



Bachelor's Thesis

Efficient GroupBy for the QLever SPARQL engine

Benke Boldizsár Hargitai

December 1, 2025

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

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

Author Benke Boldizsár Hargitai,
Matriculation Number: 5370932

Editing Time September 1, 2025 - December 1, 2025

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

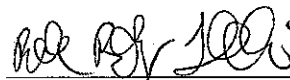
Supervisor M.Sc. Johannes Kalmbach,
Department of Computer Science
Chair for Algorithms and Data Structures

Declaration I hereby declare, that I am the sole author and composer of this Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Überlingen, 01.12.2025

Place, Date



Signature

Abstract

This bachelor's thesis describes methods developed to accelerate data grouping in the QLever SPARQL data management engine. This was accomplished by optimizing the QLever engine's **GROUP BY** algorithm. Calculating the **GROUP BY** operation requires one or more columns to be specified. The input data is then grouped along these columns. Currently, there are two strategies for this calculation:

1. Sorting the input data by the grouping columns and separating blocks of data where the values in the grouping column are the same.
2. Using a hash map.

Neither strategy has proven to be faster than the other on a consistent basis. When the hash map strategy was introduced, it initially appeared to be consistently better than the sorting strategy. However, it turned out to be slower when there were too many small groups. This common scenario occurs when the values in the grouping columns are mostly unique. This problem led to the development of a “hybrid” approach in this bachelor's thesis. The approach uses both of the previously mentioned grouping strategies. It begins with the hash map strategy. If the number of groups exceeds a certain threshold, the approach switches to sorting while extending the groups already in the hash map. Finally, the results from the two strategies are merged and sorted. Experimental evaluations based on RDF benchmark datasets show that this new approach leads to significant improvements in query time.

Zusammenfassung

In der vorliegenden Bachelorarbeit wurden Methoden entwickelt, die die Daten-gruppierung in der QLever-SPARQL-Datenverwaltungs-Engine beschleunigen. Dies konnte durch Optimierung des **GROUP BY** Algorithmusses in dem QLever-Engine erreicht werden. Um die **GROUP BY**-Operation zu berechnen, müssen eine oder mehrere Spalten angegeben werden. Die Eingabedaten werden dann entlang dieser Spalten gruppiert. Derzeit gibt es zwei Strategien in QLever für diese Berechnung:

1. Sortieren der Eingabedaten nach den Gruppierungsspalten und Trennen von Datenblöcken, bei denen die Werte in der Gruppierungsspalte identisch sind.
2. Verwendung einer Hash-Map.

Keine der beiden Strategien zeigte sich in allen Fällen schneller als die andere. Als die Hash-Map-Strategie eingeführt wurde, schien sie zunächst immer besser zu sein als die Sortierstrategie. Es stellte sich jedoch heraus, dass sie langsamer ist, wenn es zu viele kleine Gruppen gibt. Dieses häufige Szenario tritt auf, wenn die Werte in den Gruppierungsspalten größtenteils eindeutig sind. Dieses Problem führte zur Entwicklung eines "hybriden" Ansatzes in dieser Bachelorarbeit. Der Ansatz verwendet beide zuvor genannten Gruppierungsstrategien. Er beginnt mit der Hash-Map-Strategie. Wenn die Anzahl der Gruppen einen bestimmten Schwellenwert überschreitet, wechselt der Ansatz zum Sortieren, während die bereits in der Hash-Map vorhandenen Gruppen erweitert werden. Schließlich werden die Ergebnisse der beiden Strategien zusammengeführt und sortiert. Experimentelle Auswertungen anhand von RDF-Benchmark-Datensätzen zeigen, dass dieser neue Ansatz zu erheblichen Verbesserungen der Abfragezeit führt.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Thesis Objective and Contribution	2
2	Background	3
2.1	RDF Data	3
2.2	The SPARQL Query Language	5
2.3	Fundamentals of Grouping	8
2.4	Existing GROUP BY Implementations	9
2.4.1	Sort-Based Grouping	9
2.4.2	Hash-Based Grouping	9
2.5	Related Work in Database Systems	10
3	Approach	12
3.1	The Cinderella Analogy	12
3.2	Mapping the Terms of the Analogy	15
4	Implementation	17
4.1	The Hybrid Approach	17
4.1.1	Merging the Results	18
5	Experimental Setup	19
5.1	Hardware and Software Configuration	19
5.2	Datasets and Queries	20
5.2.1	Dataset Sizes	20
5.2.2	Olympics	20
5.2.3	DBLP	21
5.3	Benchmark	22
5.3.1	Linearly Increasing Number of Groups	23

5.3.2	Logarithmically Increasing Number of Groups	24
6	Results and Methodology	25
6.1	Testing with Queries	25
6.2	Benchmark Results	26
7	Discussion	30
7.1	Evaluation of the Results	30
8	Summary and Outlook	33
8.1	Conclusion	33
8.2	Directions for Future Research	33
9	Acknowledgments	35
	Bibliography	36
	Web Resources	37

Figure Index

2.1	RDF graph of legumes with type and color relations	4
3.1	The initial list of legumes with type and color.	13
3.2	Grouping the legumes by type and color.	13
3.3	Grouping the legumes by type and size.	15
6.1	Comparing sort-based and hash-map-based grouping strategies with 1.2 million entries	27
6.2	Comparing strategies: sorting, hash-map, hybrid with 1.2 million entries	27
6.3	Comparing strategies: sorting, hash-map, hybrid with 12 million entries	28
6.4	Comparing strategies: sorting, hash-map, hybrid with 1.2 million entries on a logarithmic scale	29
6.5	Comparing strategies: sorting, hash-map, hybrid with 12 million entries on a logarithmic scale	29

Table Index

2.1	Query results showing legume IDs	5
2.2	Query results showing legume entities and their labels	7
2.3	Grouped query results showing legume counts	8
5.1	Hardware configuration of the test machine	19
5.2	Software used for the experiment	19
5.3	Disk space requirements for building and indexing datasets	20
6.1	Average computation times of Query 5.2 in DBLP for each strategy	26
7.1	Slopes of fitted lines by dataset size, block configuration and grouping strategy	31

Code Index

2.1	Triples describing attributes of legumes	4
2.2	Simple SPARQL query example	5
2.3	SPARQL query example for joining	6
2.4	SPARQL query example for grouping	8
5.1	Query for testing correctness	21
5.2	Query for testing efficiency	21
5.3	Query similar to benchmark	23

1 Introduction

Being able to [query](#) existing datasets is essential for any kind of research. For example, Wikidata is a large dataset that contains millions of entries. Wikidata is used in libraries [\[11\]](#) and for research in life sciences [\[12\]](#) like biology and chemistry. The entries in Wikidata are in RDF-format (Resource Description Framework), which is a standard solution for representing structured data. This is similar to the tables (relations) of relational databases. A database can be queried to obtain the desired information. For relational databases, this is possible with the SQL [\[2\]](#) query language, while for RDF databases the [SPARQL](#) query language can be used. To actually compute queries, an engine is needed which compiles and runs the query. The [QLever SPARQL](#) engine [\[1\]](#) is such an engine. It is desirable for query engines to compute queries as quickly as possible. This thesis aims to improve the query computation time of [QLever](#), by optimizing one specific component of the engine: The [GROUP BY](#) operator.

The [GROUP BY](#) operator allows users to group query results by one or more of their attributes. For example grouping books in the library by their author.

[GROUP BY](#) is also needed for tasks such as counting, averaging, and summarizing data. This is called aggregation, e.g. if one wants to find out how many books each author has, the [COUNT](#) aggregate function can be used.

1.1 Motivation

Until now, two strategies have been used to compute the **GROUP BY** operation: (1) sorting entries then separating the groups, and (2) adding entries to a hash map. These strategies are described in detail in the theory section (2.4).

It was found that the hash map strategy is much faster than the sorting strategy, but only if the number of groups is relatively small. If there are many groups, performance deteriorates very quickly.

1.2 Thesis Objective and Contribution

As neither of the two existing strategies for computing **GROUP BY** were generally superior in computation time, a new “hybrid” approach was developed that combines these two strategies. This thesis improves the speed of the computation of the **GROUP BY** operation by showcasing the hybrid approach. How the hybrid approach works is first explained, followed by an analysis of the specific improvement.

2 Background

This section provides background knowledge to help understand the concepts used in the thesis and also discusses existing implementations of the [GROUP BY](#) operator.

2.1 RDF Data

RDF is an abbreviation for “Resource Description Framework”. Here, the word “resource” is synonymous with “data entry” (which could be a number, a text string, a boolean value, etc.). With these resources, statements can be made in the form of **subject - predicate - object** triples. Such a triple is equivalent to the statement: **this - relates to - that**. Triples are written in turtle format, which consists of the subject, the predicate, the object and a dot at the end, separated by spaces, as seen in [Listing 2.1](#). It is apparent that triples *describe* the relation between the resources, hence the name *Resource Description Framework*.

In summary, RDF can be defined as describing information resources arranged in triples. For example, it can describe attributes of different kinds of legumes. [Code 2.1](#) describes six legumes: a green pea, two brown lentils, a yellow lentil, a yellow pea and a red lentil. It does this by first stating the ID of the specific legume, the subject. Then it states which *kind* of attribute of the legume we are describing in that triple (type or color); this is the predicate. Finally it states which specific attribute we associate with the legume. Thus, each triple is a statement equivalent either to “this legume is of type X” or to “this legume has color Y”.

Listing 2.1: Triples describing attributes of legumes

```

<legume-1> <type> <pea> .
<legume-1> <color> <green> .
<legume-2> <type> <lentil> .
<legume-2> <color> <brown> .
<legume-3> <type> <lentil> .
<legume-3> <color> <yellow> .
<legume-4> <type> <lentil> .
<legume-4> <color> <brown> .
<legume-5> <type> <pea> .
<legume-5> <color> <yellow> .
<legume-6> <type> <lentil> .
<legume-6> <color> <red> .

```

A collection of triples, such as the attributes of the legumes, is very similar to a directed graph. We can interpret the **subject** and the **object** of a triple as nodes and the **predicate** as the edge between these two nodes. This is why RDF is also known as a Knowledge Graph. Figure 2.1 illustrates how the example above can be represented as a graph.

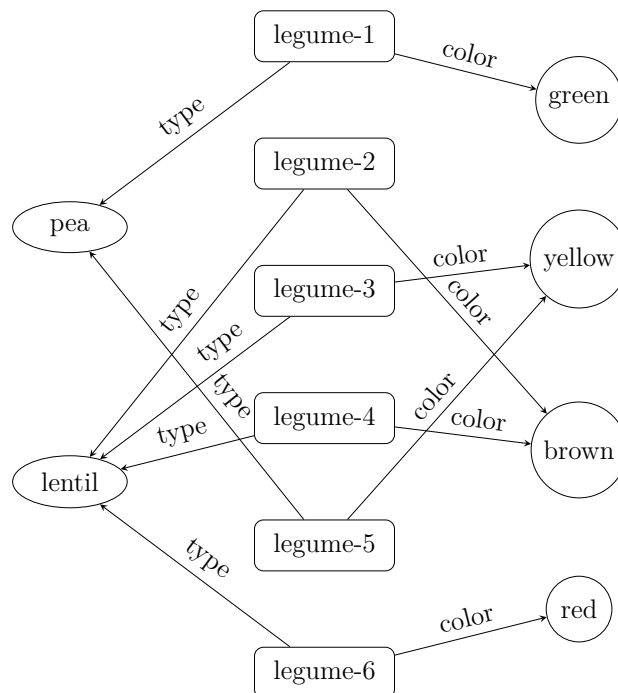


Figure 2.1: RDF graph of legumes with type and color relations

An RDF-dataset is conceptually similar to an SQL table with three columns. In that sense a triple is similar to a row in an SQL table and the rows of a table are often called entries. Therefore, in this thesis the terms “triple”, “row” and “entry” are used interchangeably.

2.2 The SPARQL Query Language

The QLever engine uses the “SPARQL Protocol and RDF Query Language”, which is abbreviated as **SPARQL**. It is a declarative query language for RDF data. The syntax of **SPARQL** is similar to SQL’s syntax. The difference between the two languages is apparent when we look inside of the WHERE clause. To demonstrate what a **SPARQL** query looks like, we can look at a simple example which lists types of legumes: Query 2.2. Note the turtle format inside the WHERE clause. There is also new syntax present. Words beginning with a question mark (e.g. `?legume_id`) are variables. Words that have a prefix and are separated from them by a colon, are resources (e.g. `wdt:P279`).

Query 2.2: Simple SPARQL query example

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?legume_id WHERE {
  ?legume_id wdt:P279 wd:Q145909 .
}
```

Let us decipher the statement in the WHERE clause. “P279” is a predicate and its meaning is “subclass of”. “Q145909” is the ID for the Wikidata item called “legume”. Therefore, this query looks for triples which are of the form **something - is a subclass of - legume**. When all triples of this form have been found, the engine shows the values after the SELECT clause in the result. In this case, this is the ID of the legume types.

	?legume_id
1	Q114832486
2	Q13138734
3	Q2727662
4	Q2987371
5	Q60963647

Table 2.1: Query results showing legume IDs

The result can be seen in Table 2.1. This result is not yet very informative. The table consists of a single column of identifiers, which cannot be understood with natural language. To understand them intuitively, the names of these legume types can be queried as well. To achieve this, a second statement is added, as shown in Query 2.3.

Query 2.3: SPARQL query example for joining

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?legume_id ?legume_name WHERE {
  ?legume_id wdt:P279 wd:Q145909 .
  ?legume_id rdfs:label ?legume_name .
}
```

In Query 2.3, there is a new resource: the `rdfs:label` predicate. This predicate associates a resource with its human-readable name. All triples which get selected by this query have to fulfill both statements in the `WHERE` clause. In other words, the legume ID has to be a subclass of legume *and* it has to have a label. After finding all triples that satisfy both statements, both the legume ID and the legume name are selected, as specified after the `SELECT` clause.

The result of executing this query is shown in Table 2.2. Here, each legume ID is associated with its name(s) in different languages.

	?legume	?legume_name
1	Q114832486	Abu
2	Q114832486	(Arabic text)
3	Q13138734	wâldbeantsje
4	Q2727662	cereal legume
5	Q2727662	civaie
6	Q2727662	getrocknete Hülsenfrüchte
:	:	:
45	Q2987371	green bean
:	:	:
51	Q2987371	grüne Bohne
:	:	:
70	Q2987371	zöldbab
:	:	:
88	Q60963647	legumes as feed
89	Q60963647	légumineuse fourragère
90	Q60963647	leguminose foraggere

Table 2.2: Query results showing legume entities and their labels

In the result shown in Table 2.2, there are already two columns. The first column shows the legume ID, while the second column shows the legume name. 90 rows are listed in total. All five types of legumes are represented in this result, each with their respective names in different languages.

The “Abu” legume (Q114832486) has two different labels, one in English and one in Arabic. The green bean (Q2987661) has seemingly the most labels. To find out in exactly how many languages each legume type is represented, the results can be grouped by legume ID and the number of labels can be counted for each group. This is done with the **GROUP BY** clause, as shown in Query 2.4 in Section 2.3.

2.3 Fundamentals of Grouping

A group, generally speaking, is a set of elements defined by shared criteria. In the specific context of this thesis, a group is defined as a set of rows that share the same value(s) in one or more specified columns. In the legume example, a group could consist of all rows where the legume ID is the same. For example, all rows with the legume ID “Q2987371” (green bean) would form a group.

To form groups, the `GROUP BY` operator is used, as shown in Query 2.4.

Query 2.4: SPARQL query example for grouping

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?legume_id (COUNT(?legume_name) as ?count_names) WHERE {
  ?legume_id wdt:P279 wd:Q145909 .
  ?legume_id rdfs:label ?legume_name .
}
GROUP BY ?legume_id
```

Here, the rows are grouped by the `?legume_id` column. For each group, the number of entries in the `?legume_name` column is counted with the `COUNT` aggregation function. The result of executing this query is shown in Table 2.3.

	?legume	?count
1	Q114832486	2
2	Q13138734	1
3	Q2727662	21
4	Q2987371	63
5	Q60963647	3

Table 2.3: Grouped query results showing legume counts

As expected, there are five groups in total, one for each legume type. The `COUNT` aggregation function shows how many names each legume type has. For example, the name of the green bean (Q2987371) is registered in 63 different languages.

To compute the **GROUP BY** operation, the **QLever** engine uses two main strategies: sorting-based grouping and hash-based grouping. These strategies are explained in detail in Section 2.4.

2.4 Existing GROUP BY Implementations

To understand how the two strategies relate to each other, it is necessary to know how they work.

2.4.1 Sort-Based Grouping

The sorting strategy starts by sorting all input entries by the grouping columns. Once this is accomplished, entries with identical values in the grouping columns are adjacent. This enables the group separation to be as straightforward as iterating through input entries. An entry belongs to a new group if any of its grouping-column values differ from those of the previous entry.

If the set of input entries is too large, sorting the entries cannot be performed in the RAM. In this case, the external storage can be utilized, as explained in Section 8.2 about the Directions for Future Research.

2.4.2 Hash-Based Grouping

The hash map strategy involves combining the grouping-column values of a specific entry into one “hash”. This is typically done by applying a hash function that converts the combination of values into an integer. The hash map then allocates buckets in memory for entries that produce the same hash value. Thus, if the hash is identical for any two entries, they are stored in the same bucket and can be grouped together.

In contrast to the sorting strategy, computation time for the hash map strategy is proportional to the number of groups. While the computation time of the hash map strategy also depends on the number of entries it gets, it is much faster if there are only a few groups.

In scenarios like this, when the number of groups is large, the hash map strategy becomes much slower, because each distinct group requires memory allocation. This allocation is required for the aggregation state, where intermediate results (e.g., counts, sums) are stored for each group.

As the number of groups rises, the hash map must be constantly resized. This is similar to the staging problem in SPARQL query engines, where operators such as `GROUP BY` must materialize all intermediate data before starting to produce the output. If intermediate result sets are large, staging has a negative effect on performance [5].

When there are only a few groups, the hash map is highly efficient both in terms of time and memory. This is because only a few rows will allocate a new aggregation state, while most of the rows update existing aggregation states.

2.5 Related Work in Database Systems

There are existing RDF-based database management systems that also support `GROUP BY`.

The database management system “DuckDB” uses an algorithm [7] for grouping, which can serve as a reference for us. DuckDB uses an approach which is very similar to the approach presented in this thesis.

The Proceedings of the VLDB Endowment published the paper “Saving Private Hash Join” [6]. Although the concern of the paper is not grouping itself, but rather the join operation, it shares similar ideas about using the hash map. In this paper they also talk about reducing memory consumption of hash maps. They refer to the problem of high memory consumption when there are many groups as the “performance cliff”.

They also work with a hybrid approach, which they call the Hybrid Hash Join (HHJ). This consists of switching between different algorithms if the memory reaches a limit. This is similar to the fallback mechanism described in this thesis (switching between the sorting and hash-map strategies). They emphasize that they do so gracefully, which this thesis also does. Graceful degradation is defined as “the ability of a computer, machine, electronic system or network to maintain limited functionality even when a large portion of it has been destroyed or rendered inoperative” by TechTarget [10].

They also discuss “spilling” data to external storage when the memory limit is reached. This is similar to external sorting, which this thesis only mentions as a potential future direction in Section 8.2.

3 Approach

The hybrid approach is described in this chapter. It is highly recommended for everyone to read the analogy presented in the analogy section (3.1) to get a clear and intuitive understanding. References to this analogy are also made in later sections. At the end of this chapter, the connection between the analogy and the technical implementation is examined.

3.1 The Cinderella Analogy

To illustrate the problem, put yourself in the shoes of Cinderella from the Grimm Brothers' fairy tale [4].

She wanted to go to a ball with her stepmother and two stepsisters, where the prince would choose his bride. To prevent this, the stepmother poured lentils into the ashes where Cinderella used to sleep and ordered her to pick the lentils out of the ashes in just two hours. When she succeeded with the help of the pigeons and other birds she had summoned, her jealous stepmother repeated this cruel game and reduced the time to one hour, only to ultimately refuse to allow her to attend the ball. As is usual in fairy tales, the story has a happy ending, and Cinderella, who manages to sneak into the ball with the help of her animal friends, wins the prince's heart.

Let us now assume that, as her stepsisters had often done before, the stepmother had instructed Cinderella to collect 12 million peas and lentils from the ashes, separate them, and divide them into different groups according to color and size – all in under 15 minutes. How would she have managed that? At the time of the Grimm Brothers, this would have been an impossible task in the time allowed. Or perhaps not?

The hypothetical Cinderella develops a strategy. First, with the help of the birds, she collects the legumes from the ashes and separates them into peas and lentils. However, the subsequent sorting is more difficult. She takes a long sheet of paper and lays out the legumes in a single column.

Then she writes the type and color of each legume next to it. This gives her a list of the legumes and their descriptions, as shown in Figure 3.1.

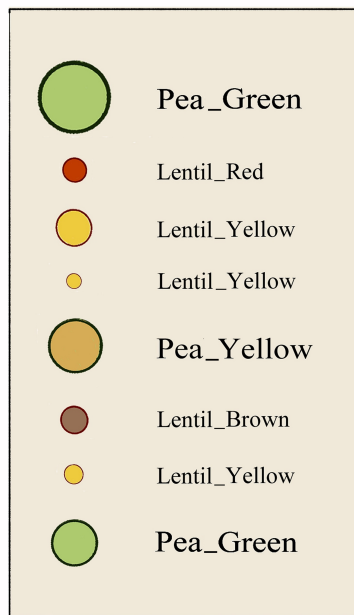


Figure 3.1

The initial list of legumes with type and color.

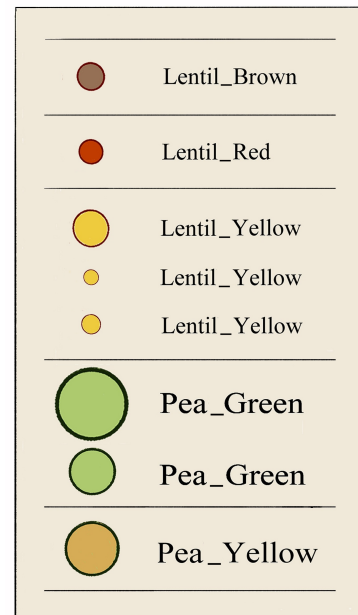


Figure 3.2

Grouping the legumes by type and color.

Then she takes another long sheet of paper, arranges the legumes in alphabetical order of description and draws a horizontal line when the description changes (Figure 3.2).

After ten minutes, Cinderella is finished. She shows this sorted and grouped collection of legumes to her stepmother, but of course she is not satisfied with it and pours all the legumes back together again, saying, “Only if you can group them by type and color in less than five minutes will you be allowed to come to the ball.”

Cinderella knows she must try a different approach. Her previous method of writing things down and sorting them is not fast enough. She remembers a large chest of drawers with many drawers and suddenly has an idea. “What if I put each group in a different drawer?” This time, she makes a list of the type and color of the legumes and gives each variety a name, e.g. ‘Pea_Green’ for green peas, ‘Lentil_Red’ for red lentils and ‘Lentil_Yellow’ for yellow lentils and then sorts them separately into a drawer with the same label. This is a great idea, because the hypothetical Cinderella finishes in less than two minutes with the new method.

However, when she shows her stepmother the result, her stepmother is still not satisfied, throws all the legumes back into a bowl and says, “You must group everything by type and size (in micrometers), not by type and color. If you can do that in less than three minutes, you can come with us.” Cinderella is confident, not only because she has the magical ability to instantly determine the size of the legumes in micrometers, but also because the drawer method is much faster than the sorting method. However, to her surprise, when she looks at the clock after completing the task, she sees that six minutes have passed since she started. “That can’t be right,” she thinks, “What took the most time?”

“Well, this time I needed a lot of drawers,” she thinks, “because there are so many unique combinations: Lentil_5374, Pea_6350, Pea_6518 ...(see Figure 3.3), I can hardly keep track of a million drawers. What if, once I have too many drawers, I stop using the chest of drawers and finish the rest using the sorting method? Actually, when I find another ‘Lentil_5374’ or one of the other legumes for which I have already labelled a drawer, I can simply put it in the appropriate drawer. That’s easy. But for the rest, I’ll use the sorting method on paper. In the end, I’ll take each group of legumes out of their drawer and put them on the paper. Then everything will be sorted.”

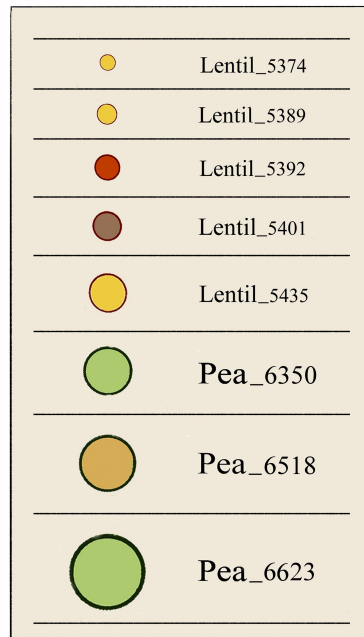


Figure 3.3
Grouping the legumes by type and size.

When she does this, the result is immediately visible. She finishes the job in less than two minutes. Proudly, she goes to her stepmother and shows her the result. “Cinderella,” her stepmother might say, “you’ve done a good job. Now you can come to the ball with us.”

3.2 Mapping the Terms of the Analogy

In the previous section, we saw how Cinderella was able to group the legumes efficiently by combining two different strategies. Now let us map the elements of the analogy to the technical terms used in this thesis.

Cinderella	→	The GROUP BY operator
legumes	→	input triples / entries / rows to be grouped
type, color, size of legumes	→	attributes of data, the value(s) by which we group
chest of drawers	→	hash map
drawers	→	groups in the hash map
paper with sorted legumes	→	IdTable to be sorted
sorting the legumes on paper	→	sorting-based grouping strategy
putting legumes in drawers	→	hash-map-based grouping strategy
“Too many drawers”	→	the group threshold

There is still one element left to explain, which was omitted from the analogy to make it more compact and understandable.

Imagine, that Cinderella receives the legumes in several bags from her stepmother, instead of all at once. She opens one bag, processes the legumes inside it, then opens the next bag and so on. She would be more efficient this way, because she can rest a little in between. Although it seems like an extraordinarily kind act, her stepmother just wants Cinderella to finish quickly.

Similarly, in **QLever**, the data is processed in multiple steps, instead of processing all of the input data at once. The unit of data being processed in one step is called a block. This way the memory is used more efficiently, often resulting in a faster query execution time. This is because the processed block of data stays small enough to fit in CPU cache.

Sometimes Cinderella’s stepmother only gives her a single large bag of legumes. Now, Cinderella is performance-oriented as well and if she gets a single large bag, she still takes breaks in between. This is especially important, because she can only check the number of drawers she has used while she is taking a break. So if she didn’t take breaks and she would only check at the end whether or not she had used too many drawers, she would process the whole bag inefficiently.

Similarly, in **QLever**, if the whole amount of entries comes in one large block, it is still broken down to smaller blocks. Then, the number of groups is examined after processing each smaller block. This is explained thoroughly in Section 4.1 about the hybrid approach.

4 Implementation

This chapter describes how the implemented algorithm of the hybrid approach works.

4.1 The Hybrid Approach

The hybrid approach is an algorithm for computing the `GROUP BY` operation. The input rows are initially processed using the hash map. Each row is assigned a bucket in the memory. These buckets are already the desired groups. If there are only a few groups, the processing is very efficient because of the reduced RAM usage, so the whole table gets processed with the hash map. However, if there are many groups, the hash map strategy is inefficient because of the higher RAM usage. After processing a block of up to 262,144 rows, the number of groups is counted. If this number exceeds a certain threshold, no new groups are added to the hash map. If the currently processed row belongs to a group that is already in the hash map, then the row is added to that group in the hash map. Otherwise, the row is appended to an `IdTable`. After all rows have been processed, groups are generated from the hash map and the `IdTable`. From the hash map the buckets can directly be exported to a list of groups. The `IdTable` is sorted, then the groups are separated and again the result is a list of groups. The two resulting lists of groups are then merged.

The threshold mentioned earlier is a number which the user can set and it has a default value. The evaluation section (7.1) discusses how this default value is chosen. The switchover to the hybrid approach occurs if the number of groups in the hash map exceeds this threshold.

As described in Section 3.2, data is often organized in blocks. If the currently processed block is too large, the algorithm of the `GROUP BY` operator breaks it into smaller blocks of size `GROUP_BY_HASH_MAP_BLOCK_SIZE`, which is currently defined as 262,144. This makes it possible to check the number of groups after processing each smaller block of 262,144 entries as well and exit the hash map strategy earlier.

4.1.1 Merging the Results

The rest table contains the values that are not stored in the hash map; consequently, the two partitions are disjoint: each group resides entirely in either the hash map or the table. The hash map yields a set of already aggregated groups. Sorting the `IdTable` produces the remaining groups. These two group lists are then concatenated.

Finally, the combined list of groups is sorted again to ensure the correct order of groups in the final output. This is a requirement of the `QLever` engine, as it expects the output of the `GROUP BY` operator to be sorted by group keys.

5 Experimental Setup

This chapter specifies the hardware and software configuration and explains how they were used for testing and benchmarking the proposed optimizations.

5.1 Hardware and Software Configuration

The programming, testing and benchmarking was all done on a remote computer using SSH remote connection and the “Remote Explorer” Visual Studio Code plugin.

The hardware specifications of the remote computer are detailed in Table 5.1.

Component	Specification
CPU	AMD Ryzen 7 3700X 8-Core Processor
RAM	133 GiB (125 GiB + 8 GiB swap)
Storage	2 x Samsung SSD 970 EVO Plus 2TB (4TB total)

Table 5.1: Hardware configuration of the test machine

The software environment used for the tests and benchmarks is detailed in Table 5.2.

OS	Ubuntu 22.04.5 LTS
Compiler (C++)	11.4.0
Boost	1.81.0
CMake, CTest	3.31.2

Table 5.2: Software used for the experiment

5.2 Datasets and Queries

Two datasets were used for testing the correctness and speed of the hybrid approach: Olympics [13] and DBLP [3]. Both of these were default datasets available for qlever. These datasets are easy to download, index and use, thanks to the qlever-control python repository [8].

5.2.1 Dataset Sizes

Table 5.3 shows how much space was needed to build and test the project on the remote computer. This is relevant if one would like to locally build and test their own copy of the project.

Folder / Dataset	Disk Space
DEBUG build	48.6 GB
RELEASE build	2.78 GB
Olympics database index + raw data	376.3 MB
DBLP database index + raw data	33.8 GB

Table 5.3: Disk space requirements for building and indexing datasets

5.2.2 Olympics

The Olympics dataset contains information on Olympic athletes, events, and competition results. It is a relatively small dataset (1,781,625 triples), making it suitable for initial testing and debugging of the hybrid approach. This dataset was only used to test correctness, not to measure efficiency. In other words it was used to verify that the hybrid approach produces the same results as the previous sorting-based and hash-map-based implementations. Query 5.1 was used for this purpose.

Query 5.1: Query for testing correctness

```

SELECT ?g (COUNT(?g) AS ?count) WHERE {
  GRAPH ?g { ?s ?p ?o }
}
GROUP BY ?g
ORDER BY DESC(?count)

```

This query is a built-in example on the [QLever](#) interface. The result of this query was only one row with a single count number, describing the number of named graphs in the dataset. First, the query was executed on the remote computer and the count number was recorded. On occasion, this number was different from the result of the query when executed on the official [QLever](#) website [9], which served as an indication that there was a bug in the implementation.

5.2.3 DBLP

The DBLP dataset is a knowledge graph that contains bibliographic information on computer science publications, extended with citation data from OpenCitations [3]. It is quite large (1,454,495,937 triples), which allows us to test the efficiency of the hybrid approach. The testing was conducted with Query 5.2.

Query 5.2: Query for testing efficiency

```

PREFIX dblp: <https://dblp.org/rdf/schema#>
SELECT ?title ?year (COUNT(?entity) AS ?count)
WHERE {
  ?entity dblp:title ?title .
  ?entity dblp:yearOfPublication ?year .
}
GROUP BY ?title ?year

```

This query groups the DBLP dataset by publication title and year, counting the number of entities (publications) for each title-year pair. The result is a list of titles with their respective publication years and the number of publications for

each title-year pair. The query feeds the **GROUP BY** operator with around 8 million triples. These

The end there are around 7.9 million groups. Almost exactly as many, as the total number of triples. This is perfect to test our main problematic scenario (which Cinderella was faced with when she grouped by type and size with drawers).

5.3 Benchmark

To test the efficiency of the hybrid approach, a benchmark was created. This measured the runtime of the **GROUP BY** operation with different strategies and configurations.

As a reminder, there are three strategies to benchmark: sorting, hash map, and the hybrid approach. The five main goals were:

1. to show that the critical scenario, in which there are many small groups, *the hash map strategy is significantly slower* than the sorting strategy;
2. to determine, *which group threshold would be best* for the **fallback**, so that the runtime of the hybrid approach is optimal;
3. to *examine how the hybrid approach performs* compared to the other two strategies;
4. to *examine the best and worst case scenarios* for the hybrid approach; and
5. to verify that the differences in runtime between the three strategies are *independent* of the number of entries and the number of blocks.

Therefore the benchmark had to vary three main parameters: the number of groups, the number of blocks and the number of entries. For the best and worst case scenarios, the distribution of group sizes also had to be varied.

To prove that the difference in runtime between the three strategies is independent of the number of entries, each strategy was executed with two different input sizes: 1.2 million and 12 million entries.

To prove that the difference in runtime between the three strategies is independent of the number of blocks, each strategy was executed twice: once with a single block of entries and once with 13 blocks.

The number of groups was varied with two methods: linearly and logarithmically increasing number of groups. In both measurements the groups were evenly distributed. This means that each group had exactly the same number of entries.

5.3.1 Linearly Increasing Number of Groups

First, to be able to plot the runtime linearly, the number of groups was incremented through 30 steps, measuring the runtime of the strategies 30 times. For example, with 1.2 million entries, the number of groups ranged from 40,000 to 1,200,000 in increments of 40,000. This was achieved by setting the values in the first column of the input data accordingly and then grouping by the first column. The details of the input data generation are explained below. Benchmarking the first measurement of this case with 40000 groups is roughly equivalent to executing the SPARQL Query 5.3.

Query 5.3: Query similar to benchmark

```
SELECT ?a (COUNT(?b) AS ?x) WHERE {
  VALUES (?a ?b) {
    (0 517) (1 12) (2 834) (3 77) (4 905) ... (39997 261)
    (39998 443) (39999 88) (0 1000) (1 3) (2 672) (3 541)
    (4 224) ... (39997 715) (39998 90) (39999 333) (0 987)
    (1 456) (2 12) (3 777) ...
  }
}
GROUP BY ?a
```

Each parenthesis in the values of Query 5.3 represents one row of the input to the `GROUP BY` operator. Each row has the format '(first_column second_column)'. We group by the values in the first column and count the number of occurrences of each group by counting the values in the second column.

The numbers in the first column are generated using the following mathematical formula: $i \bmod \text{number_of_groups}$, so if there are N groups, then the first column contains integers from 0 to $N-1$. In Query 5.3 the first column is represented by variable ‘?a’ and the number of groups is 40000.

In the second column, seeded semi-random values are generated. This means that, by providing a seed value, the same random values are generated every time the benchmark is executed. This is important to ensure the reproducibility of the benchmark results. The number of entries in each group was counted by the second column with the COUNT aggregate function.

5.3.2 Logarithmically Increasing Number of Groups

Often in real-world scenarios, the number of groups is much smaller than the number of entries. For example, when grouping people by their country of residence, there are only around 200 countries in the world, but billions of people.

To examine the performance difference between the three strategies in cases where the number of groups is small, the benchmark was repeated with logarithmically increasing number of groups. This time, the number of groups ranged from 1 to the number of entries in multiplicative increments of 1.5, measuring the runtime $\lceil \log_{1.5}(\text{number_of_groups}) \rceil$ times. The last measurement was always with the maximum number of groups, equal to the number of entries. For example, with 1.2 million entries, the number of groups started at 1, then $\lceil 1.5 \rceil = 2$, then $\lceil 2 \cdot 1.5 \rceil = 3$, then $\lceil 3 \cdot 1.5 \rceil = 5$, and so on, until it reached 1,200,000. This way, more measurements were taken in the lower range of group counts, where the hash map strategy is expected to perform better.

6 Results and Methodology

This section presents the results obtained through the specified queries and benchmarks, as well as the methods used to achieve them. To test the correctness and efficiency of the hybrid approach, the queries mentioned in Chapter 5 were used. To fully evaluate the runtime of the hybrid approach, however, a dedicated benchmark was created. First, let us look at the testing with queries.

6.1 Testing with Queries

Query 5.1 for the Olympics dataset was used in `DEBUG` mode, because the `DEBUG` logs helped to track issues with the implementation. A good example of this was when the algorithm processed a block of entries, and then switched to the hybrid approach (as explained in Chapter 4) which then processed the same block of entries again, before moving to the next one. This resulted in more groups than in the result of the query on the official [QLever](#) website. This made it possible to find the issue and fix it.

Query 5.2 for the DBLP dataset was used in `RELEASE` mode, as only the runtime of the hybrid approach built in `RELEASE` mode is relevant for the end results. The query was executed with different configurations, similar to the benchmark configurations described at the end of Section 5.3. This query fed the `GROUP BY` operator with blocks of approximately 125,000 entries each. Almost all of the entries belonged to different groups, so the number of groups quickly exceeded the threshold for switching to the hybrid approach.

Two threshold values were used for the hybrid approach to check how the threshold influences the runtime. The first group threshold was set to 100,000 entries, so that the `fallback` occurs immediately after processing the first block with the

hash map. The second value for group threshold was 1,000,000 entries, so that a few blocks are processed with the hash map before the [fallback](#) occurs.

Executing the query five times for each strategy and manually noting the computation time of the `GROUP BY` operator was enough to prove the concept. The results were calculated by taking the average of these five measurements for each strategy. The results are shown in Table 6.1.

Strategy	Runtime (s)
Sort-Only	2.932
Hash-Only	8.696
Hybrid (thr: 100,000)	4.173
Hybrid (thr: 1,000,000)	5.122

Table 6.1: Average computation times of Query 5.2 in DBLP for each strategy

The results show that the hybrid approach is significantly faster (about twice as fast) than the hash map strategy in this critical scenario. Also, the hybrid approach is slower than the sorting strategy. This is expected, as the sorting strategy has no issues with memory consumption in the critical scenario. The hybrid approach has overhead because it must first process a number of entries with the hash map before switching to sorting.

The results show a approximately 1 second difference for the two thresholds. From these two cases, the best performance is achieved with a threshold of 100,000 groups. This makes sense because in this case the [fallback](#) occurs earlier, so less entries are processed with the hash map.

6.2 Benchmark Results

A number of measurements were taken to compare the three strategies. In the first measurement (Figure 6.1), the number of entries is constant (1.2 million) and the number of groups is increasing linearly, as described in Section 5.3.1. Both strategies were measured twice: once with a single block of entries and then with 13 blocks. Latter case is indicated in the diagrams with “(Blocks)”.

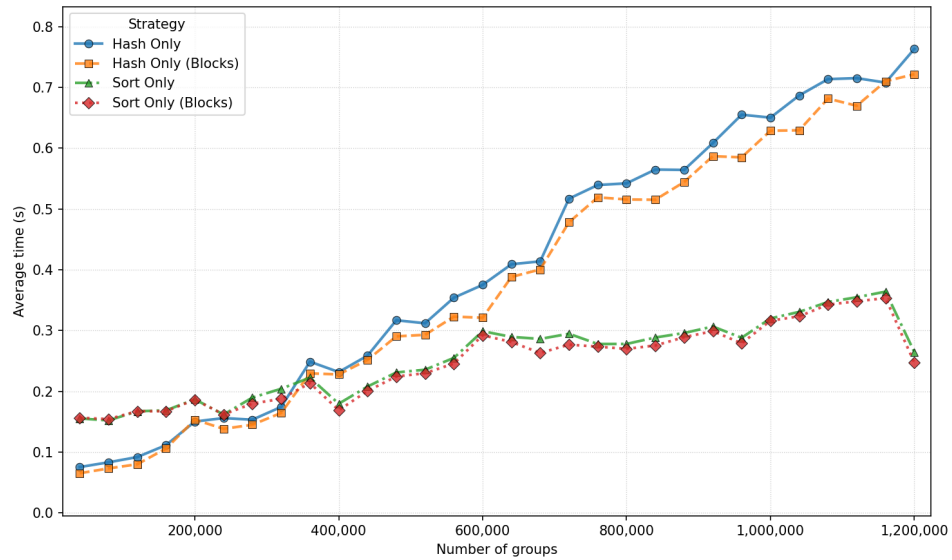


Figure 6.1: Comparing sort-based and hash-map-based grouping strategies with 1.2 million entries

Then, drawing the conclusion from Figure 6.1, the group threshold for switching strategies was set to 350,000 groups. With this threshold, a new measurement was conducted. This time the hybrid approach was also measured alongside the two previous methods. As in the previous measurement, all strategies were measured both in a single block and in multiple blocks.

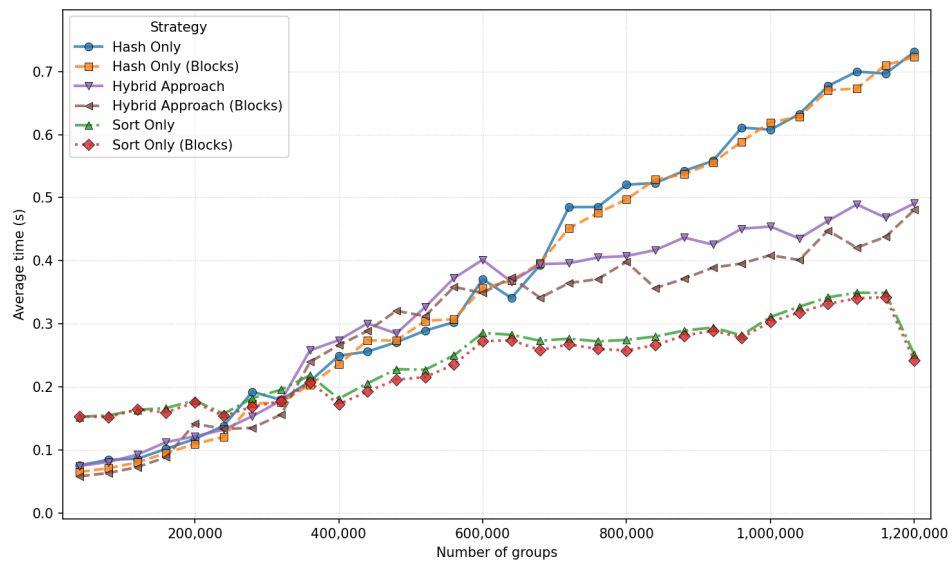


Figure 6.2: Comparing strategies: sorting, hash-map, hybrid with 1.2 million entries

The strategies were also measured with a total of 12 million entries. This was done in order to check whether the slope of the strategies was dependent on the total number of entries. The results are shown in Figure 6.3.

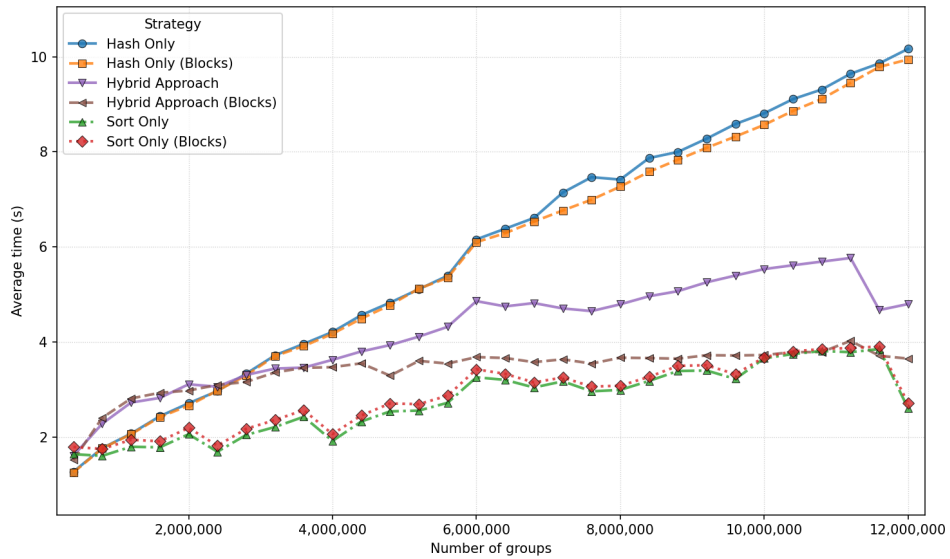


Figure 6.3: Comparing strategies: sorting, hash-map, hybrid with 12 million entries

The previous measurements did not show what is happening if the group number is a very small number. Therefore, a measurement was taken which started with just one group and kept continuously multiplying the number of groups by 1.5 and rounding up the result, until the number was greater than the total number of entries. This was done for both 1.2 million and 12 million entries. The results are shown in Figures 6.4 and 6.5. As in every previous measurement, both single-block and multi-block cases were measured.

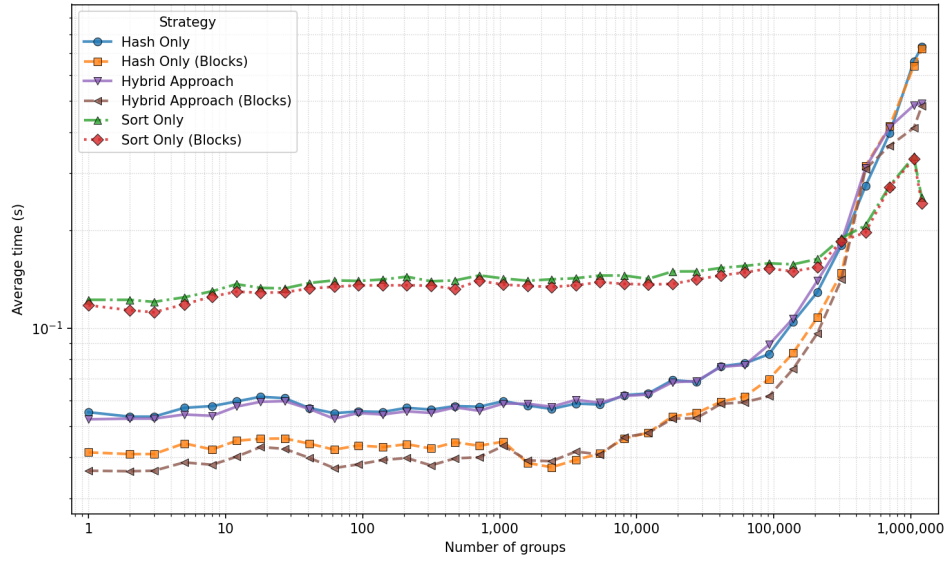


Figure 6.4: Comparing strategies: sorting, hash-map, hybrid with 1.2 million entries on a logarithmic scale

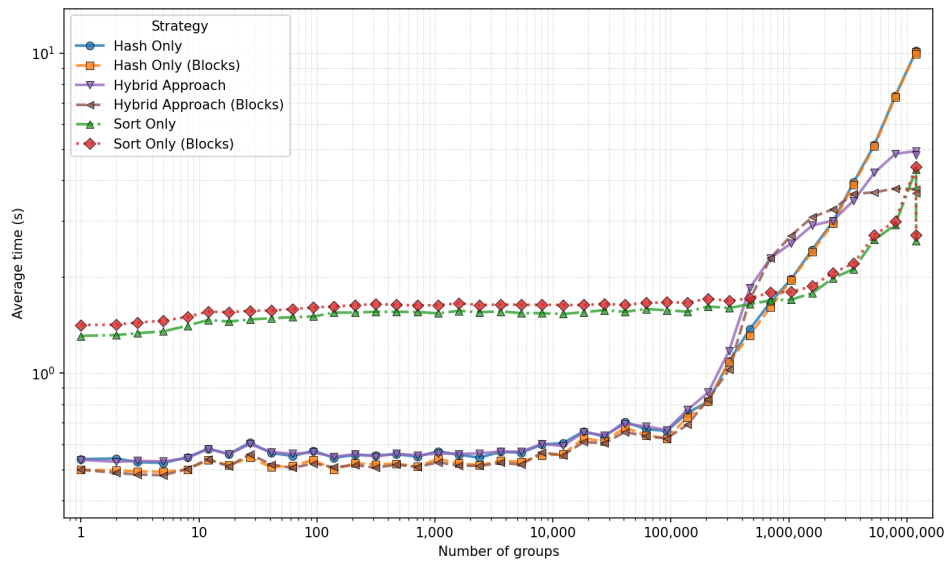


Figure 6.5: Comparing strategies: sorting, hash-map, hybrid with 12 million entries on a logarithmic scale

7 Discussion

7.1 Evaluation of the Results

To compare the runtime of the two strategies as a function of the number of groups, a benchmark was made. Figure 6.1 shows the result of this benchmark. Looking at the Figure, it is apparent that the hash-based grouping strategy outperforms the sorting-based grouping strategy up to a certain number of groups. If the number of groups is smaller than or equal to 320,000 groups, hash-based grouping is consistently faster. If it is larger than or equal to 360,000 groups, hash-based grouping is consistently slower. The threshold must therefore be in between these two numbers. For the sake of simplicity, the threshold value of 350,000 was chosen for the subsequent measurements, where also the hybrid approach was tested with the benchmark.

The next two measurements also tested the runtime of the hybrid approach. They were performed with 1.2 million (Figure 6.2) and 12 million (Figure 6.3) entries, respectively. A number of things can be observed.

1. The hybrid approach is faster than the slowest previous strategy in approximately 80% of the cases.
2. The sorting strategy depends much less on group size than the hash map strategy. The hybrid approach falls between the two.
3. In both diagrams, the multi-block case is significantly faster than the single large block case.

The number “80%” comes from the observation that, in the measurement with 1.2 million entries, the hybrid approach was slower than the slowest previous strategy in 7 out of 30 cases, whereas in the measurement with 12 million entries it was

slower in 5 out of 30 cases. On average, therefore, the hybrid approach was slower in 6 out of 30 cases, or 20% of the time.

The computation time of the sorting strategy depends mostly on the number of input entries. It depends little on how many groups there are in the end. In other words, the computation time has a flat slope when plotted against the number of groups. This can be seen in Figure 6.1.

After fitting a line on the time curve of the sorting strategy with regression analysis, the slope of the line can be calculated. The result can be seen in Figure 7.1. The time unit used for the computation of the slope was microseconds. We can see that the slopes are similar for both 1.2 and 12 million entries, with only a slight increase for the larger dataset. Also the difference between single-block and multi-block cases is small.

In the table we can see that the computation time of the hash map strategy is much more dependent on the number of groups. When fitting a line on the plot against the number of groups, the slope of the hash map strategy is much steeper than the slope of the sorting strategy, for both 1.2 and 12 million entries.

		Sorting (°)	Hash map (°)	Hybrid (°)
1.2 million entries	Single-block	9.5568	31.8700	20.3027
	Multi-block	9.1531	31.0368	18.9768
12 million entries	Single-block	10.5021	37.4765	16.1157
	Multi-block	10.0932	36.6270	6.3337

Table 7.1: Slopes of fitted lines by dataset size, block configuration and grouping strategy

Finally, it is no surprise that the multi-block case proved to be mostly faster than the single-block case. However, it is quite surprising how different these two cases are in the measurement with 12 million entries (Figure 6.3). In the multi-block case, the hybrid approach was sometimes even faster than the sorting method in more critical scenarios with a large number of groups. Also the slope of the entire line is much flatter in this case than in the single-block case. It is also flatter than any other slopes that were measured.

Looking at Figure 6.4 of the measurement on a logarithmic scale, a number of things can be observed:

1. the hash map strategy and the hybrid approach are both significantly faster than the sorting strategy in the low group count range (up to approximately 350,000 groups);
2. the hash map strategy and the hybrid approach are very close to each other in performance in most of the cases (up to approximately 700,000 groups), only diverging in the high group count range;
3. in the low group count range (up to approximately 1000 groups), the multi-block case of the hybrid approach is the fastest; and
4. the single-block and multi-block cases are otherwise very close to each other in performance.

Similar observations can be made from Figure 6.5:

1. the hash map strategy and the hybrid approach are both significantly faster than the sorting strategy in the low group count range (up to approximately 350,000 groups);
2. the single-block and multi-block cases are otherwise very close to each other in performance; however,
3. the hybrid approach is now almost indistinguishable from the hash map strategy in most cases, only diverging after approximately 300,000 groups;
4. between approximately 500,000 and 3,000,000 groups, the hybrid approach is slower than both other strategies.

It can be concluded that the hybrid approach performs very similarly to the hash map strategy in most cases, only diverging in the high group count range. The difference between the sorting strategy and the other two strategies in the small-number case can be explained by the fact that in this case, the hash map uses very little memory.

8 Summary and Outlook

8.1 Conclusion

The results from the experiments indicate that the optimizations for the **GROUP BY** operator in **QLever** led to significant improvements in query execution times, albeit not in all cases.

The hybrid approach, which combines hash map and sorting strategies, was significantly faster than the slowest of the original strategies, in both scenarios with a large number of groups and in scenarios with a small number of groups. This confirms the assumption that neither of the two original strategies is generally superior, and that a combination of both strategies can yield better performance.

However, the hybrid approach was not the fastest in all scenarios. Close to the point where the hash map strategy starts to struggle with memory consumption, the hybrid approach was slower than both original strategies. This is probably because switching between strategies and managing two different data structures is time-consuming. Further optimizations could be explored to reduce this overhead.

8.2 Directions for Future Research

There are a number of ways the implementation of the **GROUP BY** operator could be improved.

In this thesis, much emphasis has been placed on the efficiency in terms of time. However, it would also be desirable to also optimize **GROUP BY** in respect of

memory usage. Currently, if the number of triples to group is too large, the engine may crash because of memory inefficiency. This can be solved by partitioning entries and storing them on the hard drive, and then sorting them one-by-one in the RAM or even sorting them on the hard drive.

Another direction for future research could be to further improve the hybrid approach by making the threshold adaptive. Currently, the threshold is a fixed number set by the user with a default value of 350,000. However, it would be possible to make the threshold depend on characteristics of the input data, such as the distribution of group sizes or the total number of entries. This could lead to even better performance in a wider range of scenarios.

9 Acknowledgments

First, I would like to express my gratitude to my supervisor, Johannes Kalmbach, for his careful review of my code and for answering all my questions. I would also like to thank Prof. Hannah Bast for being my first examiner.

Second, I would like to thank my family and friends for sharing their ideas and for being there for me during my studies and while working on this thesis.

Bibliography

- [1] H. Bast and B. Buchhold. *QLever: A Query Engine for Efficient SPARQL + Text Search*. In: Proceedings of the 2017 ACM Conference on Information and Knowledge Management (2017), pp. 647–656. URL: <https://doi.org/10.1145/3132847.3132921>.
- [2] E. F. Codd. *A relational model of data for large shared data banks*. In: Commun. ACM (1970), pp. 377–387. URL: <https://doi.org/10.1145/362384.362685>.
- [5] S. Hong et al. *Query Optimization for Graph Analytics on Linked Data Using SPARQL*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF), July 2015. DOI: [10.2172/1215587](https://doi.org/10.2172/1215587). URL: <https://www.osti.gov/biblio/1215587>.
- [6] L. Kuiper et al. *Saving Private Hash Join*. In: Proceedings of the VLDB Endowment 18.8 (2025), pp. 2748–2760. URL: <https://www.vldb.org/pvldb/vol18/p2748-kuiper.pdf>.
- [10] TechTarget. *Graceful degradation*. Accessed: 2025-12-01. 2023. URL: <https://www.techtarget.com/searchnetworking/definition/graceful-degradation>.
- [11] K. Tharani. *Much more than a mere technology: A systematic review of Wikidata in libraries*. In: The Journal of Academic Librarianship 47.2 (2021), p. 102326. URL: <https://doi.org/10.1016/j.acalib.2021.102326>.
- [12] A. Waagmeester et al. *Wikidata as a knowledge graph for the life sciences*. In: eLife 9 (2020). Accessed: 2025-11-24, e52614. URL: <https://doi.org/10.7554/eLife.52614>.

Web Resources

- [3] DBLP Team. *DBLP Computer Science Bibliography + Citations from OpenCitations*. https://sparql.dblp.org/download/dblp_KG_with_associated_data.tar. Accessed: 2025-09-27.
- [4] Grimm Brothers. *Aschenputtel (Cinderella)*. https://www.grimmstories.com/en/grimm_fairy-tales/aschenputtel. Accessed: 2025-09-23.
- [7] H. Mühleisen and M. Raasveldt. *Parallel Grouped Aggregation in DuckDB*. Accessed: 2025-11-27. 2022. URL: <https://duckdb.org/2022/03/07/aggregate-hashtable>.
- [8] QLever Team. *QLever control*. <https://github.com/ad-freiburg/qllever-control>.
- [9] QLever Team. *QLever Olympics Dataset*. <https://qllever.cs.uni-freiburg.de/olympics>. Accessed: 2025-09-27.
- [13] Wallscope. *120 Years of Olympics*. <https://github.com/wallscope/olympics-rdf>. Accessed: 2025-09-27.

Glossary

fallback

A secondary mechanism or default behavior used when the primary option is unavailable or fails. [15](#), [17](#), [18](#)

GROUP BY

A SPARQL clause used to group query results by one or more expressions for aggregation functions such as COUNT, AVG, MIN, and MAX. [1-3](#), [6](#), [12](#), [15](#), [17](#), [18](#), [22](#), [I](#), [II](#)

QLever

A high-performance SPARQL query engine for large RDF datasets. [1](#), [2](#), [22](#)

query

A query is a user-made request for information or data to a database or a search engine to retrieve specific results. [1](#)

RDF data

Short for Resource Description Framework. A way of representing information as subject – predicate – object triples, such as

`<Gerard> <type> <human>` and

`<Gerard> <yearOfBirth> <1994>`

which would represent the information “Gerard is a human being born in 1994”. RDF has been developed by [W3C](#) as a standard model for data interchange on the web. For example, websites can publish their data in RDF format enabling search engines and other applications to automatically understand and integrate information from the website. In the case of a publication database this information could be how author profiles are linked, etc. [B](#)

SPARQL

Short for “SPARQL Protocol and RDF Query Language”. This language has a syntax similar to [SQL](#) and it enables the usage of RDF data. SPARQL supports matching graph patterns, filtering, aggregation, and data manipulation, just like SQL. [1](#), [4](#), [6](#)

SQL

Short for Structured Query Language. . [1](#), [5](#), [B](#)

W3C

Short for World Wide Web Consortium. W3C is the main organization for international standards of the World Wide Web. W3C is responsible for developing protocols and guidelines, to ensure, inter alia, the web’s accessibility and security. For example, W3C develops standards such as HTML, CSS, and the [RDF data](#) model, which define how web pages are structured, styled, and how data is represented and exchanged on the web.

[A](#)