University of Freiburg
Department of Computer Science

Master's Thesis

Lazy Evaluation of SPARQL Queries with Caching

Robin Textor-Falconi

Matriculation Number: 4544830

Supervisor and First Reviewer:

Prof. Dr. Hannah Bast

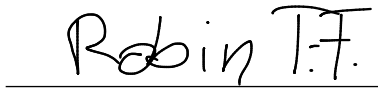
Second Reviewer:

Prof. Dr. Christian Schindelhauer

December 27, 2024

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 also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg, 27th December 2024 

Place, Date Signature

Abstract

QLever is a SPARQL engine capable of computing queries with results of billions of triples fast and efficiently. This is mainly because the operations' computations are almost exclusively done in memory. However, there are many cases where QLever has larger memory requirements than conceptually necessary. This work implements a new system that allows operations to perform their computations on partial results one after another to reduce the memory requirements up to an order of magnitude with minimal impact on performance while keeping some of the key benefits of the existing caching mechanism.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

1.1. Preliminaries

SPARQL [1] is a query language that standardizes how triples of data can be searched and processed.

QLever [2] is an engine that (currently incompletely) implements the SPARQL standard. This means that given a dataset and a valid SPARQL query, the SPARQL standard defines what the correct output should be. How the result is computed is left to implementing engines such as QLever to decide for themselves. This leaves a lot of room for implementors to decide how the computation of results should be implemented and which computations to value higher than others when performance-memory-tradeoffs have to be made eventually.

Many publicly available knowledge bases can be used as datasets for QLever. These include (but are not limited to) Wikidata [3] with roughly 20 billion triples containing information about a broad range of topics and UniProt [4] with roughly 160 billion triples containing information about protein sequences. With these large amounts of data efficiency is no longer a convenience, but rather a requirement if we expect a result to be computed reasonably fast.

1.2. Problem Definition

Processing in QLever typically happens by executing layers of operations one after another. To optimize certain queries where this chain of operations would cause inefficiencies, QLever uses hand-written optimizations when it detects a known pattern. This is predominantly used whenever data has to be read from disk where thanks to the index QLever builds, data can be accessed more efficiently without having to search through all of it, but also in other cases. This approach does unfortunately break down whenever the actual computation even just slightly deviates from the hard-coded patterns. The following trivial SPARQL query is a good example of a case where QLever is unnecessarily inefficient:

```sparql
SELECT * WHERE {
  ?a ?b ?c
}
```

Previously this kind of SPARQL query would cause QLever to make a copy of the whole dataset in memory just to export everything over the network. Of course, this specific case could be solved by writing code to handle this special case, but with a sheer unlimited amount of edge cases, this is only sustainable up to a point. Additionally, many computations could conceptually operate on partial data and produce partial results. This could make a huge difference in memory consumption because QLever is performing almost all operations purely in memory, consequently making smaller memory footprints possible. If additional handwritten optimizations were added to cover all of these cases the code would

become unmaintainable very fast. To prevent this QLever requires a generic interface to consume and process partial data so that every operation can benefit from every other operation that also supports this interface. The development and implementation of this interface is the main goal and topic of this thesis.

2. Related Work

2.1. MillenniumDB

MillenniumDB [5] is an open-source graph database system that shares goals somewhat similar to those of QLever. It achieves overall good performance across a wide range of SPARQL queries. Just like QLever, for the authors of MillenniumDB good performance on real-world data is highly valued, even outperforming QLever for some queries. Before the efforts of this thesis, MillenniumDB was able to efficiently handle queries that QLever severely struggled with[1]. Unlike QLever MillenniumDB does not seem to have a mechanism that makes use of chunked HTTP encoding [6] so large result sets have to be fully held in memory or stored on disk at some point during processing. It also does not come with a mechanism that caches query results[2]. So each invocation will trigger a new full computation chain.

To our surprise, we noticed that MillenniumDB does perform poorly on a small number of SPARQL queries that seem to be conceptually trivial. This is a query we found:

```SPARQL
SELECT ?a (COUNT(*) AS ?count) WHERE {
  ?a ?b ?c
}
GROUP BY ?a
LIMIT 1000000
```

All this query should be doing is to grab subjects from a sorted set of triples and count their unique amount of objects until a million unique subjects have been found and return that. For reasons unbeknownst to us, this takes more than 60 seconds to execute on their official demo site[3], and for smaller limits, an incorrect empty result is returned. Our efforts in this thesis made this specific query blazingly fast for QLever.

2.2. Virtuoso

Virtuoso, now an RDF graph store with built-in SPARQL and inference [7] is a well-known SPARQL engine that is often used as a baseline reference for SPARQL engines [2], [5]. For a lot of queries, it is a lot slower than QLever [2], but unlike MillenniumDB it does support chunked HTTP encoding. Unlike QLever, just like MillenniumDB, it does not come with a caching mechanism[2].

[1] See Section 5.3.2 for an example query.

[2] SPARQL is built on top of HTTP, so by using the right headers you can make use of HTTP caching under some circumstances.

[3] https://wikidata.imfd.cl

3. Background

3.1. Processing Stages

In the past, QLever used to process queries in 3 strictly sequential stages:

1. Query Planning
2. Computation
3. Export

In the query planning stage, QLever tries to determine the optimal tree of computations heuristically. This stage was not touched by the efforts of this work to not further increase the scope. However, there are many reasons to believe tweaking this stage to consider lazy operations in the heuristic has to potential to improve performance or reduce memory consumption for certain queries. More on that in Section 8.2.

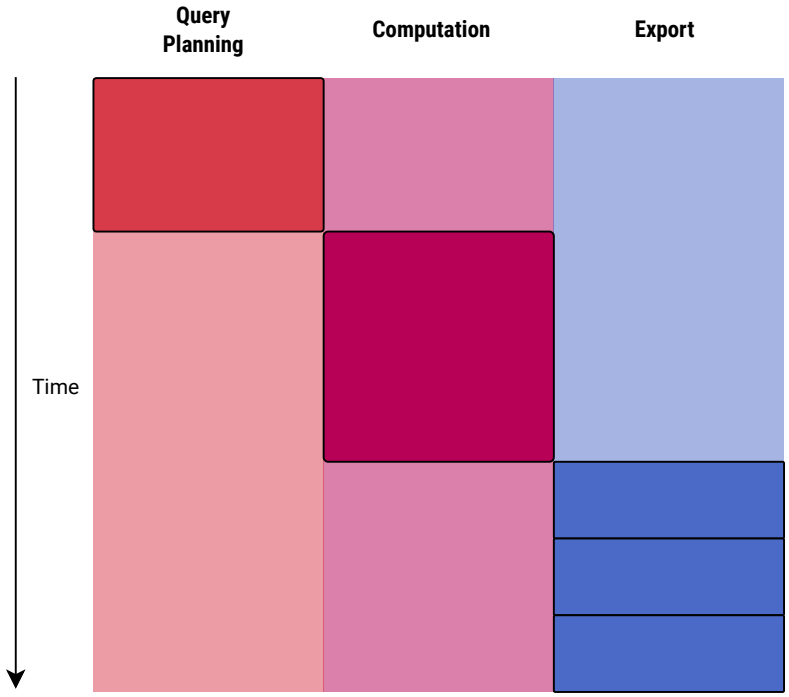


Figure 1: Model of previous processing order

Initially, the computation stage and the export stage were strictly separated[4]. This means that after planning, the complete result was computed before anything could be sent in response to the client.

The export stage then takes this fully materialized result and starts sending it to the requesting client. The heavy computation is done at this point, the data

[4]There is an exception for limits applied to the query result, which is omitted if the exporter can just apply the limit during export. So in this case the computation stage computes a slightly different result, which is then “corrected” by the export stage.

now only needs to get serialized to the HTTP stream as explored in our previous work [6]. In a nutshell, the compact internal representation is converted into a less compact, but text-based[5] format that can be understood and parsed across ecosystems. Especially for `CONSTRUCT` queries where the serialization step can multiply the number of entries of the internal representation by a large factor, this already saves a lot of memory. However, for cases where the result consists mostly of boolean values the internal representation is only marginally more compact than an equivalent CSV or TSV representation.

3.2. Caching

An important quality-of-life feature of QLever is the caching of intermediate results of a query. During query planning, QLever builds a tree of operations that (for the most part) work independently of each other. So generally an operation first fetches the result of its children and then transforms those values accordingly. The cache acts as a layer in between; when evaluating an operation the returned result is immutable, allowing it to be stored in the cache and be shared across multiple operation instances. So when an operation notices its result is already present in the cache, there is no need to recompute the result and the operation can just use it for further processing. When repeatedly making minor adjustments to SPARQL queries, which happens a lot when experimenting with queries, many partial results can be reused. This results in a better user experience because of lower execution times.

3.3. Optimizations

There is a big problem when strictly using this approach to calculate results: One of the most fundamental operations, internally called “IndexScan” is responsible for retrieving matching data from the knowledge base and turning it into a continuous memory block for further processing. This means that in the worst case, the whole knowledge base will be copied just to filter out most elements. This is not just incredibly wasteful, it would also make many computations infeasible to compute on consumer hardware because memory is a limited and precious resource there.

So to avoid this problem many operations implement specific optimizations when they detect that one of their children is a special kind of operation. The “Join” operation which as the name implies joins two tables together based on a common column, will handle the join of one or two “IndexScans” differently by skipping rows that are known not to match at all entirely, thus saving a lot of computation time and memory. The biggest limitation of this approach is that this system breaks down as soon as the “IndexScan” or any other optimizable sub-operation is no longer the direct child whenever the query demands a more complicated layout. Then the system will not detect the potential shortcut it could take and will perform a potentially multiple factors more expensive computation instead.

[5] QLever also supports binary export of data for debugging purposes.

4. Approaches

4.1. Conceptual Goals

At this point, we should have been able to sufficiently explain why a new mechanism is required for QLever to broaden the range of computations that it can efficiently perform. Luckily, as already suggested in Section 1.2 many operations conceptually do not need random access to all rows at once to make progress. The “Filter” operation for example only needs to read a single row to decide if this row can stay or needs to go. It would even be suited for perfect parallelism, where every row is processed in parallel. Many more operations share similar traits, whose implementation is explained further in Section 5.3.

Exporting as the last processing step also does not need all of the rows at once. As originally introduced in [6] and improved upon in a later effort, QLever is capable of streaming its result using the HTTP “Chunked transfer encoding” mechanism. This essentially makes it the requesting client’s responsibility to finally aggregate the data, moving the burden away from QLever’s limited memory constraints. This means that for a limited range of queries, it would theoretically be possible to compute very large result sets without QLever ever having to store more than one row in memory at the same time. Especially for operations with limits, this would allow us to avoid having to do a lot of computation when only the first n values of an operation are required.

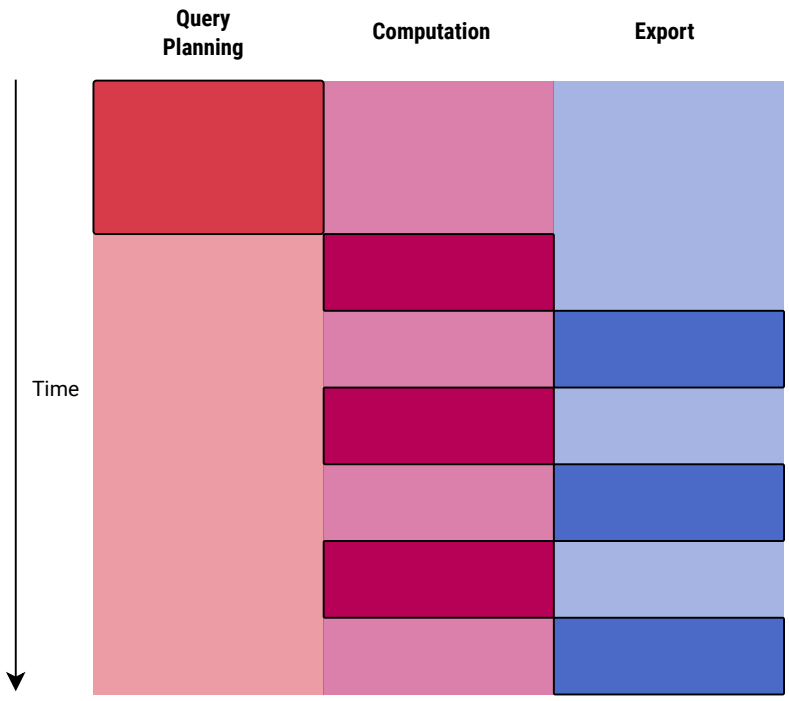


Figure 2: Model of proposed processing order

This is of course not a new concept. Purely functional programming languages like Lisp have been doing this since the late 1950s [8]. What is new here, is that an engine like QLever applies this concept to truly large amounts of data with performance in mind. In purely functional languages lists are typically represented as linked lists which has its benefits in a pure world where everything is a function and state does not exist, but is a rather inefficient layout of data in the world of today's CPU architectures. However, functional programming languages are usually compiled languages and can thus often transform their code into efficient CPU instructions. It is not a goal of QLever to be able to just-in-time-compile SPARQL queries to efficient machine code, so QLever has to be treated as a SPARQL interpreter instead of a compiler which does have some performance limits in specific cases.

4.2. Practical Goals

There are many ways to achieve our set goal in practice. However, today's CPU architectures are specialized to operate on local data, i. e. run CPU instructions that operate on data that is very close to each other in terms of memory addresses. This way a processor is much more likely to leverage the low latency and high throughput of its various caches because this increases the chances memory was already loaded into the cache by a previous instruction using nearby data. A somewhat recent optimization of this kind was performed in the Linux kernel networking stack, where optimizing memory locality resulted in up to 40% performance increase [9].

Because of this, processing each row individually will leave a lot of performance on the table[6]. So our approach has to be a compromise of grouping enough rows in a batch to continue to offer a high level of performance and few enough to not end up with the status quo where everything is materialized at once. In practice, we ended up targeting between 100 thousand and 1 million rows at once, which would be up to 8 megabytes of continuous memory per column, small enough to fit inside the L2 cache of a modern desktop CPU. This is more of an upper limit and soft objective though. For many operations, it made sense to deviate from this objective to make the code simpler or to avoid redundant copies just to meet the quota. There might be some room for optimization left here. If the generated batches end up consistently too small this will make the performance of the engine end up tanking quite a bit due to the overhead of the code. This overhead is discussed later in depth in Section 5.1.1. On the other hand, small batches can sometimes be favorable for operations that do not support limits out of the box. If the limit is smaller than the batch size the performance gains from not computing values that are not required can only ever be as large as the batch size allows it to. So if we have set a small limit, something in the single digit range, but the batch size is 10000, we are out of luck and the engine computes at least 10000 values before noticing it did compute

[6] To put this claim to the test we created a synthetic benchmark in Section 6.4.1

way too many values. It would of course be possible to make the operation limit-aware to avoid these kinds of issues.

4.2.1. Dealing with the Cache

This just leaves the question of how to combine this idea with caching. The existing code is conceptually a cache-first system. First, a result is computed, then it will be put in the cache (if the cache size allows it), and just then will the operation that requested the computation be handed over an immutable view of the result. The immutability is important to allow safe concurrent access across threads, but it also forces operations that could be implemented using cheap in-place algorithms to make potentially expensive copies of the data. So there is a significant cost to this approach, but also the benefit that if a query requires the same result of an operation twice, or if two queries try to fetch the same data concurrently it is only computed once.

Our original idea was to store the individual batches of rows individually in the cache, by augmenting the cache lookup key with an offset of the data. This would allow operations with a set offset to pick just the entries from the cache that they actually require. Upon closer inspection, this turned out to be a bad idea for a variety of reasons. If we are supposed to be able to find an arbitrary entry in the cache we either have to have fixed boundaries for the entries so calculating the lookup key becomes as simple as division or we would have to create some sort of lookup table to find cached entries for arbitrary offsets. For very large results this would essentially replace the whole cache with a result that does not even fit inside the cache in its entirety. To add insult to injury the latter case could also lead to the scenario where a result in batches is almost completely stored in the cache, except for the first batch, triggering a computation that eliminates the second batch from the result to make room for the newly computed first batch, ultimately resulting in a series of cache misses even though initially all batches except for a single one were present, demonstrating the worst-case for the implemented least recently used caching policy.

It also does not consider what to do if an operation can only guarantee a partial order of its rows. Currently, we assume all results to be ordered deterministically, even those that rely on hash-based data-structure implementations, but this might be something that might change in the future and is hard to prove beyond doubt for every single existing operation. Thinking a step further this also means that operations cannot efficiently use cached batches where one or more batches have been dropped from the cache. After all, to resume to this exact point the operation needs to be able to skip rows up until this point, which is potentially expensive because of the nature of the operation (a filter operation would not know how many entries to skip without evaluating them explicitly) or even impossible because the order might not be deterministic and so it needs to recompute everything to prevent duplicate and/or missing values. So this approach is not able to solve our problems.

Our second idea involves changing what a “result” is itself. Previously a result was little more than a wrapper around a single table of entries. We would always know exactly how many rows there are in the result set and be able to access any piece of data in constant time. But what if it could be one of two things? One mode where the existing behavior persists and one where we would be able to consume one batch after another. To make caching possible the result now stores those batches within itself once they are computed, allowing a second consumer to cheaply retrieve them, allowing the actual cache to drop everything at once if it deems the whole result too big. The main benefit of this approach is that it has similar characteristics as the status quo when it comes to caching. A single query that needs to run the same operation twice avoids the expensive computation the second time if the result fits inside the cache. The same goes for subsequent or concurrent queries.

This approach, however, has its downsides too. For one the caching mechanism becomes more complicated as its entries now suddenly can change in size. This is a direct result of making the result self-caching, they are no longer immutable and thus they are stored in the cache before the computation has started. This also means that we suddenly would need a mechanism to synchronize the threads. Error handling also becomes more complicated. If two concurrent queries are running the same operation simultaneously and one query runs into a user-defined[7] timeout or the user cancels the query computation, obviously this should only abort the respective query, not the one that just happens to piggyback off the computation. It also has the downside that if two queries want to compute the same operation, but one of them can process it an order of magnitude faster than the other, this would mean that either the faster query has to wait for the slower query to process its data all of the time or the slower query ends up wasting time. In this case, the faster query would be so far ahead that the values it computed have already been purged from the cache due to size constraints, causing the slower query to become even slower because it has to start over.

This leads us to our third idea and the one we ultimately settled on. The basic assumption here is that only one of the cases where caching yields huge benefits is very common. It is rather rare to see one or more queries require the exact same result from an operation. The only case where the cache is regularly hit is when iteratively altering queries to see how the results differ when different parameters are tweaked, as it often happens when performing experiments with QLever. So what if the cache becomes an afterthought? What if every operation would be forced to compute its own dedicated result if it wants to process only small batches of the whole result? Then we could have mutable tables without having to worry about concurrency issues and thread safety, allowing some operations to resort to cheaper algorithms. To allow this mechanism to still benefit from the cache, these mutable tables would have to be copied for a while into a single aggregate table

[7] While the user can lower this timeout on a per-query basis, the upper limit is set by the server and cannot be overridden without special privileges

until either everything is stored there and it still fits inside the cache or it is deemed too big and discarded. The only major drawback of this approach is that suddenly the part that does the caching becomes somewhat expensive because of the newly introduced step that copies memory, especially for large results. Section 5.2.2 discusses how these problems can be partially mitigated.

Besides that, it is way simpler than the other approaches and does not involve a new concurrency mechanism of any sort while still keeping some of the benefits of the existing caching system for a lot of cases.

5. Lazification

5.1. Implementation in C++

5.1.1. Generators and State Machines

How can we implement our proposed goal in code? QLever is written in C++[8], which means that we can use a language feature called “coroutines”. These can be interpreted as a generalization of a classic subroutine, also known as a function or method, that can suspend and resume execution at fixed suspension points. In practice, we use this feature to suspend execution after computing a batch of rows (from now on referred to as a “chunk”) for a parent operation to pick it up, perform its operation, and resume execution to compute the next chunk of data.

```cpp
#include <generator>
#include <vector>
#include <iostream>
// Yield all integers within [0, 10)
std::generator<int> tenValuesWithGenerator() {
  for (int i = 0; i < 10; i++) { co_yield i; }
}

// Return a vector containing all integers within [0, 10)
std::vector<int> tenValuesWithVector() {
  std::vector<int> result;
  for (int i = 0; i < 10; i++) { result.push_back(i); }
  return result;
}

// Both variants print the same output
int main() {
  for (int i : tenValuesWithGenerator()) { std::cout << i << ' '; }
  std::cout << std::endl;

  for (int i : tenValuesWithVector()) { std::cout << i << ' '; }
  std::cout << std::endl;
}
```

Listing 1: Code example to illustrate the basic structure. It requires C++23.

As we can see in Listing 1 both variants can be used interchangeably in this particular scenario. The only semantic difference is that the variant using the vector needs memory for all 10 values at the same time, whereas the variant using the

[8] At the time of writing C++20 specifically.

generator only requires memory for a single value at once[9]. To achieve this C++ turns the coroutine into an object that will run a function up to the point where a value is yielded based on the current state of the coroutine. So it is nothing more than syntactic sugar for classic state machines. Instead of integers, QLever yields a class it calls `IdTable`, which is pretty much just a table of IDs, 64 bits in size each. This also means that the function call to `tenValuesWithGenerator()` will not do any actual computation and return immediately. Only when we start iterating over the returned `std::generator<int>` object the actual code within the coroutine is invoked. This is why we will refer to this type of result as a "lazy result" and the previous type of result as a "fully materialized result" from now on. It is worth noting, however, that as of right now generators have somewhat of a bad reputation for being hard to optimize by C++ compilers. This will likely change in the future when more projects adopt coroutines in their code, but for now, this is another reason why it is so important that the chunks that are yielded are large enough to make the overhead from generators negligible in comparison. (See Section 6.4.1 for a concrete example.)

5.1.2. Local Vocabularies

Of course, some queries produce values that are not representable in just 64 bits respectively, mostly queries that involve creating a string of some sort. To handle these cases a result typically bundles an `IdTable` together with a `LocalVocab`. The IDs within the table then represent tagged pointers to a value in the vocabulary. To avoid vocabularies that store all values of a complete result, when all we really need is just the vocabulary for a partial result, for the case of lazy results the generator also needs to bundle a `LocalVocab` with the `IdTable` it yields. Unfortunately, this introduces new challenges. A `LocalVocab` does not know which parts of an `IdTable` correspond to each stored value, so when processing values originating from multiple sources we always have to assume that all entries of a local vocabulary are necessary which is incredibly wasteful. This is not a new problem, it is also present when dealing with fully materialized results. However, in the lazy case, because we no longer have a single `LocalVocab` for each result but thousands if not ten thousands of them worst-case with a lot of potential redundancy, merging a lot of them can suddenly become very expensive. This is especially noticeable when operations consume a lazy result but are constrained to produce a fully materialized result, requiring them to merge thousands if not tens of thousands of vocabularies (possibly with duplicates) into a single one.

Previously merging two local vocabularies involved just appending shared pointers[10] to the stored[11] values to an internal vector without any sort of de-

[9] Of course the generator needs to track its state between invocations, so technically this overhead needs to be taken into account as well, but it is negligible when the values yielded are an order of magnitude bigger than the variables on the stack.

[10] By shared pointer we are referring to an instance of `std::shared_ptr`, which is a pointer that manages its allocated memory using a reference counter.

duplication. In some cases where the same non-empty local vocabulary was yielded over and over again due to the nature of the operation, this meant a single local vocabulary suddenly held thousands of the same shared pointers instead of just a single instance, wasting a lot of memory. To combat this the vector was replaced with a hash set to provide an acceptable amount of de-duplication[12].

5.2. Implementation Quirks

5.2.1. Backwards Compatibility

While not strictly a requirement, it generally makes adopting new technologies and/or features easier if they allow for gradual opt-in. So what we did is that the default remains a fully materialized result for every operation. Only if an operation explicitly requests one of its children to provide a lazy result only then will try to provide one if it is supported and it makes sense to do so. This way all of the existing code continues to work without any changes and operations can be lazified step by step. Note that this means that just because we request a lazy result we are not entitled to actually get one, the only guarantee we have is that when we do not request a lazy result we are guaranteed to get a fully materialized result.

5.2.2. Generators in the Cache

A small engineering challenge that remains is to make lazy results interact nicely with the cache. As mentioned in Section 4.2.1 caching of lazy results should be nothing more than an afterthought. It is expensive to keep copying values into an `IdTable` that is just there in case the values might be small enough to be cached. Because of this, some additional constraints have to be met for the code to invest precious CPU cycles into maintaining a large copy that might not even be used in the future. One of those constraints is a configurable threshold that indicates how large the aggregation table is allowed to grow before it is deemed too large and its values are discarded. In addition, if we know in advance that the result size will be larger than this threshold, which is the case for index scan operations then it will not even try to cache it and avoid the expensive copy operations entirely. Also if aggregating tables results in a memory allocation error because there is just no memory left, then the value is also gracefully discarded to free memory. A caveat here is that the mere presence of a table for the cache could lead to such memory allocation errors elsewhere because it takes up space that other operations might need more urgently. And then, only if the generator is consumed in its entirety and thus the aggregation table can be deemed complete if it is still around, then, the aggregated table along with an aggregated local vocabulary will be offered to the cache. So the results in the cache are always fully materialized. Additionally, if an operation requests a lazy result and a fully materialized result is already present in

[11] The values are not stored inside the vocabularies. All the vocabularies do is to ensure the lifetime of the value pointed to by the stored shared pointer is at least as long as the lifetime itself.

[12] If the same Value is created multiple times independent of each other, they will not get de-duplicated because the hash implementation only considers the address in memory.

the cache, it will receive the cheap fully materialized version instead of having to compute a new one.

Because of this, an operation cannot know in advance if it is going to receive a lazy result or not and consequently if it is going to be able to provide one itself. This circumstance means that due to QLever's code architecture, we cannot possibly know if another result that is currently getting computed will turn out to be lazy, so the best we can do is wait until the computation is done and see if it turned out to be a fully materialized result we can use or a lazy result where the computation is still pending. This is not a huge problem though as long as we follow some guidelines. For one, if we happened to have waited for a result that turned out to be lazy, we just compute our own without querying the cache again. This is to prevent starvation of queries that would otherwise repeatedly wait for results turning out to be all lazy and thus unusable for direct cache storage. It is also important to make sure that when an operation creates a lazy result it must be able to quickly return and notify the cache of it. In practice, this is not always given, even though it likely would be possible with some refactoring effort. However, on one hand, this case should be rare enough that it will not become an issue as long as QLever does not handle multiple very similar queries at once regularly. On the other hand, if an operation did in fact trigger a full computation of a fully materialized result, chances are that this result will be stored in the cache so that waiting to find out a result is lazy and then starting the computation was not for nothing because we can then cheaply use the cached value instead. This of course only works when lazy results are never evaluated outside of the parent's lazy result because lazy results are not necessarily cached.

5.2.3. Live Query Updates

Another minor adjustment that had to be made to make lazy result works is to change the system that provides users with updates about the current query state in realtime. When all operations only produce fully materialized results, all that was necessary to provide updates in real-time, was to send an initial state at the beginning of a query computation and whenever a child operation finished, regardless if it failed or succeeded. For lazy results, we cannot do that. Instead, we have to send updates whenever the generator that computes lazy results yields a new value. This works but does cause a drastic increase in traffic, so much in fact, that it can overwhelm a web browser when it tries to parse all the sent JSON. To avoid this we put a limit in place that reduces the amount of updates to once every 50 milliseconds per operation[13].

5.3. Implementing Laziness Support for Operations

There are many types of operations QLever has implemented so far. A subset of those operations was changed to support laziness as part of this thesis. For

[13]Of course the last update will always be sent, regardless of the limit and two different operations might accumulatively send updates more frequently than the limit.

some operations, it makes more sense to support laziness than for others. The reasoning is explained on a case-by-case basis below. As a general rule of thumb, it makes sense to consume a result lazily and produce a non-lazy result whenever the incoming result is potentially really large and the outgoing result is expected to be much smaller than that to reduce the memory footprint. When an operation has only fully materialized results to deal with but its output is expected to be larger than its inputs it still makes sense to produce lazy values when requested because this way a large memory overhead can also be avoided.

5.3.1. `IndexScan`

The `IndexScan` operation is one of the most fundamental building blocks of almost any SPARQL query in QLever. It is responsible for reading the indexed dataset from disk, so anything that interacts with the loaded knowledge base is using an `IndexScan` one way or another. Because it is infeasible for a lot of operations to fully materialize a full scan, a "lazy" implementation already existed which was used as an optimization by a handful of operations. So all we did for this operation was to use the pre-existing functionality that read data from disk into memory in chunks (by default up to 30000 rows each) and made it available via the general interface introduced by this thesis if requested.

This allows the most basic SPARQL query (see Figure 5[14]) to be run completely lazy:

```SPARQL
SELECT * WHERE {
  ?a ?b ?c
}
```

5.3.2. `Filter`

The `Filter` operation for the most part is pretty basic. It computes the result of its child operation and eliminates entries if they do not match a set predicate[15]. This means that the `Filter` operation can only ever reduce the amount of entries, never increase it. So it makes sense to always request its data lazily. If the child operation supports it and returns a large result and the `Filter` operation result itself is not requested lazily, chances are that the predicate is strict enough to eliminate enough entries such that the fully materialized result is considerably smaller than the original, which has the potential to save a lot of memory. When the `Filter` operation itself is lazily requested, then the result type depends entirely on the type of the result the child operation produces. If it is lazy, then the filter algorithm is applied to every yielded table individually, otherwise, it is applied to the fully materialized result. The chunk size is entirely dependent on the chunk size of the child operation and the predicate. This means that currently if a predicate would

[14] This figure shows the execution tree for the respective operation.

[15] There is an optimization that avoids having to evaluate an expression for every entry if it is known that the data is continuous regarding the evaluation result e.g. y < x on a sorted range of values.

eliminate all rows within a chunk, the `Filter` operation would simply yield an empty table. Or, if the table is not completely empty this might lead to very small chunks, which is undesirable but the alternative would be to merge tables, including their respective local vocabularies which will likely cause a high memory consumption too because there currently is no mechanism to selectively remove entries from a local vocabulary, so we would have to keep memory allocated that might no longer be used instead. An idea to solve this is discussed in Section 8.3. Filtering is currently not using in-place algorithms, but it might be an optimization worth considering for lazy evaluation.

Here is an example for a query (see Figure 6[14]) where support for laziness makes a huge difference:

```sparql
SELECT * WHERE {
  ?a ?b ?c .
  FILTER(?a = ?c)
}
```

When we run this on Wikidata, this will result in ~17 million rows, out of the ~20 billion rows of the whole knowledge base. QLever is never even close to seeing the whole data in memory. Only the requesting client will ever have to store the resulting couple of gigabytes of data this creates.

5.3.3. `GroupBy`

The `GroupBy` operation is responsible for "grouping" i.e. aggregating multiple rows into common values. It can be used to calculate the maximum values and count rows, simply for concatenation across rows or other things. So most of the time `GroupBy` ends up producing a result that is smaller than the result provided by the child operation. If every row in the child's result is unique the result produced by `GroupBy` will have the same number of rows. So just like `Filter` it makes sense to always request the result lazily.

`GroupBy` expects its input to be sorted by default, which allows the code to assume that once a value is gone, no equivalent value will ever appear again in the input. Consuming a result lazily differs from the case where all values are known in advance. We need a mechanism that keeps track of intermediate values. Luckily in the past support was added for a hash-based `GroupBy`, which was initially developed to avoid sorting operations when beneficial and already does what we need with a caveat. It does not support all kinds of aggregation functions. Support for some was added where it was trivial as an effort of this thesis, but the non-trivial cases were left for future efforts. There is also a lot of room for optimization here, because when the input is sorted for aggregation functions like `COUNT(*)` we do not need to process each row individually but we could process them all at once, which is currently not done.

The hash-based variant of `GroupBy` is currently disabled by default. When enabled it is only used whenever it can be used to avoid an explicit sorting operation before it, but not if the data is sorted already or the aggregation function is not supported. This variant does not support consumption of lazy results, because the benefits are not as big as for the sorted case. A future contribution could add support for this case too.

When producing values lazily the chunk segmentation is mostly inherited from the child operation. This can also lead to very small chunks whenever a chunk contains only very few distinct groups. We do not consider this case to be too much of a problem, because if we assume operations generally produce large chunks this will mean that the `GroupBy` operation will reduce the amount of rows by a lot so the produced result will be an order of magnitude smaller in comparison.

A query (see Figure 7[14]) that counts the number of objects for every subject could look like this:

```SPARQL
SELECT ?a (COUNT(*) AS ?count) WHERE {
  ?a ?b ?c
} GROUP BY ?a
```

5.3.4. `Join`

The `Join` operation is a centerpiece of most SPARQL queries and does benefit a lot from being able to consume lazy inputs, even though a lot of the time this requires pre-sorting. As the name already implies it is used to find all matching rows of two matching columns. The algorithm to allow this already partially existed, primarily to optimize the join of two `IndexScan` operations. The `IndexScan` operation never produces results containing `UNDEF` values, which is why `UNDEF` support did not exist so far when only being used to join two lazy results. To be able to always request the child operations lazily, support for `UNDEF` was added and lazy results are always requested from the child operations[16].

When both children are fully materialized `Join` will also compute a fully materialized result. The basic assumption here is that most of the time only a few rows will match so a result of the size of a Cartesian product (which would happen if both join columns only contain the same single value) is highly unlikely and most of the time a join will reduce the combined amount of rows. Under this assumption, it makes sense to use the faster algorithm without any indirections.

For all the other cases if the result is requested lazily it will produce a lazy result. Joins in QLever internally use a flush mechanism to group write operations together. Whenever flush is called, the code checks if the amount of written rows exceeds the targeted chunk size, and if that is the case creates a new chunk (if it is

[16]There is an exception when one side of the join is already cached or its size is known to be below a certain threshold, which will cause the child to be fully materialized.

not empty) to be consumed by the next operation. After the last call to flush a final chunk is created with the remaining rows.

Joins where one side is of type `IndexScan` get special treatment, because in those cases we can ask the `IndexScan` to skip blocks that cannot possibly match the current value during a join. This kind of pre-filtering already existed for joins with fully materialized results and a variant was added that also supports the same mechanism when the tables are computed on the fly.

A query that combines two `Joins` (see Figure 8[14]) after another to illustrate this efficiency can be seen here:

```SPARQL
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?label ?birth_date WHERE {
  ?person wdt:P31 wd:Q5 .
  ?person wdt:P569 ?birth_date .
  ?person rdfs:label ?label .
}
```

5.3.5. `Bind`

All the `Bind` operation does is add a new column, where each row of the new column is created by a transformation based on the values of the other columns in the respective row. For this operation, it makes sense to just forward the request for a lazy result to the child. If the caller wants laziness and the child supports it too, `Bind` can just add a column "in-place", in the sense that the `IdTable` yielded from the child operation is modified, and yield it again with the newly added column. For the non-lazy case, everything stays as-is. The local vocabulary is of course extended analogously in both cases.

An example query making use of laziness in `Bind` (see Figure 9[14]) would look like this:

```SPARQL
SELECT (CONCAT(?a, ?b, ?c) AS ?string) WHERE {
  ?a ?b ?c
}
```

5.3.6. `Union`

The `Union` operation combines the results of two child operations by concatenating them and sometimes it also permutates the columns[17] so they match. If the result of `Union` is requested lazily it will always return a lazy result (apart from caching). This makes sure that even if both children return fully materialized, potentially very

[17] Tables in QLever have their columns stored independently of each other, so applying column permutations is linear in the number of columns, not rows, which makes it very cheap.

large results it never has to produce a table that has the combined size of both. If one or both children return lazy results `Union` does not even have to copy any data for lazy results because the tables are mutable. So all it does is to apply a permutation in this case, making the whole operation trivially cheap to perform.

An example query making use of laziness in `Union` (see Figure 10[14]) would look like this:

```SPARQL
SELECT * WHERE {
  { VALUES (?a ?b ?c) { ("S" "P" "O") } }
  UNION
  { ?a ?b ?c }
}
```

5.3.7. `Distinct`

The `Distinct` operation eliminates duplicated rows[18]. It expects its input to be sorted. On one hand, this is potentially a huge advantage because it makes this operation a perfect fit to consume a lazy input and provide a lazy output whenever a new distinct value is found. On the other hand, most of the time its child is a sorting operation which is inherently non-lazy[19], which means that very often we will not be able to capitalize on this new feature.

In the future, QLever might bring a hash-based implementation that avoids sorting. Such a variant would perform rather well in cases where unsorted data contains a lot of duplicates, but if most of the values are unique our hash-based data structure would eventually contain every unique value because it cannot possibly know if new values from the input are duplicates of any of its preceding values.

For the current implementation, however, it is rather simple to implement. Since `Distinct` can only ever reduce the number of rows, never increase it, just like `Filter` it makes sense to always ask its child operation for a lazy result. If the child operation returns a fully materialized result, `Distinct` will also return a fully materialized result, where only the distinct rows are copied over. If the child operation does return a lazy result and it is asked for a lazy result, it can even use an in-place algorithm since the data from generators is mutable.

For cases where the `Distinct` operation is non-lazily asked for a result, but the child operation does return a lazy result, `Distinct` aggregates all tables into a single one.

The simplest query to use `DISTINCT` (see Figure 11[14]) would look like this:

```SPARQL
SELECT DISTINCT * WHERE {
  ?a ?b ?c
}
```

[18] Of course we can selectively configure which columns should be checked for equality.
[19] See Section 5.4.5.

5.3.8. `TransitivePath`

The `TransitivePath` operation[20] can be used whenever the dataset needs to be traversed similarly to a graph. For example, if we wanted to find all subclasses of a certain class, but also the transitive subclasses of those subclasses. It can also come in handy whenever we want to track down transitive dependencies of an element in a knowledge graph. The size of the results of this operation heavily depends on the connectivity i.e. the amount of matching edges of the knowledge graph. So it can create very small results with only a bunch of rows but also very large results with millions if not billions of rows. Especially for the cases where the result size explodes with every newly traversed edge, it makes sense to produce a lazy result whenever requested.

For the result of the child operation to be traversed like a graph, we need constant time random access to each individual element in the result to be efficient. So we do not gain anything from requesting this result lazily, we would have to aggregate it anyway. A slight exception to this is an optimized case where a subsequent/parent join would eliminate a lot of rows not present in the result of another sibling operation. We still need the full graph to traverse it, but the sibling operation can be requested and subsequently processed lazily as part of the `TransitivePath` operation.

Whenever the result of the operation is requested lazily, one chunk is created for the results of every depth-first search started for every "node" on the starting side of the operation. This can often lead to very small chunks of varying sizes, which is undesirable as we explained, but this segmentation of chunks is very natural to implement because it allows a nice separation of concerns. For efficiency reasons, this should probably be changed in the future.

A simple query demonstrating the `TransitivePath` operation (see Figure 12[14]) on the Wikidata dataset could look like this:

```sparql
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT * WHERE {
  ?class wdt:P279+ ?subclass
}
```

5.3.9. `CartesianProductJoin`

The name of the `CartesianProductJoin` operation is a little bit misleading because it does not perform a join operation whatsoever. It could of course become a join operation when combining it with a `Filter` operation, which would represent the most naive `Join` algorithm one could think of. But as a standalone operation, all it does is produce a result that represents the Cartesian product of the results all of its children compute. So its size will be the product of the sizes of all of its children,

[20] The transitive path operation comes in 2 distinct variants that have the same behavior to the outside world, but differ in time and space requirements.

which means that its result becomes huge regularly. This characteristic makes it a good choice to produce values lazily. As a nice bonus, it can even consume up to a single child lazily.

Producing values lazily is as simple as it gets. Instead of producing one big table containing all of the values, we produce many smaller tables. We can even use the existing `LIMIT` and `OFFSET` support to pretend we just want to repeatedly create different slices of the actual result. This is why this is one of the few operations that can produce chunks that all strictly adhere to a fixed size.

Consuming a child lazily is where it becomes tricky. Consuming multiple children lazily, while in theory possible, is most likely a bad idea in practice. Due to the nature of Cartesian products for all but one of the operands we eventually need to go back to the start and repeat the values again and again. The generator model we introduced in this thesis does not allow that and even if we just acquired a new instance every time we need to start over, that would for one cause a lot of overhead and also potentially cause expensive computations to run that compute the same result over and over again. So simply requesting a non-lazy variant is almost certainly the better option. Even though the existing implementation of this operation did not guarantee any ordering, it always returned an ordered result if the rightmost input was ordered. In the future, this ordering guarantee might come in handy. However, as of right now, it must be known in advance if a result will be returned ordered or not. So to keep the possibility open to guarantee an ordering, we can only ever request the rightmost[21] operation to return a lazy result, without being able to fall back to one of the other child operations and permutating the columns afterward. Ideally, the rightmost operation would be the operation producing the biggest result that can be produced lazily. This would cut memory consumption down the most. Unfortunately, the order of the children is currently arbitrary, the result of a hash function.

A query that makes heavy use[22] of the laziness of a single lazy child operation (see Figure 13[14]) could look like this:

```SPARQL
SELECT * WHERE {
  VALUES (?a) { ("A") }
  VALUES (?b) { ("B") }
  ?c ?d ?e
}
```

5.4. Other Operations

There are many operations that were not touched as part of this work. For completeness, we list the reasons for this down below.

[21] This is an arbitrary implementation detail, the algorithm might as well be mirrored.

[22] 2 out of 3 times, the hash-function does not order the `IndexScan` last, so this is not guaranteed to make use of laziness using the current version of QLever.

5.4.1. `CountAvailablePredicates`

This is a special operation that optimizes auto-completion of SPARQL queries explored in [10]. As of right now, it de-duplicates its inputs similarly to `Distinct` by first sorting everything before eliminating duplicate values. This makes it a rather bad candidate for laziness.

5.4.2. `HasPredicateScan`

`HasPredicateScan` is another variant of the regular `IndexScan`. So similarly `HasPredicateScan` could support laziness. However, it was not important enough to make the cut given the limited time we had.

5.4.3. `Minus` & `OptionalJoin` & `MultiColumnJoin`

All of those operations are conceptually very similar operations to regular `Join`. The most common case that involves no `UNDEF` values would most likely be fairly easy to implement, given the current code base. However, the behavior specified by the SPARQL standard when it comes to joining columns that contain `UNDEF` values makes this algorithm rather complicated to implement correctly. A future bachelor's thesis might be able to unify all of these join variants to provide a single coherent interface. In any case, these operations serve a small enough niche so the implementation became a lower priority, low enough to make it not fit within the time constraints of this thesis.

5.4.4. `NeutralElementOperation`

This is an operation that always returns a single row with zero columns. It is a pure helper construct for joins. So there is nothing to do here.

5.4.5. `OrderBy` & `Sort`

Sorting algorithms are inherently non-lazy. We need to consume the whole input before being able to know which one is the first element and return that. This is why intermediate sorting algorithms often "break" lazy chains where nothing is ever fully materialized. This is why this operation was not touched as part of this work. There might be potential to optimize this issue by adjusting query planning to prefer query plans that avoid sorting operations if there are alternatives that work without them.

Some ideas on how to take advantage of lazy processing will be discussed in Section 8.1.

5.4.6. `PathSearch`

`PathSearch` is somewhat similar to `TransitivePath`, it can be seen as a generalization of it that not only returns the result but also the paths that lead to it. So it would be possible to add support for lazy processing just like `TransitivePath`, but it simply was not a priority given the niche it serves.

5.4.7. `Service`

The `Service` operation represents federated queries that fetch data from other SPARQL engines remotely. Because data transfer is inherently some sort of stream,

it makes perfect sense to also support laziness. In fact, this operation was changed to support laziness during the efforts of this thesis, but not as part of this thesis, even though it only was made possible because of it.

5.4.8. `SpatialJoin`

A `SpatialJoin` is not really a join in the classic sense. It is used to match geo-spatial data, where two locations have no direct match and we are searching for either the nearest match or all matches within a certain distance. To achieve this an index is built for one of the two join sides, to allow the other side to quickly search for matching data. This means that the side used to build the index will not benefit from laziness because it would have to aggregate all data anyway. The other side however could be consumed lazily, so ideally the smaller side of both would be used to build the index and the bigger side requested lazily to achieve the smallest memory footprint possible. Most definitely a good candidate for future improvement.

5.4.9. `TextIndexScanForEntity` & `TextIndexScanForWord` & `TextLimit`

These operations were added as part of [11], a somewhat recent addition to QLever. This allows QLever to quickly perform full-text searches using additional indexed data. In a somewhat similar fashion to `IndexScan` and `HasPredicateScan` supporting laziness would be feasible here as well if this becomes a common use-case.

5.4.10. `Values`

This operation is used to represent hard-coded values from the SPARQL query. Therefore we can assume that the amount of values present is comparatively low and always fully materialized anyway.

6. Experiments

6.1. Setup

CPU	AMD Ryzen 9 7950X
Memory	48GB[23]
Storage	2x 4TB WD SN850X NVME, Software-RAID 0
OS	Ubuntu 24.04.1 LTS
Git hash	`4237e0d4af70e6e400f4357f61756eb5873fe98a`
Build configuration	Release, Parallel, GCC 13.3.0
Dataset	Wikidata 2024-11-14

Table 1: Benchmark environment

To measure performance in all of the following experiments we submit a query via HTTP and set the expected output format to QLever's proprietary JSON format which also includes a computation time with millisecond precision. To avoid waiting for the potentially slow export via the network and for it to skew the peak memory usage, we always set the `GET` parameter `send=100` which limits the output but still calculates the whole result.

All tests are repeated twice. The reported value will represent the arithmetic mean of all 3 values.

6.2. Comparing Performance and Memory

For this experiment, we compare all example queries from Section 5.3 on the Wikidata dataset plus a variant of the `GroupBy` query, also found in Section 2.1, and a variant of the `CartesianProductJoin` example presented in Section 5.3.9 with an added `LIMIT 1000000`, as well as the following query cherry-picked from qlever-ui's[24] example query set to act as more of a real-world example.

[23] Artificial memory limit via configuration.

[24] The official qlever-ui demo can be found at https://qlever.cs.uni-freiburg.de.

```sparql
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX schema: <http://schema.org/>
SELECT ?name ?pic ?sitelinks WHERE {
  ?person wdt:P31 wd:Q5 .
  ?person ^schema:about/wikibase:sitelinks ?sitelinks .
  ?person wdt:P18 ?pic .
  ?person schema:name ?name .
  FILTER (lang(?name) = "en") .
}
ORDER BY DESC(?sitelinks)
```

Listing 2: Example query “People with Pictures”

6.2.1. Methodology

To directly compare the performance and memory overhead of laziness we use a second variant of the code that never requests lazy results[25]. This is to avoid any performance differences resulting from any other aspects of the code.

To measure the memory footprint of queries we measure the memory usage at the point before the query and compute the difference of this value and the peak memory usage after the query has finished.

6.2.2. Results

The raw data for this experiment can be found in Table 4 and Table 5. Below we can find a prettified version that is easier to interpret.

All values denoted in red performed worse than their respective counterpart. All values represented as a single dash indicate that the query did not successfully finish due to memory allocation errors. For Wikidata these queries would typically require more than 160GB of memory which is more than our test machine is capable of providing.

[25]The exact code change can be found in Listing 3.

Query	Processing Time		Memory Delta	
	Lazy	Non-Lazy	Lazy	Non-Lazy
`IndexScan` example	20s	-	702MB	> 48GB[23]
`Filter` example	127s	-	702MB	> 48GB[23]
`GroupBy` example	608s	-	702MB	> 48GB[23]
`Join` example	3715ms	4833ms	702MB	2151MB
`Bind` example[26]	447min	-	1130MB	> 48GB[23]
`Union` example	39s	-	702MB	> 48GB[23]
`Distinct` example	82s	-	702MB	> 48GB[23]
`TransitivePath` example	13s	15s	702MB	2244MB
`CartesianProductJoin` example	111s	-	702MB	> 48GB[23]
`GroupBy` example variant	385ms	-	705MB	> 48GB[23]
`CartesianProductJoin` ex. variant	144ms	122ms	702MB	702MB
People with pictures	2342ms	2222ms	851MB	996MB

Table 2: Comparison of queries with and without laziness capabilities

The measured data mostly confirms our initial assumptions and expectations. Of course, the selection of queries to benchmark is biased towards worst-case scenarios for QLever, where without any laziness functionality the memory requirements are really high. This is why most of the queries fail when forced to run without laziness enabled. There are some exceptions though.

A lot of the queries were measured to have 702MB of peak memory usage. This is almost certainly an artifact of the measurement approach. It indicates that QLever at some point requires 702MB of additional memory to serve a response for a single request. We do not know if this spike in memory usage occurs at some point between the start of the HTTP request and the start of the computation, or if it is an artifact that is itself caused by every query. To find out an involved memory analysis would be required which is non-trivial to do in the worst case. Regardless our approach is good enough to be able to isolate memory-intensive non-lazy operations that are significant enough to stand out.

The `Join` query represents a little bit of a special case, because for `Join` QLever already had optimizations in place specifically for the case where the results of two `IndexScan` operations had to be joined. This case can be drastically sped up by making use of the index so blocks of rows that are known not to match at all can be skipped altogether. This is why even if the newly introduced laziness mechanism is disabled, this optimization allows the `IndexScan` operations to essentially perform the same actions as before. The increased memory overhead is a direct effect

[26]For `Bind` the values were extrapolated because creating a string for every triple would take an unreasonable amount of time. Based on a `LIMIT 100000000` and 20,054,336,444 triples in total we estimated a linear growth in runtime and the memory consumption to be roughly constant.

of the circumstance that even with this specific optimization still working `Join` itself has no other choice than to store the computed result set in a single fully materialized table. When executing lazily only the individual chunks end up fully materialized, allowing QLever to never fully materialize the result at any point in the computation chain. Somewhat surprisingly the lazy variant of the computation is significantly faster than its non-lazy counterpart. We have to assume this is because expensive allocations and reallocations of large memory areas can be avoided in the lazy case, but a definitive answer would require a closer inspection.

The `TransitivePath` example query likely succeeds because it only operates on a relatively small subset of the Wikidata graph, on roughly 5 million rows, leading to a result with roughly 130 million rows. Even when the results have to be fully materialized this amount of data will fit comfortably inside the memory of most consumer hardware. The drastically increased memory requirement of the non-lazy variant is unsurprising because in this case the 130 million rows never have to exist in memory all at once. The 2-second reduction in execution time is also somewhat unexpected but within a range that can reasonably be attributed to measurement uncertainty.

Neither the `GroupBy` query nor its variant with a limit do successfully complete in the non-lazy case. It is worth pointing out though that in the lazy case the variant with limit runs significantly faster, even though the `GroupBy` operation does not natively support limits. This is because the mechanism that enables lazy consumption does allow ending consumption on every chunk boundary and thus enables this huge improvement to execution times without specifically implementing it for every operation.

The original query for `CartesianProductJoin` does not complete in the non-lazy case but does complete for the variant with a stricter limit. This is because `CartesianProductJoin` is one of the few operations that can natively handle limits and propagate them accordingly to its children. For this query, the slight increase in processing time for the lazy case can be explained by all of the overhead this mechanism brings along. The memory overhead with limit for both cases, lazy and non-lazy is too small to stand out and thus too small to reason about.

The “People with pictures” (Listing 2) query was chosen to be more of a real-world example rather than a worst-case scenario. This is why both the lazy and non-lazy cases finish reasonably fast. The lazy case brings about 5% performance overhead, but a reduction of 15% in terms of peak memory consumption. These gains and losses are not representative of a wide range of queries, every query is different, but they show that laziness does allow to trade memory for performance in some cases.

6.3. Performance Penalty of Caching

For this experiment, we use the same queries as Section 6.2, but we apply a `LIMIT 1000000`. Exceptions to this rule are the query for `Join`, which is left unchanged,

and `CartesianProductJoin` which is used with `LIMIT 100000000` instead to account for the different characteristics of the operations. Because our two variants in the previous experiment only differed by their `LIMIT` they are no longer of any interest to us and thus we have 2 fewer queries to observe.

6.3.1. Methodology

To better understand the overhead the newly introduced caching mechanism for lazy results introduces we measure the computation time for the respective queries, using different maximum caching thresholds. This threshold can be controlled via a CLI flag `--lazy-result-max-cache-size`. It defaults to 5MB. For this experiment, we also try 500MB and 5000MB.

6.3.2. Results

The raw data for this experiment can be found in Table 6. Figure 3 visualizes this data by representing the default case with green bars and the other thresholds in yellow and red respectively. The error bars indicate the standard error of the 3 individual test runs, the actual value of the bars represents the arithmetic mean. To represent all measured values in the same figure, they are all divided by the mean time of the 5MB configuration. So the green bars all start at 1 for reference.

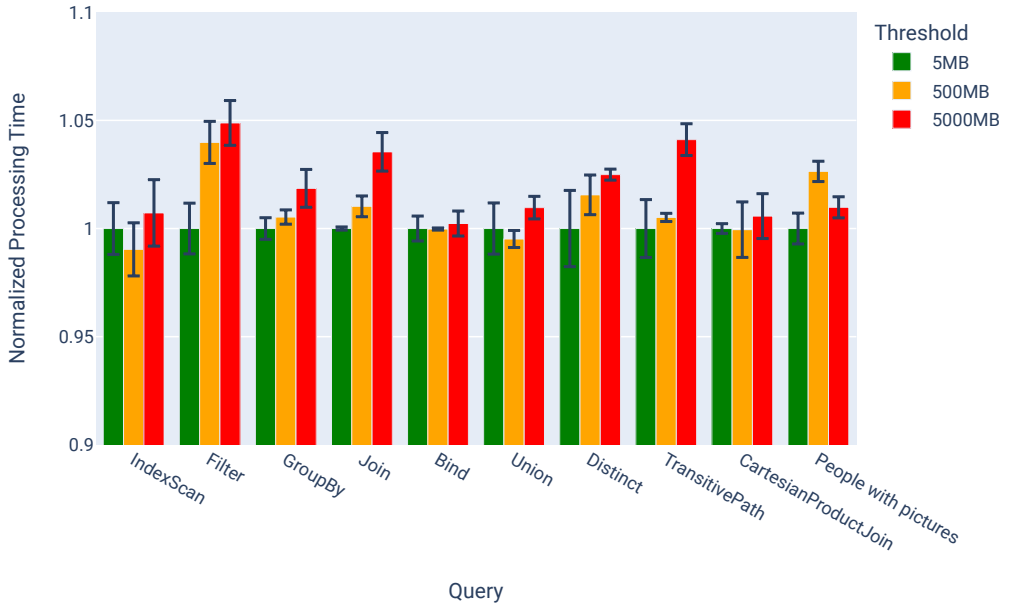


Figure 3: Overhead caused by caching lazy results

We would expect that by increasing the threshold the overhead and thus the processing time would increase. And for the most part, this is what we can observe in the plot. An exception to this can be observed for `IndexScan`. This makes sense because `IndexScan` is the only operation that does exactly know the size of its results in advance so if the result is too large it will not even start to aggregate the

tables to be able to cache it. This is why the standard errors of those three bars overlap each other so clearly.

It is not completely clear why for `Bind` the bars are all so similar. We assume this is because, for this particular query, most of the time is likely spent creating strings and allocating the required memory instead of doing any actual computation. This could be a potential reason for this observation. For the “People with pictures” query, there seems to be an oddity with the 500MB and 5000MB configuration. However, in this case, the actual result sizes might slightly exceed 5MB, but will never get to sizes where the 500MB threshold is too small. This is why increasing the threshold even further will not change anything for this particular query and so the oddity can be explained by the variance of the measurement process.

6.4. Testing the Effects of Chunk Sizes

We want to observe the implications for the performance of different chunk sizes.

6.4.1. Synthetic Benchmark

To empirically test the claim made in Section 4.2 we created a synthetic micro-benchmark using “Google benchmark”[27]. The implementation for this benchmark can be found in Listing 4 and Listing 5.

We ran this benchmark using the same compiler configuration as all other experiments in this chapter. We have 8 variants of the same generator that all try to create 100 million 32-bit integers and subsequently sum all of them up (this is not enough to result in an integer overflow). To make it a little bit harder for the branch predictor, the values generated start at a random value, which is then incremented for every newly inserted value.

Chunk Size	Ø Time	#Samples
1	972 ms	1
(inline) 1	253 ms	3
10	134 ms	5
100	39 ms	18
1000	31 ms	22
10000	28 ms	24
100000	27 ms	25
1000000	28 ms	25
10000000	159 ms	4

Table 3: Total average processing time of the benchmark code

As we can see in Table 3 we reach a sweet spot at a chunk size of 100 thousand elements. For sizes any lower it seems that the overhead the generator introduces

[27] Google benchmark: https://github.com/google/benchmark

slows things down a bit, especially when the chunk size is just a single element where it becomes unreasonably high. We assume this is because the benchmark repeatedly allocates memory on the heap for just 4 bytes which is very inefficient. To have a closer look at this effect, we created a second variant of the benchmark (Listing 5) that yields the integers directly instead of wrapping them inside a vector. As we can see the overhead is still rather high in comparison, but way lower than before. Once we reach a chunk size of 1 million the time is increasing again, presumably because the memory regions grow so large they no longer fit inside the CPU cache they could previously comfortably reside in.

6.4.2. Methodology

To see the practical effects of chunk sizes concerning runtime, we run 2 queries with 4 different chunk sizes, namely the example queries shown in Section 5.3.4 (`Join`) and Section 5.3.9 (`CartesianProductJoin`) because those are currently the only operations with fixed chunk sizes. They were also used in the previous experiment in Section 6.2. Because those chunk sizes are not configurable, we need to modify the code directly. See Listing 7 and Listing 6 for the respective changes[28]. We compare chunk sizes of 1,000, 10,000, 100,000, and 1,000,000, so the code is adjusted for each case accordingly.

6.4.3. Results

The raw data for this experiment can be found in Table 7. Below it is visualized in Figure 4. The shown values indicate the mean of three runs, the error bars show the respective standard error.

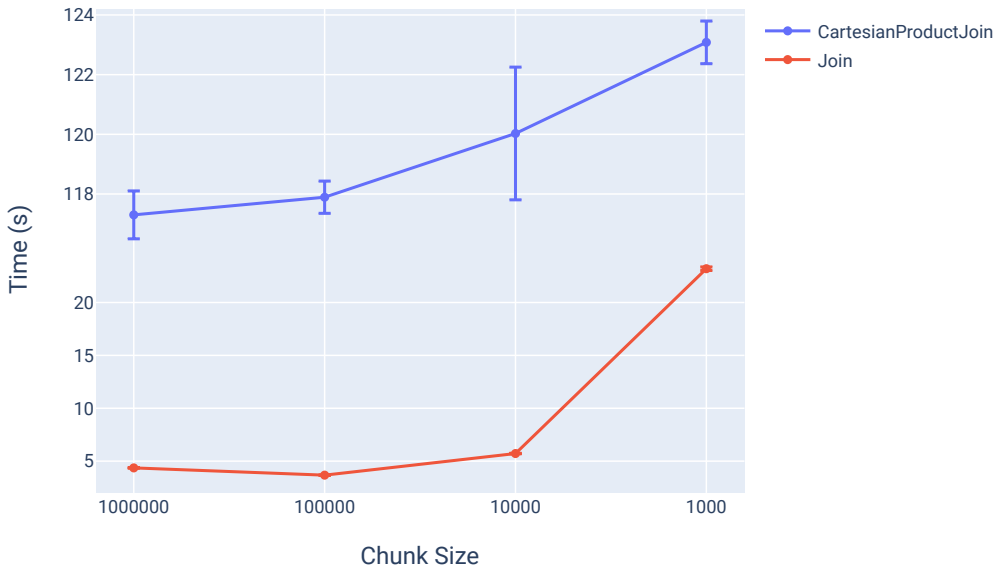


Figure 4: Real-world effects of different chunk sizes.

[28]The referenced patches only show the changes to specific other chunk sizes, so for all sizes that are not shown the values need to be changed accordingly.

As we can see the results do align with our synthetic example to a reasonable degree. We can clearly see that for `Join` the measurements for 1 million, 100 thousand, and 10 thousand are very similar, just like we observed in the synthetic benchmark. The fastest option of them all (100,000) is used by QLever for this operation. For `CartesianProductJoin` we see a similar picture but the standard error makes it clear that the processing time itself can vary a lot, regardless of chunk size configuration. For this operation, 1 million is used as the chunk size. In both cases, a chunk size of 1000 introduces a significant overhead making it clear a chunk size this low should be avoided.

7. Conclusion

In this thesis, we have shown that extending QLever's code to provide a general mechanism to evaluate results lazily provides huge benefits for memory consumption on a variety of queries. We also showed that in some cases this newly introduced mechanism can even provide drastic improvements to performance whenever small limits are present. For queries that do experience a statistically significant performance regression, we also showed that it is a reasonable tradeoff to reduce memory consumption. It should be clear that there is still room left for further improvement by enabling even more operations that have not been modified yet to make use of this newly introduced mechanism.

8. Future Work

8.1. Lazy Sorting

In Section 5.4.5 we mentioned that sorting is inherently non-lazy. We cannot produce a value until we have consumed the whole input and once everything is consumed there does not seem to be a reason to yield smaller slices of the data, because when everything is sorted the data is already present in its entirety.

One improvement that can be made here is to use the laziness mechanism to reduce the memory footprint. This can be done by sorting all input chunks individually when they come in and writing them to disk for later access. Once the input is fully consumed small parts of it can be read to memory and passed to subsequent operations. This way the operation will not be any faster (and almost certainly slower because reading and writing from disk is orders of magnitudes slower than from memory), but it will save a lot of memory.

8.2. Improving Query Planning

Query planning was not changed by the efforts of this work, as mentioned in Section 3.1, but it is likely some room for optimization is left on the table here. But this is far from being a trivial problem. As soon as memory constraints come into play, optimizing query plans turns into a multi-objective optimization problem. No longer does the query planner just have to find a sequence of operations that likely compute the correct result as fast as possible, now we also have to consider how much memory is at our disposal. It does not help that the amount of allocated memory is constantly changing, so the actual available memory has to be estimated conservatively which might leave a lot of performance on the table when memory stays unused even though it could have been used.

It remains an open question if a hybrid approach where a faster but memory-intensive operation is preferred over a slower, but memory-efficient plan and falling back to the alternative if not enough memory is available does provide any practical advantages. This would heavily depend on real-world scenarios and implementation details that might change in the future.

In case a small limit is present it would be highly desirable to prefer query plans that only chain operations together so that all of them can start producing partial results without having consumed the whole input. So if a limit is present we can stop the computation as soon as enough rows have been produced, saving a lot of CPU resources and time. This also increases the complexity of query planning.

8.3. Minimal Local Vocabularies

As indicated in Section 5.1.2 and explicitly mentioned in Section 5.3.2 the current implementation of the class for local vocabularies `LocalVocab` has limitations that become noticeably problematic with the introduction of lazy results. The local vocabulary just keeps a set of “vocabulary data” in memory without any metadata

where it is being used. Conceptually it is an extension of QLever's internal `ValueId` type, which stores everything in 64 bits as already explained in Section 5.1.2. But to trivially and efficiently write large amounts of data they need to be separated. Otherwise, the code would have to follow pointers into arbitrary memory whenever modifying instances of `IdTable` just to keep track of various reference counters which is really inefficient.

Once something is written into a `LocalVocab` object it is no longer associated with the actual `ValueId` that it supports. So when a subsequent operation removes an entire column from an `IdTable` or only keeps some rows from the result, we either have to keep all entries from the local vocabulary (which is the current behavior) or check every value individually if it is still required (which is unreasonably expensive to do). In other words, the current implementation trades a higher memory consumption for faster execution times.

This does not have to be the case though. Obviously, there is always some kind of tradeoff that has to be made, but simply having one instance of `LocalVocab` associated with a certain column of an `IdTable` would already allow to drop a lot of values whenever a whole column is dropped. This would allow for lower memory footprints for a range of queries, even if it does increase the memory footprint whenever multiple columns have lots of overlapping vocabulary data. The former case should be more common because the latter should only ever occur when a query is specifically crafted to have redundancy across multiple columns. Maybe there is even a more sophisticated approach out there that improves even more cases. In any case, this is something to think about in the future.

8.4. Avoiding Aggregation Tables when Unnecessary

As mentioned in Section 5.2.2 currently the `IndexScan` operation does not even try to build an aggregation table when it will inevitably exceed the memory threshold, because `IndexScan` does know exactly how many elements it will return. This mechanism could be extended to other operations as well, even if it means relying on heuristics to estimate the expected size. It remains unclear how much of a margin of error should be taken into account when making this decision. Regardless, changing this could speed up processing in several cases even if this means not caching some results where the heuristic is off by a lot.

8.5. Preventing Out-Of-Memory Issues for Lazy-Caching

Whenever a lazy result is consumed the values are copied over into an aggregation table to be able to store the whole result in the cache in the future (see Section 5.2.2 for details). Whenever this table becomes too big because no more memory is available or it has reached the maximum allowed threshold it is discarded to not terminate the computation for this silly reason. This works well enough for the most part, especially because the default threshold is rather small (5 MB). But if the memory requirements of a specific operation are just perfectly so that it would have succeeded without any aggregation tables in the memory, but failed because an

aggregation table consumed just enough memory to run out of it this is obviously undesirable. Just to be clear, it is unlikely that an operation, potentially requiring multiple gigabytes of memory will ever fail specifically because of this but ideally, it should be impossible. With fully materialized results this does never happen. Whenever an operation tries to allocate memory exceeding the configured memory limit QLever clears the cache first and checks again. Because the aggregation table is not stored in the cache during its creation it is not affected by this even though it should. By extending this mechanism it should be possible with reasonable effort to also clear any aggregation tables that might still be in the process of creation.

9. Acknowledgements

I want to thank Prof. Dr. Hannah Bast for her encouragement throughout this work. I want to thank Johannes Kalmbach for his continuous support, feedback, and constructive criticism throughout the practical implementation of this work, which ultimately made this work a better version of itself.

I want to thank my family and my friends for their ongoing support and encouragement over the years which enabled me to pursue my master's degree without any social or economic pressure.

Bibliography

[1] C. B. Aranda *et al.*, “SPARQL 1.1 Overview,” Mar. 2013.

[2] 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*, in CIKM '17. Singapore, Singapore: Association for Computing Machinery, 2017, pp. 647–656. doi: 10.1145/3132847.3132921.

[3] “Wikidata, a free, collaborative, multilingual, secondary knowledge base, collecting structured data to provide support for Wikipedia, Wikimedia Commons, the other wikis of the Wikimedia movement, and to anyone in the world..” Accessed: Dec. 08, 2024. [Online]. Available: https://www.wikidata.org/

[4] T. U. Consortium, “UniProt: the Universal Protein Knowledgebase in 2025,” *Nucleic Acids Research*, p. gkae1010, 2024, doi: 10.1093/nar/gkae1010.

[5] D. Vrgoč *et al.*, “MillenniumDB: An Open-Source Graph Database System,” *Data Intelligence*, vol. 5, no. 3, pp. 560–610, 2023, doi: 10.1162/dint_a_00229.

[6] Robin Textor-Falconi, “Efficient Export of SPARQL Query Results,” 2022. Accessed: Dec. 08, 2024. [Online]. Available: https://ad-publications.cs.uni-freiburg.de/theses/Bachelor_Robin_Textor-Falconi_2022.pdf

[7] O. Erling, “Virtuoso, a Hybrid RDBMS/Graph Column Store.,” *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.

[8] J. McCarthy, “History of LISP,” *SIGPLAN Not.*, vol. 13, no. 8, pp. 217–223, Aug. 1978, doi: 10.1145/960118.808387.

[9] Coco Li, “Analyze and Reorganize core Networking Structs to optimize cacheline consumption.” Accessed: Dec. 08, 2024. [Online]. Available: https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/

[10] H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle, “Efficient SPARQL Autocompletion via SPARQL.” 2021.

[11] Nick Göckel, “Efficient Keyword Search For The QLever SPARQL Engine,” 2024. Accessed: Dec. 08, 2024. [Online]. Available: https://ad-publications.cs.uni-freiburg.de/theses/Bachelor_Nick_G%C3%B6ckel_2024.pdf

Appendix

Section 5: Lazification

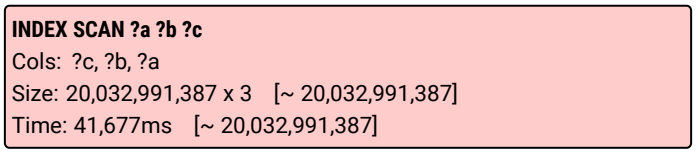


Figure 5: Execution Tree for the `IndexScan` example query

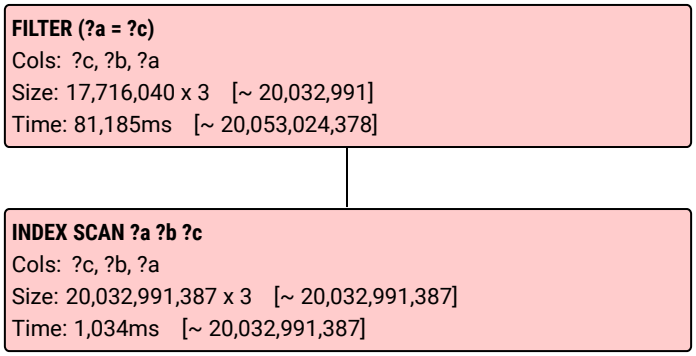


Figure 6: Execution Tree for the `Filter` example query

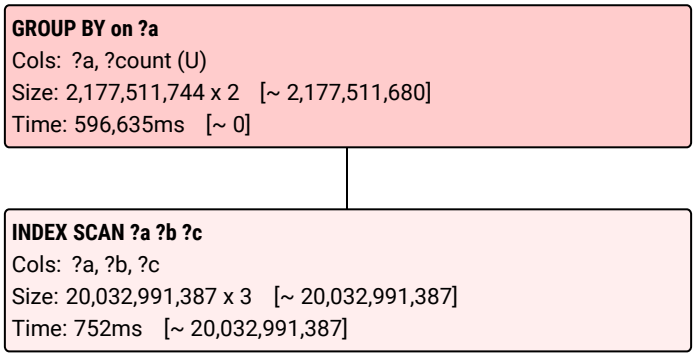


Figure 7: Execution Tree for the `GroupBy` example query

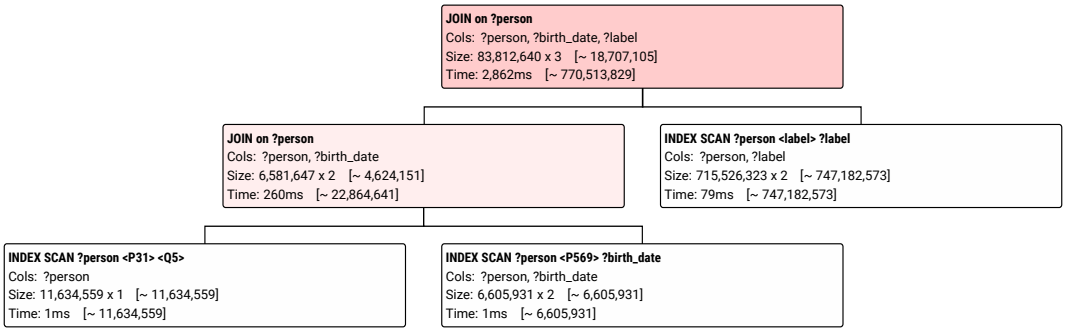


Figure 8: Execution Tree for the `Join` example query

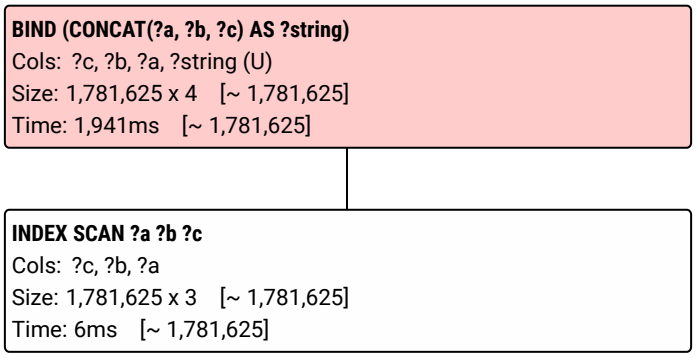


Figure 9: Execution Tree for the `Bind` example query

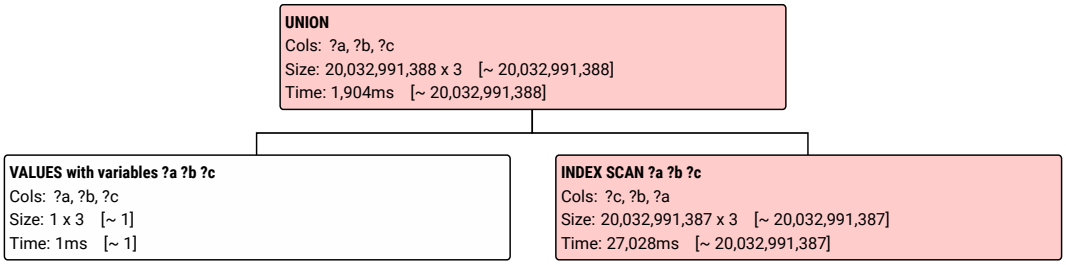


Figure 10: Execution Tree for the `Union` example query

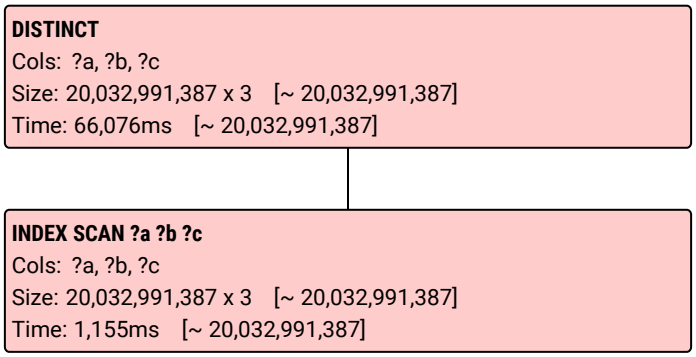


Figure 11: Execution Tree for the `Distinct` example query

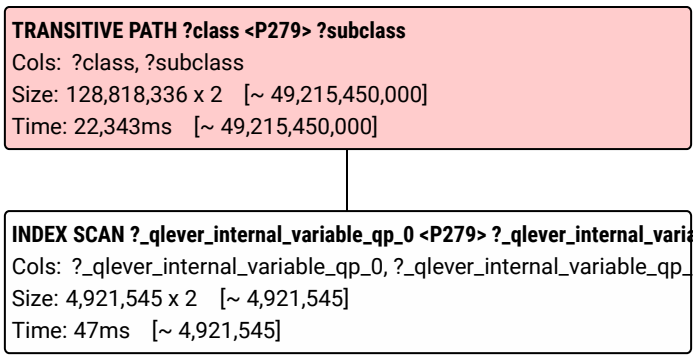


Figure 12: Execution Tree for the `TransitivePath` example query

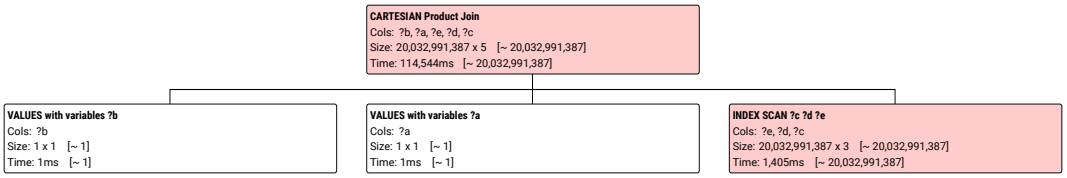


Figure 13: Execution Tree for the `CartesianProductJoin` example query

Section 6: Experiments

Query	Processing Time			Memory Delta		
	1	2	3	1	2	3
0	23597ms	18637ms	18898ms	702436KB	702448KB	702928KB
1	126627ms	129626ms	124991ms	702812KB	702676KB	702968KB
2	608942ms	608612ms	606451ms	702812KB	702804KB	702896KB
3	3749ms	3667ms	3729ms	702576KB	702840KB	702768KB
4	135250ms	135075ms	134275ms	1129792KB	1131180KB	1131640KB
5	38543ms	39217ms	39414ms	702708KB	702732KB	703060KB
6	82289ms	82222ms	82581ms	702476KB	702800KB	703120KB
7	13028ms	13534ms	13478ms	702644KB	703064KB	702920KB
8	109903ms	111920ms	111551ms	702536KB	702988KB	702920KB
9	387ms	384ms	386ms	702912KB	710600KB	702920KB
10	141ms	145ms	146ms	702804KB	702536KB	702668KB
11	2381ms	2329ms	2317ms	881880KB	805020KB	867932KB
12						

Table 4: Raw data for experiments Section 6.2 and Section 6.3

```
diff --git a/src/engine/Operation.cpp b/src/engine/Operation.cpp
index 3a25e752..c872064f 100644
--- a/src/engine/Operation.cpp
+++ b/src/engine/Operation.cpp
@@ -139,8 +139,7 @@ ProtoResult Operation::runComputation(const
ad_utility::Timer& timer,
   checkCancellation();
   runtimeInfo().status_ = RuntimeInformation::Status::inProgress;
   signalQueryUpdate();
-  ProtoResult result =
-      computeResult(computationMode == ComputationMode::LAZY_IF_SUPPORTED);
+  ProtoResult result = computeResult(false);
   AD_CONTRACT_CHECK(computationMode == ComputationMode::LAZY_IF_SUPPORTED
||
                     result.isFullyMaterialized());
```

Listing 3: Git patch to disable lazy results for direct comparison

Query	Processing Time			Memory Delta		
	1	2	3	1	2	3
0	OOM	OOM	OOM	OOM	OOM	OOM
1	OOM	OOM	OOM	OOM	OOM	OOM
2	OOM	OOM	OOM	OOM	OOM	OOM
3	4803ms	4774ms	4924ms	2150120KB	2153188KB	2151608KB
4	OOM	OOM	OOM	OOM	OOM	OOM
5	OOM	OOM	OOM	OOM	OOM	OOM
6	OOM	OOM	OOM	OOM	OOM	OOM
7	15471ms	15480ms	15391ms	2244908KB	2244224KB	2244472KB
8	OOM	OOM	OOM	OOM	OOM	OOM
9	OOM	OOM	OOM	OOM	OOM	OOM
10	133ms	116ms	117ms	702388KB	702832KB	702812KB
11	2245ms	2243ms	2179ms	982380KB	1014724KB	991012KB
12						

Table 5: Raw data for the second experiment of Section 6.2

Query	5MB			500MB			5000MB		
	1	2	3	1	2	3	1	2	3
0	135ms	138ms	142ms	138ms	133ms	140ms	135ms	139ms	144ms
1	125.97s	124.30s	120.02s	129.86s	129.77s	125.40s	132.17s	129.45s	126.75s
2	381ms	373ms	377ms	380ms	376ms	381ms	389ms	387ms	376ms
3	51ms	52ms	50ms	52ms	48ms	53ms	52ms	53ms	50ms
4	1430ms	1424ms	1457ms	1437ms	1435ms	1438ms	1420ms	1451ms	1450ms
5	140ms	134ms	140ms	138ms	138ms	136ms	138ms	139ms	141ms
6	103ms	111ms	107ms	107ms	111ms	108ms	110ms	109ms	110ms
7	516ms	506ms	535ms	521ms	520ms	524ms	539ms	549ms	533ms
8	139ms	149ms	141ms	138ms	145ms	141ms	143ms	141ms	145ms
9	2484ms	2410ms	2450ms	2522ms	2531ms	2485ms	2490ms	2483ms	2443ms
10									

Table 6: Raw data for the experiment of Section 6.3

```cpp
#include <vector>
#include <random>
#include <ranges>
#include <benchmark/benchmark.h>
#include "Generator.h"

constexpr size_t TOTAL_AMOUNT = 100'000'000;

cppcoro::generator<std::vector<uint32_t>> generateValues(size_t chunkSize) {
  if (TOTAL_AMOUNT % chunkSize != 0) {
    throw std::runtime_error{"Must be exactly divisible"};
  }
  size_t seed = std::random_device{}();
  for (size_t i = 0; i < TOTAL_AMOUNT / chunkSize; i++) {
    std::vector<uint32_t> elements;
    elements.resize(chunkSize);
    for (uint32_t& element : elements) {
        element = static_cast<uint32_t>(seed++);
    }
    co_yield elements;
  }
}

uint64_t sum(cppcoro::generator<std::vector<uint32_t>> generator) {
  uint64_t result = 0;
  for (uint32_t element : std::ranges::ref_view{generator} | std::views::join) {
    result += element;
  }
  return result;
}

static void SUM_VALUES(benchmark::State& state) {
  for (auto _ : state) {
    benchmark::DoNotOptimize(sum(generateValues(state.range(0))));
    benchmark::ClobberMemory();
  }
}

BENCHMARK(SUM_VALUES)->RangeMultiplier(10)->Range(1, TOTAL_AMOUNT / 10);
```

Listing 4: Code used to create a synthetic benchmark for chunk sizes.

```cpp
#include <random>
#include <ranges>
#include <benchmark/benchmark.h>
#include "Generator.h"

constexpr size_t TOTAL_AMOUNT = 100'000'000;

cppcoro::generator<uint32_t> generateValues() {
  size_t seed = std::random_device{}();
  for (size_t i = 0; i < TOTAL_AMOUNT; i++) {
    co_yield static_cast<uint32_t>(seed++);
  }
}

uint64_t sum(cppcoro::generator<uint32_t> generator) {
  uint64_t result = 0;
  for (uint32_t element : generator) {
    result += element;
  }
  return result;
}

static void SUM_VALUES_INLINE(benchmark::State& state) {
  for (auto _ : state) {
    benchmark::DoNotOptimize(sum(generateValues()));
    benchmark::ClobberMemory();
  }
}

BENCHMARK(SUM_VALUES_INLINE);
```

Listing 5: Code used to create a synthetic benchmark with inline values

`Generator.h` is a replacement for the `generator` header in versions before `C++23`. The version used by QLever and this benchmark can be found here: https://github.com/ad-freiburg/qlever/blob/4237e0d4af70e6e400f4357f61756eb5873fe98a/src/util/Generator.h

```
diff --git a/src/engine/CartesianProductJoin.h b/src/engine/
CartesianProductJoin.h
index 8c0a071c..3b572669 100644
--- a/src/engine/CartesianProductJoin.h
+++ b/src/engine/CartesianProductJoin.h
@@ -37,7 +37,7 @@ class CartesianProductJoin : public Operation {
   // custom `chunkSize` for chunking lazy results.
   explicit CartesianProductJoin(QueryExecutionContext* executionContext,
                                 Children children,
-                                size_t chunkSize = 1'000'000);
+                                size_t chunkSize = 100'000);

   /// get non-owning pointers to all the held subtrees to actually use the
   /// Execution Trees as trees
```

Listing 6: Git patch to change the chunk size of `CartesianProductJoin` from the default 1 million to 100 thousand

```
diff --git a/src/engine/Join.h b/src/engine/Join.h
index 8c8978c8..22e04afb 100644
--- a/src/engine/Join.h
+++ b/src/engine/Join.h
@@ -36,7 +36,7 @@ class Join : public Operation {

   using OptionalPermutation = std::optional<std::vector<ColumnIndex>>;

-  static constexpr size_t CHUNK_SIZE = 100'000;
+  static constexpr size_t CHUNK_SIZE = 1'000'000;

   virtual string getDescriptor() const override;
```

Listing 7: Git patch to change the chunk size of `Join` from the default 100 thousand to 1 million

Query		1000000	100000	10000	1000
`CartesianProductJoin`	1	118804ms	117425ms	114862ms	121655ms
	2	115451ms	117054ms	121113ms	122917ms
	3	117645ms	119197ms	124109ms	124674ms
`Join`	1	4300ms	3637ms	5733ms	23565ms
	2	4393ms	3682ms	5651ms	22858ms
	3	4390ms	3704ms	5769ms	23167ms

Table 7: Raw data for the experiment of Section 6.4