

The result computation of the Service operation was extended using the WebSocket client in the following way:

1. generate a random query-id
2. connect WebSocket client to the endpoint with the given query-id
3. request result from endpoint over HTTP with set query-id

At first, a query ID is randomly generated. Then the WebSocket Client is connected to the endpoint with the given query ID. In order to not delay the retrieval of the actual result, the WebSocket client is run in a separate thread. The Service operation then requests the result from the endpoint over HTTP with the set query-id. During the computation of the result on the remote endpoint, the WebSocket client receives updates on the runtime information of the service query. It is then added as a child to the Service operations runtime information, resulting in a detailed runtime information of the whole query.

4 Benchmarks

This chapter contains benchmarks for two of the presented implementations, the LazyJsonParser and the efficient federated queries.

4.1 Setup

The following benchmarks were conducted on a machine with the following specifications:

- CPU: AMD Ryzen 5 2600 (12 cores) @ 3.400GHz
- RAM: 16GB
- OS: Ubuntu 22.04.5 LTS

Each benchmark was run ten times, and the average time was calculated.

4.2 LazyJsonParser

For the LazyJsonParser, we have tested the performance against parsing an entire JSON object at once. As input data, we have used a JSON results object with 10,000,000 bindings with a total file size of 1020 MB $\approx$ 1 GB. The JSON object is structured as shown in Listing 4.1.
We have tested the parser with different sized chunks of the input data (in bytes), and the results are shown in Table 4.1.
With the LazyJsonParser parsing the JSON object essentially twice, once to determine the struture and once to extract the data, the time taken is expected to be longer than parsing the entire object at once. Compared to parsing the JSON object at once, the LazyJsonParser is slightly slower. The results show that the runtime of the LazyJsonParser depends on the chunk size, with a chunk size of 2000 bytes being the most efficient. However, there is not a big difference between them, therefore a non-optimal chunk size will not have a big impact on the performance. With the main improvement due to the LazyJsonParser being the reduced RAM usage, a minimal difference in runtime is acceptable. While parsing the given input data at once results in a `nlohmann::json` object of $\approx$ 5GB, the LazyJsonParser only requires e.g. $\approx$ 12MB when parsing with the optimal chunk size of 2000 bytes.

Parser	chunk size (bytes)	execution time
nlohmann/json	-	19,935ms
LazyJsonParser	100	37,490ms
	1000	22,173ms
	1500	21,421ms
	2000	21,084ms
	2500	21,321ms
	3000	21,858ms
	4000	23,065ms
	8000	23,034ms
	16000	22,986ms
	32000	23,051ms
	64000	23,222ms

Table 4.1: Benchmark of the LazyJsonParser for different chunk sizes

```
{
  "head": {
    "vars": ["index"]
    },
  "results": {
    "bindings": [
      {
        "index": {
          "type": "literal",
          "datatype": "http://www.w3.org/2001/XMLSchema#int",
          "value": "1"
          }
        },
      ⋮
      {
        "index": {
          "type": "literal",
          "datatype": "http://www.w3.org/2001/XMLSchema#int",
          "value": "10000000"
          }
        }
    ]
  }
}
```

Listing 4.1: Structure of the input data

4.3 Efficient Federated queries

For the following benchmarks, we have sent the given queries to a QLever instance running on the previously mentioned setup. The service queries where computed on the respective endpoints at `https://qlever.cs.uni-freiburg.de/api/`. We provide the execution time of the Service operation available in the runtime information.
At first, we have tested the performance of the Query 3.4 which retrieves all movies directed by Ethan Coen. With the small local and large Service result, the optimization of the constraint service query shows in a significantly reduced runtime.
Next, we tested the JOIN of a small Service with a large sibling result. To do so, we have switched the subqueries of query 3.4 such that the service queries result is the smaller one. To simulate different thresholds for the size of the `VALUES` clause, we have limited the result size of the sibling query using a `LIMIT` with the respective threshold size. For the constraint/unconstraint measurements, we have set the threshold for the `VALUES` clause size to 50,000 and 0 resepectively, in order to use/reject the optimization. This is the worst case for the constraint service query, as it already computes its result in a reasonable time without the optimization. Adding a `VALUES` clause with only 10,000 values increases the runtime by $\approx$ 2250ms. The results show, that the runtime of the constraint service query increases linearly with the number of values in the `VALUES` clause. Therefore, the current threshold set to 10,000 is a good compromise between the best and worst case scenario.

Query	**unconstraint**	**constraint**
Movies directed by Ethan Coen (3.4)	13,150ms	103ms
JOIN of small Service with large sibling result (4.2)	253ms	-
`VALUES` clause size: 10000	-	2522ms
`VALUES` clause size: 20000	-	5203ms
`VALUES` clause size: 30000	-	7701ms
`VALUES` clause size: 40000	-	10200ms

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX schema: <http://schema.org/>
PREFIX imdb: <https://www.imdb.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX p: <http://www.wikidata.org/prop/>
SELECT ?movie ?title ?year ?imdb_id ?imdb_votes ?imdb_rating WHERE {
  {
    SELECT * WHERE {
      ?movie_imdb imdb:id ?imdb_id .
      ?movie_imdb imdb:type "movie" .
      ?movie_imdb imdb:numVotes ?imdb_votes .
      ?movie_imdb imdb:averageRating ?imdb_rating .
    }
    LIMIT 10000
  }
  SERVICE <https://qlever.cs.uni-freiburg.de/api/wikidata> {
    {
      SELECT ?movie (YEAR(MIN(?date)) AS ?year) WHERE {
        ?movie wdt:P31 wd:Q11424 ;
              wdt:P577 ?date
      }
      GROUP BY ?movie
    }
    ?movie wdt:P345 ?imdb_id .
    ?movie wdt:P57 wd:Q13595531 .
    ?movie rdfs:label ?title FILTER (LANG(?title) = "en") .
  }
}
ORDER BY DESC(?imdb_votes)
```

Listing 4.2: Small service, large sibling result modification of query 3.4

5 Conclusion

We have presented several improvements to the implementation of federated queries in the QLever query engine. First, we have ensured that all valid RDF results can be handled as input to the Service operation, allowing seamless use of federated queries. We introduced the LazyJsonParser, allowing us to significantly reduce memory usage during the import of JSON results. We also presented a more efficient implementation of the Service operation, which simplifies federated queries using the query context. Finally, we have extended the Service operation with a WebSocket client to retrieve the runtime information of federated queries run on remote QLever endpoints.

5.1 Future Work

During the development of the presented improvements, several ideas for further improvements have come up.

5.1.1 SERVICE variables

Due to the SPARQL standard, the URI of the SPARQL endpoint in a `SERVICE` clause can also be given as a variable[1]. This requires the computation of the Service operation to be delayed until the value for the variable is computed.

5.1.2 WebSocket client

The WebSocket client described in Section 3.6 could be extended beyond the retrieval of a queries Runtime information from a QLever endpoint. Due to the WebSocket endpoint being a QLever specific feature, a successful connection already implies that the endpoint is QLever. We could however check that explicitly, and provide the following functionality if it is:
If the computation of the query is cancelled (either manually by the user or due to an error), the WebSocket client can be used to notify the endpoint such that it can cancel the computation of the service query aswell. The server-side functionality for this is already implemented in QLever and is currently used by QLever UI.
We could also request the query result in a custom format, which is more efficient than the SPARQL standard.

[1] `https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/#variableService`

6 Acknowledgments

First and foremost, I would like to thank my advisor, Johannes Kalmbach, for their invaluable feedback, encouragement and patience. I always had the opportunity to ask questions and discuss different approaches in countless meetings.
I also want to thank Prof. Dr. Hannah Bast for supervising this thesis.
Last but not least, i would like to thank my friends and family for their support and encouragement throughout my studies.

Figure Index

2.1 RDF triple as directed graph 2
2.2 Graph of the knowledge base example 3
2.3 Query execution tree 4

3.1 Runtime information of the query in 3.4 13
3.2 Runtime information for query 3.4, using the improvement 15
3.3 Runtime information of an optimized Service operation 16
3.4 Runtime information of a service query, visualized in the QLever UI 20

Table Index

2.1 Example: RDF knowledge base . . . 2
2.2 Result of query 2.1 . . . 3

4.1 Benchmark of the LazyJsonParser for different chunk sizes . . . 22

Code Index

2.1 Simple SPARQL query 3
2.2 SERVICE clause pattern 5
2.3 Example query: TV series ordered by ImDb rating 5
2.4 Query sent to the service endpoint 6

3.1 Example of a TSV result 7
3.2 SPARQL 1.1 Query Results JSON Format example 8
3.3 parseChunk method 10
3.4 Query for the movies directed by Ethan Coen 13
3.5 Resulting service query (`VALUES` clause truncated for readability) 15
3.6 LocalBlankNodeManager getId() function 17
3.7 Block allocation 18

4.1 Structure of the input data 22
4.2 Small service, large sibling result modification of query 3.4 24

7 Bibliography

References

[1] H. Bast and B. B., “Qlever: A query engine for efficient sparql+ text search,” *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 647–656, 2017.

[2] R. Textor-Falconi. “Lazy evaluation of sparql queries with caching.” (2025), [Online]. Available: `https://ad-publications.cs.uni-freiburg.de/theses/Master_Robin_Textor_2025.pdf` (visited on 02/20/2025).

Web Resources

[3] D. Beckett and T. Berners-Lee. “Turtle - terse rdf triple language.” (2011), [Online]. Available: `https://www.w3.org/TeamSubmission/turtle/` (visited on 01/16/2025).

[4] R. W. Group. “Resource description framework (rdf).” (), [Online]. Available: `https://www.w3.org/RDF/` (visited on 01/16/2025).

[5] R. W. Group. “Sparql 1.1 query results csv and tsv formats.” (), [Online]. Available: `https://www.w3.org/TR/sparql11-results-csv-tsv/` (visited on 01/16/2025).

[6] R. W. Group. “Sparql 1.1 query results json format.” (), [Online]. Available: `https://www.w3.org/TR/sparql11-results-json/` (visited on 01/16/2025).

[7] N. Lohmann, *Json for modern c++*, `https://github.com/nlohmann/json`, Version 3.11.3, accessed on 2025-02-20, 2021.