

Undergraduate Thesis / Bachelorarbeit

Optimizing SPARQL FILTER
Expressions in QLever by Introducing
Prefilter Procedures

Hannes Baumann

Examiner: Prof. Dr. Hannah Bast

Advisor: M. Sc. Johannes Kalmbach

Albert-Ludwigs-University Freiburg
Department of Computer Science
Chair of Algorithms and Data Structures

June 19th, 2025

Writing period

19. 03. 2025 – 19. 06. 2025

Examiner

Prof. Dr. Hannah Bast

Advisers

M. Sc. Johannes Kalmbach

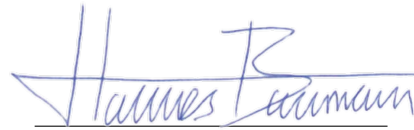
Declaration

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

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

Riegel, 19.06.2025

Place, Date

A handwritten signature in blue ink, reading "Hannes Baumann", written over a horizontal line.

Signature

Abstract

QLever (clever!) is a SPARQL engine developed at the Chair of Algorithms and Data Structures at the University of Freiburg. QLever enables users to efficiently query large knowledge graphs, such as Wikidata [1].

This thesis is an optimization contribution related to the frequently used **FILTER** expression. For example, **FILTER(?var > 5)** enforces that only rows with a value greater than 5 in the column associated with the variable **?var** are included in the result.

The previous evaluation of a query involved the procedure of scanning all index sections for a given scan specification through the **INDEX SCAN** operation related to the variable **?var**. The involvement of variables in **FILTER** expressions was not taken into account during the scanning process. As a consequence, the evaluation of **FILTER** expression required iterating over a mostly unnecessary large number of rows, checking in this case the **?var**-associated column value on the condition **?var > 5**. A significant number of rows typically contain values that do not satisfy the **FILTER** expression. Thus, scanning all rows and subsequently iterating over them to evaluate the **FILTER** expression introduced an unnecessary computational overhead, especially since QLever stores its indexes in sorted order. These sorted indexes allow us to efficiently prefilter and subsequently scan only the index sections that contain values satisfying the expression **?var > 5**. The objective of this thesis is to implement a prefilter procedure that reduces the evaluation overhead for such **FILTER** expressions.

Zusammenfassung

QLever (clever!) ist eine SPARQL-Engine, welche am Lehrstuhl für Algorithmen und Datenstrukturen der Universität Freiburg entwickelt wird. QLever ermöglicht Anwendern effiziente SPARQL-Abfragen über sehr große Wissensgraphen, wie beispielsweise Wikidata [1], auszuführen.

Diese Thesis fokussiert sich auf die Optimierung der häufig verwendeten **FILTER** Operation. Der Ausdruck **FILTER(?var > 5)** bewirkt, dass das Ergebnis nur Zeilen enthält, welche die Bedingung **?var > 5** erfüllen.

Es ist offensichtlich, dass in den meisten Fällen ein Großteil der in den Indexsektionen enthaltene Werte die Filterbedingung nicht erfüllt. Bisher wurden jedoch Indexsektionen ohne Berücksichtigung der Filterbedingung mittels der **INDEX SCAN** Operation gescannt, und alle daraus resultierenden Zeilen anschließend auf die Erfüllung der Filterbedingung überprüft. Diese Vorgehensweise ist ineffizient, insbesondere im Kontext von QLever, wo jeder Index sortiert vorliegt. Ein sortierter Index ermöglicht eine Vielzahl von Optimierungen. Unter anderem kann auch die Auswertung einiger Filterausdrücke optimiert werden. Die Optimierung wird durch einen Vorfilter realisiert, der genau die Indexsektionen vorfiltert, welche Werte enthalten, die die Filterbedingung erfüllen. Für das eben genannte Beispiel wird bezüglich **?var > 5** vorgefiltert. Folgend werden nur filterrelevante Indexsektionen per **INDEX SCAN** Operation gescannt. Dies hat zur Folge, dass auch die Filterbedingung über eine geringere Anzahl an Zeilen geprüft wird, was den bisherigen Aufwand an Rechenressourcen reduziert. Ziel meiner Arbeit ist die Implementierung dieser Vorfilterstrategie, was zur Folge hat, dass der zuvor beschriebene Overhead reduziert wird.

Contents

1. Introduction	1
1.1. SPARQL Standard	1
1.2. Resource Description Framework and Knowledge Graphs	1
1.3. Large RDF Knowledge Graphs and QLever	3
1.4. FILTER Expressions and Queries	4
1.5. Problem Definition: Overhead in FILTER Expression Evaluation . .	7
2. Related Work	8
3. Further Background	9
3.1. RDF Knowledge Graph Indexing with QLever	9
3.2. Vocabulary Identifiers	10
3.3. General 64-bit Identifiers	11
3.4. Index Block Metadata	12
3.5. The INDEX SCAN Operation	13
4. Approach	14
4.1. Prefilter supported SPARQL Expressions	14
4.2. The Algorithm Constructing Prefilter Expressions	15
4.2.1. The Base Case	16
4.2.2. Logical AND	17
4.2.3. Logical OR	19
4.2.4. Logical NOT	20
4.2.5. Complexity Analysis	21
4.3. Prefilter Expressions	22
4.3.1. Relational Expressions	22
4.3.2. AND and OR	24
4.3.3. NOT	24
4.3.4. STRSTARTS and REGEX	25
4.3.5. IN and NOT IN	26

4.3.6. IsDatatype	26
4.3.7. Prefilter Dates by Year	27
4.3.8. Complexity Analysis	28
4.4. The Prefilter Procedure in the Context of Query Planning	29
4.4.1. Overview of Prefilter Procedure Integration	30
4.4.2. Prefilter Pushdown during Query Planning	30
4.4.3. Challenges	30
5. Empirical Analysis	32
5.1. Benchmark	32
5.2. Observations	34
6. Conclusion	36
7. Acknowledgments	37
A. Benchmark Queries and Their Execution Trees	38
A.1. Query 1	38
A.2. Query 2	39
A.3. Query 3	41
A.4. Query 4	42
A.5. Query 5	43
A.6. Query 6	44
A.7. Query 7	47
A.8. Query 8	50
A.9. Query 9	53
A.10. Query 10	53
Bibliography	55

1. Introduction

1.1. SPARQL Standard

The current SPARQL standard [2] is defined and maintained by the W3C (World Wide Web Consortium). The name SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language. As the name suggests, it defines a standardized language used to query data stored in the RDF (Resource Description Framework) format¹.

1.2. Resource Description Framework and Knowledge Graphs

RDF represents data as descriptive triples in the format of **subject predicate object**; or abbreviated **s p o**. In the following, we refer to this concept as RDF triples. Such an RDF triple describes a directed relationship between the **subject** and **object** entities via a **predicate**. A single component of such an RDF triple, hence **subject**, **predicate**, or **object**, is referred to as an RDF term.

Brief example. `<Ada_Lovelace> <Occupation> <Mathematician>` declares that Ada Lovelace's occupation was that of a professional mathematician. Such an RDF triple can also be interpreted as a directed graph. The **subject** and **object** represent nodes, and the **predicate** represents the directed edge (see Figure 1).

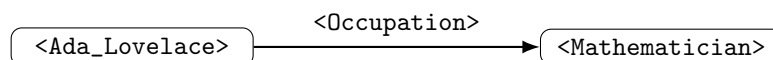


Figure 1.: The RDF triple `<Ada Lovelace> <Occupation> <Mathematician>` represented as a directed graph.

The RDF triple presented above is slightly simplified. The `<...>` structure is used to indicate that the RDF term is an IRI (Internal Resource Identifier). An IRI provides a

¹<https://www.w3.org/TR/rdf11-concepts/>

globally unique identifier for the resource represented by the RDF term. In most cases, an IRI is simply an URL (Uniform Resource Locator) that refers to publicly available information about the entity it represents. Table 1 provides an example with concrete IRI values for the respective components of an RDF triple. In addition to an IRI value, RDF terms can also represent simple literals, such as "Ada Lovelace". It is also possible to specify the intended interpretation of literals by appending a language tag or type specifying IRI. "Ada Lovelace"@en indicates that "Ada Lovelace" is written in english, while "33"^^<http://www.w3.org/2001/XMLSchema#int> specifies that "33" should be interpreted as an integer.

Triple Component	IRI (Wikidata)
<Ada_Lovelace>	<http://www.wikidata.org/entity/Q7259>
<Occupation>	<http://www.wikidata.org/prop/direct/P106>
<Mathematician>	<http://www.wikidata.org/entity/Q170790>

Table 1.: Wikidata IRIs representing the entities <Ada_Lovelace>, <Mathematician>, and the property <Occupation>.

Figure 1 illustrated that RDF triples which define directed relationships can be interpreted as a directed graph. The sample of RDF triples provided in Table 2, along with its corresponding representation as a directed graph in Figure 2, visualizes how the RDF triples interconnect to form a larger knowledge graph. This representation demonstrates how such triples can collectively represent rich semantic information.

RDF Triples
<Ada_Lovelace> <Occupation> <Mathematician>.
<Ada_Lovelace> <Has_name> "Ada Lovelace".
<Ada_Lovelace> <Known_for> <First_Computer_Algorithm>.
<Ada_Lovelace> <Known_for> <The_Analytical_Engine>.
<Ada_Lovelace> <Worked_with> <Charles_Babbage>.
<Ada_Lovelace> <Daughter_of> <Lord_Byron>.
<Charles_Babbage> <Occupation> <Mathematician>.
<Charles_Babbage> <Known_for> <The_Analytical_Engine>.

Table 2.: Sample RDF Triples

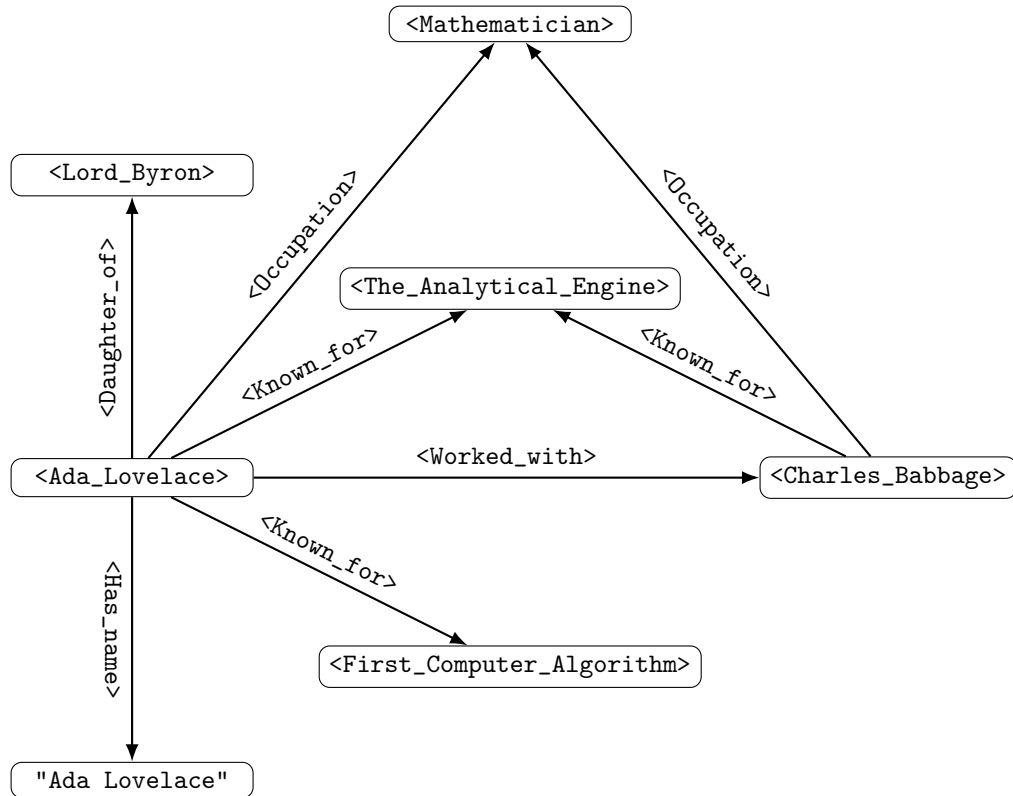


Figure 2.: RDF triples represented as a graph.

Note that the RDF knowledge graph shown in Figure 2 not only directly connects Ada Lovelace to Charles Babbage via the triple `<Ada_Lovelace> <Worked_with> <Charles_Babbage>`, but also indirectly associates them. Both entities are indirectly associated through their respective relationships to the same triple objects, `<The_Analytical_Engine>` and `<Mathematician>`.

1.3. Large RDF Knowledge Graphs and QLever

QLever² is a SPARQL engine developed at the Chair of Algorithms and Data Structures at the University of Freiburg. It is developed with an emphasis on performance in modern C++ and already applies a wide range of optimization strategies, some of which are mentioned in Chapter 2.

The result of those optimization efforts is that QLever enables users to efficiently query popular RDF knowledge graph datasets, such as complete Wikidata, UniProt,

²<https://github.com/ad-freiburg/qllever>

or PubChem [1, 3, 4]. Wikidata is one of the largest collaborative general-purpose knowledge graphs, UniProt represents comprehensive information on all relevant proteins, and PubChem is the largest database for chemical compounds. A significant reason for their popularity is the scope of available information, which enables in-depth knowledge discovery; Wikidata, UniProt and PubChem consist of billions of RDF triples.

1.4. FILTER Expressions and Queries

FILTER is defined as a result-constraining operation in the SPARQL standard. The name **FILTER** already provides an intuitive understanding of the semantics behind it. For a conditional expression **EXPR** that evaluates true or false, the result of the **FILTER(EXPR)** expression includes only those rows in the result for which its expression evaluates true.

The following SPARQL queries using the **FILTER** expression outline the type of queries whose evaluation procedure we aim to optimize. These queries are applied to the Olympics dataset [5]. The corresponding result tables and query execution trees were obtained through the public QLever user interface³. Note that retrieving the entity-identifying IRI, corresponding to `?athlete_id` in the queries below, is often useful. However, these identifiers are omitted from the presented results to keep them compact. The execution trees serve as visual support for Section 1.5, discussing the current evaluation overhead of **FILTER** expressions.

Query 1

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?athlete_id ?athlete_name WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  FILTER REGEX(?athlete_name, "^Sebast")
}
LIMIT 5
```

Run Query 1 in QLever: <https://qllever.cs.uni-freiburg.de/olympics/VbrLKs>
This query retrieves the name of five athletes whose name starts with "Sebast" from the Olympics dataset.

³<https://qllever.cs.uni-freiburg.de>

	?athlete_name
1	Sebastiaan Clemens Verschuren
2	Sebastiaan Jacques Henri "Bas" van de Goor
3	Sebastian "Sebu" Kuhlberg
4	Sebastian Bachmann
5	Sebastian Bayer

Figure 3.: Result Table of Query 1

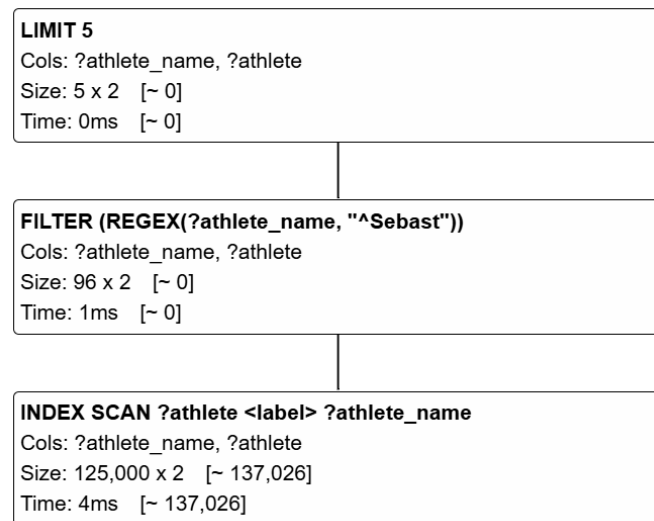


Figure 4.: Execution Tree of Query 1

Query 2

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?athlete_id ?athlete_name ?athlete_age WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  ?athlete_id foaf:age ?athlete_age .
  FILTER(?athlete_age >= 60 && ?athlete_age <= 65)
}
LIMIT 5
  
```

Run query in QLever: <https://qllever.cs.uni-freiburg.de/olympics/ZmVvxR>
 This query retrieves the name and age of five athletes whose age is between 60 and 65 from the Olympics dataset.

	?athlete_name	?athlete_age
1	Adrienne Jouclard	65
2	Afanasijs Kuzmins	61
3	Afanasijs Kuzmins	65
4	Alberto Guerrero Recio	65
5	Alfred Egerton Cooper	65

Figure 5.: Result Table of Query 2

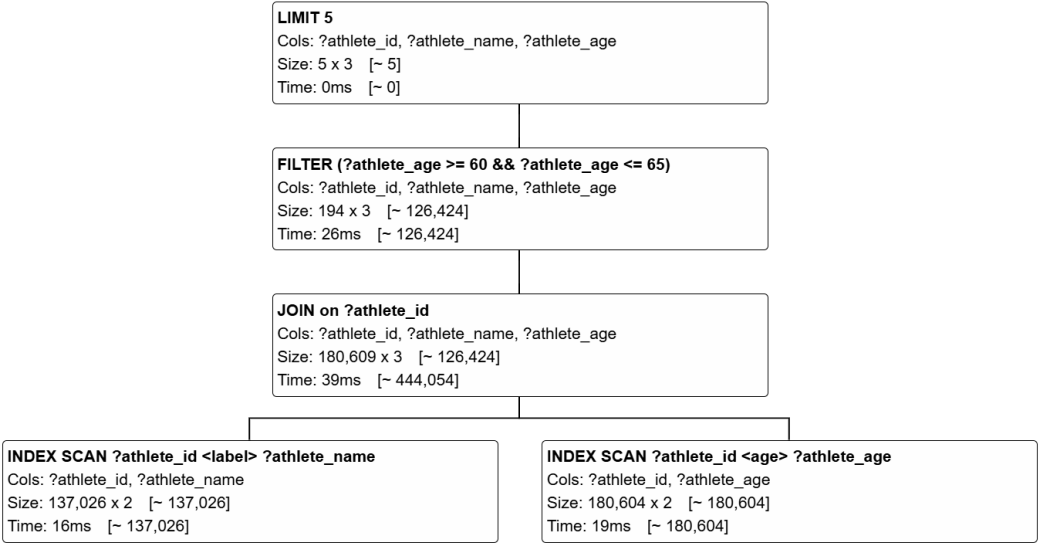


Figure 6.: Execution Tree of Query 2

1.5. Problem Definition: Overhead in FILTER Expression Evaluation

To emphasize the current **FILTER** evaluation overhead, we begin this section with an analysis of the evaluation structure of Query 1 from Section 1.4. The analysis is based on its corresponding query execution tree, as depicted in Figure 4. Note that the query execution tree provides the result table size for each operation, formatted as $num_rows \times num_cols$.

Query 1 retrieves the first five athlete names that start with "Sebast" from the Olympics dataset, using the constraint `REGEX(?athlete_name, "^Sebast")`. It is intuitively clear that the evaluation of this filter constraint expression is computationally expensive, since the evaluation of `REGEX` involves the direct comparison of strings. The complete evaluation of Query 1 involves three concrete steps. Scanning the index with the **INDEX SCAN** operation, evaluating the **FILTER** clause with constraint `REGEX(?athlete_name, "^Sebast")`, and finally evaluating **LIMIT** to retrieve only the first five matches. We can observe that the **INDEX SCAN** operation returns a table of size $125,000 \times 2$. Without further optimizations, the expression `FILTER(REGEX(?athlete_name, "^Sebast"))` must be evaluated against 125,000 potential string values to verify if they begin with "Sebast". In contrast, the result table of the **FILTER** expression is only of size 96×2 , while 125,000 string value comparisons were executed. It is obvious that scanning all index sections without consideration of the **FILTER** expression via the **INDEX SCAN** operation leads to computational overhead, as only 96 rows are relevant to the result.

A similar overhead is observable for Query 2 through its execution tree depicted in Figure 6. The **INDEX SCAN** operation, scanning the index column corresponding to the variable `?athlete_age`, produces a table of size $180,604 \times 2$. In contrast, only 194 values satisfy the expression `(?athlete_age >= 60 && ?athlete_age <= 65)`, resulting in an intermediate result table of size 194×3 .

We observe with both queries the same problem. Their **FILTER** expressions are notably restrictive. This results in a large number of rows being discarded during a compute-intensive **FILTER** expression evaluation process. The root cause of this problem is the **INDEX SCAN** operation, which scans index sections that are known in advance to be irrelevant to the result. The following chapters provide a feasible solution. They introduce an optimization that reduces the number of scanned index sections by pushing prefilter predicates (i.e., prefilter expressions) down to the **INDEX SCAN** operation.

2. Related Work

The `FILTER` expression optimization approach discussed in this thesis is primarily based on existing infrastructure and optimizations already embedded in QLever. As a result, the optimization is quite specific to QLever. Bast et al. [6] introduced the SPARQL engine QLever, which enables efficient querying over RDF knowledge graphs combined with a text corpus. Their work introduces fundamental techniques to build an efficient SPARQL engine, such as a general projection of RDF terms into compact integer identifiers (IDs), permuted and compressed indexes, and an efficient query planner that selects the optimal evaluation order for a given query based on cost estimates. Another notable paper is Bast et al. [7], which specifically presents in detail techniques on how to prepare RDF knowledge graphs for efficient querying, as well as strategies for optimizing query evaluation. The discussed approaches align with the optimization strategies applied by QLever. It also involves sorted and permuted indexes, and additionally discusses monotonic OID (Object Identifier) mapping of RDF terms through the vocabulary.

The optimization technique discussed here is closely related to the strategy of pushing predicates down to the scan or reading phase, a common optimization in database query engines. For example, the support of pushing down predicates for Spark SQL is discussed by Armbrust et al. [8]. They leverage the approach of predicate pushdown, with predicates being directly evaluated while reading the table from the file. Similarly, Lin et al. [9] also employ the predicate pushdown technique, as they introduce a comprehensive and safe extension algorithm to support it. Their goal is to optimize the evaluation of SQL queries in data processing pipelines by reducing the amount of data generated at the beginning of the pipeline. Although their focus is on table-based databases, their approach is transferable to QLever for optimizing `FILTER` expressions. This thesis similarly aims to discard irrelevant data early during the `INDEX SCAN` operation.

3. Further Background

This chapter explains the techniques behind optimized indexing, as the structure of the optimized index plays a key role in the optimization of **FILTER** expressions. The first section outlines the general approach on which QLever builds its RDF knowledge graph index. The next two sections explain how all RDF terms are projected into compact integer IDs (identifiers). These IDs are essential for reducing the index footprint and enhancing query evaluation performance. Next, the concept of index compression is briefly presented, along with the associated structure that keeps track of the compression metadata. The structure that keeps track of the compressed index plays a key role in the optimization approach discussed in Chapter 4. The final section concludes by presenting the relationship between the **INDEX SCAN** operation and the compression metadata of the index.

3.1. RDF Knowledge Graph Indexing with QLever

Knowledge graphs provided as raw RDF triples cannot be directly queried with QLever, since its engine that evaluates queries requires a precomputed index of the respective knowledge graph. For index construction, QLever features a dedicated index-builder component, building not just one index but multiple permutations of it.

The concept of storing permuted and sorted RDF triples for indexing with QLever was introduced by Bast et al. [6]. When we refer to an index as sorted, it means that the RDF terms are first ordered by the primary term, then by the secondary term, and finally by the tertiary term, whereby only the first terms are overall fully sorted as a column. For the RDF triples forming the knowledge graph, there are $3! = 6$ possible permutations.

The six RDF term permutations

- | | |
|-----------|-----------|
| (1) s p o | (4) p o s |
| (2) s o p | (5) o s p |
| (3) p s o | (6) o p s |

Bast et al. found that the permutations **pos** and **pso** are sufficient for typical

queries. Given this finding, QLever lets the user decide whether all six permutations should be built or only the two mentioned above. Note that the permutation **pos** is of particular relevance to the optimization introduced, as explained later on.

The availability of sorted and permuted indexes enable significant optimizations of crucial operations such as **INDEX SCAN** and **JOIN**. They are also crucial for the optimization of **FILTER** expressions.

3.2. Vocabulary Identifiers

For QLever, Bast et al. [6] introduced separate vocabularies for RDF data and text corpus. We refer to the vocabulary used for storing RDF data as the RDF vocabulary. All IRI and literal RDF terms in the knowledge graph are added to the RDF vocabulary during construction of the index. QLever implements a vocabulary-based monotonic ID (identifier) mapping. Since these IDs are exclusively associated with entries in the RDF vocabulary, they are here referred to as Vocab-IDs. These Vocab-IDs are implemented as unsigned integer values. The mapping assigns each distinct IRI and literal a unique Vocab-ID while preserving the intended < order of the RDF terms. This enables efficient relational comparison operations by performing them directly on the assigned Vocab-ID integers, without processing the original string values. Given the Vocab-ID characteristics, the index can be built without relying on the RDF term strings by substituting them with their corresponding Vocab-IDs. The SPARQL-compliant Vocab ID mapping and its properties are formally described by Bast et al. [7], the concept is introduced as OID (Object Identifier) mapping.

The concept behind Vocab-ID mapping is well presentable through an example. Five RDF triples are given in Table 3. The related RDF terms and their assigned Vocab-IDs are listed in Table 4, where the order of the RDF terms is preserved for their corresponding Vocab-ID.

RDF Triples
<Ada_Lovelace><Occupation> <Mathematician>.
<Ada_Lovelace> <Has_name> "Ada Lovelace".
<Ada_Lovelace> <Known_for> <First_Computer_Algorithm>.
<Ada_Lovelace> <Known_for> <The_Analytical_Engine>.
<Charles_Babbage> <Has_name> "Charles Babbage".

Table 3.: Sample RDF Triples

they fall into separate ranges, primarily ordered by their type bits. The payload bits that contain the actual value determine the secondary order for values of the same datatype. This defines the order logic of ID-mapped RDF terms within the sorted index.

Entries from the RDF vocabulary and the query-local vocabulary are also encoded as distinct data types. The ID mapping for RDF vocabulary entries is done via their associated Vocab-IDs, preserving the underlying lexicographic order. Query-local IRI and literal values are assigned to IDs through their entry index in the query-local vocabulary. The relational comparison between an ID encoding an RDF vocabulary entry and another ID encoding a query-local vocabulary entry is not performed using their raw bit representation. Instead, when comparing an RDF vocabulary ID to another query-local vocabulary associated ID, we effectively compare the Vocab-ID of the RDF vocabulary entry to the corresponding Vocab-ID range of the query-local IRI or literal. The Vocab-ID range is defined by an upper and lower Vocab-ID, representing the potential position in the RDF vocabulary. Determining the relation between two IDs that encode query-local vocabulary entries is done by directly comparing their underlying string values. For all other ID pairs encoding different datatype values, the relational comparisons are directly performed on the bit representation. Since QLever builds the index adhering to ascending order through the $<$ operator directly on the IDs introduced here, the IDs encoding query-local and RDF term entries are contained in mixed and in lexicographic order.

3.4. Index Block Metadata

To optimize memory usage, the permuted indexes are not stored as a complete arrangement of ID-mapped RDF triples. Instead, QLever compresses each index permutation into blocks and stores the compressed representation in the memory file. This compression approach for QLever was introduced by Bast et al. [6].

block0	block1	block2	...	blockN
--------	--------	--------	-----	--------

Index stored as a sorted sequence of compressed blocks.

The compression is performed by first partitioning the index through fixation of the first RDF term, while grouping and compressing the second and third terms. In the context of an `INDEX SCAN`, the second and third terms are typically considered "free", which means that they represent scan columns of two associated variables. All second and third terms are treated as pairs in the next step of the compression.

Using these pairs, the index is compressed by grouping all third terms for each unique second RDF term. The grouped third terms and their associated second term are subsequently written to the index file, while recording their position within the file via byte offsets. Each index block corresponds to a fixed first RDF term and contains all grouped third terms to a distinct single second term. However, to maintain practical block sizes in memory when loading the data, each block adheres to an upper size limit. As a consequence, the grouped values may be split over multiple blocks in the file. The byte offsets of these blocks, along with the offset for the associated fixed first term within the file, are recorded. These byte offsets serve as lookup keys for locating each block in the index file.

The structure that records and tracks the offset information of the index blocks within the file is referred to as the Index Block Metadata. In addition to the offset information, each Index Block Metadata value contains the complete first and last RDF triples of its associated index block. This is particularly relevant to the **FILTER** expression optimization approach, as the information about the value range of the associated index block is available through the first and last RDF triples without decompression. If decompressing the index blocks was necessary to access their value range information, the discussed optimization could potentially introduce an overhead that outweighs its performance benefits.

3.5. The INDEX SCAN Operation

The **INDEX SCAN** operation always represents a leaf node in the query execution structure, as illustrated by Figure 4 and Figure 6. For a given scan specification and defined permutation, this operation retrieves the scan-relevant Index Block Metadata values. The provided scan specification indicates whether the scan is performed over all three columns (full scan). This corresponds to a scan with respect to three variables, while scans over one or two columns are invoked for specifications involving one or two associated variables, respectively. Note that only the first scanned column is fully sorted, according to the order of the sorted index, as explained in Section 3.1. When the **INDEX SCAN** operation is subsequently evaluated, it decompresses the index sections through the scan-relevant Index Block Metadata values by retrieving the index blocks from the file via the recorded byte offsets. The result is returned as a table and further passed on to subsequent operations.

4. Approach

We optimize the evaluation performance of `FILTER` expressions by discarding irrelevant index blocks. Index blocks are considered irrelevant if we can safely assume that they contain only values that do not satisfy the `FILTER` expression. Scanning those irrelevant index blocks via the `INDEX SCAN` operation introduced an evaluation overhead for the `FILTER` expression downstream, as analyzed in Section 1.5. Section 3.4 introduced the Index Block Metadata structure, which effectively represents an index block. Thus, index blocks can be omitted by discarding their associated Index Block Metadata prior to performing the `INDEX SCAN` operation. Irrelevant index blocks are discarded using a prefilter expression that mimics the logic of the `FILTER` expression, as we keep only the index blocks with values that satisfy the expression. As a result, the `INDEX SCAN` operation subsequently decompresses and yields only the index blocks associated with the Index Block Metadata values relevant to the `FILTER` expression and its result.

We refer to the entire process, from retrieval of the prefilter expression to discarding irrelevant index blocks using the corresponding Index Block Metadata values in the `INDEX SCAN` operation during query planning, as the prefilter procedure. The following sections outline each relevant step involved in the prefilter procedure, while the conclusive section of this chapter elaborates on the prefilter procedure in the broader context of query planning and its associated challenges.

4.1. Prefilter supported SPARQL Expressions

The logical SPARQL operators listed in Table 6 are supported by prefilters, as well as the SPARQL expressions listed in Table 5. In addition, composite expressions such as `FILTER(YEAR(?date) = 2000)`, which retrieve dates in relation to a year specified as an integer, are also supported.

Operator	Description
Relational Expressions	
?var < val	Check ?var is less than val
?var <= val	Check ?var is at most val
?var > val	Check ?var is greater than val
?var >= val	Check ?var is at least val
?var = val	Check ?var equals val
?var != val	Check ?var does not equal val
isDatatype Expressions	
isIRI(?var)	Check ?var is an IRI
isBlank(?var)	Check ?var is a blank node
isLiteral(?var)	Check ?var is a literal
isNumeric(?var)	Check ?var is a numeric literal
Prefix Filter Expressions	
STRSTARTS(?var, "prefix")	Check ?var starts with "prefix"
REGEX(?var, "^prefix")	Check ?var starts with "prefix"
IN and NOT IN Expression	
?var IN (expr1, ..., exprN)	Check if ?var matches any of the expression results
?var NOT IN (expr1, ..., exprN)	Check if ?var does not match any of the expression results

Table 5.: Supported SPARQL expressions

Operator	Description
Logical Expressions	
&&	AND
	OR
!	NOT

Table 6.: Supported logical SPARQL expressions

4.2. The Algorithm Constructing Prefilter Expressions

We refer here to the original SPARQL expression (*SE*) with the notation EXPR_{SE} , while EXPR_{PE} denotes its corresponding prefilter expression (*PE*). This section introduces the algorithms involved in the construction of the prefilter expressions, by briefly explaining their behavior. Section 4.2.1 introduces the base case, involving the expressions listed in Table 5. This algorithm is rather trivial. However, the algorithms that implement the merge procedures of **AND** and **OR** are somewhat com-

plex, particularly when negated through a logical NOT (discussed in Section 4.2.4). This complexity arises from the goal to support prefiltering for FILTER expressions involving multiple variables in the future, such as `FILTER(?x > 33 && ?y < 33)` or `FILTER((?x < 0 && ?y < 0) || (?x > 20 && ?y > 33))`.

Given that the base prefilter expressions are evaluated via binary search, the requirement is a fully sorted variable-related column. However, given the order of the RDF triples when sorted for the index, only a single column is effectively fully sorted. This fully sorted scan column is the first non-fixated variable-related column for which the INDEX SCAN operation is performed. As a consequence, our approach requires the decomposition of expression EXPR_{SE} into multiple variable-specific prefilter expressions $\text{EXPR}_{PE}^{var_0}, \dots, \text{EXPR}_{PE}^{var_n}$, such that $?var_i \neq ?var_j$ for all $i \neq j$. $\text{EXPR}_{PE}^{var_i}$ denotes the prefilter expression applied to the scan column associated with $?var_i$. Each of these expressions $\text{EXPR}_{PE}^{var_i}$ is successfully matched with a distinct INDEX SCAN operation under the condition that each variable $?var_i$ corresponds to a fully sorted column. Under this condition, the set of expressions $\text{EXPR}_{PE}^{var_0}, \dots, \text{EXPR}_{PE}^{var_n}$, each successfully applicable to its corresponding INDEX SCAN, can be interpreted as a logical conjunction. The following algorithms operate on a prefilter expression list PF:

$$\text{PF} = [\text{EXPR}_{PE}^{var_0}, \dots, \text{EXPR}_{PE}^{var_n}], \quad (1)$$

which is semantically interpreted as the logical conjunction:

$$\bigwedge_{i=0}^n \text{EXPR}_{PE}^{var_i}. \quad (2)$$

4.2.1. The Base Case

Algorithm 1 GETPREFILTEREXPRESSIONOFEXPR()

- 1: $var \leftarrow \text{GETVARIABLE}()$
 - 2: $value \leftarrow \text{GETREFERENCEVALUE}()$
 - 3: $expr \leftarrow \text{EXPR}_{PE}(value)$
 - 4: **return** $[(var, expr)]$ ▷ List with (variable, expression) pair
-

Algorithm 1 outlines how we handle the base case involving the SPARQL expressions listed in Table 5. The procedure here is simple: access the associated variable `?var` and the reference value(s), then return the pair consisting of the corresponding EXPR_{PE} and the associated variable `?var`. The assigned variable associates EXPR_{PE} with the INDEX SCAN operation that scans the index column that corresponds to

variable `?var`. For SPARQL expressions that are not listed in Table 5, the algorithm returns an empty list.

4.2.2. Logical AND

We assume that the argument `isNegated` is set to `false` here, since the case `isNegated = true` is implicitly discussed in Section 4.2.4.

Algorithm 2 MERGEANDPREFILTEREXPRESSION (*leftExpr, rightExpr, isNegated*)

```

1:  $pf\_in\_1 \leftarrow leftExpr.GETPREFILTEREXPRESSION(isNegated)$ 
2:  $pf\_in\_2 \leftarrow rightExpr.GETPREFILTEREXPRESSION(isNegated)$ 
3:  $pf\_out \leftarrow []$ 
4:  $i \leftarrow 0, j \leftarrow 0$   $\triangleright pf\_in\_1$  and  $pf\_in\_2$  are sorted by variable
5: while  $i < |pf\_in\_1|$  and  $j < |pf\_in\_2|$  do
6:    $(var_1, expr_1) \leftarrow pf\_in\_1[i]$ 
7:    $(var_2, expr_2) \leftarrow pf\_in\_2[j]$ 
8:   if  $var_1 = var_2$  then
9:      $merged \leftarrow isNegated ? OR_{PE}(expr_1, expr_2) : AND_{PE}(expr_1, expr_2)$ 
10:    APPEND( $pf\_out, (var_1, merged)$ )
11:     $i \leftarrow i + 1, j \leftarrow j + 1$ 
12:   else if  $var_1 < var_2$  then
13:     APPEND( $pf\_out, (var_1, expr_1)$ )
14:      $i \leftarrow i + 1$ 
15:   else
16:     APPEND( $pf\_out, (var_2, expr_2)$ )
17:      $j \leftarrow j + 1$ 
18:   end if
19: end while
20: while  $i < |pf\_in\_1|$  do
21:   APPEND( $pf\_out, pf\_in\_1[i]$ )
22:    $i \leftarrow i + 1$ 
23: end while
24: while  $j < |pf\_in\_2|$  do
25:   APPEND( $pf\_out, pf\_in\_2[j]$ )
26:    $j \leftarrow j + 1$ 
27: end while
28: return  $pf\_out$ 

```

This algorithm effectively implements the strategy of decomposing SPARQL expression $EXPR_{SE}$ into multiple variable-specific prefilter expressions $EXPR_{PE}^{?var_0}, \dots, EXPR_{PE}^{?var_n}$, while adding them to the result list PF_{OUT} . The splitting occurs during the merge procedure of the sorted prefilter expression lists PF_{IN1} and PF_{IN2} , since only a pair

of expressions associated with the same variable can be meaningfully combined by conjunction $\text{AND}_{PE}(\text{expr}_1, \text{expr}_2)$. All prefilter expressions that do not form a pair through matching variables (i.e., where $\text{var}_1 \neq \text{var}_2$), are added individually to the result. This is possible given the implicit logical-and semantics behind a prefilter expression list PF (see Equation 1). Adding an expression associated with a distinct variable technically adds a conjunct to the result PF_{OUT} . This behavior is illustrated in Figure 7 for the expression $?x > 33 \ \&\& \ ?y < 33$.

Note that prefilter expressions are contained in the order according to their associated variables within the lists PF_{IN1} and PF_{IN2} . In combination with the algorithm's merging logic (linear-time), the prefilter expressions are added to PF_{OUT} in the order of their associated variables, thereby preserving the sorted order and uniqueness.

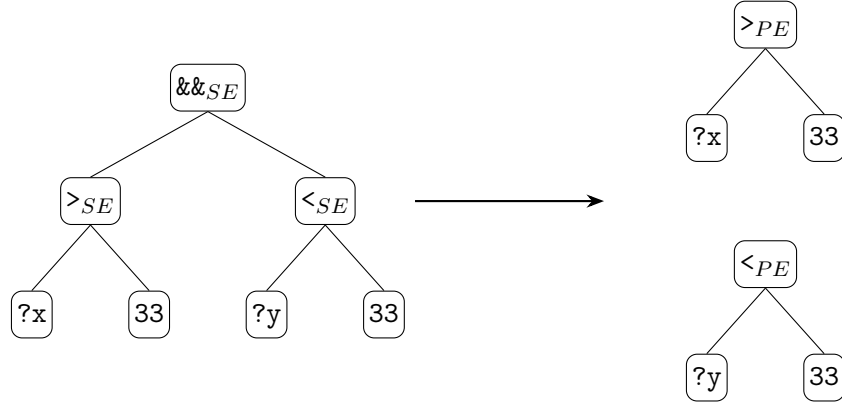


Figure 7.: SPARQL expression (left) and the corresponding prefilter expressions (right). Each prefilter expression is associated with a distinct variable.

4.2.3. Logical OR

We assume that the argument `isNegated` is set to `false` here, since the case `isNegated = true` is discussed in Section 4.2.4.

Algorithm 3 MERGEORPREFILTEREXPRESSION(*leftExpr*, *rightExpr*, *isNegated*)

```

1: pf_out  $\leftarrow []$ 
2: pf_in_1  $\leftarrow$  leftExpr.GETPREFILTEREXPRESSION(isNegated)
3: pf_in_2  $\leftarrow$  rightExpr.GETPREFILTEREXPRESSION(isNegated)
4: i  $\leftarrow 0$ , j  $\leftarrow 0$   $\triangleright$  lists are sorted by variable
5: while i < |pf_in_1| and j < |pf_in_2| do
6:   (var1, expr1)  $\leftarrow$  pf_in_1[i]
7:   (var2, expr2)  $\leftarrow$  pf_in_2[j]
8:   if var1 = var2 then
9:     merged  $\leftarrow$  isNegated ? ANDPE(expr1, expr2) : ORPE(expr1, expr2)
10:    APPEND(pf_out, (var1, merged))
11:    i  $\leftarrow$  i + 1, j  $\leftarrow$  j + 1
12:   else if var1 < var2 then
13:     i  $\leftarrow$  i + 1
14:   else
15:     j  $\leftarrow$  j + 1
16:   end if
17: end while
18: return pf_out

```

As presented with Algorithm 3, the general merging strategy of the **OR** merge procedure resembles that of the **AND** merge implemented by Algorithm 2. If two prefilter expressions of PF_{IN1} and PF_{IN2} are associated with the same variable, they are combined using the corresponding merge operator (i.e., $\text{OR}_{PE}(\text{expr}_1, \text{expr}_2)$ here) and subsequently added to the result. However, in contrast to the **AND** merge procedure, prefilter expressions that do not form a match via their associated variables are discarded. The reason is that the result list must semantically reflect $\text{expr}_1 \parallel \text{expr}_2$ across the two lists PF_{IN1} and PF_{IN2} , as it is obviously the case when two expressions are joined via OR_{PE} . But when we instead add a prefilter expression expr associated with a distinct variable individually to the result PF_{OUT} , we subsequently prefilter with the assumption that its condition must be satisfied. This is because a prefilter expression list PF semantically represents a conjunction across multiple **INDEX SCAN** operations (see Equation 1). However, the assumption that a prefilter expression

`expr` with a distinct associated variable must be satisfied is not sound in the context of an `OR` merge, as a disjunction `expr || exprother` may still be satisfied even if `expr` evaluates to false. Thus, the algorithm excludes prefilter expressions for variables without a matching counterpart in the other list from the resulting list `PFOUT`.

4.2.4. Logical NOT

Algorithm 4 `GETNOTPREFILTEREXPRESSION(isNegated, childExpr)`

```

1: pf ← childExpr.GETPREFILTEREXPRESSION( $\neg$ isNegated)
2: for all (var, expr) in pf do
3:   expr ← NOTPE(expr)
4: end for
5: return pf

```

The implementation of function `NOT` is algorithmically simple, since all `EXPRPE` are capable of performing the logical complement. For `ANDPE` and `ORPE`, this involves applying the De Morgan laws:

$$\neg(A \wedge B) = (\neg A) \vee (\neg B) \quad (3)$$

$$\neg(A \vee B) = (\neg A) \wedge (\neg B) \quad (4)$$

Let us now briefly discuss how the Boolean `isNegated` flag used in the `AND` merge Algorithm 2 and the `OR` merge Algorithm 3, as it relates to the algorithm responsible for constructing prefilter expressions for logical `NOT`. Assume that we have the expression `!(?x <= 33 || ?y >= 33)`, and want to retrieve the corresponding `EXPRPE`. If `isNegated` is set to `false` during its retrieval, the `OR` merge is performed as explained in Section 4.2.3 on the child expressions `?x <= 33` and `?y >= 33`. During this merge procedure, the variables `?x` and `?y` of the prefilter expression do not match. Consequently, these prefilter expressions are discarded. As a result, the merge procedure returns an empty list `PFOUT` to its `NOT` expression parent, the algorithm presented here. This is obviously wrong, since `!(?x <= 33 || ?y >= 33)` is equivalent to `(?x > 33 && ?y < 33)` by the De Morgan law 4. Thus, the result is expected to contain the prefilter expressions corresponding to `?x > 33` and `?y < 33`. The solution is to perform an `AND` merge for the `OR` child, while using the disjunction `ORPE` to join the expressions whose variables match. This happens under the condition that `isNegated` is toggled to `true`, and the `AND` merge algorithm from Section 4.2.2 is executed corre-

spondingly with `isNegated = true`. This approach results in prefilter expressions for `?x <= 33` and `?y >= 33`, which are subsequently complemented to `?x > 33` and `?y < 33` with Algorithm 4 of this section. Note that in case of previously `ORPE` joined prefilter expressions `expr1` and `expr2` (i.e., matching variables), the consolidation `ORPE` is logically complemented according to De Morgan 4. Analogously, for De Morgan law 3, an `OR` merge with `isNegated = false` is performed on an `AND` child node.

4.2.5. Complexity Analysis

We omit cases with redundant negations. For worst-case expressions, the `AND` merge (see Algorithm 2) dominates the complexity, as it merges two sorted lists without discarding any expressions. Given the retrieved lists `PFIN1` and `PFIN2`, the complexity of the merge procedure is $O(k)$, where $k = |\text{PF}_{IN1}| + |\text{PF}_{IN2}|$. The size of the output `PFOUT` depends on the number of matched expressions through their associated equal variables. If all variables are distinct, all prefilter expressions are added individually. Thus, $|\text{PF}_{OUT}| = |\text{PF}_{IN1}| + |\text{PF}_{IN2}|$ in the worst case. Let us assume an expression which consists only of conjunctions. The worst composition of such an expression is represented by a highly unbalanced binary tree with n `AND` nodes, while introducing $n + 1$ distinct variables across $n + 1$ base expressions. This implies that for every merge, either `PFIN1` or `PFIN2` is of size 1. For the first merge performed, the size of the other list is also 1, which results in $|\text{PF}_{OUT}| = 2$. Subsequently, the next merge is performed on input lists of size 1 and 2, thus $|\text{PF}_{OUT}| = 3$. The final merge operates on lists of sizes 1 and n , with $|\text{PF}_{OUT}| = n + 1$. At each step, the merge procedure iterates over $|\text{PF}_{OUT}|$ prefilter expressions across the lists `PFIN1` and `PFIN2`. Thus, the total complexity is given by:

$$\sum_{i=2}^{n+1} i = \frac{(n+1)(n+2)}{2} - 1,$$

which simplifies to $O(n^2)$ when merging n conjunctions of base expressions, if internally structured as a highly unbalanced tree.

For the base algorithm (see Section 4.2.1) that covers the expressions listed in Table 5, the time complexity is $O(1)$. Since most `FILTER` expressions typically involve at most one or two variables, the procedure to retrieve the corresponding prefilter expression is unlikely to impose a significant performance overhead.

4.3. Prefilter Expressions

We categorize the prefilter expressions of relational expressions and logical operators as primary, as they form the basis on which all other (composite) expressions are built. Section 4.3.1 and Section 4.3.2 discuss these primary prefilters and the corresponding evaluation procedure. Based on these primary prefilter expressions, the following sections elaborate on the prefilter logic for secondary SPARQL expressions. For example, the prefilters associated with expressions such as **STRSTARTS** or **IN** are considered secondary.

4.3.1. Relational Expressions

Let us briefly revisit the order of the index and its corresponding Index Block Metadata values. The index consists of permuted ID-mapped RDF triples, as discussed in Section 3.3. As a consequence, the index is sorted with respect to these ID values. The IDs introduced are 64-bit integers, which enable efficient relational comparison operations. In addition, QLever compresses the index into blocks, while tracking those index blocks through the component introduced as Index Block Metadata in Section 3.4. The evaluation-relevant information of an Index Block Metadata value is its first and last triple. In particular, relevant to the evaluation is the term of those triples at the position associated with the filter-related variable. The Index Block Metadata values are provided as a sorted span, reflecting the order of the index. Note that it is essential to maintain the sorted order. Otherwise, the **INDEX SCAN** operation subsequently decompresses and returns an unsorted index, resulting in false operation evaluations downstream.

The evaluation procedure of a relational prefilter expression is best explained by an example. Let us assume that the objective is to prefilter the index for the expression **FILTER(?x > 10)**. The corresponding retrieved prefilter checks for the condition $> ID_{10}$, where ID_{10} represents the ID mapping of 10. The following four steps outline how the filter-relevant Index Block Metadata values are located.

1. **Retrieving the ID Range:**

Since we are given a span of Index Block Metadata values that contain the first and last ID-mapped RDF triple of the associated compressed index blocks, the first step is to access the filter-relevant ID range. This ID range is represented by all RDF terms at position i , the column index corresponding to the filter-related variable $?x$, across both the first and last triples of all Index Block Metadata values. The terms are successively accessible, effectively representing a column

over the triples of the Index Block Metadata values as a sorted ID range. Thus, a span consisting of n Index Block Metadata values yields an ID range of size $2 \times n$, as they contain two triples.

2. Locating the relevant Datatype Ranges:

For our concrete example, IDs satisfying $ID > ID_{10}$ are considered relevant. However, the ID range may contain IDs of various datatypes, and it is obviously not possible to meaningfully compare ID_{10} with an ID representing a vocabulary entry, or an ID representing a date value. In addition, most ID pairs are directly compared on their underlying raw bit representation, an exception from the vocabulary entries, as explained in Section 3.3. This poses an issue because the first four type bits distort the comparison when the goal is to evaluate an integer ID against a double ID based on the actual content of the payload bits. Thus, this step involves locating the subranges relevant to the comparison within the ID range retrieved previously. For ID_{10} encoding the integer value 10, the comparison-relevant datatype ranges are those associated with integers and doubles. Analogously, the comparison-relevant datatype subranges are located for reference IDs that represent values of other datatypes. Locating these subranges associated with the comparison-relevant datatypes is efficient, since the given ID range is sorted and binary search is applicable.

In practice, each subrange associated with a specific datatype section is defined by a pair of iterators indicating its start and end within the overall ID range retrieved in Step 1.

3. Locating the Relevant Ranges:

Having located the subranges consisting of the comparison-relevant datatypes in the previous step, the objective is to refine these subranges for the given condition $ID > ID_{10}$. The localization of the relevant subranges consisting of IDs that satisfy $ID > ID_{10}$ is also performed by binary search. The direct comparison of IDs encoding numerical values is done bit-wise. Thus, the value ID_{10} must be decoded and re-encoded as a double $ID_{10.00}$, for performing binary search over the subranges associated with double values. Otherwise, the comparison would not make sense since the bits encoding the datatype would dominate the relational comparison.

4. Get the Relevant Index Block Metadata:

The retrieved subranges defining the sections of IDs which satisfy $ID > ID_{10}$ are represented by a pair of iterators referring to the total ID range obtained in

Step 1. Since the ID range has size $2 \times n$ for a span consisting of n Index Block Metadata values, the iterator values of the pairs that define the relevant ID subranges are consequently offset by a factor of 2. As a result, the sections of relevant Index Block Metadata values can be obtained by dividing the iterator values of each pair by two. Thus, we effectively map back the ID subranges to their corresponding subranges in the Index Block Metadata span. Each Index Block Metadata subrange is defined by a pair of iterators that mark the start and end positions within the span. Consequently, we can omit a direct materialized copy of the relevant Index Block Metadata values, since the corresponding subranges are recorded by iterator pairs. This evaluation procedure within the recursive context is quite efficient, as only pairs of subrange-defining iterators are passed around, instead of the actual Index Block Metadata values.

Evaluations involving different relational operators, or IDs of a different datatype, are performed analogously to the presented example.

4.3.2. AND and OR

Both AND (`&&`) and OR (`||`) are considered binary in the sense that they possess two child expressions. Both of these child expressions return the subranges corresponding to the relevant Index Block Metadata values for their respective expressions.

The evaluation of an AND expression is performed by computing the intersection over the iterator-based subranges returned by its child expressions. Analogously, the evaluation of an OR expression is performed by computing the union over the respective subranges. Since we internally keep these ranges in sorted order with respect to the associated relevant Index Block Metadata values, the evaluation of the expressions AND an OR can be performed through a linear iteration procedure over the ranges while merging them according to the union or intersection logic.

4.3.3. NOT

As already introduced with Algorithm 4 in Section 4.2.4, creating a negated prefilter expression involves performing the logical complement on the respective (single) child expression. If the child expression represents a logical AND or OR, its complement is determined by De Morgan's laws (see Equations 3 and 4). Complementing a relational expression involves changing the relational operator with its corresponding logical counterpart:

(`>` \leftrightarrow `<=`), (`>=` \leftrightarrow `<`), and (`=` \leftrightarrow `!=`).

Thus, given that the whole child expression is effectively already logically negated and accordingly evaluated, there is no actual evaluation procedure involved here.

4.3.4. STRSTARTS and REGEX

Note that we can only prefilter **REGEX** expressions involving a prefix check on the string value, such as **REGEX(?name, "^Bob")**. **STRSTARTS** is effectively equivalent to a case-sensitive **REGEX** prefix check. In addition, case-insensitive prefix matching can be performed using **REGEX** with flag value "i" (e.g., **REGEX(?name, "^Bob", "i")**). This will result in matches where the difference between upper and lowercase letters is ignored.

Section 3.2 discussed the RDF vocabulary that represents the static index vocabulary. This vocabulary implements a monotonic Vocab-ID mapping that preserves the order of all defined entries during index construction. QLever's vocabulary entries include both IRI and literal values, which are ordered lexicographically in a case-insensitive manner. Consequently, both case-sensitive and case-insensitive prefix checks can be prefiltered using the same approach. Our goal is to construct a precise prefilter using both upper- and lower-bound reference values, represented by bounding IDs that define the range for vocabulary entries that match our prefix.

The corresponding prefix range with respect to the RDF vocabulary is determined by two binary search operations. The first search determines the lower bound, which is the position of the first vocabulary entry that matches the provided prefix. The second search finds the upper bound, which is the position of the first entry that no longer matches the prefix. These search results are effectively the Vocab-IDs corresponding to the position of the bounding entries in the vocabulary. However, the lower-bound Vocab-ID needs to be adjusted by decrementing it by one. This adjustment is necessary because query-local literal values are compared according to their hypothetical position in the static RDF vocabulary, as explained in Section 3.3. This ensures that the prefilter also includes potential IDs encoding prefix-relevant query-local values, as they may be placed in order before the entry associated with the unadjusted lower-bound Vocab-ID. Next, the decremented lower-bound Vocab-ID and the upper-bound Vocab-ID are mapped to their corresponding general IDs, **lowerID** and **upperID**. Based on these bounding IDs, the resulting prefilter expression is as follows:

(> lowerID) AND (< upperID).

4.3.5. IN and NOT IN

The expression `FILTER(?var IN ("Bob", 2, 33.00, <http://iri>))` restricts the result returned by `FILTER` to rows where the respective value in the column associated with `?var` was either equal to "Bob", 2, 33.00 or `<http://iri>`. The SPARQL standard defines the `IN` expression as the disjunction:

$lhs = \text{EXPR}_0 \ || \ lhs = \text{EXPR}_1 \ || \ \dots \ || \ lhs = \text{EXPR}_N$. `IN` checks if the value on the left-hand side is contained in the list of expressions on the right-hand side when evaluated. However, since our prefilter cannot involve the active evaluation of expressions $\text{EXPR}_0, \dots, \text{EXPR}_N$, the prerequisite for correct prefiltering is that the reference list contains only constant expressions. Constant expressions are literals, typed literals such as "2000-11-28T00:00:00.000"^^`xsd:dateTime`, IRIs, and the Boolean constants. Thus, as explained in Section 3.3, the values of these constant expressions can be directly mapped to a corresponding ID value, as needed for prefiltering. Given a list consisting only of constant expressions $\text{EXPR}_1, \dots, \text{EXPR}_N$, the corresponding prefilter expression is a disjunction over the equality conditions on the retrieved IDs: $(= \text{ID}_1) \ || \ (= \text{ID}_2) \ || \ \dots \ || \ (= \text{ID}_N)$.

In contrast, $lhs \neq \text{EXPR}_0 \ \&\& \ lhs \neq \text{EXPR}_2 \ \&\& \ \dots \ \&\& \ lhs \neq \text{EXPR}_N$ represents the logical interpretation behind expression `NOT IN`. To prefilter `NOT IN`, akin to the complement of `IN`, we construct its prefilter expression as a conjunction of non-equality conditions: $(\neq \text{ID}_1) \ \&\& \ (\neq \text{ID}_2) \ \&\& \ \dots \ \&\& \ (\neq \text{ID}_N)$.

The evaluation of `IN` and `NOT IN` prefilter expressions is based on the procedures explained in Section 4.3.1 and Section 4.3.2.

4.3.6. IsDatatype

The prefilter of `isBlank(?var)` is simple, since blank terms are handled as a separate datatype, the associated Index Block Metadata values can be simply located via binary search. Its evaluation follows the same approach as that of relational prefilters (see Section 4.3.1), but by skipping the third step, since the aim is to locate only the datatype range. For `isNumeric(?var)`, the prefilter retrieves the datatype ranges corresponding to integer and double values.

The expressions `isIri(?var)` and `isLiteral(?var)` require a different evaluation approach, since the vocabulary entries include both IRI and literal values, ordered lexicographically and in a case-insensitive manner. The consequence is that both IRI and literal values are included in the ID range associated with the RDF vocabulary and the query-local vocabulary entries, as explained in Section 3.3. And given that

the ID projections of vocabulary entries are order-preserving, the IDs corresponding to literal values precede those associated with IRIs, since `''` (quotation marks indicating a literal) is lexicographically smaller than `<` (angle brackets indicating an IRI). Thus, we can prefilter for the expression `isLiteral(?var)` by using the ID representation of `<` as an upper bound. The expression `isIri(?var)` is prefiltered in accordance by choosing `<>` as a lower bound.

4.3.7. Prefilter Dates by Year

This prefilter optimizes expressions such as `FILTER(YEAR(?date) = 2000)` and `FILTER(YEAR(?date) < 0)`. The inner `YEAR` expression extracts the year component of a date value as an integer, which is then compared to the reference year, provided as an integer. The result is that the first example retrieves all dates from the year 2000, while the second example retrieves all dates preceding year 0.

A prefilter is feasible, since QLever also maps date values into the general ID space in an order-preserving manner. For a given a SPARQL-compliant `xsd:dateTime` or `xsd:date` value, such as the literals `"2025-05-31T15:30:00Z"^^xsd:dateTime` and `"2025-05-31"^^xsd:date`, QLever internally dissects the value into its individual components: year, month, and day; hour, minute, second; and the optional timezone. The corresponding integer values of the individual components are then bit-shifted into an unsigned 64-bit integer, representing the complete date value. The bit-shifting procedure is designed so that the bit position of each component reflects its significance in the overall date value. High-order bits are used for more significant date components, such as the year and month, while the low-order bits are occupied by the time components. This projection approach preserves the chronological order of date and date-time values. Thus, it is ensured that the IDs corresponding to dates are primarily ordered by the year component, exactly what is required for evaluating a prefilter expression using binary search.

The corresponding prefilter for `FILTER(YEAR(?date) = 2000)` captures the range of dates from the year 2000 through a lower and upper reference bound. The lower bound is represented by the ID that encodes a partial date where only the year component is set, while the month and day components remain undefined, effectively corresponding to `"2000-00-00"` in the internal encoding. The upper bound is represented by an ID that represents the partially defined internal date `"2001-00-00"`. The resulting prefilter expression is the conjunction: `(> lowerID) AND (< upperID)`.

The prefilter for `FILTER(YEAR(?date) < 0)` involves only an upper bound: the ID

representing the partially defined internal date "0000-00-00". The resulting prefilter expression is simply: `< upperID`. The implementation generalizes the approach illustrated by the two given examples to all possible filter-by-year compound expressions, using the primary prefilter expressions presented in Section 4.3.1 and Section 4.3.2.

Note: The undefined date components are internally set to the placeholder value 0, making partially specified dates comparable and sortable alongside fully defined date values. These partially defined date values are in particular useful for prefiltering, as they represent safe and exclusive bounds.

4.3.8. Complexity Analysis

Section 4.3.1 presented the evaluation approach for the relational prefilters that also serve as base expressions for more specific prefilters, such as the prefilter for **STRSTARTS** (see Section 4.3.4) and the prefilter locating dates with a certain year (see Section 4.3.7). The evaluation of relational prefilter expressions is based on binary search. The first search identifies the ranges of the comparison-relevant datatypes, while subsequent searches locate the ranges within that satisfy the relational comparison on the corresponding reference ID.

Let us briefly analyze the complexity of a binary search procedure. Binary search requires a sorted range of values and proceeds as follows for a given target value:

1. Consider the current value range. In the first iteration, this corresponds to the entire range.
2. Check the value in the middle of the range. If this value equals the target, the search is complete.
3. If the target value is smaller, repeat the search in the left half of the range.
4. If the target value is greater, repeat the search in the right half of the range.
5. If the range is not further dividable, the target value is not contained. This corresponds to the worst-case scenario of binary search.

The best-case scenario is obviously that the value defining the first split is equivalent to our target value. This would correspond to $O(1)$. The worst-case occurs if our target value is not contained. If we perform a prefilter on n Index Block Metadata values, the ID range on which the search is performed consists of $2 \times n$ IDs, since each Index Block Metadata value holds the first and last triple of its corresponding index block. As a result, the upper limit on the number of range halvings performed

is $\log_2(2 \times n)$. This is equivalent to $\log_2(2) + \log_2(n)$, which simplifies to $1 + \log_2(n)$. Thus, for relational prefilter expressions, the complexity is $O(\log(n))$.

As explained in Section 4.3.2, evaluating prefilter expressions containing disjunctions and/or conjunctions involves computing the union or intersection of the subranges returned by the child expressions using a linear iteration approach. This is possible because the order of the subranges is preserved throughout all evaluation steps. Thus, compound prefilter expressions that require multiple union or intersection computations lead to a complexity worse than $O(\log(n))$. The worst-case scenario occurs when evaluating multiple unions (OR-prefilter evaluations) sequentially over a highly unbalanced prefilter expression tree, where none of the subranges can be merged, and the subsequent resulting union is simply a set containing all subranges from both children. The complexity in this case is similar to the analysis presented in Section 4.2.5, with the difference that the defining factor is not the number of distinct variables in a conjunction, but the number of non-overlapping subranges produced by disjunctive prefilter expressions. This case results in a quadratic time complexity on the number of disjunctions. Note that the upper limit for the number of subranges returned at each step is $\frac{n}{2}$, since the adjacent ranges are merged, where n is the total number of Index Block Metadata values. However, this scenario is largely hypothetical. Most compound prefilter expressions involve only a few disjunctions and/or conjunctions, in particular too few to cover the upper bound $\frac{n}{2}$ subranges separately.

Thus, given that typical `FILTER` expressions involve only few disjunctions and/or conjunctions, if any, the evaluation of its prefilter expression should remain efficient.

4.4. The Prefilter Procedure in the Context of Query Planning

The integration of the prefilter procedure not only requires its implementation in the query evaluation process but also requires alignment with the overall query planning and evaluation framework of QLever. Its query planner, as introduced by Bast et al. [6], optimizes for efficiency by preferring query execution trees with the lowest estimated cost. Each query execution tree defines the order in which the operations of the query are evaluated. Thus, the objective of the query planner is the selection of the query execution order with the least associated computational cost. A visual representation of such query execution trees is provided in Section 1.4.

4.4.1. Overview of Prefilter Procedure Integration

As briefly mentioned in the introduction of this chapter, we evaluate the prefilter expressions in advance of the actual `INDEX SCAN` evaluation. As a result, this approach not only improves the evaluation efficiency of the `FILTER` expression but also that of the variable-associated `INDEX SCAN` operation, since definitely irrelevant index blocks are subsequently excluded from the scan. Given that the query planner selects execution orders based on cost estimates, the selection of the correct query execution order depends on accurate cost estimates. Consequently, `INDEX SCAN` operations with an applied prefilter expression must provide an accurate cost estimate that reflects the benefit of prefiltering, since its cost benefit must also be reflected in the cost estimate of the subsequent `FILTER` expression. If the benefit of prefilter application is not signaled via beneficial cost estimates, the resulting advantage is ignored by the query planner. The cost estimate for the `INDEX SCAN` operation is related to the size of its result table, where the size correlates with the number of Index Block Metadata values it must decompress. Thus, it is necessary to evaluate the prefilter expression already during query planning, so that we accurately influence the cost estimates of the `INDEX SCAN` operation and the `FILTER` expression, as well as the total query execution order.

4.4.2. Prefilter Pushdown during Query Planning

The procedure during query planning is as follows for the current implementation. The `FILTER` expressions retrieve their corresponding prefilter expression(s), as presented in Section 4.1, subsequently passing them down to their variable-associated `INDEX SCAN` operation. With the implementation tested in Chapter 5, prefilter expression pushdowns are only successfully performed if the associated `INDEX SCAN` operation is a direct child of `FILTER`. As a consequence, the current implementation only supports prefiltering with respect to a single variable.

4.4.3. Challenges

My initial intention was to propagate the prefilter expressions in addition through the `SORT` and `JOIN` operations. All prefilterable variables contained in a `FILTER` expression are associated with a distinct `INDEX SCAN` operation, since each `INDEX SCAN` scans only a single fully sorted column. As explained previously, a fully sorted column is the prerequisite for applying a prefilter expression. Thus, the results of these variable-associated `INDEX SCAN` operations must first be joined using the `JOIN` operation

before the actual **FILTER** expression can be meaningfully evaluated. However, a **JOIN** operation typically requires the **pso** permutation, where the **predicate** is fixed and the **subject** column is fully sorted, since **JOIN** operations are mostly performed on the **subject** columns representing entity identifiers. In contrast, for a prefilter expression to be applicable, the **INDEX SCAN** must typically operate on the **pos** permutation. The **object** column (e.g., containing the name or other characteristics) is often the target of the variables contained in the **FILTER** expression, and must subsequently correspond to the first fully sorted column for a fixed **predicate**. Thus, for the result of a scanned and prefiltered **pos** permutation to be joinable by operation **JOIN**, a fully sorted **subject** column is required. Consequently, when the **pos** permutation is used for the **INDEX SCAN**, the query planner must consider inserting a **SORT** operation between the **INDEX SCAN** and the **JOIN**, since the **subject** column is only partially sorted. However, the query planner with its current implementation does not consider the possibility of a deep prefilter expression push-down and its potential benefits. If I analyzed the query planning procedure correctly, it does not contemplate combining the **pos** permutation mostly required for applying a prefilter expression on the **INDEX SCAN** operation, with a subsequent **SORT** on the **subject** column to make the result joinable with **JOIN**. However, as explained above, incorporating this combination into the query planning phase is essential for the effective application of multiple prefilter expressions, each associated with a distinct variable and **INDEX SCAN** operation. Due to time constraints, it was not feasible for me to adapt the query planner's strategy accordingly. Note that this adaptation would introduce critical changes, as the suggested approach would potentially lead to an increase in the number of available query execution trees. As a consequence, the performance of the query planner could be noticeably degraded. Thus, the prefilter expression pushdown and necessary query planning adaptations should initially be extended further only to the most essential operations required for prefiltering across multiple variables: the operations **JOIN** and **SORT**. If the pushdown strategy proves to be beneficial for the overall query evaluation performance, it should be extended to other operations. This requires for the tested implementation that the average performance gain achieved by the evaluation procedure must outweigh the overhead introduced by the prefilter procedure during query planning (see Chapter 5).

5. Empirical Analysis

The objective of this chapter is to assess the performance gain achieved through the prefilter procedure, while we expect that its overhead during query planning is outweighed by the gain achieved. Thus, our goal is also to determine whether a net performance gain is achieved on average in the context of query evaluation and planning. We perform a benchmark using ten representative queries. Some queries used for the benchmark contain highly restrictive `FILTER` expressions (e.g., checking for equality (`=`)), while others contain less restrictive `FILTER` expressions (e.g., checking for inequality (`!=`)). We also assess the performance for datasets of different sizes. The Olympics knowledge graph [5] serves as an evaluation reference for smaller datasets, whereas Wikidata [1] is used as a reference for larger datasets. The Olympics dataset consists of approximately 1.8 million RDF triples, and the Wikidata dataset consists of approximately 20 billion triples. This is done to assess the performance gain relative to the size of the knowledge graph, as this variable may influence the effectiveness of the introduced prefilter procedure.

Note that this analysis represents the least extensive prefilter procedure, as prefilter expressions are only applied if the corresponding `INDEX SCAN` operation is a direct child of the `FILTER` expression. This is because no further adjustments were made to the query planning strategy given the reasons discussed in Chapter 4.4.3.

5.1. Benchmark

The runtime for each query has been obtained through the QLever User Interface¹, which provides a detailed analysis. Each query and its corresponding execution tree is provided in Section A of the Appendix. The queries were executed three times under both conditions: with the prefilter procedure code enabled and with it disabled (our baseline). The server was restarted before computing each query. Table 7 lists the total times required for query computation, together with the times for the individual steps of query planning and evaluation. In addition, the times of the

¹<https://github.com/ad-freiburg/qllever-ui>

evaluation steps **FILTER** and **INDEX SCAN** are provided, representing the evaluation steps directly affected by the introduced prefilter procedure. Since each query was executed three times, the provided times are the average of the three runs. The last column provides the number of rows scanned by the **INDEX SCAN** operation associated with the filter-related variable.

The QLever server was hosted on machine @prut, equipped with an AMD Ryzen 9 7950X 16-Core Processor and a total of 124 GiB of RAM. The Olympics index used for benchmarking was built based on default settings. The corresponding Wikidata index was used as already provided on machine @prut.

Query	Prefiltering enabled	Query Comput. (ms)	Query Planning (ms)	Query Eval. (ms)	Filter Eval. (ms)	Index Scan Eval. (ms)	Num. Rows Scanned
Olympics Dataset							
Q1 (A.1)	No	77.67	2	75.67	73.33	1.67	125,000
	Yes	20.67	2	18.67	17	1	31,250
Q2 (A.2)	No	17.67	2	15.67	3.67	2	180,604
	Yes	6.33	2	4.33	0	0.67	24,354
Q3 (A.3)	No	8.67	2	6.67	1	1.33	139,277
	Yes	7.67	2	5.67	0	1.67	93,750
Q4 (A.4)	No	17	2	15	4	3	139,277
	Yes	17	2	15	4.33	2.67	139,277
Q5 (A.5)	No	6.67	1.33	5.33	0	5	1,781,625
	Yes	4.67	1.67	3	0	2	136,153
Wikidata Dataset							
Q6 (A.6)	No	151.67	16.33	135.33	71.33	57	91,582,414
	Yes	24	16.67	7.33	0	0.67	31,250
Q7 (A.7)	No	270.67	22	248.67	1	20.66	11,558,486
	Yes	217	20.67	193.33	0	2	89,736
Q8 (A.8)	No	222.33	20.33	202	9.33	10	48,931,776
	Yes	145.33	27.33	118	7.33	2	306,776
Q9 (A.9)	No	timeout (60s)	-	-	-	-	-
	Yes	931.66	277.66	654	643.67	10	1,076,194
Q10 (A.10)	No	4,510.67	72.33	4,438.33	1,072	3,365.67	7,009,302,883
	Yes	164.33	149.33	15	0	15	31,250

Table 7.: Benchmark results on the Olympics and Wikidata datasets, with and without prefiltering. The times are presented in milliseconds (ms). The benchmark queries and their corresponding execution trees are provided in Section A of the Appendix.

5.2. Observations

We now analyze the benchmark results presented in Table 7. The prefilter procedure significantly reduces the evaluation overhead of the **FILTER** expression (see column **Filter Eval.**). This is because the result of the **INDEX SCAN** operation consists of fewer rows (see column **Num. Rows Scanned**), which also reduces the evaluation time required for the **INDEX SCAN** operation (see column **Index Scan Eval.**).

We also observe that query Q4 represents an exception, as the planner discards the execution tree that includes the applied prefilter. This is illustrated by the query execution tree with the prefilter procedure enabled, as shown in Figure 15. This tree is identical to the one generated when the prefilter procedure is disabled (see Figure 14). For more complex queries involving **JOIN** operations, the successful prefilter procedure results in a pushdown of the **FILTER** clause down to the **INDEX SCAN** operation. This structure is required for a successful prefilter procedure, as the implementation only applies a prefilter expression when **INDEX SCAN** is a direct child of a **FILTER** clause. This structure can be observed for optimized execution trees, such as in the case of query Q2 with Figure 11 and query Q8 with Figure 23. In both cases, the **FILTER** clause appears as a child of the **SORT** operation, which precedes the **JOIN** operation. The reason for this planning strategy is already explained in Section 4.4.3, where we examined a similar strategy for deep prefilter pushdowns instead of the already applied **FILTER** clause pushdown. Successfully applying a prefilter expression for our benchmark queries requires a sorted **object** column, whereas the **JOIN** operation requires the **subject** column to be sorted (for a fixed **predicate**). For query Q4, the planner presumably discards the execution tree with a pushed-down **FILTER** clause because cost estimates show that the **FILTER** expression is not selective enough, since the prefilter removes too few index blocks (and thus rows).

For all queries except Q4, fewer rows are scanned. As a result, both the **INDEX SCAN** and subsequent **FILTER** evaluations are faster (see columns **Index Scan Eval.** and **Filter Eval.**), leading to improved overall evaluation times (see **Query Eval.**). For more complex queries, such as query Q7 and query Q8, the directly related **JOIN** operations also experience a performance gain. This can be observed when comparing their optimized and non-optimized execution trees (compare Figure 20 vs. Figure 21, and Figure 22 vs. Figure 23). The evaluation times of these two queries are reduced from 248.67 to 193.33 ms and 202 to 118 ms overall, respectively. Note that this improvement is not fully attributable to the gains in **FILTER** and **INDEX SCAN** performance alone, as their combined time savings do not add up to the observed

speedup. Thus, in case of complex queries, the prefilter procedure potentially also speeds up the evaluation procedure of downstream operations.

The time reduction is particularly significant when the query contains a **FILTER** expression that requires direct checking of string values, such as the queries Q1, Q6, and Q9, which involve the evaluation of the expressions **STRSTARTS** and **REGEX**. For queries Q1 and Q6, the query evaluation time is reduced from 75.67 to 18.67 ms and from 135.33 to 7.33 ms, respectively. The benchmark results also highlight significant performance gains for queries on a large index containing highly restrictive **FILTER** expressions, while previously requiring scanning large portions of the overall index. This can be observed for queries Q9 and Q10, where the query evaluation time is reduced from more than 60 seconds to 654 ms and from 4438.33 ms to 15 ms, respectively.

For all queries for which the prefilter was successfully performed, the time required for the (total) query computation (see column **Query Comput.**) is reduced. This appears to hold even when the prefilter procedure results in only a small reduction in the number of scanned rows, as indicated by query Q3. The time required for query computation is the sum of the time needed for query planning and query evaluation. Thus, our assumption is correct that the prefilter overhead during query planning is outweighed by the performance gain during query evaluation.

As discussed above, query Q4 represents an exception, since the query planner discards the execution tree where the prefilter expression has been applied. As a consequence, there is no performance gain achieved during query evaluation, but the compute overhead during query planning remains. However, this overhead appears to be too small to measure in most cases (at least through the QLever UI), since the values in the columns **Query Planning** and **Query Comput.** remain identical for query Q4. These observations are promising, indicating that the prefilter procedure is rather efficient and introduces only minimal compute overhead during query planning for typical queries.

6. Conclusion

The conducted work demonstrates that the prefilter procedures introduced for **FILTER** expressions are an effective optimization approach. The previous evaluation overhead, as discussed in Section 1.5 has been successfully reduced. This is demonstrated by the benchmark results presented in Table 7. Moreover, its results indicate that the prefilter procedure itself remains mostly efficient, with the performance gain achieved during query evaluation outweighing its introduced overhead during query planning. Because the prefilter expression is applied before the actual evaluation, the performance of the **INDEX SCAN** is also improved. As the benchmark shows, the prefilter procedure not only enhances the evaluation performance of the **FILTER** expression by smaller intermediate results. In addition, some downstream operations, particularly **JOIN** operations, can also benefit. These observations suggest that the deep pushdown of prefilter expressions, as discussed in Section 4.4.3, may also be beneficial.

The **FILTER** expressions now mostly evaluate across index blocks that overall contain values that satisfy the expression. To be more precise, only the bounding index blocks of the prefiltered Index Block Metadata range may contain values that do not satisfy the **FILTER** expression. Thus, if the **FILTER** expression is not highly restrictive, it will evaluate non-bounding index blocks in which all rows satisfy the expression. It is safe to assume that all their rows are contained in the **FILTER** expression result. Evaluating the values of non-bounding index blocks represents an evaluation overhead itself. A potential solution is to evaluate the bounding index blocks already during the evaluation of the **INDEX SCAN** operation, eliminating non-satisfying values. Consequently, evaluating the expression of the corresponding **FILTER** clause in the execution structure can be omitted. Its input can be routed directly to the next operation downstream, as an intermediate result. This approach would further enhance the evaluation performance of queries for which the prefilter expression yields non-bounding blocks.

7. Acknowledgments

First and foremost, I would like to thank my advisor, Johannes Kalmbach, for his invaluable advice, the many code reviews and testing, the patience that was sometimes required, and for holding me accountable to a high standard. Also, a big thanks to my examiner, Prof. Dr. Hannah Bast, for also testing some implementations as early as the development phase, and in particular for the opportunity to contribute to a meaningful and interesting project. This was really motivating, and I learned a lot during the completion of this work. I would also like to thank my parents for their continuous support and for providing a stable environment throughout my studies.

A. Benchmark Queries and Their Execution Trees

This chapter lists all ten benchmark queries used for the evaluation in Section 5.1. All queries are presented with their two corresponding execution trees. The first tree shows the execution order for the baseline reference, while the second tree shows the execution order with the prefilter procedure enabled.

A.1. Query 1

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?athlete_id ?athlete_name WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  FILTER STRSTARTS(?athlete_name, "Sebast")
}
LIMIT 5
```

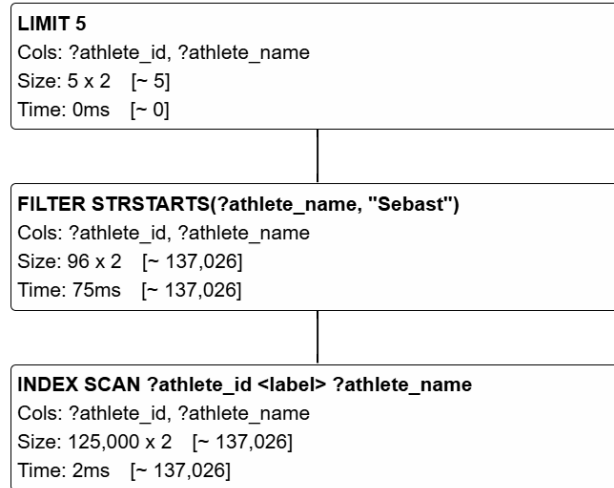


Figure 8.: Execution Tree Benchmark Query 1 (baseline)

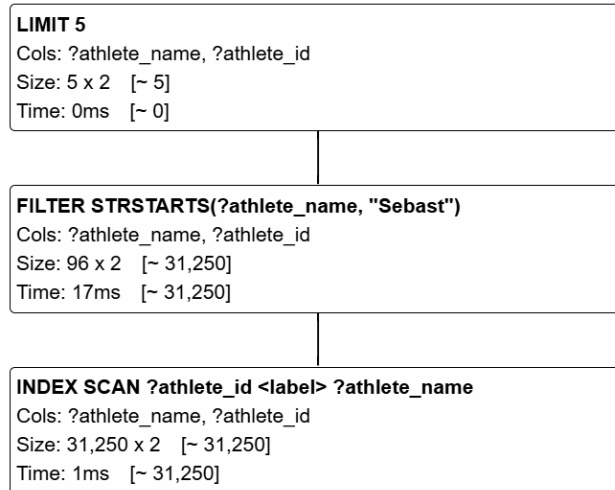


Figure 9.: Execution Tree Benchmark Query 1 (with prefilter)

A.2. Query 2

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?athlete_id ?athlete_name ?athlete_age WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  ?athlete_id foaf:age ?athlete_age .
  FILTER(?athlete_age >= 60 && ?athlete_age <= 65)
}
LIMIT 5
  
```

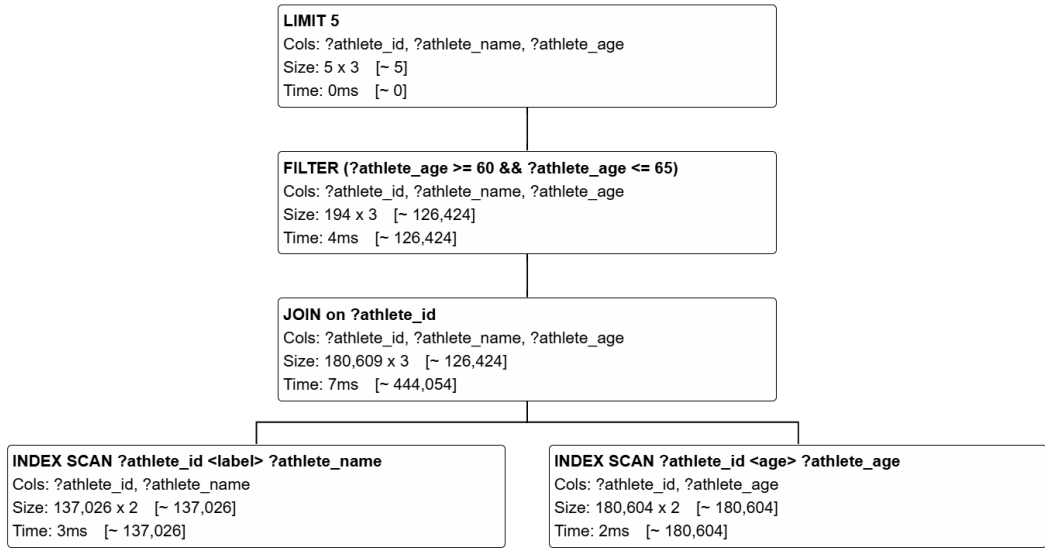


Figure 10.: Execution Tree Benchmark Query 2 (baseline)

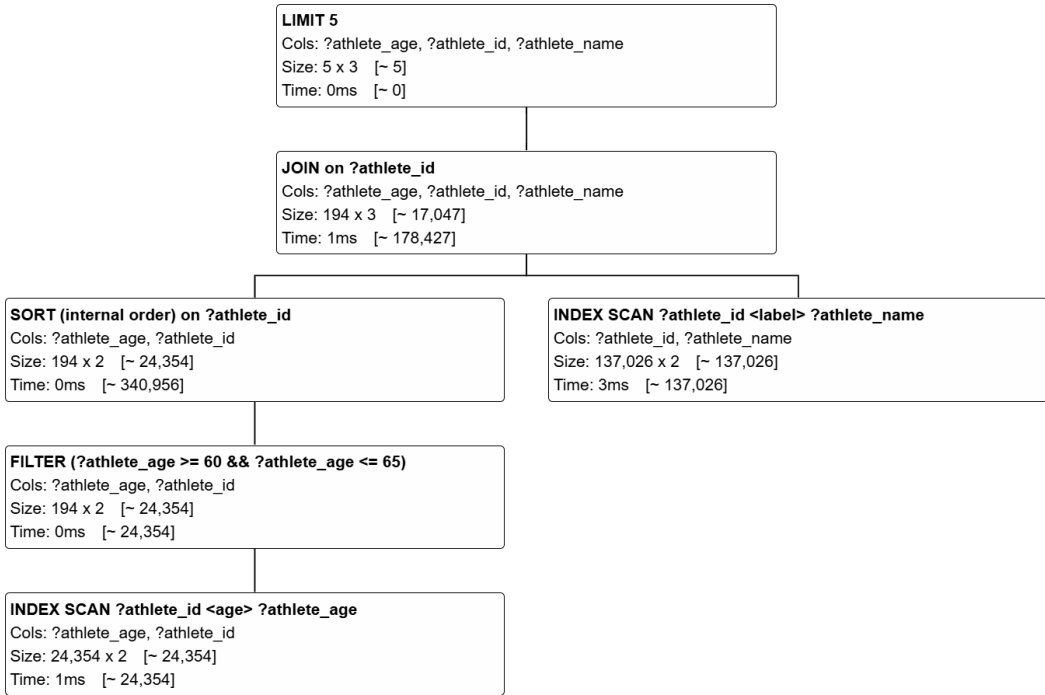


Figure 11.: Execution Tree Benchmark Query 2 (with prefilter)

A.3. Query 3

```

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX team: <http://wallscope.co.uk/resource/olympics/team/>
SELECT ?athlete_id ?athlete_name ?athlete_team WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  ?athlete_id dbo:team ?athlete_team .
  FILTER (?athlete_team IN (team:Germany, team:Spain, team:
    Netherlands, team:Switzerland))
}

```

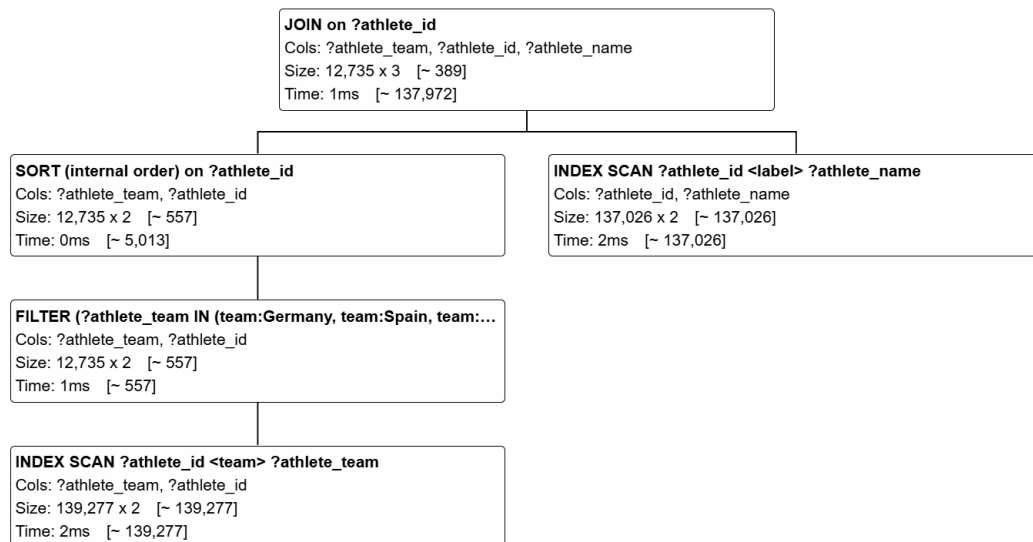


Figure 12.: Execution Tree Benchmark Query 3 (baseline)

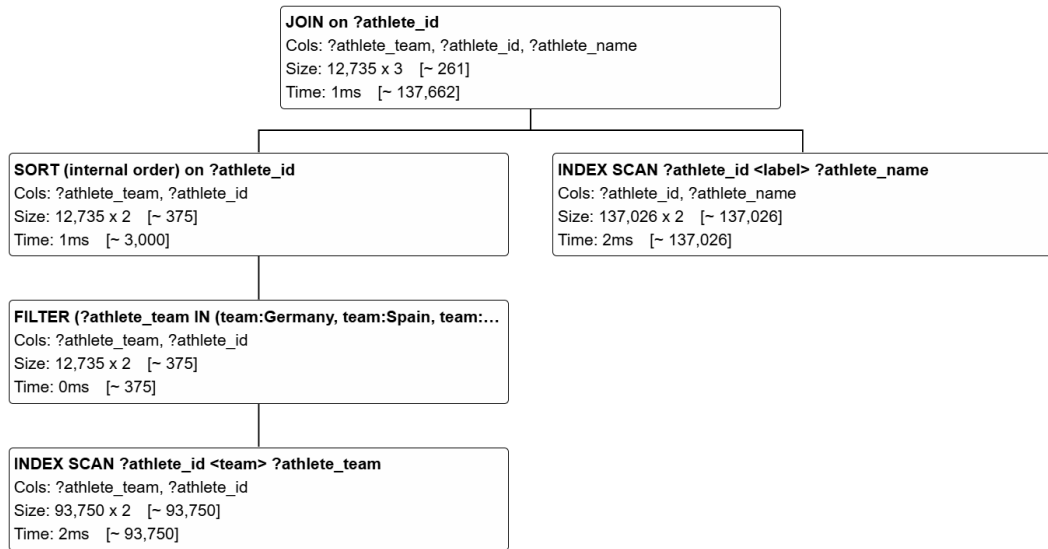


Figure 13.: Execution Tree Benchmark Query 3 (with prefilter)

A.4. Query 4

```

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX team: <http://wallscope.co.uk/resource/olympics/team/>
SELECT ?athlete_id ?athlete_name ?athlete_team WHERE {
  ?athlete_id rdfs:label ?athlete_name .
  ?athlete_id dbo:team ?athlete_team .
  FILTER (?athlete_team NOT IN (team:France, team:Poland))
}

```

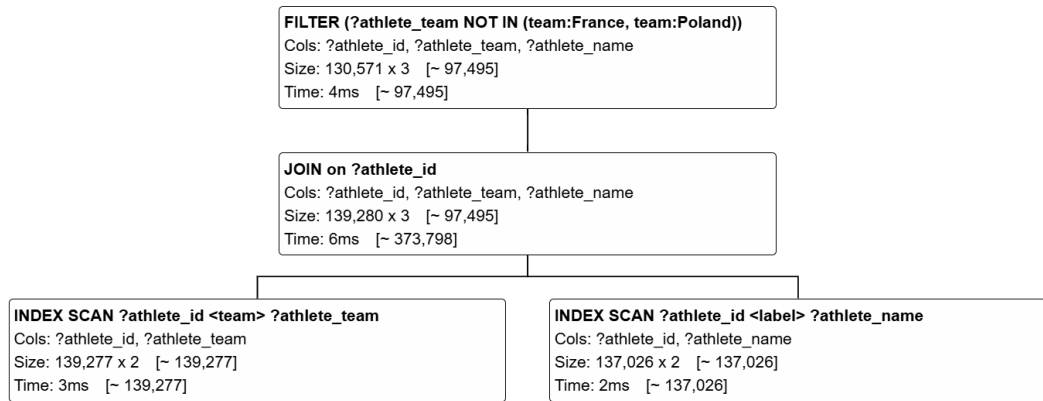


Figure 14.: Execution Tree Benchmark Query 4 (baseline)

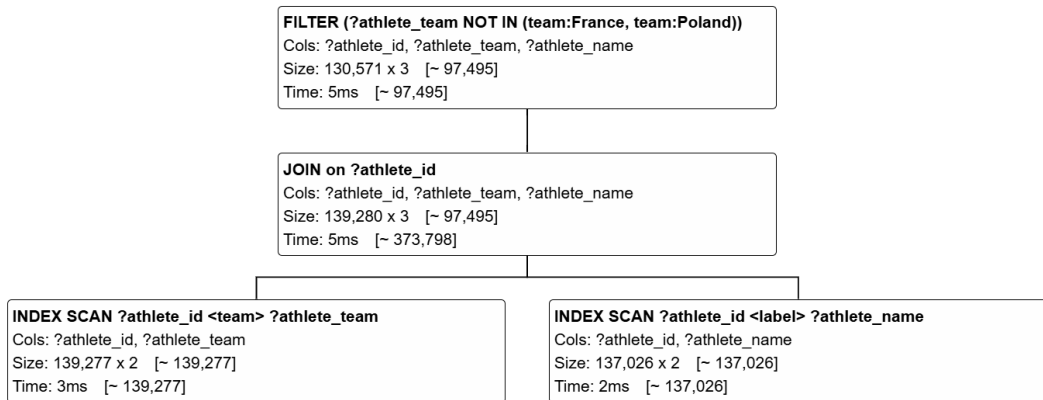


Figure 15.: Execution Tree Benchmark Query 4 (with prefilter)

A.5. Query 5

```

PREFIX medal: <http://wallscope.co.uk/resource/olympics/medal/>
SELECT * WHERE {
    ?x0 ?x1 ?x2 .
    # object is gold medal
    FILTER (?x2 = medal:Gold)
}

```

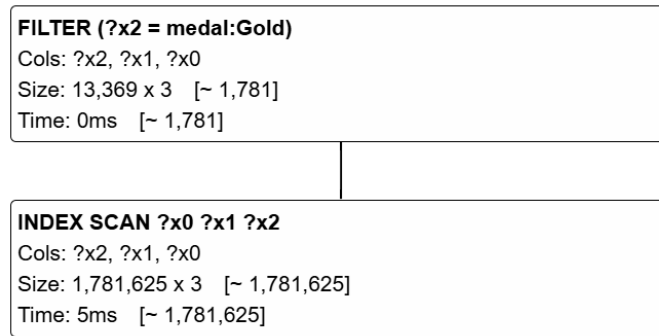


Figure 16.: Execution Tree Benchmark Query 5 (baseline)

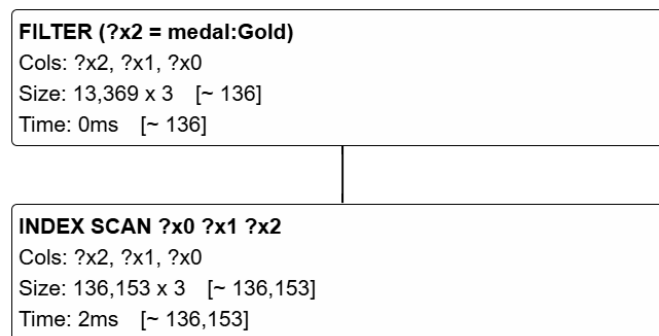


Figure 17.: Execution Tree Benchmark Query 5 (with prefilter)

A.6. Query 6

```

PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?person_id ?person_name WHERE {
  ?person_id wdt:P31 wd:Q5 .
  ?person_id skos:prefLabel ?person_name .
  # occupation tennis player
  ?person_id wdt:P106 wd:Q10833314 .
  FILTER (LANG(?person_name) = "en")
  FILTER REGEX(?person_name, "^Roger")
}
  
```

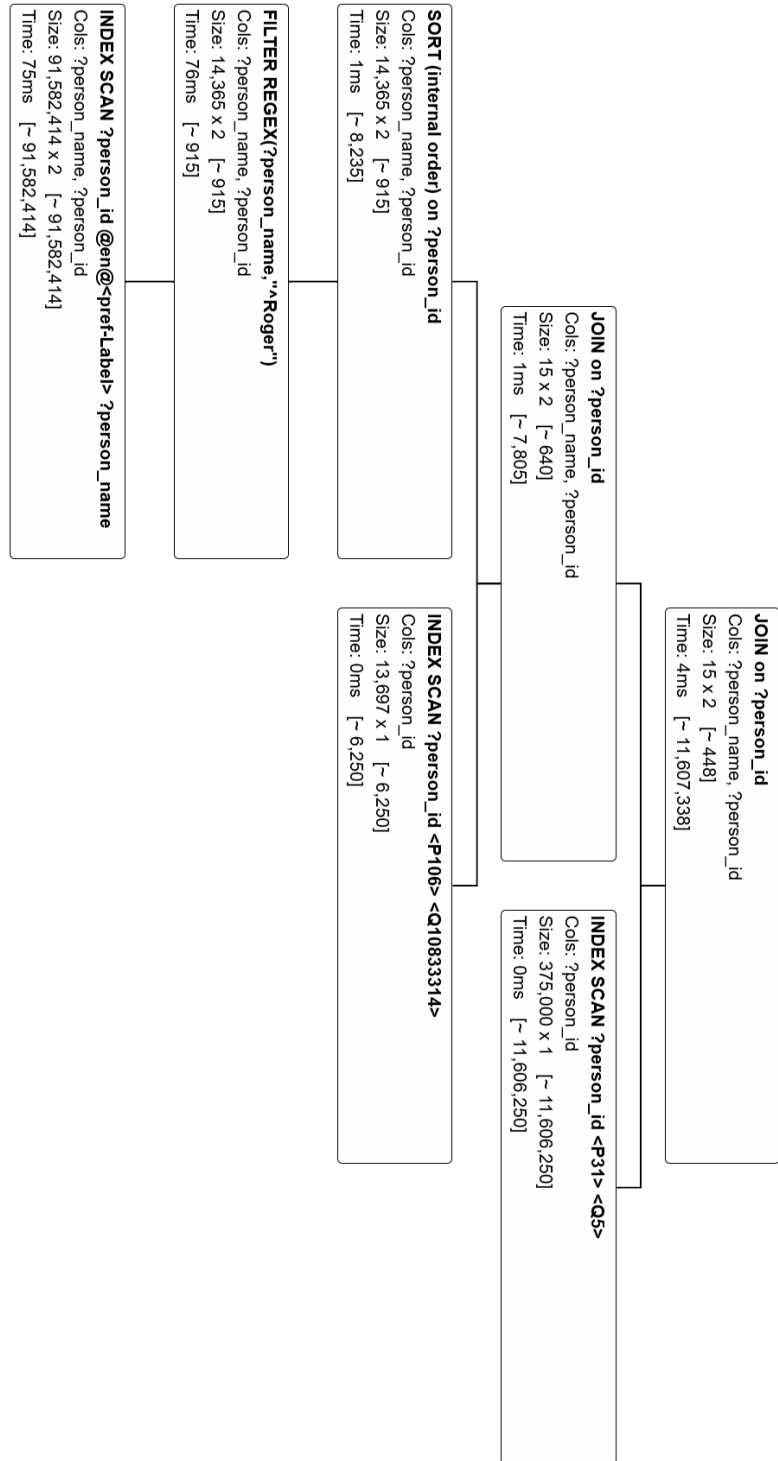


Figure 18.: Execution Tree Benchmark Query 6 (baseline)

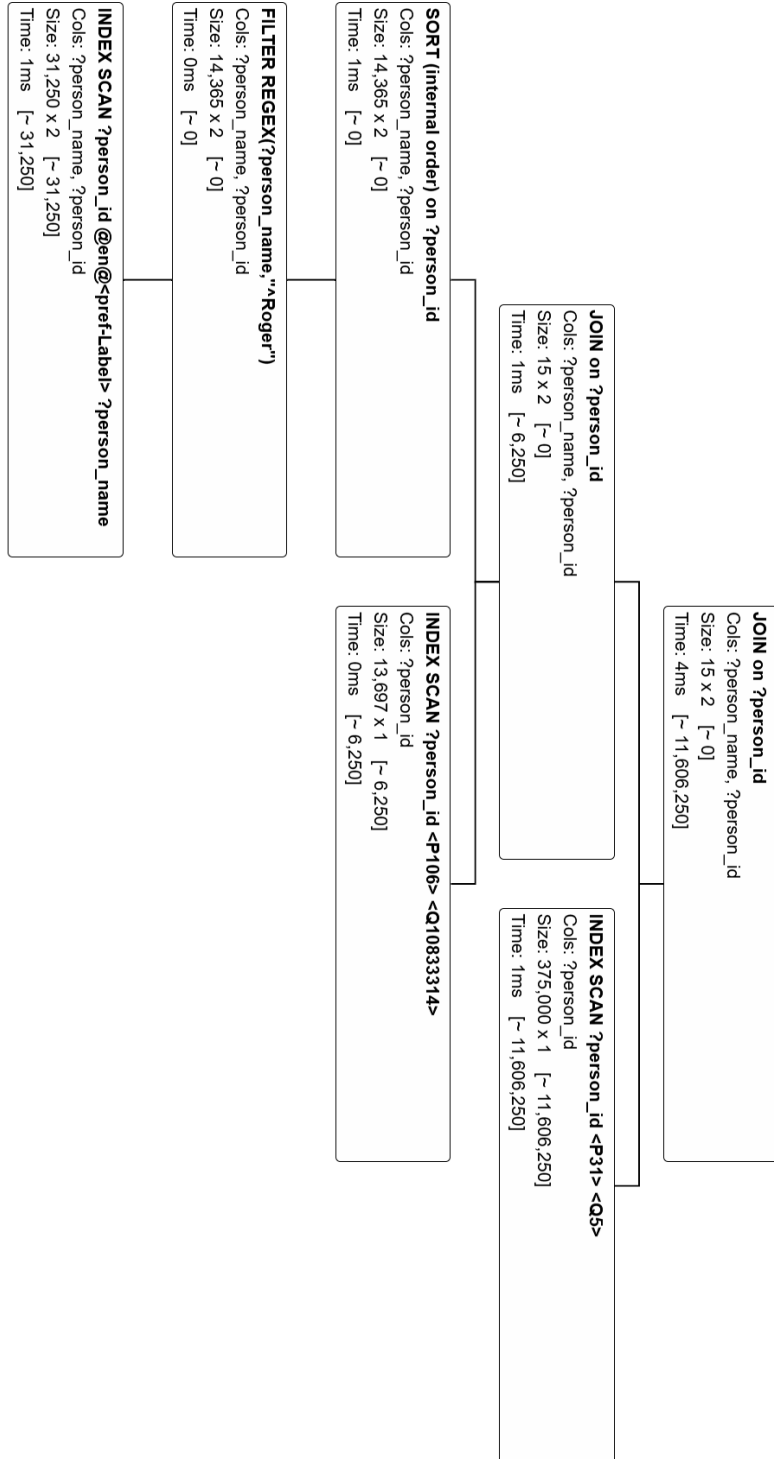


Figure 19.: Execution Tree Benchmark Query 6 (with prefilter)

A.7. Query 7

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?person_id ?person_name ?occupation WHERE {
    ?person_id wdt:P31 wd:Q5 .
    ?person_id skos:prefLabel ?person_name .
    ?person_id wdt:P106 ?occupation .
    FILTER (LANG(?person_name) = "en") .
    # filter occupation is tennis player, f1 driver or
    # professional golfer
    FILTER (?occupation IN (wd:Q10833314, wd:Q10841764, wd:
        Q490253))
}
```

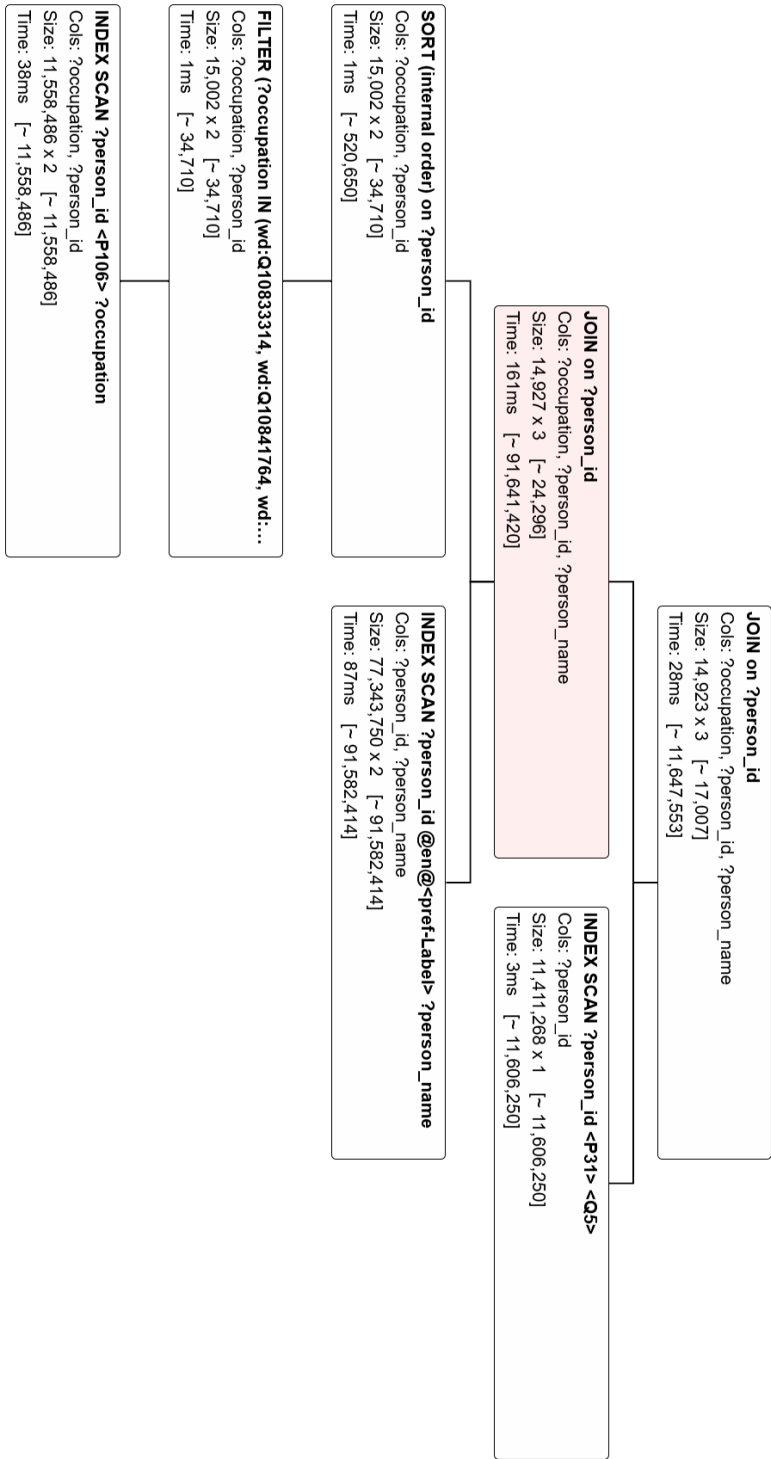


Figure 20.: Execution Tree Benchmark Query 7 (baseline)

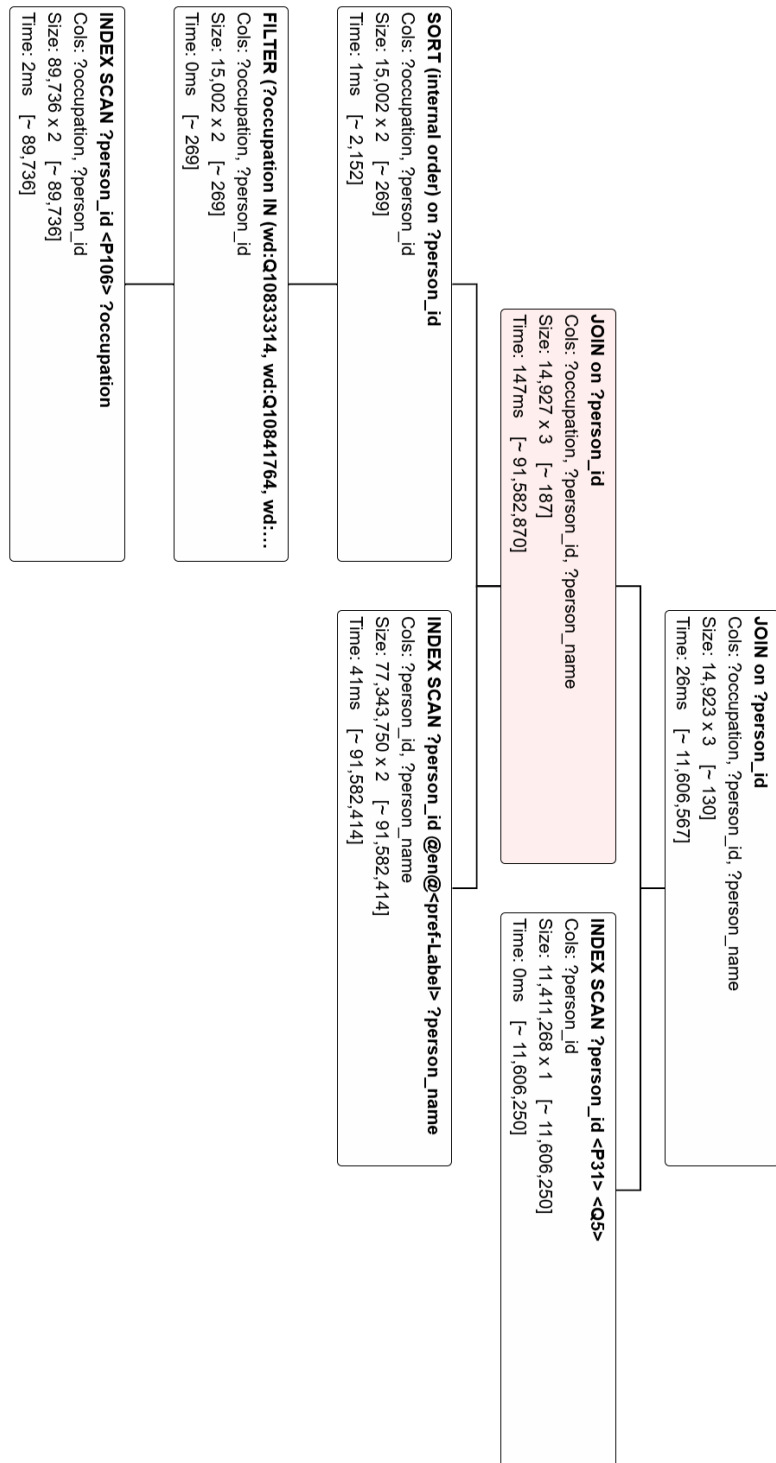


Figure 21.: Execution Tree Benchmark Query 7 (with prefilter)

A.8. Query 8

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?film_id ?film_name ?pubdate WHERE {
  ?film_id wdt:P31 wd:Q11424 .
  ?film_id wdt:P577 ?pubdate .
  # filter for films that were published after 2023
  FILTER (YEAR(?pubdate) > 2023) .
  ?film_id rdfs:label ?film_name .
  FILTER (LANG(?film_name) = "en") .
}
ORDER BY ASC(?pubdate)
```

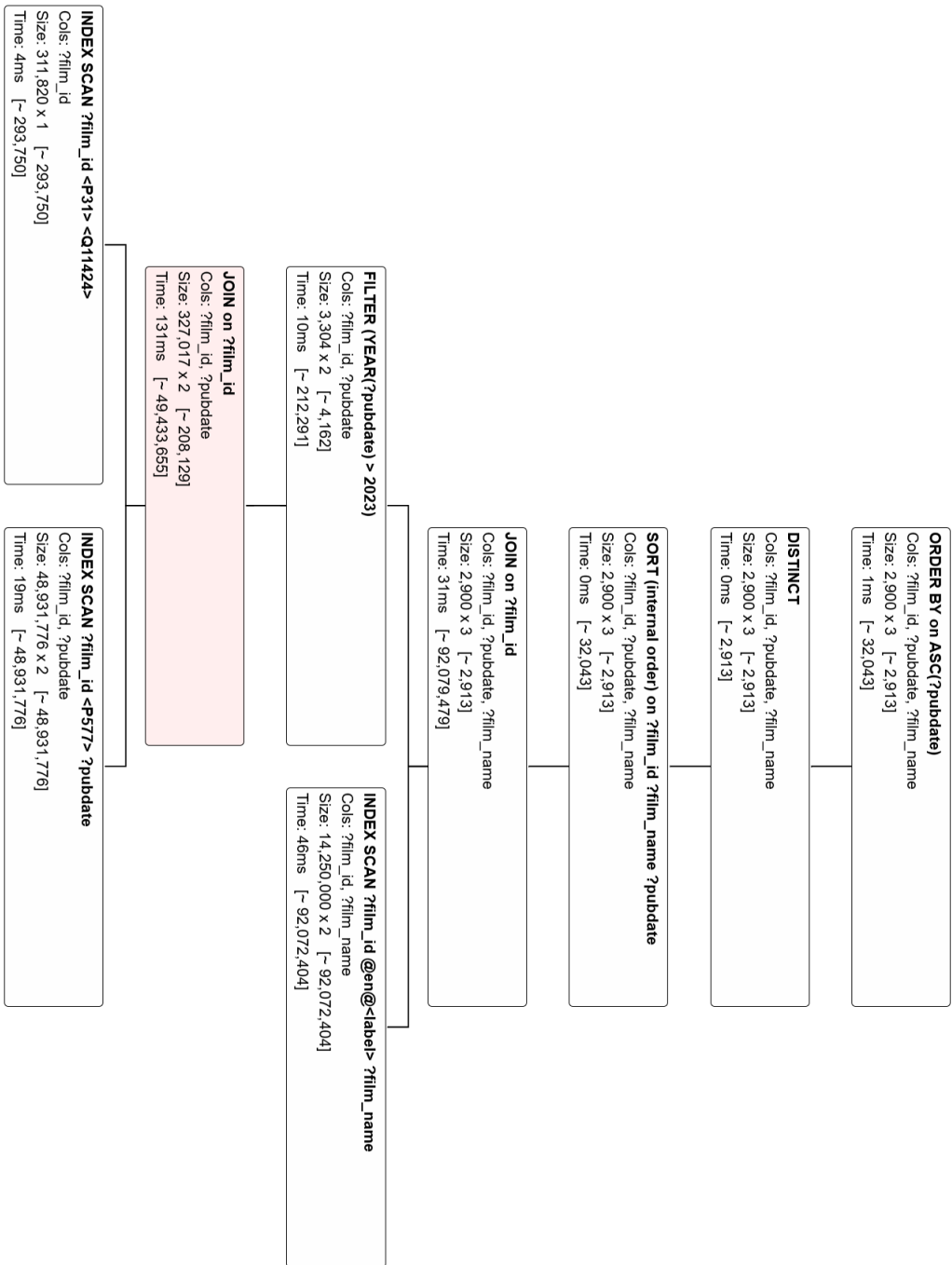


Figure 22.: Execution Tree Benchmark Query 8 (baseline)

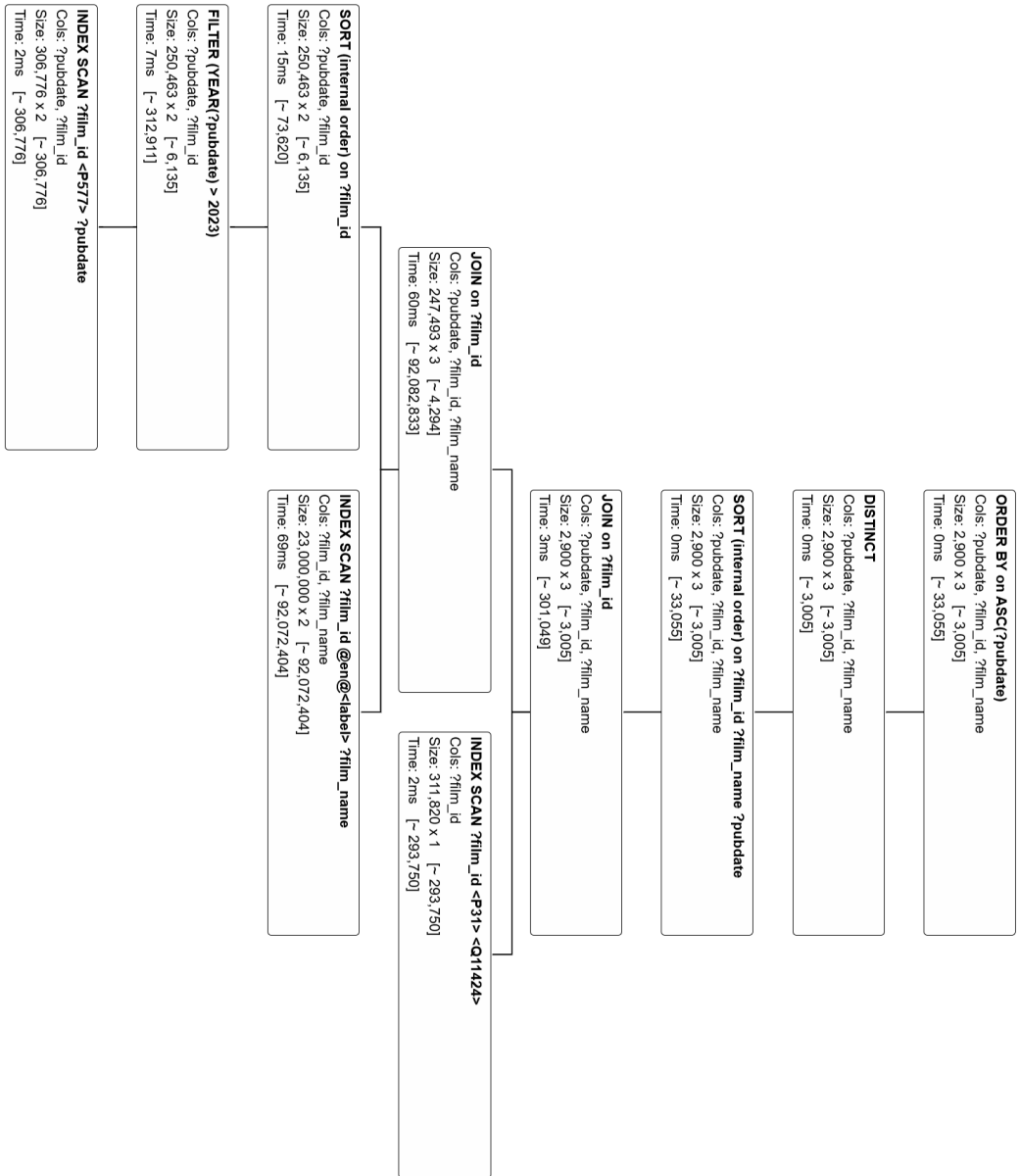


Figure 23.: Execution Tree Benchmark Query 8 (with prefilter)

A.9. Query 9

```
SELECT * WHERE {  
  ?x0 ?x1 ?x2 .  
  FILTER STRSTARTS(?x2, "Bob")  
}
```

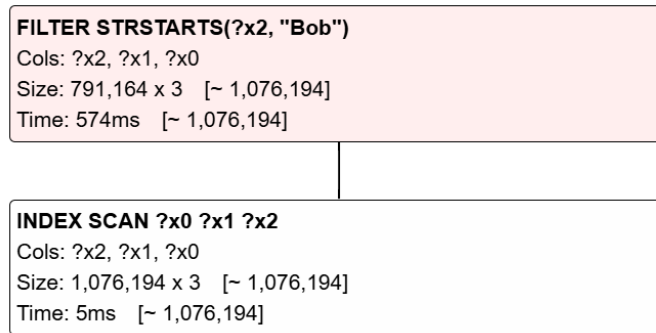


Figure 24.: Execution Tree Benchmark Query 9 (with prefilter)

A.10. Query 10

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>  
SELECT * WHERE {  
  ?x0 ?x1 ?x2 .  
  FILTER (?x1 IN (rdf:type, skos:prefLabel))  
}  
LIMIT 10
```

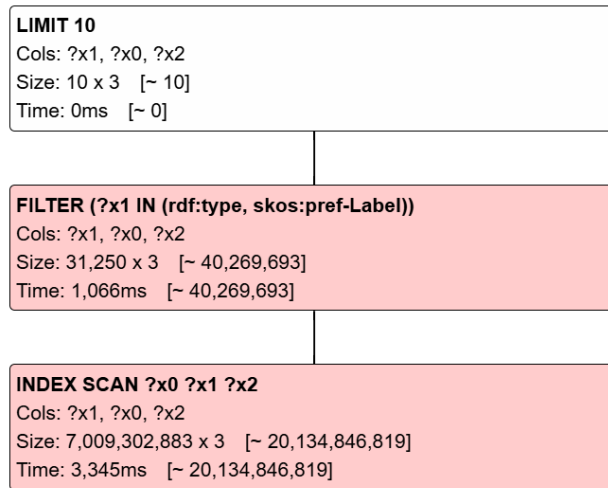


Figure 25.: Execution Tree Benchmark Query 10 (baseline)

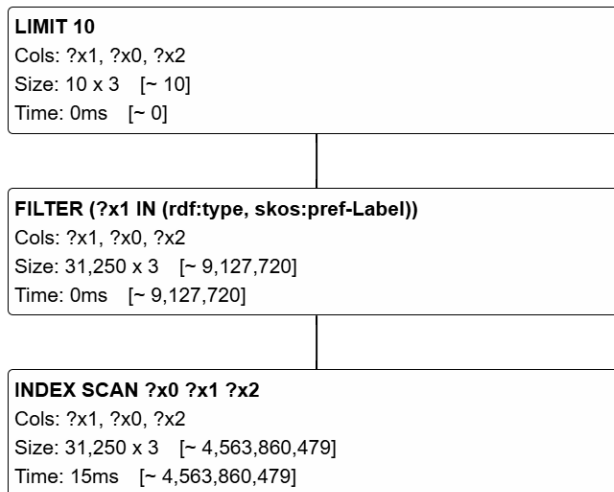


Figure 26.: Execution Tree Benchmark Query 10 (with prefilter)

Bibliography

- [1] W. Contributors, “Wikidata: Main page,” 2025. [Online]. Available at: <https://www.wikidata.org/>.
- [2] S. Harris, A. Seaborne, and E. Prud’hommeaux, “Sparql 1.1 query language.” W3C Recommendation, 2013. [Online]. Available at: <https://www.w3.org/TR/sparql11-query/>.
- [3] T. U. Consortium, “Uniprot: The universal protein resource,” 2025. [Online]. Available at: <https://www.uniprot.org/>.
- [4] G. Fu, C. Batchelor, M. Dumontier, J. Hastings, E. Willighagen, and E. Bolton, “Pubchemrdf: towards the semantic annotation of pubchem compound and substance databases,” *Journal of Cheminformatics*, 2015. [Online]. Available at: <https://doi.org/10.1186/s13321-015-0084-4>.
- [5] W. Ltd, “Olympics linked data: Github repository,” 2025. [Online]. Available at: <https://github.com/wallscope/olympics-rdf>.
- [6] H. Bast and B. Buchhold, “Qlever: A query engine for efficient sparql+text search,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM)*, pp. 647–656, ACM, 2017. [Online]. Available at: <https://dl.acm.org/doi/10.1145/3132847.3132921>.
- [7] H. Bast, J. Kalmbach, T. Klumpp, and C. Korzen, “Knowledge graphs and search,” in *Information Retrieval*, pp. 231–281, ACM, 2023. [Online]. Available at: <https://dl.acm.org/doi/10.1145/3674127.3674134>.
- [8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1383–1394, ACM, 2015. [Online]. Available at: <https://dl.acm.org/doi/10.1145/2723372.2742797>.

- [9] Y. Lin, C. Yan, and Y. He, “Predicate pushdown for data science pipelines,” in *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, ACM, 2023. [Online]. Available at: <https://dl.acm.org/doi/10.1145/3589281>.

