Bachelor's Thesis

Efficient Export of SPARQL Query Results

Robin Textor-Falconi

Examiner: Prof. Dr. Hannah Bast
Adviser: Johannes Kalmbach

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

March 10th, 2022

Writing Period

21. 12. 2021 – 21. 03. 2022

Examiner

Prof. Dr. Hannah Bast

Adviser

Johannes Kalmbach

Declaration

I hereby declare that I am the sole author and composer of my 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, 10th March

Place, Date

[illegible]

Signature

Abstract

This bachelor's thesis documents an implementation of CONSTRUCT queries largely compliant with the SPARQL standard. It also discusses several approaches to make the process of exporting query results as fast as possible and compares the performance of a selection of those approaches in a real-world example. This includes environments with a limited bandwidth to the server by making use of compression algorithms and asynchronous processing.

Contents

1 Introduction

Knowledge bases are important tools to work with large amounts of data in a modern age. The Resource Description Framework, commonly referred to as RDF is a framework to represent information in a standardized manner [1]. It can be used for internal representation of data within a knowledge base. The World Wide Web Consortium (W3C) has created a well-established recommendation of 'SPARQL' [2], a query language for accessing RDF graphs. 'QLever' is a knowledge base, developed at the chair of Algorithms and Data Structures of the University of Freiburg that is largely compliant with the SPARQL standard [3]. QLever is written in C++, making use of some of the most recent features the C++20 standard [4] defines. It offers support to process OpenStreetMap data [5] and efficient context sensitive auto-completions of queries [6].

SPARQL's most important functionality are 'SELECT' queries where a list of 'variables' is provided along with a 'WHERE' condition to limit the result to a subset matching a given criteria. For every matching data entry in the knowledge base a row of a new table will be exported where the variables are substituted with the actual contents of the knowledge base.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX sc: <http://schema.org/>

SELECT ?x ?name WHERE {
  ?x wdt:P31 wd:Q13442814 .
  ?x sc:name ?name
}
```

Listing 1: Example SELECT query that selects all scientific papers with their id and name

An RDF graph is basically a set of triples consisting out of a ‘Subject’, a ‘Predicate’ and an ‘Object’ each. In Listing 1 we can see how variables like `?x` and `?name` are used as placeholders to indicate a variable value in the query result. All other values can be considered as filtering requirements of the RDF graph.

2 Construct Queries

In addition to SELECT queries, the SPARQL recommendation also defines the concept and syntax of ‘CONSTRUCT’ queries [7]. At a first glance they look rather similar, but instead of returning a table where each column represents a variable or expression, a CONSTRUCT query returns a single RDF graph. This RDF graph is formed by taking the graph template supplied with the CONSTRUCT query, substituting any variables in the template with the values from the computed result and joining those individual substituted templates by union into a single graph. Any resulting illegal RDF constructs are not included in the final graph. This feature is especially useful when trying to transform existing data sets to add or remove information because no further post-processing is required to supply a SPARQL engine with this kind of data.

```
CONSTRUCT {
  ?name a <http://xmlns.com/foaf/0.1/Person>
} WHERE {
  ?x <http://xmlns.com/foaf/0.1/name> ?name
}
```

Listing 2: A simple CONSTRUCT query

2.1 Implementing CONSTRUCT query parser logic

At the time of writing, the QLever code base is still in a transitional state of the parsing code. Most of the logic to parse SPARQL grammar for all kinds of queries is still self-written and hard-coded to work for a supported variant of the official SPARQL grammar. This means the code is hard to understand and therefore even harder to maintain. So implementing support for the whole SPARQL grammar without accidentally breaking something is not trivial. To tackle this maintenance issue, efforts are being made to migrate the parsing logic to ANTLR (**antlr.org**). ANTLR is a parser generator, which turns a formally defined grammar into executable code automatically. This way it is sufficient to use the official SPARQL grammar in the recommendation [7] and let ANTLR generate the correct code for it. With that in place development efforts can be focused on implementing semantic parsing logic i.e. how to interpret the parsed query rather than having to worry if the query is syntactically well-formed. To allow for an easy migration process, the legacy parser starts parsing the individual queries and invokes the ANTLR parser to handle more specific details. The ultimate goal is to completely replace the legacy parser in favour of the ANTLR solution.

ANTLR is able to generate code for a variety of languages including C++. The downside of this variety is that instead of generating code that adheres to common practices and conventions of each language individually, it seems like the original Java based class structure was forced onto all languages. This implies heavy use of `antlrcpp::Any`, a wildcard Type similar to Java's `Object`. It can represent any arbitrary type at run-time, but does not provide any type information at compile-time. This approach makes lots of boilerplate code necessary with the single purpose of casting to the proper types that could be known if writing the code by hand, but at least allows for a lot of flexibility with individual types being used during the 'visit'. Before being able to properly implement such a parser using ANTLR we need to understand all of the requirements for this project.

2.1.1 Unwrapping RDF triples

A typical CONSTRUCT query is made up of a graph template containing a fixed amount of triples and a corresponding WHERE clause. At a first glance it looks like all the logic a parser needs to implement is about storing a collection of triples. The SPARQL recommendation however provides a lot of syntactic sugar allowing for some recursively nested expressions. This means there are two ways a parser could be implemented: Either by storing the SPARQL graph template as is and developing a data structure in C++ that can store it, or by unwrapping the whole syntactic sugar during parsing to generate an equivalent collection of triples. Both approaches have their benefits. The former allows for a potentially denser data representation which might be useful when exporting the result as discussed in Chapter 6, Section 6.1.2. The latter approach allows for a simpler data structure, at the cost of storing potentially redundant information that could be avoided. We chose the latter due to its simplicity.

```
CONSTRUCT {
  ?a a <http://xmlns.com/foaf/0.1/Person> ,
       <http://xmlns.com/foaf/0.1/givenName> ;
     <http://xmlns.com/foaf/0.1/knows> ?a .
  [?x (?x ?a)] ?a [?x [?x ?x]] .
} WHERE {
  ?x <http://xmlns.com/foaf/0.1/name> ?a
}
```

Listing 3: A more complicated CONSTRUCT query, equivalent to 11 triples

Listing 3 shows how a small amount of characters can make a construct query hard to read. This is a well-formed SPARQL query that the parser has to interpret correctly. Note that this is a rather synthetic query. It serves as a good example to break down, but there is most likely no practical use for this particular query. Let's start breaking it down!

Subjects, Predicates, Objects and other honorable mentions

Instead of using the IRI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` as a Predicate, the short token `a` can be used instead. This is simply a convenient short-hand notation, for an IRI the authors of the SPARQL recommendation assumed to be a very frequent use-case. Very easy to substitute.

The token `.` is used to separate triples from each other. Optionally it can also be used as a terminator i.e. the last triple in a list can also end with a `.` but it doesn't have to. Within a triple the tokens `;` and `,` each have special meaning. In a scenario where two or more triples share the same Subject, `;` can be used to avoid having to type the same Subject twice. This way `?a ?b ?c . ?a ?d ?e` can be written as `?a ?b ?c ; ?d ?e` for example. Similar to the `.` token, the token `;` can also be used as a terminator when preferred, making `CONSTRUCT { ?a ?b ?c ;. }` a completely valid syntax even though it looks weird.

If two or more triples share the same Subject and Predicate the token `,` can be used to make the term even shorter. Instead of writing `?a ?b ?c . ?a ?b ?d` only the sort form `?a ?b ?c , ?d` is required to express the same information.

Another concept worth noting are PREFIX declarations. Instead of having to write IRIs sharing the same common prefix over and over again, PREFIX declarations can be used to keep queries (all kinds of them) tidy and avoid potential typos. Listing 2 could be turned into Listing 4. All the code has to do is to replace the prefix with the content of the PREFIX declaration. There is also a so called BASE declaration for relative IRI resolution with a similar purpose, but QLever does currently not support those at all.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT {
  ?name a foaf:Person
} WHERE {
  ?x foaf:name ?name
}
```

Listing 4: Listing 2, but using PREFIX declarations

Blank Nodes

Blank Nodes are nodes within an RDF Graph without any special meaning assigned to them. Their sole purpose is to create a purely logical connection between multiple RDF triples. If two Blank Nodes share the same label, they are the same. The label is no attribute of the Blank Node itself, just a mechanism to identify the same Blank Node in 2 different locations. Because the label has no meaning, this means that any SPARQL engine is not required to store the label at all, as long as it knows which Blank Nodes are the same. Using SPARQL syntax, labelled Blank Nodes start with an underscore followed by a colon, followed by the label. (See Listing 5)
That being said there is also the concept of anonymous Blank Nodes, where the label is auto-generated by the SPARQL engine. This becomes important for the following two concepts.

```
CONSTRUCT {
  ?a a _:label .
  _:label a ?b
} WHERE {
  ?a <http://xmlns.com/foaf/0.1/name> ?b
}
```

Listing 5: CONSTRUCT query using labelled Blank Nodes

Property Lists

The concept of Property Lists, as the grammar calls them [7], provide an alternative mechanism to create Blank Nodes. In its simplest form `[]` refers to a Blank Node with a unique label by design. This means no other Blank Node can refer to this exact instance of the Blank Node no matter what. This becomes interesting once adding information inside those brackets. The SPARQL grammar expects a `;` separated list of pairs of Predicate and Objects. The generated Blank Node becomes the Subject of all of those Predicate-Object-pairs. Therefore all of the following 3 expressions are semantically equivalent: `[] ?a ?b`, `[?a ?b]` and `_:node ?a ?b`. By using this notation Property Lists can be nested in each other like this: `[a []]`. A key concept to take away from this is that an expression can generate triples in itself, but are still able to act as Subjects or Objects within another expression. This means that the code that traverses this parse tree has to be prepared to always accept the top level information from each expression to potentially make use of it in subsequent steps as well as a list of all triples that might have been generated in the process.

Collections

Collections provide syntactic sugar to deal with Blank Nodes when trying to express data in the form of a linked list. Triples generated by the Collection syntax act as the links between Blank Nodes, using well-defined IRIs. The Predicate `<http://www.w3.org/1999/02/22-rdf-syntax-ns#rest>` is used to connect two subsequent Blank Nodes to each other, where the leading Blank Node acts as the Subject and the trailing Blank Node as the Object. If the Object in such a triple is `<http://www.w3.org/1999/02/22-rdf-syntax-ns#nil>` this means the Blank Node acting as Subject is the last item of the list. Consequently the single term `<http://www.w3.org/1999/02/22-rdf-syntax-ns#nil>` is equivalent to the empty Collection `()`. Each link can refer to its item using a triple, where the Blank Node acts as the Subject, the IRI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#first>` as the Predicate and the item as the Object. A typical Collection would look like this: `(1 2 3 4)`. It would result in 4 generated Blank Nodes and 8 triples connecting those Blank Nodes and the corresponding values together.

```
CONSTRUCT {
  ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://xmlns.com/foaf/0.1/Person> .
  ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://xmlns.com/foaf/0.1/givenName> .
  ?a <http://xmlns.com/foaf/0.1/knows> ?a .
  _:c0 ?x _:c1 .
  _:c1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#first> ?x .
  _:c1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest> _:c2 .
  _:c2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#first> ?a .
  _:c2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest>
            <http://www.w3.org/1999/02/22-rdf-syntax-ns#nil> .
  _:c0 ?a _:c3 .
  _:c3 ?x _:c4 .
  _:c4 ?x ?x .
} WHERE {
  ?x <http://xmlns.com/foaf/0.1/name> ?a
}
```

Listing 6: Equivalent query to Listing 3, but unwrapped

Using all of those rules Listing 3 can be unwrapped to Listing 6. Bonus points if you are able to comprehend each individual step in this example.

2.2 Constructing RDF graphs from existing logic

Once the parser is able to compute equivalent triples, connecting the newly written code to the existing query engine is pretty simple in comparison.

To achieve this some of the pre-existing query processing code has to be adjusted to be able to deal with seemingly missing SELECT query variables. Interestingly enough this refactoring process revealed some minor bugs and non-standard behaviour in the code that went unnoticed before. Once CONSTRUCT queries correctly compute a valid result set, implementing a second serialization process is straightforward. To export an RDF graph instead of the table format typical for a SELECT query, the code has to substitute the placeholders in the newly parsed CONSTRUCT query instead. These generated RDF triples are only exported when the resulting RDF construct is considered valid in order to be compliant with the SPARQL recommendation.

The preferred format to export CONSTRUCT queries should be the 'Turtle' format. It provides a non-ambiguous way of representing RDF triples and is similar enough to the parsed input so it can be serialized with very little effort. Even though the SPARQL recommendation defines export formats for XML [8], JSON [9], CSV and TSV [10], none of them are defined to be used with CONSTRUCT queries. QLever supports the use of JSON, TSV or CSV to serialize the resulting RDF graph in a non-standard way. The nature of those formats lead to lost type information in certain cases (is `<http://example.com>` an IRI, or just a plain string which happens to have the same format as an IRI?), which might explain why they haven't been standardized at all. XML is not supported by QLever at all because the age of this format seems to keep the demand for it at zero.

3 Sending partial results

3.1 Problem Definition

QLever's main advantage over the comparable query engines "RDF-3X", "Virtuoso" and "Broccoli" is its low average query time for a large variety of SELECT queries for large datasets [3]. However these timings were achieved in a controlled high-end testing environment, a rather specific scenario that completely ignores the fact that the result of such SPARQL queries might need to be exported to different machines in the same network, or even over the world wide web.

The QLever engine currently supports the export of SELECT queries with arbitrary result sizes - in theory. In practice exporting large result sets used to be error-prone in the original implementation on systems with lower amounts of memory. A lot of effort was put into optimizing the efficiency of actual queries, but the HTTP protocol was implemented with a far more pragmatic approach to the problem - in comparison. QLever is using the Boost library Beast (**github.com/boostorg/beast**). This makes sure a solid basis is available to perform optimizations on.

This is how it used to work for all export formats (CSV, TSV, JSON) in the original code: A SPARQL query is sent by the client to the server, the server processes the query and stores the query result in memory. Depending on the required export format, another subroutine then creates a string in memory using CSV, TSV or the JSON format and then starts sending it to the client once the whole result is serialized

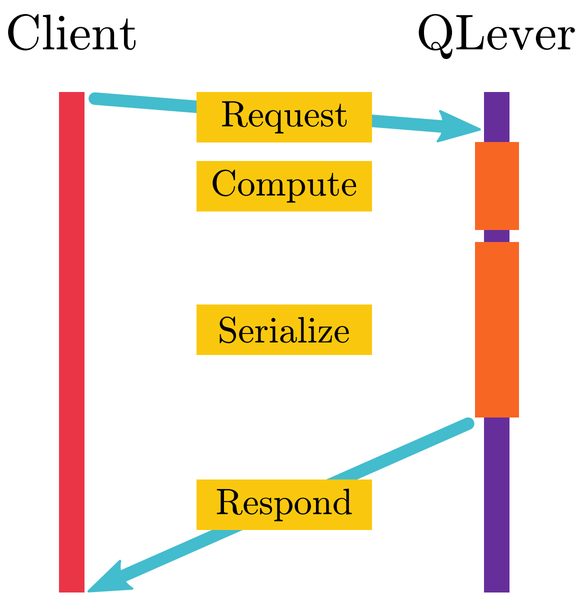

Figure 1: Original timing of request handling

completely. This has some implications for the efficiency of the export of the query result. This means that the server has to keep the entire result set in memory twice, a non-negligible overhead when dealing with large data sets. Send partial data once it is ready instead of waiting until the whole string has been serialized in memory is also a nicer experience for the client because less time is spent waiting until the server's first response. Having to acquire so much additional memory is what makes exporting error-prone in practice. There have been reports of people who wanted to extract subsets of large data sets, where the server didn't respond for a very long time, only to get killed by the Out-of-Memory-Killer before any data is sent, even though the memory would have been enough to calculate the result. Not only is this very annoying for end users, it also made a workaround necessary where individual 'pages' of the required result are queried using 'LIMIT' and 'OFFSET' SPARQL modifiers and concatenating those pages locally to get the entire result into a single file. Needless to say this is far from being an optimal workflow.

CONSTRUCT queries suffer the exact same problem as SELECT queries when it comes to large query results. A lot of memory needs to be allocated just to serialize a string in memory to immediately send it to the client and discard it afterwards. To add insult to injury due to the nature of CONSTRUCT queries, they often create a more than one triple for every matching item in the result table, thus making the effect even worse.

3.2 Serialization Formats

To reduce this seemingly unnecessary memory consumption we need to look at the individual export formats that are available: CSV, TSV, JSON and Turtle.

3.2.1 JSON

JSON (short for JavaScript Object Notation) is a lightweight data-interchange format. One of the key strengths of this serialization format is its simplicity. It is easy to parse, easy to generate and due to its text-based nature also kind of easy to read for humans, which may be the reason why it is so commonly used among the web. It is typically used as follows: A JSON object is created and some other 'objects' (which are associative maps in disguise), 'arrays' or 'values' which is itself a 'number', a 'string', a 'boolean' or 'null' are assigned to it. Once this object tree is complete, the whole tree is serialized into a string to be stored or sent elsewhere. This makes it rather unsuited for our idea where the response is supposed to be sent in smaller chunks. As of the time of writing Niels Lohmann's JSON library (**github.com/nlohmann/json**), which is currently being used by QLever does not support this niche use-case, even though it would be possible in theory. That's why we ignore this format for now. While it would be possible to optimize this as well, the effort required exceeds the scope of this thesis.

3.2.2 CSV and TSV

CSV (short for Comma Separated Values) and TSV (short for Tab Separated Values) are probably the most basic serialization formats one could think of. So basic even, that depending on the system several variants of the format exist, each with its own little catch. The format is suited to represent table-like data in a plain text format, the result of a SPARQL SELECT query for example. In their simplest form both formats use a comma or tab character respectively to separate columns and a newline character as separator so that each line contains all cells of a row. This structure makes it an ideal candidate to send partial data to the client once the partial data is ready.

At this point it should be noted that the official SPARQL recommendation settles on the CSV Definition RFC4180 [11] and the TSV definition by the Internet Assigned Numbers Authority (IANA) [12] when referring to those formats and acknowledges the limitations of those formats [10]. This means that this approach, even though it is suitable for a lot if not most purposes, is not the best choice for every case.

3.2.3 Turtle

The Turtle format [13] is SPARQL's preferred format to represent RDF Graphs. It shares a lot of similarities with SPARQLs syntax. Triples for example are separated with a dot, Blank Nodes and IRIs are represented using the exact same syntax and so on. It also has a special syntax equivalent to PREFIX statements, but we won't use that to keep things consistent with the CSV and TSV representations. Just like CSV and TSV the data is stored without any form of nesting, which makes it ideal to send partial data. Every triple can be sent on its own. Newlines are not required because dots are used as separators. For human readability the newlines will stay for now though, more on that can be found in Chapter 6, Section 6.1.1.

3.3 Hyper Text Transport Protocol 1.1

“The SPARQL 1.1 Protocol is built on top of HTTP” [14], which means that requirements for requests and responses have to be supported by software implementing the SPARQL 1.1 protocol. This makes it important to understand how HTTP, or rather HTTP/1.1 [15] in particular works on a low level, which is the HTTP version the SPARQL recommendation is built on. As a side note: It is possible to use HTTP/2 [16] or the upcoming HTTP/3 [17] by using a reverse-proxy to potentially make use of the theoretical performance gains newer versions promise, but the SPARQL recommendation predates those HTTP revisions so it does neither mention this possibility nor propose native support in any shape or form. (Chapter 6, Section 6.2 provides some further thoughts on this topic.)

Client:

```
GET / HTTP/1.1
Host: example.com
```

Server:

```
HTTP/1.1 200 OK
Content-Length: 13

Hello, World!
```

Figure 2: A simple HTTP/1.1 request and response

HTTP (at least for versions <3) uses TCP as its underlying transport protocol. So any data that is sent via HTTP is guaranteed to arrive sequential and in-order. Note that the HTTP protocol uses ‘CRLF’-style line breaks (carriage return + line feed), commonly found in the Windows world. For readability those characters are not explicitly visible in the figures. In Figure 2 everything after the empty line is called the response body. The ‘Content-Length’ header indicates how many bytes the client has to read until the body ends. If we want to start sending partial responses this header cannot possibly be set, because the server cannot know how many bytes the result will end up containing in advance. The most obvious solution to this problem would be to omit the header entirely from the response and just close the underlying TCP connection once everything has been sent. It is completely legal to

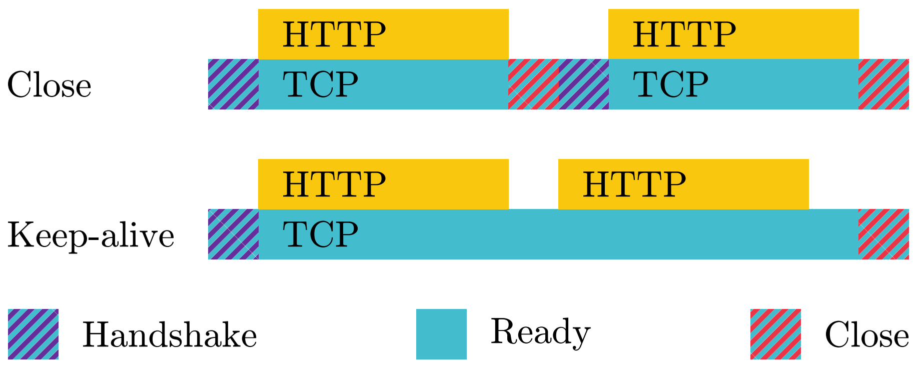

Figure 3: How HTTP/1.1's keep-alive mechanism can reuse TCP connections

do this in HTTP/1.1, but in most cases it is undesirable to solve the problem like this. On regular websites multiple requests are usually made in quick succession to load resources like CSS, JavaScript, images, sound and/or video after the initial HTML has been loaded, so closing and reopening a new TCP connection for every single request has a high overhead (see Figure 3). This is the reason HTTP/1.1 introduced the keep-alive concept. Basically if client and server agree via headers to keep the TCP connection alive, it will not be closed after each request so it can be reused in subsequent requests. This also introduced the so-called 'chunked transfer encoding'.

Client:

```
GET / HTTP/1.1
Host: example.com
```

Server:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked

6
Hello,
7
 World!
0
```

Figure 4: A simple HTTP/1.1 request with chunked response

Using this concept the request shown in Figure 2 could look like the one presented in Figure 4. As you can see now every chunk is prefixed using its length, so an arbitrary amount of chunks can be sent to the client until a chunk with length 0 is sent, which indicates the end of the body. This way the TCP connection can stay open for another subsequent request, without knowing the response body size in advance. The downsize of this approach is that if an error or exception occurs during body serialization (which is unlikely but not impossible) there is no way to indicate that to the client by using a HTTP status code like ‘500 Internal Server Error’ for example, simply because the status code is sent before the HTTP body is completely serialized. Mechanisms to provide a short error description are discussed in Chapter 6, Section 6.5.

Luckily the Beast framework allows to implement this chunked encoding. There is no built-in class that can be used, but by implementing a class that fulfills the so-called ‘Body’ concept and supplies the framework with the individual chunks, Beast will properly handle chunked transfer encoding. Concepts in Beast are basically requirements a certain class has to fulfill. If those requirements are sufficiently fulfilled by a class (there may be optional requirements) it can be used as a template type for a beast HTTP response. By omitting the optional static member function `Body::size` beast will interpret the whole response as chunked and set the required headers for us, leaving out the ‘Content-Length’ header completely. The only requirement that needs to be properly implemented here is the subclass ‘writer’ the ‘Body’ has to provide if it wants to support HTTP response serialization. This subclass needs to meet the requirements of a different concept ‘BodyWriter’. Basically any ‘BodyWriter’ has to implement an initialization function and a function `BodyWriter::get` that returns a new buffer to read from on every call. This is the core of the whole operation, where the magic happens. The tricky part of the implementation now is to figure out how to write code that is able to retrieve partial results whenever the Beast framework decides to call this `BodyWriter::get` member function.

3.4 Writing a generator function

There are several possible approaches to write code that allows to combine 2 seemingly different call stacks. The most obvious solution would be to spawn another thread that calls a blocking callback signaling the main thread to accept the generated data and unblocking the call once it has been processed by Beast. This repeats until no data is left and a zero length buffer is sent.

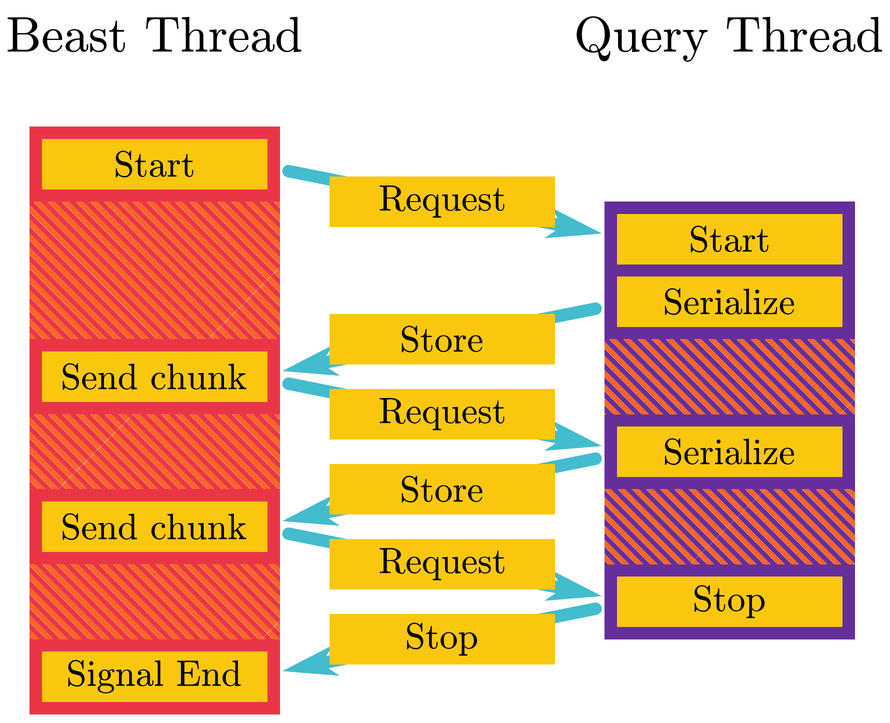

Figure 5: Threads are not working concurrently

This approach however, is a pretty wasteful use of resources. As you can see in Figure 5 both threads just pass the control to each other which causes only a single thread to be active at once, but consumes resources for 2 threads. Another approach using only a single thread would be to keep track of some sort of state and pass a pointer or reference to the generating function every time it needs to gather new data. This would work but requires the caller to initialize and manage this state memory-wise. This introduces a lot of boilerplate code and moves responsibility that should belong to the called function to the caller of the function. This makes this second approach rather undesirable purely because of the maintenance burden it introduces. Luckily with C++20 there is another way that works just like the second approach without the mentioned downsides: Coroutines. [4]

Coroutines look like regular functions (also known as subroutines) at a first glance. The only thing that makes them different is that they make use of one or more of the following three keywords: `co_await`, `co_yield` and/or `co_return`. The killer feature of coroutines is that they can suspend execution at any point during their lifetime. In a sense subroutines can be seen as a special case of coroutines that never suspend their execution. This suspendable execution is key here. The existing subroutine can be adjusted to `co_yield` any partial data to the caller instead of writing it into a stream. This returns control to the caller (Beast in our case) that can store the yielded data in a buffer and return a pointer to it for Beast to send it to the client over HTTP. Once the next piece of data is requested Beast can then resume the execution of the coroutine and the process repeats itself until all data has been yielded. Internally coroutines work very similar to the second proposed solution from earlier, but the state is managed by the compiler instead of the caller, which is why they are so convenient to use.

Coroutines in C++20 are designed to work with user defined types. There are open source implementations for common types like generators, tasks and so on, but none of them provided an interface with the same convenience `std::ostream::operator<<` provided up until this point. It allows for automatic formatting and conversion to a string. The solution to sort of keep this syntactic sugar and avoid having to convert the values by hand, was to implement a suiting type ourselves. This `stream_generator` as we called it accepts any type for `co_yield` that is accepted by `std::ostream::operator<<`, converts it to a string before finally storing it in a buffer. When `stream_generator::next` is called, the execution of the coroutine is started or resumed.

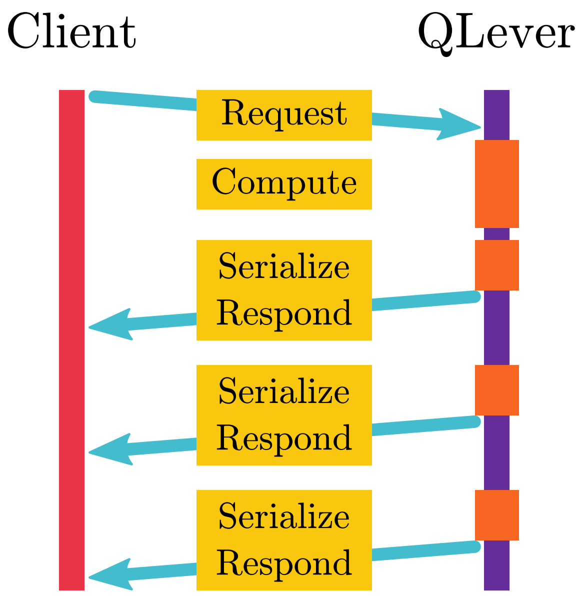

Figure 6: Timing of request handling when using a chunked response

Once new data is yielded and execution is suspended, `stream_generator::next` returns a read-only view to the buffer so Beast can use it. By using this approach to start the transport of data earlier in the pipeline, the required memory usage of the server can be decreased by a great amount. However, you might have noticed that the total export time would not decrease at all (see Figure 6). Instead, due to the overhead of chunked encoding it might even increase by a small amount. There is definitely some room for optimization there.

4 Optimization

4.1 Batching yields to reduce overhead

Using `co_yield` is nice to write readable code. On closer inspection of a HTTP request using this newly written code something odd becomes apparent. Figure 7 demonstrates the issue on a query with a single RDF triple as a result. Beast does not buffer our data in any way, but instead directly sends out the partial data directly as a dedicated chunk.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked

1c
<http://example.com/subject>
1
 
1e
<http://example.com/predicate>
1
 
1b
<http://example.com/object>
3
 .

0
```

Figure 7: Example of a chunked response without any intermediate buffer

This is a pretty severe problem and a direct result of code that looks roughly like Listing 7, which actually is the case for convenience reasons. The small little separating characters get the whole overhead of a chunk added to the total transfer size. This means that instead of sending just a single byte for the space character ' ' (U+0020), a total of 6 bytes are actually sent. The chunk size and the actual payload each take a single byte. The remaining 4 bytes are occupied by the necessary 'CRLF' separators in the HTTP protocol. Now imagine that this happens multiple times for a single triple, potentially for multiple millions of triples. This is an unacceptable overhead.

```
stream_generator generateResults() {
  for (size_t i = offset; i < offset + limit; i++) {
    auto triple = getTriple(i);
    co_yield triple.getSubject();
    co_yield ' ';
    co_yield triple.getPredicate();
    co_yield ' ';
    co_yield triple.getObject();
    co_yield " .\n";
  }
}
```

Listing 7: Example of a `stream_generator` in action

To keep this overhead as low as possible we introduced a small intermediate buffer where individual yields were aggregated before actually sending the bytes to be handled by Beast. An arbitrary buffer size of 1 MiB was chosen in the hopes to find a good balance for a value that is high enough the overhead is kept low, but low enough the memory consumption continues to stay low and the latency between chunks does not diminish any gains in throughput that were made so far. Especially for small query results this reduces the overhead to almost zero and even for results that are several GiB in size this adds an overhead of roughly 10 KiB for every GiB (an overhead of approx. $00,000953674\%$), making it almost negligible. It remains unclear how different buffer sizes affect overall performance. More on that in Chapter 6, Section 6.8

4.2 Benchmark Setup

CPU	AMD Ryzen 7 3700X 8/16 x 3,6-4,4 GHz
Memory	128GB DDR4
Storage	3.4TB SSD RAID 0
OS	Ubuntu 20.04

Table 1: Specifications of benchmark system

At this point it becomes increasingly harder to tell whether or not a certain change will actually increase performance or even decrease it because some newly introduced overhead starts taking up more CPU time than expected. This makes it necessary to have a setup that provides reproducible benchmark results. The chair of 'Algorithms and Data Structures' kindly provided a system to run example queries on to measure reproducible runtimes. The system in question has the specifications listed in Table 1 running QLever in a docker environment without virtualization. We hand-picked 4 different queries to measure the time for within the benchmarks.

All queries are performed on the full wikidata knowledge base. The trailing `LIMIT 18446744073709551615` on all queries is added to avoid the default result limit of 100000 of QLever. The simulating client is a python3 script using urllib3 (**github.com/urllib3/urllib3**) with a buffer size of 1MiB. When receiving the data it is immediately discarded without any further processing to avoid measuring the overhead of potentially storing or parsing the data. All benchmarks are performed using the CSV format 3 times in a row and the presented data represents the average of all 3 runs. Only the time of the body transfer is measured.

Query A

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?person_id ?person
       (COUNT(?profession_id) AS ?count)
       (GROUP_CONCAT(?profession; separator=", ") AS ?professions)
WHERE {
  ?person_id wdt:P31 wd:Q5 .
  ?person_id wdt:P106 ?profession_id .
  ?profession_id rdfs:label ?profession .
  ?person_id rdfs:label ?person .
  FILTER (LANG(?person) = "en") .
  FILTER (LANG(?profession) = "en")
}
GROUP BY ?person_id ?person
ORDER BY DESC(?count)
LIMIT 18446744073709551615
```

This is an example query provided by qlever-ui (**github.com/ad-freiburg/qlever-ui**) called ‘People and their professions’. It should be seen as a non-trivial query with a large expected result size.

Query B

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?person_id ?person WHERE {
  ?person_id wdt:P31 wd:Q5 .
  ?person_id rdfs:label ?person .
}
GROUP BY ?person_id ?person
LIMIT 18446744073709551615
```

This query is very similar to Query A. It is not limited to the english language and thus is expected to have to read from disk more often (due to QLever's memory layout), potentially revealing bottlenecks that have not as much to do with the overall performance of the code.

Query C

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?x ?y WHERE {
  ?x wdt:P31 ?y
}
LIMIT 18446744073709551615
```

This is supposed to be a rather simple query, designed to maximize the result size without making the filter essentially no-op.

Query D

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
CONSTRUCT {
  ?x wdt:P31 ?y .
  ?x wdt:P31 ?y .
  [a ()]
} WHERE {
  ?x wdt:P31 ?y
}
LIMIT 18446744073709551615
```

This query is similar to Query C, but wrapped as a CONSTRUCT query. The idea behind this setup is to synthetically increase the result size without altering the WHERE clause.

4.3 Trying to avoid redundant string copies

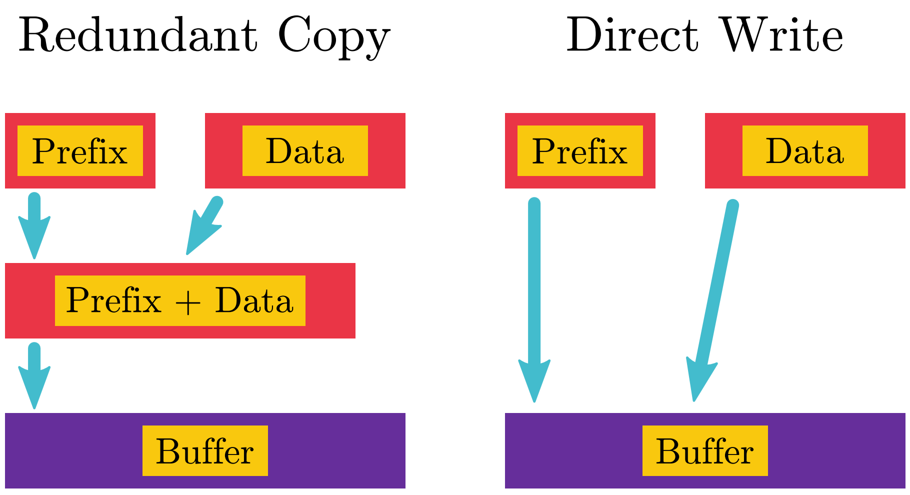

Figure 8: Strings are copied unnecessarily

While observing the coroutine that generates SELECT queries, we noticed there is a function that allocates a new string for every read from the result set even if it was just a simple concatenation and no extra allocation would have been necessary: The code could have just used the original strings instead of creating a copy to perform any read operations on (see Figure 8). The reason for this allocation is pure convenience. There are cases where allocating a string really is necessary, for example when parts of the vocabulary that have to be exported are not stored in memory but on disk instead. In theory this should always be the uncommon case, most of the data should be accessible in-memory. We theorized that if this extra allocation could be avoided for potentially millions of calls of this function this might be beneficial to the total runtime of the export. To put this theory to the test we implemented a proof of concept, a datatype that either stores a `std::string` or two instances of `std::string_view` at once. In theory this approach keeps dynamic allocations at a minimum level at the cost of a more complicated API. To quickly check the plausibility of this idea, the rudimentary changes were deployed to the benchmark machine.

	`std::string`	Custom datatype
Query A	60.99	57.44
Query B	55.23	53.79
Query C	105.78	97.34

Table 2: Average throughput in MB/s for the string allocation experiment

Unexpectedly, this resulted in a significantly lower throughput across the board of up to 8.44 MB/s as seen in Table 2. It looks like allocating small amounts of bytes is pretty well optimized and outperforms code that tries to unify the interface of the underlying `std::string` or `std::string_view`s. Maybe other factors played a more significant role to performance, we have to guess. After seeing those results during early development this idea was quickly scrapped.

4.4 Using the deflate compression algorithm

On the benchmark system the overall throughput is already pretty decent. However the average throughput is still way beyond the average download speed of 59.86 Mbit/s worldwide according to **speedtest.net** (as of January 2022) [18]. So most internet plans will not be able to match the throughput QLever is capable of. Additionally, if a single QLever instance has to handle multiple requests simultaneously the server's upstream might become the bottleneck. For this reason using compression algorithms to reduce the amount of data sent becomes very tempting. Luckily HTTP and therefore SPARQL allows for compression out of the box using the so-called 'Content-Encoding' header. The original RFC for HTTP/1.1 defines 3 compression methods that can be potentially used for responses: 'gzip', 'compress' and 'deflate'[15]. According to 'MDN Web Docs' the method 'compress' "is not used by many browsers today, partly because of a patent issue (it expired in 2003)" [19]. Therefore it was not considered as an option in this thesis. 'gzip' and 'deflate' are rather unfitting names for the same underlying deflate algorithm wrapped in the gzip and the zlib format accordingly, where ‘deflate’ has slightly lower overhead [20]. Therefore in the following benchmarks only ‘deflate’ is considered when referring to compressed data. Luckily Boost Iostreams (**github.com/boostorg/iostreams**) comes with implementations for both variants of the same algorithm. Instead sending the generated data directly to the client, it is now compressed first if they confirmed support using the ‘Accept-Encoding’ header. The HTTP client will then automatically decompress the received HTTP to get back the original data (see Figure 9).

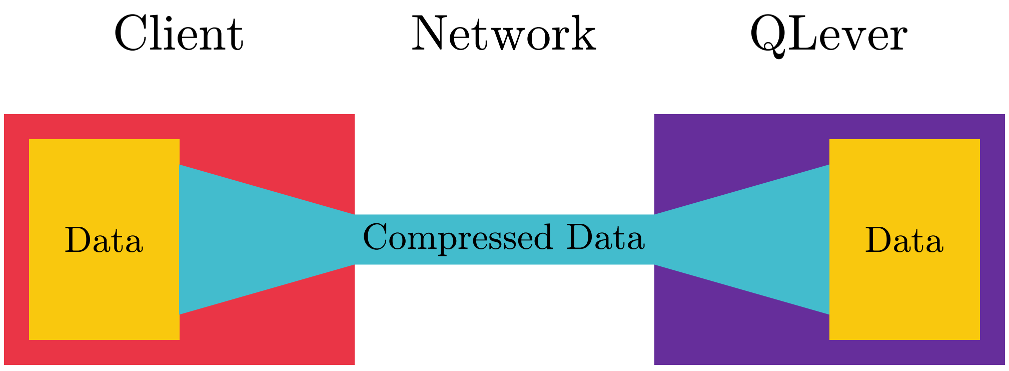

Figure 9: Reduce required bandwidth using compression

4.4.1 Different compression levels

The zlib (**zlib.net**) implementation of the Deflate algorithm is configurable using several compression levels, allowing to find a good balance between fast compression speed and good compression ratios. It ranges from 0 (referred to as ‘Fast’ from now on) over 6 (‘Default’) to 9 (‘Best’). It was unclear if the saved bytes of a better compression ratio would compensate for the additional processing time compared to faster approaches. This is why we performed benchmarks to find the best level for this use-case. For better comparison a so-called ‘effective throughput’ is provided, that indicates how many bytes per second would have been sent if the data was sent uncompressed. The actual throughput is provided in parentheses if it differs. The compression overhead has a hefty toll on the total throughput for all levels (see Table 3).

	None	Fast	Default	Best
Query A	106.18	(11.16) 55.14	(6.16) 35.99	(3.02) 18.01
Query B	123.74	(8.50) 76.75	(4.75) 48.43	(2.22) 23.75
Query C	457.53	(9.94) 192.83	(4.45) 99.75	(2.09) 51.39
Query D	139.71	(3.38) 105.03	(1.90) 68.43	(1.02) 39.70

Table 3: Comparison of synchronous compression levels

We expect that the mode 'Fast' would come closest to the original throughput, which it does. However, it still does not come close to the original throughput by a considerable margin in most cases. Regardless it is safe to say that using 'Fast' is the clear winner when using a high-end broadband connection ($\geq$ 100Mbit/s). For slower broadband connections ($30 - 100$Mbit/s) 'Default' is also a considerable candidate. Only for very slow connections ($<$30Mbit/s) 'Best' should even be considered, but at this point an upgraded internet connection might be a better choice overall if a truly high throughput is desired. More on optimizing the compression level is discussed in Chapter 6, Section 6.4.

4.5 Generating data asynchronously

A lot of time is wasted while waiting for the client to receive the data. Time that could be used to continue producing data concurrently as illustrated in Figure 10. Using an additional thread for this task would increase the overall throughput because the server can continue processing while another thread sends the data to the client. This comes at the cost of increased CPU load and memory usage.

We use a thread-safe queue to implement such a system. One thread runs until some partial data is yielded by the coroutine, adds it to the queue and resumes execution. Another thread waits until the queue is no longer empty and starts compressing and sending the data afterwards (see Figure 11). To avoid a buildup of memory when slow connections cause the sending thread to fall behind the generating thread a maximum

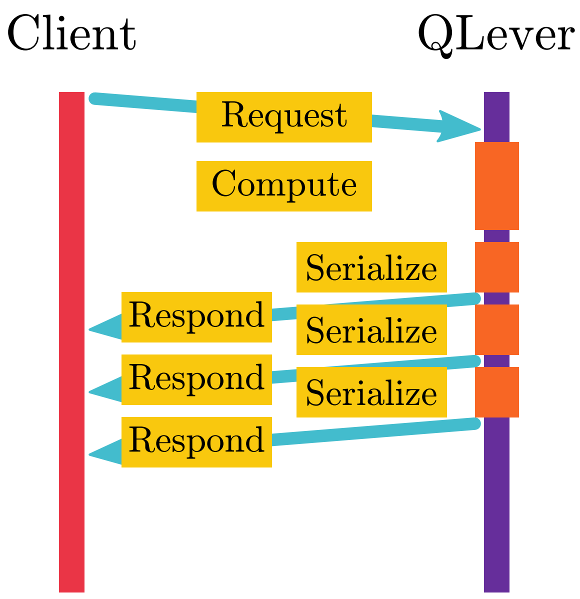

Figure 10: Timing of request handling when using a chunked response with an asynchronous layer

limit for the queue is necessary. We chose an arbitrary limit of 100, because it would consume up to about 100 MiB in memory worst-case, which seems like a reasonable amount to reserve for an individual query. Of course depending on the system there will most likely exist better limits.

	None	Fast	Default	Best
Query A	109.16	(21.97) 108.48	(9.10) 53.19	(3.57) 21.35
Query B	126.02	(13.85) 125.01	(7.46) 75.94	(2.70) 28.90
Query C	477.07	(15.68) 304.25	(5.42) 121.55	(2.31) 56.76
Query D	141.40	(4.48) 139.10	(3.56) 128.74	(1.44) 55.87

Table 4: Comparison of asynchronous compression levels

The benchmarks using this asynchronous approach shown in Table 4 clearly show that throughput increased across the board. Note how even uncompressed transmissions benefit slightly from this change. This makes asynchronous processing a very nice approach to increase efficiency because modern systems tend to have plenty of cores available that could be used.

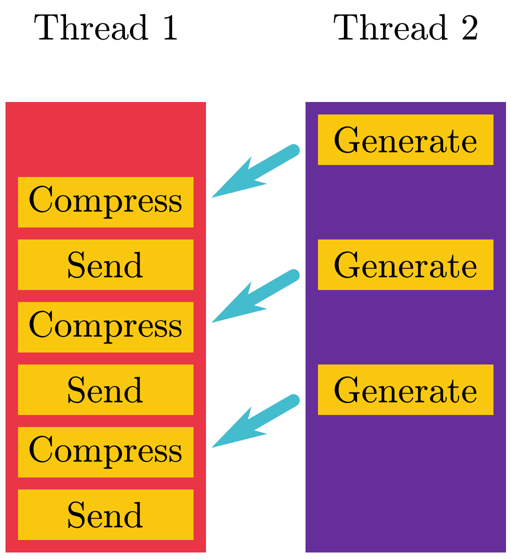

Figure 11: How an asynchronous layer would distribute work

4.5.1 Trying out two asynchronous processing layers

In the previous example only the data generation was executed on an asynchronous layer. The actual compression is still performed synchronously with Beast code. Could additional asynchronous compression increase performance even further?

	Fast	Default	Best
Query A	(21.92) 108.30	(9.12) 53.28	(3.59) 21.43
Query B	(13.76) 124.28	(7.45) 75.89	(2.71) 29.03
Query C	(15.95) 309.44	(5.44) 121.95	(2.30) 56.60
Query D	(4.41) 137.04	(3.60) 130.19	(1.45) 56.00

Table 5: Comparison of double asynchronous compression levels

The same layer introduced in the previous step can simply get applied twice. So making code changes to test this scenario is easy. Turns out the benchmark results disagree with our theory, as seen in Table 5. For 'Default' and 'Best' the throughput is barely increasing at all, and in some cases the throughput is even decreasing, presumably due to the increased overhead that comes with the required synchronization. So while this approach could increase performance in theory, it is not observable in practice.

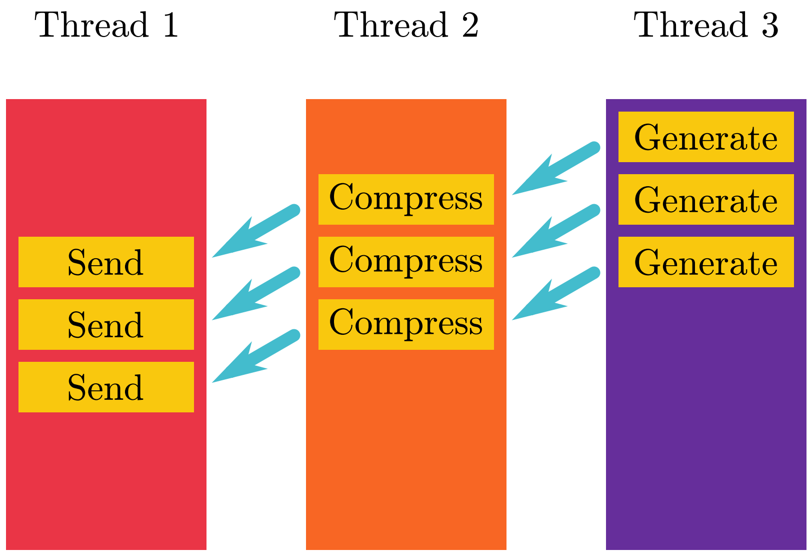

Figure 12: How 2 asynchronous layers could increase performance

5 Conclusion

QLever is now able to export query results at speeds that are able to exceed the typical internet bandwidth of regular consumers while keeping the required system resources at an acceptable level. This makes it easy to export large query results to a local machine, without any mandatory manual post-processing. Efforts that tried to apply clever tricks to avoid seemingly unnecessary operations were unsuccessful. Large parts of the achieved performance gains can be attributed to moving load off to newly spawned, dedicated threads. This increased the throughput by a large amount on certain types of queries, even without any opposing bandwidth limit. However there are still a lot of approaches to explore to increase throughput even further. During the works of this thesis several ideas came to mind. They could not be explored further for a variety of reasons, some of which are briefly discussed in Chapter 6.

6 Future work

6.1 PREFIX statements for Turtle export and other tricks

The Turtle syntax, similar to SPARQL, supports BASE and PREFIX declarations [13]. Just like in SPARQL these allow to reuse common prefixes in IRIs. Unlike SPARQL however, they can be declared at any point within the Turtle document. In theory this allows for a mechanism that counts how many times a certain prefix is used and starts using PREFIX and/or BASE declarations once a certain threshold has been reached to shorten any subsequent IRIs making use of this prefix.

```
<http://example.com/character#Isabella>
  <http://example.com#crushOn>
    <http://example.com/character#Phineas> .
<http://example.com/character#Isabella>
  <http://example.com#hasPet>
    <http://example.com/character#Pinky> .
@base <http://example.com> .
</character#Pinky> <#profession> </character#SecretAgent> .
@prefix char: </character#>.
char:Perry <#profession> char:SecretAgent .
```

Listing 8: Compression example making use of Turtle Syntax

As you can tell in Listing 8 the required amount of text is reduced the more triples are sent. This approach could greatly increase the density of data before applying an general purpose compression algorithm on the Turtle output, but would certainly introduce processing overhead and consume more memory to keep track of allowed prefixes.

6.1.1 Omitting newlines for Turtle

The CSV and TSV formats have newlines built into their syntax as a mandatory separator. Turtle uses a dot '.' (U+002E) to terminate RDF triples. The newline, as well as decorative white-spaces are only added by QLever to provide a nicer human-readable output. Obviously this is just wasting bytes if the output is supposed to be exclusively processed by another application. So removing any redundant characters from the output would be an easy way to reduce the total amount of bytes that need to be sent, even though it is expected general-purpose compression mitigates a large portion of this redundancy.

6.1.2 Keeping a dense CONSTRUCT representation

The initial CONSTRUCT query being used to generate the Turtle output in the first place might already use certain types of syntactic sugar. Currently this dense logic is unwrapped as explained in Chapter 2, Section 2.1.1 to provide a uniform representation regardless of the used export format. For export using the Turtle format, this dense representation could be kept as-is for the most part to avoid redundant repeating sequences of text. This would require a change to QLever's SPARQL parser and might introduce overhead for all other formats because they have to unwrap the dense representation on the fly.

6.2 Looking at HTTP 2 and 3

HTTP 2 [16] and 3 [17] each introduce major improvements to the transport layer of HTTP, making the chunked transfer mechanism of HTTP/1.1 redundant. It would be interesting to measure potential differences in performance when using those never versions of HTTP. This can be achieved either by using a non-standard native HTTP 2 or 3 server as a basis for the SPARQL protocol or by using a reverse-proxy supporting newer versions of HTTP in front of the existing QLever implementation (see Figure 13). It is unclear if the improvements to the transport layer promised by newer versions of HTTP provide any benefits for actual, slower connection speeds over the internet.

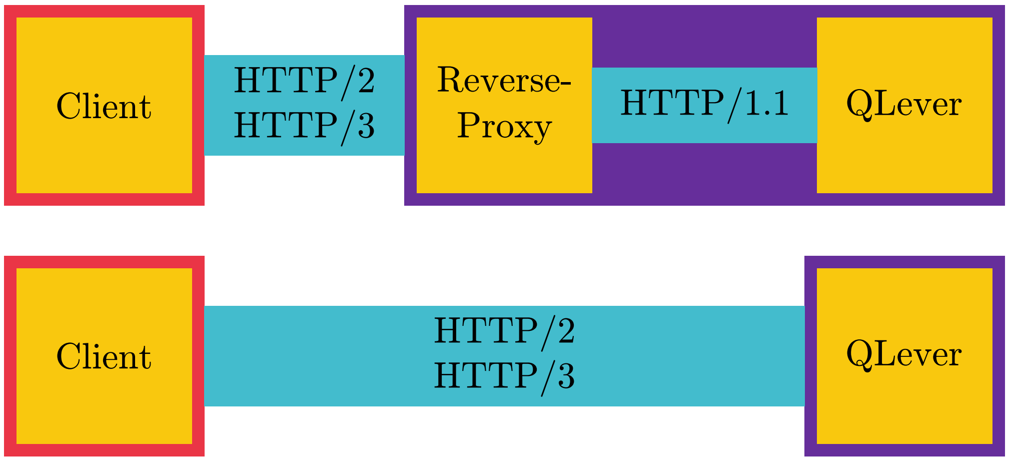

Figure 13: Reverse-Proxy vs native support

6.3 Brotli

In 2015 Google introduced 'brotli' [21] (**github.com/google/brotli**) as another alternative the the pre-existing compression algorithms for HTTP. According to 'MDN Web Docs' it is fully supported by all major, modern browsers [19] and provides competitive compression speeds when compared with the Deflate algorithm [22]. The Boost library which is conveniently already being used by QLever already provides a nice API to implement ‘gzip’ and ‘deflate’ compression for HTTP respectively. It would be interesting if the ‘brotli’ algorithm could provide any benefits to the effective throughput. The efforts required to implement a correct and fast compatibility layer of the reference implementation to work with Boost’s `boost::iostreams::filtering_stream` (currently being used for the other compression algorithms in QLever), would have exceeded the scope of this thesis.

6.4 Dynamic compression levels

A problem that was briefly touched in Chapter 4, Section 4.4.1 is that it is hard to find a good general-purpose compression level. On slow connections better compression can save a lot of time, but the overhead will increase the transmission time by a non-negligible amount. The gzip format allows for an interesting quirk. Concatenated gzip streams are a valid single gzip stream [23]. This means the compression level could be increased or decreased based on the most recent transmission times, by ending the gzip stream and immediately starting a new one using a different compression level. The client’s decompressor would then correctly decompress the original data. Another possibility would be to implement some sort of parallel compression algorithm where smaller chunks are compressed using gzip in parallel and concatenated at the end at the cost of a potentially worse compression factor.

This can not be applied to other compression formats, but it raises the question if the client should be able to tell the server the desired compression level using a custom header or GET parameter. Front-ends like qlever-ui (**github.com/ad-freiburg/qlever-ui**) might then be able to automatically select an appropriate compression level based on sample data of a smaller example request.

6.5 Error reporting after payload

A minor issue briefly mentioned in Chapter 3, Section 3.3 is that if an exception would occur during export it would be hard supply the client with any details about the error. HTTP/1.1 defines the concept of a so-called 'trailer' [15]. This allows any response with chunked transfer encoding to send a predefined set of headers once the chunked body has been sent completely. There is no header in the HTTP standard that is reserved for error reporting, because usually the body and status code can be used for that. By using a custom header to report error details a developer could find potential problems. We haven't implemented a mechanism like this, because it would be hard to make use of this additional information. It is simply unclear who would benefit from it. QLever developers can just look at the QLever log directly and end-users won't be notified by any regular HTTP client because this not standardized in any way.

6.6 String aggregating on different threads

In case the current transfer speed is not high enough there would be the possibility to split the string generation into batches. Thread 1 generates the string for the first 1000000 triples, Thread 2 works on triples 1000001 to 2000000 and so on (see Figure 14). It is unclear how much overhead this would produce and if there is a legitimate use-case where the current transfer speeds are not high enough. It is also unclear to which degree of multi-threading Beast would be able to handle the increasing amount of generated data or how tasks should be scheduled and joined to achieve optimal efficiency. This approach could be an alternative or addition to the approach tested in Chapter 4, Section 4.5.

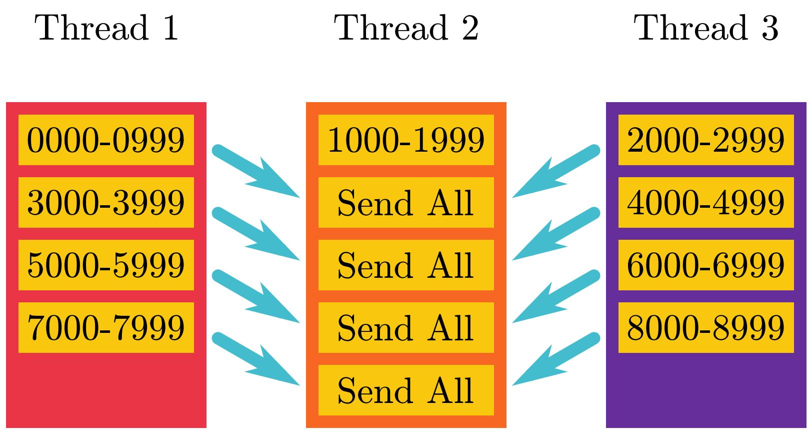

Figure 14: Splitting tasks among workers

6.7 Thread pools instead of new threads

The current code that provides asynchronous execution of the query result export process spawns a new thread for every incoming request. This is fine because the way Beast is setup within QLever does limit the amount of concurrent requests by using a fixed set of threads to handle them, so the CPU and memory impact has an implicit upper bound. For large scale deployments however, this can become a very wasteful use of resources. Similar to how requests are handled, a set of worker threads could be provided for general purpose tasks to avoid stopping and spawning threads for every incoming request. This would be especially useful for the approach discussed in Section 6.6.

6.8 Sizes of intermediate storage

One thing that we didn't look at in more detail are the sizes of intermediate storage. The inner `stream_generator` stores data until the 1 MiB threshold is reached. If this buffer threshold is too low there is no point of having this buffer in the first place and the introduced overhead mentioned in Chapter 4 becomes increasingly relevant. If it is too high, we end up increasing the latency between transmissions up until the point where the whole response is held in memory reintroducing the original problem the chunked transfer tries to solve. Similarly, the asynchronous layer's arbitrary limit of 100 could benefit from benchmarks on different systems using different values. It remains unclear if the limit affects performance in any way once it passes a magic threshold somewhere.

7 Acknowledgements

I want to thank my examiner Prof. Dr. Hannah Bast for her open mind towards fresh ideas and being very welcoming and encouraging regarding QLever development. I want to thank my adviser Johannes Kalmbach for his continued support, advise and for his insightful discussions about C++ development in general and QLever development in particular with me. I want to thank my parents for the privilege of being able to study without any financial or social pressure and my friends and family for the ongoing support and encouragement regarding my education.

Bibliography

[1] M. Lanthaler, R. Cyganiak, and D. Wood, "RDF 1.1 concepts and abstract syntax," W3C recommendation, W3C, Feb. 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ last accessed on 04th March, 2022.

[2] C. B. Aranda, O. Corby, S. Das, L. Feigenbaum, P. Gearon, B. Glimm, S. Harris, S. Hawke, I. Herman, N. Humfrey, N. Michaelis, C. Ogbuji, M. Perry, A. Passant, A. Polleres, E. Prud'hommeaux, A. Seaborne, and G. T. Williams, "SPARQL 1.1 overview," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/sparql11-overview/.

[3] H. Bast and B. Buchhold, "Qlever: A query engine for efficient sparql+text search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, (New York, NY, USA), p. 647–656, Association for Computing Machinery, 2017.

[4] ISO, *ISO/IEC 14882:2020 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, sixth ed., Dec. 2020. Final draft freely available at https://isocpp.org/files/papers/N4860.pdf last accessed on 04th March, 2022.

[5] H. Bast, P. Brosi, J. Kalmbach, and A. Lehmann, "An efficient rdf converter and sparql endpoint for the complete openstreetmap data," in *Proceedings of the*

29th International Conference on Advances in Geographic Information Systems, SIGSPATIAL '21, (New York, NY, USA), p. 536–539, Association for Computing Machinery, 2021.

[6] H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle, "Efficient sparql autocompletion via sparql," 2021.

[7] S. Harris and A. Seaborne, "SPARQL 1.1 query language," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ last accessed on 04th March, 2022.

[8] J. Broekstra and D. Beckett, "SPARQL query results XML format (second edition)," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/ last accessed on 04th March, 2022.

[9] A. Seaborne, "SPARQL 1.1 query results JSON format," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/ last accessed on 04th March, 2022.

[10] A. Seaborne, "SPARQL 1.1 query results CSV and TSV formats," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321/ last accessed on 04th March, 2022.

[11] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," Tech. Rep. 4180, Internet Engineering Task Force, Oct. 2005.

[12] P. Lindner, "Definition of tab-separated-values (tsv)." https://www.iana.org/assignments/media-types/text/tab-separated-values last accessed on 04th March, 2022.

[13] E. Prud'hommeaux and G. Carothers, "RDF 1.1 turtle," W3C recommendation, W3C, Feb. 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/ last accessed on 04th March, 2022.

[14] L. Feigenbaum, K. Clark, G. Williams, and E. Torres, "SPARQL 1.1 protocol," W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/ last accessed on 04th March, 2022.

[15] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Tech. Rep. 2616, Internet Engineering Task Force, June 1999.

[16] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," Tech. Rep. 7540, Internet Engineering Task Force, May 2015.

[17] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," Internet-Draft draft-ietf-quic-http-34, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[18] Ookla, LLC, "Speedtest Global Index," Jan. 2022. Data available at https://web.archive.org/web/20220301022049/https://www.speedtest.net/global-index last accessed on 04th March, 2022.

[19] MDN contributors, "Content-Encoding," Oct. 2021. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding last accessed on 04th March, 2022.

[20] M. Adler, "Why are major web sites using gzip?." StackOverflow, Feb. 2012. https://stackoverflow.com/a/9186091 last accessed on 04th March, 2022.

[21] J. Alakuijala and Z. Szabadka, "Brotli Compressed Data Format," Tech. Rep. 7932, Internet Engineering Task Force, July 2016.

[22] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne, “Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms,” *Google Inc*, pp. 1–6, 2015.

[23] L. P. Deutsch, “GZIP file format specification version 4.3,” Tech. Rep. 1952, Internet Engineering Task Force, May 1996.