UNIVERSITÄT KOBLENZ · LANDAU
Faculty 4: Computer Science

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Distributed Query Optimization

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.)
in Informatik

submitted by
Samuel Gros

First supervisor:	Prof. Dr. Steffen Staab Institute for Web Science and Technologies
Second supervisor:	M. Sc. Daniel Janke Institute for Web Science and Technologies

Koblenz, January 2020

Statement

I hereby certify that this thesis has been composed by me and is based on my own work, that I did not use any further resources than specified – in particular no references unmentioned in the reference section – and that I did not submit this thesis to another examination before. The paper submission is identical to the submitted electronic version.

	Yes	No
I agree to have this thesis published in the library.	☐	☐
I agree to have this thesis published on the Web.	☐	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐
The source code is available under a GNU Lesser General Public License (LGPLv3).	☐	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐

.....

(Place, Date) (Signature)

Zusammenfassung

Diese Arbeit befasst sich mit der Anfragenoptimierung in Graphdatenbanksystemen, welche sich aus mehreren Rechenknoten zusammensetzen. Ein Großteil solcher verteilten Graphdatenbanken optimiert Anfragen zentral und führt den optimierten Anfrageplan anschließend auf allen Rechenknoten aus. Da allerdings jeder Rechenknoten unterschiedliche Daten speichert, kann eine solche zentrale Optimierung auf manchen Rechenknoten zu einer ineffizienten Anfragenabarbeitung führen. Daher wird in dieser Arbeit ein *verteilter Optimierungsansatz* entwickelt und untersucht. Dieser Ansatz optimiert unter Berücksichtigung der gespeicherten Daten einen individuellen Anfrageplan auf jedem Rechenknoten. Dadurch wird versucht die Anzahl der Zwischenergebnisse, die während der Bearbeitung der Anfrage anfallen, zu reduzieren, um somit die Abarbeitung des Anfrageplans auf allen Rechenknoten zu beschleunigen. Ein Nebeneffekt des verteilten Optimierungsansatzes ist die Verdopplung bestimmter Zwischenergebnisse. Die Verarbeitung der duplizierten Zwischenergebnisse verursacht während der Anfragenabarbeitung einen Mehraufwand. Dementsprechend zeigt die Auswertung, dass die Leistungssteigerung des Optimierungsansatzes aufgrund dieses Mehraufwands nicht genau bestimmt werden kann.

Abstract

This thesis focuses on the query optimization of graph databases that are distributed over several compute nodes. Most of these distributed graph databases optimize the query centrally. The resulting optimized query plan is then executed on all compute nodes. Since the different compute nodes store different data items, the centrally optimized query plan may be inefficient on some compute nodes. To overcome this limitation, this thesis proposes and investigates a *distributed optimization approach* that optimizes an individual query plan for each compute node while considering the stored data. Thereby the presented approach aims to improve the query performance by speeding up the processing of the query plan on all slave nodes. A side effect of the distributed optimization approach is the duplication of certain intermediate results during query processing. The evaluation shows that the actual performance gain of the approach cannot be determined, given the additional workload caused by processing these duplicate intermediate results.

Contents

Page

1. Introduction **1**
- 1.1. Research Questions . . . 3
- 1.2. Outline . . . 5

2. Foundations **6**
- 2.1. Resource Description Framework . . . 6
- 2.2. SPARQL Syntax . . . 8
- 2.3. SPARQL Semantics . . . 10
- 2.4. RDF Store Koral . . . 13
  - 2.4.1. Graph Loading . . . 14
  - 2.4.2. Query Processing . . . 15
- 2.5. Problem Domain . . . 18

3. Related Work **22**
- 3.1. Query Optimization in Centralized RDF Stores . . . 22
- 3.2. Query Optimization in Distributed RDF Stores . . . 27
  - 3.2.1. Query Optimization in Federated RDF Stores . . . 27
  - 3.2.2. Query Optimization in Distributed RDF Stores . . . 28

4. Distributed Query Optimization **30**
- 4.1. Join Order Optimization . . . 30
- 4.2. Distributed Join Order Optimization . . . 34
- 4.3. Distributed Query Processing . . . 35
  - 4.3.1. Tracing of Variable Mappings . . . 36
  - 4.3.2. Routing of Traceable Variable Mappings . . . 37
  - 4.3.3. Finishing the Query Processing . . . 42

5. Evaluation **45**
- 5.1. Experimental Setup . . . 45
- 5.2. Evaluation of the Centralized Optimization Approach . . . 47
- 5.3. Evaluation of the Distributed Optimization Approach . . . 53
- 5.4. Discussion . . . 59

6. Conclusion **60**
- 6.1. Future Work . . . 61

A. Evaluation Queries **62**

1. Introduction

A commonly used data model to represent data is the *Resource Description Framework* (RDF) [13, 19]. The Resource Description Framework allows it to describe resources by making statements about them. These statements are called RDF triples. RDF data consists of several RDF triples and represents a directed graph with labeled nodes and edges.

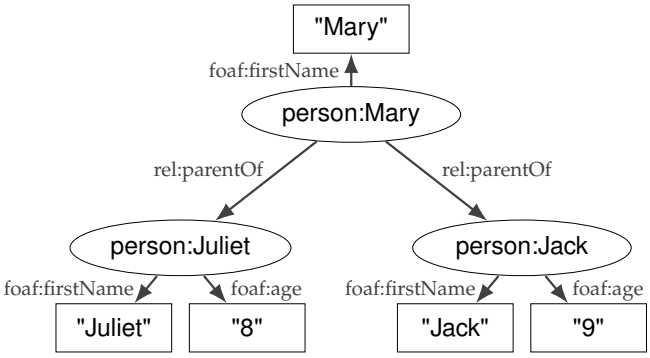


Figure 1.1.: Example of an RDF graph.

The RDF graph depicted in figure 1.1 provides information about the resources `person:Mary`, `person:Juliet` and `person:Jack`. All of the resources are assigned a first name, and in addition to that, the resources `person:Juliet` and `person:Jack` are assigned an age. Moreover, the graph describes the relationship of the family members, namely Mary is one parent of the children Juliet and Jack.

Specialized databases, called RDF stores, have been developed to store and manage such RDF data. To request specific data from an RDF store, the user can formulate queries using the standard query language SPARQL [6]. An example of a SPARQL query is presented in listing 1.1. This SPARQL query contains three triple patterns (see lines 5-7) and retrieves the name of a parent as well as the age of his/her child. Mary has two children, who are 8 and 9 years old, respectively. Accordingly, the query returns the following two results:

- "Mary", 8
- "Mary", 9

In order to properly scale RDF stores with the increasing capacity of RDF data that becomes available each year, distributed RDF stores have been developed [8]. Distributed RDF stores exceed the storage and query processing capabilities of regular

```
1. PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2. PREFIX rel: <http://purl.org/vocab/relationship/>

3. SELECT ?name ?age
4. WHERE {
5.   ?parent foaf:firstName ?name .  // Triple Pattern 1
6.   ?parent rel:parentOf ?child .   // Triple Pattern 2
7.   ?child  foaf:age ?age .         // Triple Pattern 3
8. }
```

Listing 1.1: Example of a SPARQL query.

centralized RDF stores by combining the computational power of several computers, which are employed as compute and storage nodes.

In general a distributed RDF store consists of a single master node and several slave nodes. The master node is responsible for coordinating the processing of queries, whereas every slave node stores a portion of the whole RDF graph. Figure 1.2 shows a possible distribution of the RDF graph from figure 1.1 among two slave nodes. In this scenario, the first slave node stores the RDF triples about the resources `person:Mary` and `person:Jack` and the second slave node stores the RDF triples about the resource `person:Juliet`.

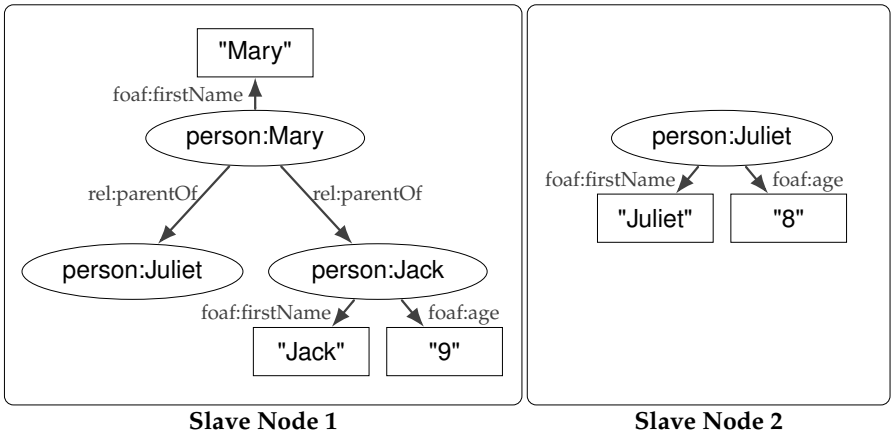


Figure 1.2.: Example of the distribution of the RDF graph from figure 1.1 among two slave nodes.

To answer a query request, a query plan which corresponds with the incoming request is created by the master node. The query plan specifies all operations necessary to produce the query results and the order in which the operations have to be executed. In a subsequent step the master node optimizes the created query plan by utilizing statistics about the stored RDF graph. Afterwards, the optimized query plan is passed to all slave nodes, and the slave nodes start processing the query plan in parallel.

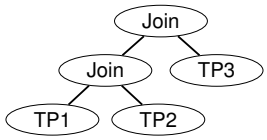
The structure of the query plan is crucial for the query performance of RDF stores. By adjusting the join order, it is often possible to reduce the number of intermediate results at an early stage of query processing. Consequently, fewer results have to be transferred and handled by the slave nodes to produce the final query results. In the setting of distributed RDF stores, centralized query optimization on the master node might be a suboptimal solution in order to improve query execution times. Since each slave node stores a different portion of the RDF graph, it is difficult to determine a single optimized query plan that benefits all slave nodes at once. Therefore, this thesis proposes a *distributed optimization approach* that optimizes the query plan of every single slave node individually. The optimization process takes the distribution of the RDF graph into account. Accordingly, this approach may reduce the query execution time since a query plan optimized for the local data may lead to a reduced number of intermediate results.

1.1. Research Questions

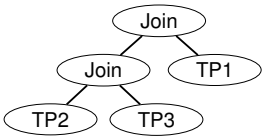
In the context of developing the *distributed optimization approach*, the following research questions arise:

Research Question 1 How to optimize the query plan distinctly for each slave node?

Consider the query plans shown in figure 1.3. These query plans correspond to the SPARQL query in listing 1.1. Each of the two query plans defines a different join order. Accordingly, the results of the three triple patterns are combined in a different order when processing either of these query plans. Instead of assigning either of the query plans to both slave nodes of the RDF store, the distributed optimization approach might determine that processing query plan (b) on the first slave node and query plan (a) on the second slave node produces the least intermediate results on both slave nodes combined.



a) Query plan whose first join joins the results of triple pattern 1 (TP1) and triple pattern 2 (TP2) and whose second join joins the results of the first join with the results of triple pattern 3 (TP3)



b) Query plan whose first join joins the results of triple pattern 2 (TP2) and triple pattern 3 (TP3) and whose second join joins the results of the first join with the results of triple pattern 1 (TP1)

Figure 1.3.: Query plans with different join order.

Research Question 1.1 How can the optimization of the query plan be performed in a distributed manner?

In order to distribute the optimization among all slave nodes, an initial query plan is constructed on the master node when a query request is received. Afterwards, the initial query plan is transferred to all slave nodes. This query plan is based solely on the structure of the incoming query. Upon receiving the query plan, each slave node optimizes it independently according to statistics about its local RDF storage.

Research Question 1.2 How can intermediate results from different slave nodes be combined?

The scenario in *Research Question 1.1* describes how the optimization of the query plan can be distributed among the slave nodes. The intermediate results produced on different slave nodes have to be combined to obtain the final query results. For this purpose, the intermediate results are usually forwarded to the subsequent operation defined by the query plan. However, since each of the slave nodes may potentially process a different query plan, the intermediate results have to be forwarded to operations at different positions in the query plan. Thus, it is not possible to process these query plans sequentially. In the described setting, each slave node only possesses knowledge about its own query plan. Therefore, identifying the operations that the intermediate results have to be forwarded to remains a challenge.

Research Question 1.3 When is the processing of a query finished?

As described in the scenario of *Research Question 1.2*, the query plans cannot be processed in sequential order anymore. As a consequence, it is more difficult to determine whether a query request has been processed entirely and whether all query results have been found. Since the regular approach to detect finished queries is no longer applicable, an alternative method has to be developed.

Research Question 2 How to evaluate the distributed optimization approach?

The performance of the distributed optimization approach can be compared to the performance of other systems in order to evaluate how effective it is. The other systems serve as baselines for the evaluation. As the first baseline, the default query processing strategy of the distributed RDF store Koral[1] can be used. The first baseline can only be used to determine if the optimization approach achieved any performance gain. A system, which optimizes a single query plan on the master node and processes the same optimized query plan on all slave nodes, constitutes the second baseline. The second baseline is more strict and allows to assess whether optimizing the query plan distinctly for each slave node is beneficial compared to more established centralized optimization approaches.

[1] https://github.com/Institute-Web-Science-and-Technologies/koral [Accessed: 18 December 2019]

1.2. Outline

The remainder of this thesis is structured as follows. In section 2 fundamental background knowledge regarding the RDF data model, the SPARQL query language and the distributed RDF store Koral is introduced. Section 3 provides an overview of related work in the context of SPARQL query optimization. In section 4 the developed distributed optimization approach is presented. Section 4.2 describes the distributed join order optimization that is used to optimize the query plan of each slave node individually. Thereby *Research Question 1* and *Research Question 1.1* can be answered. Section 4.3 details the adjustments that have been made to the query processing strategy of Koral in order to process potentially different query plans on the slave nodes. This section provides insight into the routing strategy that is used to forward intermediate results during query processing. Accordingly, an answer to *Research Question 1.2* is given. Furthermore, this section introduces a method to detect finished queries in the setting of the distributed optimization approach. This new method allows for answering *Research Question 1.3*. In section 5 the distributed optimization approach is evaluated in comparison to two baseline approaches. Moreover, the obtained results are presented and discussed to answer *Research Question 2*. Finally, section 6 summarizes the contributions of this thesis as well as possible directions for future work.

2. Foundations

This section focuses on providing background knowledge as well as formal definitions, which are fundamental for the remainder of this thesis. Section 2.1 introduces the *Resource Description Framework* (RDF) and the related terminology. In section 2.2 and section 2.3, the syntax and semantics of the relevant subset of the RDF query language SPARQL are defined. Section 2.4 describes the distributed RDF store *Koral*, how it loads RDF graphs and how it processes queries. Finally, section 2.5 describes the problem domain of join order optimization.

2.1. Resource Description Framework

The *Resource Description Framework* (RDF) is a data model that is used to represent data [13, 19]. Using RDF, statements can be made about resources in order to describe them. These statements are referred to as RDF triples and consist of a subject, a predicate and an object. In the context of RDF, resources are arbitrary entities that are unambiguously identified by an *Internationalized Resource Identifier* (IRI).

Example 2.1. Listing 2.1 shows an example of several RDF triples. This example depicts the five members of a family and the relationship between them. To represent the RDF triples TriG syntax [3] is used. `foaf`, `xsd`, `rel`, `ex`, `person` and `dog`, as defined in lines 1-6, serve as prefixes abbreviating full IRIs. For example, the prefixed name `person:Craig` is an abbreviation for `http://www.example.org/-person/Craig`. The first four RDF triples (lines 7-10) describe the resource referred to by the IRI `person:Craig`. The first triple, which is shown in line 7, has the subject `person:Craig`, the predicate `foaf:firstName` and the object `"Craig"`. This triple states that the first name of `person:Craig` is `"Craig"`. The fact that Craig is 33 years old is provided in line 8. As seen in lines 9 and 10, Craig is the father of `person:Juliet` and `person:Jack`. Similarly, it can be derived that Mary, age 34, is the mother of Juliet and Jack. Juliet, who is 8 years old, and Jack, who is 9 years old, are siblings. Furthermore, Craig and Mary are the owners of a dog named Merlin, who is 10 years old.

```
1. @prefix  foaf:   <http://xmlns.com/foaf/0.1/> .
2. @prefix  xsd:    <http://www.w3.org/2001/XMLSchema#> .
3. @prefix  rel:    <http://purl.org/vocab/relationship/> .
4. @prefix  ex:     <http://www.example.org/> .
5. @prefix  person: <http://www.example.org/person/> .
6. @prefix  dog:    <http://www.example.org/dog/> .
```

```
 7. person:Craig    foaf:firstName   "Craig" .
 8. person:Craig    foaf:age         "33"^^xsd:integer .
 9. person:Craig    rel:parentOf     person:Juliet .
10. person:Craig    rel:parentOf     person:Jack .

11. person:Mary     foaf:firstName   "Mary" .
12. person:Mary     foaf:age         "34"^^xsd:integer .
13. person:Mary     rel:parentOf     person:Juliet .
14. person:Mary     rel:parentOf     person:Jack .

15. person:Juliet   foaf:firstName   "Juliet" .
16. person:Juliet   foaf:age         "8"^^xsd:integer .
17. person:Juliet   rel:siblingOf    person:Jack .

18. person:Jack     foaf:firstName   "Jack" .
19. person:Jack     foaf:age         "9"^^xsd:integer .
20. person:Jack     rel:siblingOf    person:Juliet .

21. dog:Merlin      foaf:firstName   "Merlin" .
22. dog:Merlin      foaf:age         "10"^^xsd:integer .
23. dog:Merlin      ex:ownedBy       person:Craig .
24. dog:Merlin      ex:ownedBy       person:Mary .
```

Listing 2.1: Example of an RDF graph of a family with five members.

Definition 2.1 RDF Triple [14]**.**
An RDF triple (s, p, o) *is an element of the set* $(I \cup B) \times I \times (I \cup B \cup L)$*. I is an infinite set of IRIs, B is an infinite set of blank nodes, and L is an infinite set of literals. The sets I, B, L are pairwise disjoint.*

A formal definition of RDF triples is given in definition 2.1. In general RDF triples consist of three elements. The first element is the *subject*, the second element is the *predicate*, and the third element is the *object*. These elements are referred to as *triple components*.

The subject identifies the resource that is being described by the RDF triple. It can be either an IRI or a blank node. A blank node serves as the identifier of an anonymous resource. Unlike IRIs, blank nodes are not globally unique and only reference the same resource within the local scope of the RDF graph or the RDF store, in which they are defined [19].

The type of relationship is provided by an IRI in the predicate position of the RDF triple. In order to indicate a relationship between two resources, the object can be an IRI or a blank node. Line 9 of listing 2.1 shows a triple indicating the relationship `rel:parentOf` between the resources `person:Craig` and `person:Juliet`. Alternatively, literals can be used in the object position to provide additional information about the subject. An example of this can be seen in line 8 of listing 2.1. As described earlier, this triple provides information about the property `foaf:age` of the resource `person:Craig`.

Literals are strings which indicate typed data such as boolean values, numbers, strings, or time and date [19]. To specify how a literal string should be interpreted,

the literal is assigned a data type by providing a data type IRI. Line 8 of listing 2.1 displays an RDF triple whose object is assigned a data type. In this case the data type IRI `xsd:integer` indicates that the literal string `"33"` refers to the integer value 33.

Definition 2.2 RDF Graph & RDF Data Set [14]**.**
An RDF graph $G \subseteq ((I \cup B) \times I \times (I \cup B \cup L))$ *is a set of RDF triples. In the remainder of this thesis, an RDF graph may also be referred to as an RDF data set.*

As defined in definition 2.2, an RDF graph is a set of RDF triples. Consequently, the RDF triples in listing 2.1 form an RDF graph. RDF graphs are labeled directed graphs [2] and can easily be visualized. Subjects and objects of the RDF triples represent the set of labeled nodes. Each RDF triple represents a directed edge that leaves the subject node, enters the object node and is labeled with the predicate of the RDF triple. Figure 2.1 visualizes the RDF graph from listing 2.1. In this figure IRI nodes are visualized using ellipses, whereas literal nodes are represented by rectangles. Labeled edges are visualized as an arrow pointing from the subject node towards the object node of the corresponding RDF triple.

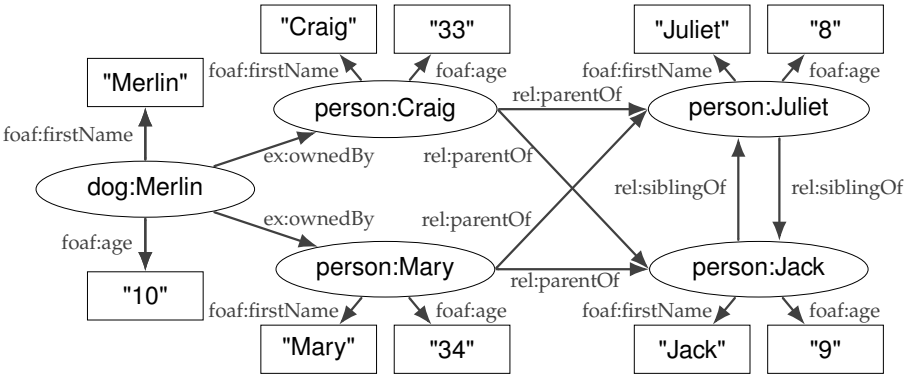


Figure 2.1.: Visualization of the RDF graph from listing 2.1.

2.2. SPARQL Syntax

RDF stores are databases specifically designed to store and manage RDF data. By means of queries, they allow the user to extract specific data from the stored RDF data set. The standard query language SPARQL 1.1 [6] is used to formulate these queries. In this section the syntax of the relevant subset of SPARQL 1.1 is defined. The definitions in sections 2.2 and 2.3 are based on the work of [2, 6, 14].

A SPARQL query consists of a query form and a pattern matching part. The *pattern matching part* consists of a SPARQL graph pattern. The graph pattern is matched against the stored RDF data set to obtain a set of results. The *query form* specifies how the results matching the graph pattern should be processed in order to produce the final query result. SPARQL supports four different query forms, namely `SELECT`,

ASK, CONSTRUCT and DESCRIBE [2]. However, only queries utilizing the SELECT query form are considered in this thesis. These queries are referred to as *SELECT queries*.

Example 2.2. Listing 2.2 shows a SPARQL SELECT query that returns all names of children who have a 34-year-old parent. Line 1 and 2 of listing 2.2 constitute the *query prologue* [6], which allows the definition of abbreviations that can be used in the query. In this case the abbreviations `foaf` and `rel` are defined. The actual query is located in lines 3-8. Line 3 indicates that the `SELECT` query form is used. Lines 4-8 show the graph pattern of the query.

```
1. PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2. PREFIX rel: <http://purl.org/vocab/relationship/>

3. SELECT ?name
4. WHERE {
5.   ?parent foaf:age "34" .
6.   ?parent rel:parentOf ?child .
7.   ?child  foaf:firstName ?name .
8. }
```

Listing 2.2: Example of a SPARQL query.

Definition 2.3 Triple Pattern [14]**.**
A triple pattern (s, p, o) *is a tuple of the set* $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$*. V is an infinite set of variables, which is disjoint from the sets I, B and L.* $\mathscr{P}$ *denotes the set of all triple patterns.*

The graph pattern of the example query in listing 2.2 consists of three triple patterns (lines 5-7). Triple patterns are similar to RDF triples, except that the elements of a triple pattern may also be variables. The first triple pattern of the query (line 5) consists of the variable `?parent` in the subject position, the IRI `foaf:age` in the predicate position and the literal `"34"` in the object position. A formal definition of triple patterns is introduced in definition 2.3. Notice that this definition does not consider blank nodes in order to simplify the semantics of triple patterns.

Based on the definition of triple patterns, SPARQL graph patterns are defined in definition 2.4. A graph pattern can either be a single triple pattern (case 1) or the conjunction of several graph patterns using the operator `.` (case 2). The SPARQL operator `.` (dot) will be referred to as `AND`. Additionally, graph patterns can also be grouped using curly braces `{ }` (case 3). It is important to note that SPARQL graph patterns can be constructed using other operators besides the `AND` operator. However, in the context of this thesis, only the defined subset of graph patterns is considered.

Definition 2.4 Graph Pattern [2]**.**
Let P, P_1, P_2 *be graph patterns.*
A graph pattern *is defined recursively as:*

1. $(s, p, o) \in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$*; a triple pattern*
2. $(P_1$ *AND* $P_2)$*; the conjunction of two graph patterns*
3. $\{P\}$*; a group pattern*

Finally, SELECT queries can be defined as shown in definition 2.5. A SELECT query is an expression composed of the `SELECT` keyword, a subset of the variables contained in the graph pattern and the `WHERE` keyword followed by the graph pattern itself. Instead of explicitly stating a set of variables, the wild card symbol * may be used to denote all variables that appear in the graph pattern.

Definition 2.5 SELECT Query [2]**.**
Let P be a graph pattern. Let $var(P)$ denote the set of variables contained in the triple patterns comprising the graph pattern P. Let $W \subseteq var(P)$ be a set of variables, called the projection variables.
A SELECT query *is an expression* `SELECT W WHERE {P}`*.*
The expression `SELECT * WHERE {P}` *utilizes the wild card symbol * and is equivalent to the SELECT query* `SELECT W WHERE {P}`*, where $W = var(P)$.*

2.3. SPARQL Semantics

For the definition of the semantics of SPARQL SELECT queries, it is necessary to understand variable mappings, their domain and their compatibility property as introduced in definitions 2.6 to 2.8. In addition to that, definition 2.9 introduces restricted variable mappings.

Definition 2.6 Variable Mapping [14]**.**
A variable mapping $\mu : V \nrightarrow (I \cup B \cup L)$ *is a partial function which assigns values to variables.*

Definition 2.7 Domain of Variable Mappings [14]**.**
The domain $dom(\mu) = d$ *of a variable mapping μ is the subset $d \subseteq V$ in which μ is defined.*

Definition 2.8 Compatible Variable Mappings [14]**.**
Let μ_1, μ_2 be variable mappings.
μ_1 and μ_2 are compatible*, denoted by $\mu_1 \sim \mu_2$, if all of their common variables share the same value: $\forall v \in (dom(\mu_1) \cap dom(\mu_2)) : \mu_1(v) = \mu_2(v)$. Common variables are those variables that appear in the domain of both variable mappings.*

Definition 2.9 Restriction of Variable Mappings [2].
Let μ be a variable mapping and let $W \subseteq V$ be a set of variables.
The restriction of μ to variables in W is denoted as $\mu_{|_W}$ and is defined as follows:

$$\mu_{|_W}(x) := \mu(x) \text{ for all } x \in (W \cap dom(\mu)), x \in V$$

The domain of the restricted variable mapping $\mu_{|_W}$ is $dom(\mu_{|_W}) = W \cap dom(\mu)$.

Example 2.3. In order to clarify the definitions 2.6 to 2.9, consider the variable mappings $\mu_1 = \{(\texttt{?child}, \texttt{person:Juliet}), (\texttt{?name}, \texttt{"Juliet"})\}$, $\mu_2 = \{(\texttt{?child}, \texttt{person:Jack}), (\texttt{?name}, \texttt{"Jack"})\}$, $\mu_3 = \{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Juliet})\}$ and the triple pattern $t = (\texttt{?child}, \texttt{foaf:firstName}, \texttt{?name})$. The value that is assigned to a variable by a variable mapping can be an IRI, a blank node or a literal. The notation of variable mappings is expanded to triple patterns. Consequently, $\mu_1(t)$ denotes the RDF triple that results from replacing all variables in triple pattern t according to variable mapping μ_1. Accordingly, the application of μ_1 to t results in the RDF triple $\mu_1(t) = (\texttt{person:Juliet}, \texttt{foaf:firstName}, \texttt{"Juliet"})$. The variables `?child` and `?name` of the triple pattern are replaced with the corresponding values provided by the variable mapping μ_1.

The domains of the variable mappings μ_1, μ_2 and μ_3 are $dom(\mu_1) = \{\texttt{?child}, \texttt{?name}\}$, $dom(\mu_2) = \{\texttt{?child}, \texttt{?name}\}$ and $dom(\mu_3) = \{\texttt{?parent}, \texttt{?child}\}$, respectively. The variable mappings μ_1 and μ_2 are not compatible. These variable mappings share the common variables `?child` and `?name`, but assign different values to both of these variables. The variable mappings μ_1 and μ_3 are compatible, as they both assign the value `person:Juliet` to their shared variable `?child`.

Restricting the variable mappings μ_1, μ_2 and μ_3 to $W = \{\texttt{?child}\}$ results in the restricted variable mappings $\mu_{1_{|_W}} = \{(\texttt{?child}, \texttt{person:Juliet})\}$, $\mu_{2_{|_W}} = \{(\texttt{?child}, \texttt{person:Jack})\}$ and $\mu_{3_{|_W}} = \{(\texttt{?child}, \texttt{person:Juliet})\}$. In this case applying the restriction removes all variables that are not included in W from the variable mappings. Thus, only values for the variable `?child` are kept.

Definition 2.10 Join of Sets of Variable Mappings [14].
Let Ω_1, Ω_2 be sets of variable mappings.
The `join` of Ω_1 and Ω_2 is defined as:

$$\Omega_1 \bowtie \Omega_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$$

Definition 2.10 introduces the `join` operator for sets of variable mappings. The `join` operator takes two sets of variable mappings and creates a new set containing the union of each pair of compatible variable mappings. The introduction of the `join` operator allows it to define the semantics of graph patterns and SELECT queries. Definition 2.11 introduces the semantics of graph patterns as a function $[[\cdot]]_G$, which takes a graph pattern P and an RDF graph G and returns a set of variable mappings matching the graph pattern P. In definition 2.13 the semantics of SELECT queries are defined. For this purpose, the function $[[\cdot]]_G$ from definition 2.11 is expanded so that it may alternatively take a SELECT query S as its argument.

Definition 2.11 Evaluation of Graph Patterns [14]**.**
Let t be a triple pattern, let P, P_1, P_2 be graph patterns and let G be an RDF graph. The evaluation $[[P]]_G$ *of a graph pattern P over graph G is defined recursively as:*

1. $[[t]]_G := \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in G\}$
2. $[[\{P\}]]_G := [[P]]_G$
3. $[[P_1 \; AND \; P_2]]_G := [[P_1]]_G \bowtie [[P_2]]_G$

Definition 2.12 Selectivity of Graph Patterns**.**
The selectivity of a graph pattern *P on an RDF graph G is a number that indicates the size of the result set produced by evaluating P over graph G.*

In definition 2.11 the evaluation of graph patterns is separated into three different cases. Case 1 is dedicated to the evaluation of triple patterns. Evaluating a triple pattern results in a set of variable mappings. When applied to the triple pattern t, each of these variable mappings produces an RDF triple that is contained in the RDF graph G. Furthermore, the domain of these variable mappings coincides with the variables contained in triple pattern t. Therefore, the evaluation of a triple pattern can be described as a lookup of RDF triples matching the IRIs and literals contained in triple pattern t. Case 2 defines the evaluation of a group pattern to be equivalent to the evaluation of the contained graph pattern. Lastly, in case 3 the conjunction of two graph patterns is realized by separately evaluating the two graph patterns and applying the `join` operator to the obtained sets of variable mappings afterwards. Definition 2.12 introduces the selectivity of graph patterns.

Example 2.4. To clarify the evaluation of graph patterns, consider listing 2.3. The shown graph pattern consists of the conjunction of the first and second triple pattern of the query in listing 2.2. To provide an example this graph pattern is evaluated over the RDF graph G in listing 2.1. The graph pattern asks for 34-year-old persons who are also a parent. The evaluation results in a set containing the variable mappings $\{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Juliet})\}$ and $\{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Jack})\}$, since `person:Mary` is 34 years old and a parent of `person:Juliet` and `person:Jack`.

$$
\begin{aligned}
&[[\texttt{?parent foaf:age "34" AND ?parent rel:parentOf ?child}]]_G \\
=&[[\texttt{?parent foaf:age "34"}]]_G \bowtie [[\texttt{?parent rel:parentOf ?child}]]_G \\
=&\{\{(\texttt{?parent}, \texttt{person:Mary})\}\} \bowtie \\
&\{\{(\texttt{?parent}, \texttt{person:Craig}), (\texttt{?child}, \texttt{person:Juliet})\}, \\
&\ \{(\texttt{?parent}, \texttt{person:Craig}), (\texttt{?child}, \texttt{person:Jack})\}, \\
&\ \{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Juliet})\}, \\
&\ \{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Jack})\}\} \\
=&\{\{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Juliet})\}, \\
&\ \{(\texttt{?parent}, \texttt{person:Mary}), (\texttt{?child}, \texttt{person:Jack})\}\}
\end{aligned}
$$

Listing 2.3: Evaluation of the conjunction of the first two triple patterns in listing 2.2.

Definition 2.13 Evaluation of SELECT Queries [14]**.**
Let S be a SELECT query, let P be a graph pattern and let $W \subseteq var(P)$ be a set of variables occurring in P.
The evaluation $[[S]]_G$ *of a SELECT query S over graph G is defined as follows:*

1. $[[\textit{SELECT W WHERE P}]]_G := \{\mu_{|_W} \mid \mu \in [[P]]_G\}$
2. $[[\textit{SELECT * WHERE P}]]_G := [[\textit{SELECT W WHERE P}]]_G$, *where* $W = var(P)$

The evaluation of a SELECT query applies a projection to the set of variable mappings that is obtained by evaluating the graph pattern of the query. In case 1 this projection is achieved by restricting each variable mapping to the set of projection variables W, which is provided by the SELECT query. Case 2 defines the evaluation of SELECT queries utilizing the wild card symbol *, which indicates that the resulting variable mappings should not be restricted. In this case the query is simply evaluated using $var(P)$ as the set of projection variables. The variable mappings obtained by evaluating graph pattern P have the domain $var(P)$ (see definition 2.11, case 1). Therefore, restricting these variable mappings to $var(P)$ will leave them unchanged.

2.4. RDF Store Koral

Koral[2] is an open-source distributed RDF store [8]. It consists of a single master node and several slave nodes. The master node is responsible for the graph loading, managing the network connection to the slave nodes, handling incoming query requests and coordinating the query processing. Each slave node stores a portion of the RDF graph. In addition to that, the slave nodes process the query plan provided

[2]https://github.com/Institute-Web-Science-and-Technologies/koral [Accessed: 18 December 2019]

by the master node. During the query processing they are responsible for matching triple patterns and joining the found intermediate results according to the query plan. Moreover, the intermediate results are transferred from slave node to slave node until the final results are sent back to the master node.

This section focuses on providing insight into how the RDF store Koral operates in its current state. Section 2.4.1 illustrates the graph loading process of Koral, whereas the query processing of Koral is explained in section 2.4.2.

2.4.1. Graph Loading

During the initial loading of an RDF graph each contained IRI, blank node and literal is replaced by a unique integer identifier. The master node maintains a dictionary mapping these identifiers to their plain representation. Therefore, the process is referred to as dictionary encoding. This replacement is done to reduce the graph size as early as possible.

Afterwards, a graph cover is created. A graph cover partitions the encoded triples into n different graph chunks, where n is the number of slave nodes. The assignment of a triple to a specific graph chunk is based on the graph cover strategy and may, for example, depend on the contained triple components or on the graph structure. Depending on the graph cover strategy a triple may also be assigned to multiple graph chunks. A commonly used graph cover strategy is the hash graph cover. This strategy utilizes a hash function to compute the hash value corresponding to the subject of a triple. Each triple is assigned to a graph chunk according to the hash value modulo the number of slave nodes [8].

Once all graph chunks have been created, statistics about the individual graph chunks are gathered. Per graph chunk the frequency of each triple component is collected individually at each triple position and stored in Koral's statistics database. The statistics are also used to determine the join responsibility for all triple components. A slave node is granted the join responsibility of a triple component if its assigned graph chunk contains the component at the subject position most frequently among all graph chunks. In case of a tied number of occurrences between two graph chunks, the number of occurrences at the object and predicate position is compared instead. The ID of the responsible slave node is stored in the first two bytes of the unique identifier.

Thereafter, the master node transfers the graph chunks to their assigned slave nodes. Each slave node stores the received graph chunk in its local RDF storage [7]. At this point the graph is completely loaded into the RDF store.

Example 2.5. As a running example assume that Koral is running with two slave nodes and the RDF graph from listing 2.1 is being loaded. Consider a graph cover strategy that assigns triples with the subjects `person:Craig` and `person:Mary` to the first slave node and the remaining triples to the second slave node. The graph chunks created by this graph cover are depicted in table 2.1. To maintain readability the plain triple representation is used, even though the graph chunks exclusively

contain dictionary-encoded triple components at this stage. Additionally, each triple component is annotated with the ID of the slave node that is responsible for joining it. The notation `person:Craig|1` indicates that slave 1 has been assigned the join responsibility of the resource `person:Craig`. In this case slave 1 holds the join responsibility because the resource `person:Craig` occurs more often in the subject position in graph chunk 1 than it does in graph chunk 2.

Slave Node 1	**Slave Node 2**

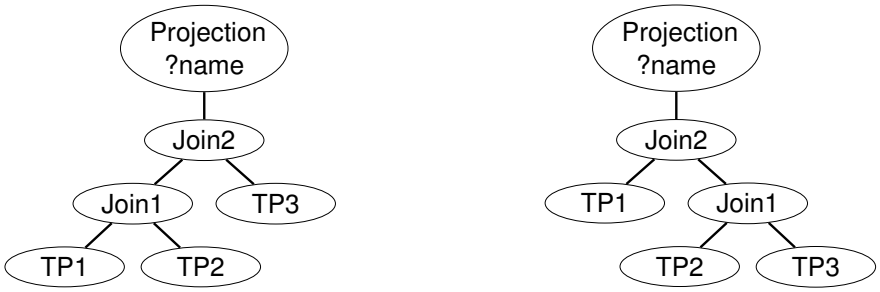
`person:Craig|1 foaf:firstName|2 "Craig"|1`
`person:Craig|1 foaf:age|2 "33"|1`
`person:Craig|1 rel:parentOf|1 person:Juliet|2`
`person:Craig|1 rel:parentOf|1 person:Jack|2`
 	
`person:Mary|1 foaf:firstName|2 "Mary"|1`
`person:Mary|1 foaf:age|2 "34"|1`
`person:Mary|1 rel:parentOf|1 person:Juliet|2`
`person:Mary|1 rel:parentOf|1 person:Jack|2`
	`person:Juliet|2 foaf:firstName|2 "Juliet"|2`
	`person:Juliet|2 foaf:age|2 "8"|2`
	`person:Juliet|2 rel:siblingOf|2 person:Jack|2`
	
	`person:Jack|2 foaf:firstName|2 "Jack"|2`
	`person:Jack|2 foaf:age|2 "9"|2`
	`person:Jack|2 rel:siblingOf|2 person:Juliet|2`
	
	`dog:Merlin|2 foaf:firstName|2 "Merlin"|2`
	`dog:Merlin|2 foaf:age|2 "10"|2`
	`dog:Merlin|2 ex:ownedBy|2 person:Craig|1`
	`dog:Merlin|2 ex:ownedBy|2 person:Mary|1`

Table 2.1.: An example graph cover of the RDF graph in listing 2.1.

2.4.2. Query Processing

The first stage of query processing is called *query planning*. The query planning starts whenever the master node receives a query request. During query planning, the master node creates a query execution tree, which serves as a summary of all operations required to process the query and their exact order. At the same time the triple components included in the triple patterns of the query are dictionary-encoded, which allows the slave nodes to properly match them against their local RDF storages. Koral only supports the subset of SPARQL SELECT queries introduced in sections 2.2 and 2.3. Accordingly, queries that do not coincide with this subset of SPARQL are rejected by the store. Therefore, the only query operations that are considered are triple pattern matches, joins and projections.

In its current state Koral does not perform any optimizations regarding the order of operations. Instead, the user can manually choose from three predefined query execution tree types, namely “left linear”, “right linear” and “bushy”. This choice determines the structure of the query execution tree that is created during query planning. By default Koral builds a left linear query execution tree. A left linear query execution tree arranges the join operations in the same order as defined by the query. The right linear query execution tree reverses this join order. Lastly, the bushy query execution tree aims at minimizing the height of the tree. Figure 2.2 shows the left and right linear query execution trees corresponding to the query in listing 2.2. The nodes labeled `TP1`, `TP2` and `TP3` refer to the first, second and third triple pattern match operation, respectively. In this case the bushy tree is identical to either of the two depicted trees depending on the implementation.



a) Left linear query execution tree. **b)** Right linear query execution tree.

Figure 2.2.: Query execution trees of the query in listing 2.2.

Following the creation of the query execution tree, a copy of it is passed to each slave node. As soon as all slave nodes have acknowledged the receipt of the query execution tree, the master node starts the *query execution* stage by instructing the slave nodes to process the query execution tree. To illustrate the query execution, consider the running example shown in table 2.1. Moreover, assume that both slave nodes have received the query execution tree depicted in figure 2.2a.

A triple pattern match operation is processed by looking up matching triples from the local RDF storages of the slave nodes. The result of a triple pattern match operation is a set of variable mappings corresponding to the matching triples. Matches for the first and second triple pattern are found exclusively in the graph chunk assigned to the first slave node. The reason for this is that the only triples containing the predicate `rel:parentOf` are the ones about `person:Craig` and `person:Mary`. Additionally, `person:Mary` is the only person who is 34 years old. Matches for the third triple pattern can be found in both graph chunks as the predicate `foaf:firstName` is not exclusive to either of them. An overview of matching triples found by the slave nodes for each triple pattern (abbreviated as TP) is provided in table 2.2. The intermediate results, produced by the triple pattern match operations, are immediately transferred to the subsequent join operations.

TP	Matching triples on slave 1	Matching triples on slave 2
TP1	**?parent** – – person:Mary|1 foaf:age|1 "34"|1	*No matches for TP1*
TP2	**?parent** – **?child** person:Craig|1 rel:parentOf|1 person:Juliet|2 person:Craig|1 rel:parentOf|1 person:Jack|2 person:Mary|1 rel:parentOf|1 person:Juliet|2 person:Mary|1 rel:parentOf|1 person:Jack|2	*No matches for TP2*
TP3	**?child** – **?name** person:Craig|1 foaf:firstName|2 "Craig"|1 person:Mary|1 foaf:firstName|2 "Mary"|1	**?child** – **?name** person:Juliet|2 foaf:firstName|2 "Juliet"|2 person:Jack|2 foaf:firstName|2 "Jack"|2 dog:Merlin|2 foaf:firstName|2 "Merlin"|2

Table 2.2.: Results of triple pattern match operations `TP1`, `TP2`, `TP3` on both slave nodes.

To reduce the imposed workload of join operations, their processing is distributed among all slave nodes. Each individual variable mapping is only joined by a single slave node. This slave node is selected according to the join responsibility of the first join variable in the variable mapping. The triple pattern match operations involved in `Join1` are `TP1` and `TP2` and the join variable of `Join1` is `?parent`. All variable mappings returned by `TP1` and `TP2` assign either `person:Craig` or `person:Mary` to the variable `?parent`. Slave node 1 holds the join responsibility for these two resources. Accordingly, all of these matches are sent to the operation `Join1` on slave 1. Row 1 of table 2.3 depicts the results of the operation `Join1`. Join operation `Join2` combines the results of triple pattern match operation `TP3` and join operation `Join1`. In case of `Join2` the join variable is `?child`. Therefore, all results of `Join1` and `TP3`, which assign `person:Juliet`, `person:Jack` or `dog:Merlin` to the variable `?child`, are sent to operation `Join2` on slave 2. The remaining variable mappings, which originate from `TP3` on slave 1, are forwarded to join operation `Join2` on slave 1. As shown in row 2 of table 2.3, `Join2` yields no results on slave 1 since no join candidates are received from join operation `Join1`. On slave 2, operation `Join2` produces two variable mappings.

Join	**Results on slave 1**	**Results on slave 2**
`Join1`	**?parent** **?child** person:Mary|1 person:Juliet|2 person:Mary|1 person:Jack|2	*No results* *Slave 2 is not responsible for any of the resources*
`Join2`	*No results* *No join candidates received from `Join1`*	**?parent** **?child** **?name** person:Mary|1 person:Juliet|2 "Juliet"|2 person:Mary|1 person:Jack|2 "Jack"|2

Table 2.3.: Results of the join operations `Join1` and `Join2` on both slave nodes.

All results produced by `Join2` are forwarded to the projection operation on the same slave node. At this point the whole query execution tree has been processed since the projection operation is the root of the query execution tree. As a consequence, all slave nodes transfer the results of the projection operation to the master node. The master node decodes the unique identifiers contained in the received results and delivers the decoded results to the client that has sent the query request.

The client is awaiting further results until the master node signalizes that the query request is finished. To determine that the query request has been processed completely, the query operations communicate by sending a finish notification to their counterpart operations on the other slave nodes.
Triple pattern match operations *send a finish notification*, if:

- all matching triples have been found and
- the corresponding intermediate results have been transferred

Join and projection operations *send a finish notification*, if:

- all received variable mappings have been processed and
- all preceding operations in the query execution tree are finished

A query operation *is finished*, if:

- it has received a finish notification from its counterpart operation on every other slave node and
- it has already sent out finish notifications by itself

Once the root operation of the query execution tree is finished, the corresponding slave node informs the master node that it has finished processing the query execution tree. As soon as all slave nodes are finished, the master node concludes the query execution stage and informs the client that no further results will be transferred.

Example 2.6. The triple pattern match operation `TP1` looks up matching triples on both slave nodes and finds the results shown in table 2.2. On slave node 1, the operation `TP1` finds one matching triple and transfers the corresponding variable mapping to the parent join operation `Join1`. Thereafter, it sends a finish notification to the operation `TP1` on slave node 2. On slave node 2, the operation `TP1` does not find any matching triples. Accordingly, no variable mappings have to be transferred. Now the operation `TP1` on slave node 2 sends a finish notification to the operation `TP1` on slave node 1. Upon receiving the finish notification from slave node 2, the operation `TP1` on slave node 1 is finished, since it has now received a finish notification from its counterpart operations on all other slave nodes and has already sent a finish notification to slave node 2. Similarly, the operation `TP1` on slave node 2 is also finished as it had already received the finish notification from slave node 1 and has sent out a finish notification as well.

During the query processing the operation `Join1` receives intermediate results from the triple pattern match operations `TP1` and `TP2`. After the operation `Join1` on slave node 1 has processed all intermediate results and has produced the variable mappings shown in table 2.3, it checks whether both of the preceding operations, `TP1` and `TP2`, are finished. If this is the case, it sends a finish notification to the operation `Join1` on slave node 2. Meanwhile, the operation `Join1` on slave node 2 does not receive any intermediate results from the operations `TP1` and `TP2`. Consequently, it does not have to process or transfer any intermediate results either. As soon as its preceding operations `TP1` and `TP2` are finished, the operation `Join1` on slave node 2 sends a finish notification to the operation `Join1` on slave node 1. At this stage either of these join operations is finished once it has received the finish notification from the other slave node.

2.5. Problem Domain

This section explains the problem domain of *join order optimization* in the context of distributed RDF stores. Join order optimization is one of the problems that can be addressed in order to optimize the query processing. It refers to the process of rearranging the sequence of join operations in such a way that the total workload of the query plan is minimized. A cost function is used to estimate the workload

associated with triple pattern match operations and join operations. Accordingly, the total workload of a query plan can be approximated. To make proper estimates, cost functions usually require precomputed statistics about the stored RDF graph, e.g. the frequency of occurrences of triple components. For the join order optimization only a small fragment of the SPARQL query language, as introduced in section 2.2, is considered. In the following an example regarding the importance of the join order is given. Afterwards, the different dimensions associated with join order optimization are presented.

Example 2.7. The order in which the triple patterns of a query are processed may be crucial to the performance of an RDF store. Consider the query processing as described in section 2.4.2. Table 2.2 shows that the first, second and third triple pattern match a total of 1, 4 and 5 triples, respectively. Joining the triple patterns in the order defined by the query execution tree in figure 2.2a is optimal since only two intermediate results are generated by `Join1`. The join operation keeps variable mappings regarding `person:Mary`, since she is 34 years old, whereas variable mappings regarding `person:Craig` are discarded. The second join operation only extends the variable mappings with values for the variable `?name`. Thus, the number of results does not change when processing `Join2`.

An alternative join order is represented by the query execution tree in figure 2.2b. This join order reverses the previous one. It is less optimal in this case, since the join operation `Join1` produces four intermediate results instead. Afterwards, the second join operation reduces the size of the result set once again. In the end the different join orders do not influence the final results of the query. However, the query performance may be impacted by a suboptimal join order as more intermediate results have to be processed or transferred over the network. Therefore, avoiding the additional workload imposed by a suboptimal join order is an important optimization task.

Dimension 1: When should the join order be optimized?

Static join order optimization is performed during query planning. All potential join orders are compared in regard to their estimated cost. A query execution tree is then created based on the join sequence with the lowest associated cost. Since this query execution tree remains unchanged during the query execution, this approach is referred to as *static* query optimization.

Dynamic join order optimization tries to optimize the order of join operations during the query execution. The intermediate results that arise can be utilized to make more precise assumptions about the processed data. According to this information the order of subsequent join operations can be adjusted.

Hybrid join order optimization is the combination of both static and dynamic join order optimization. As a starting point the query plan determined by the static join order optimization can be used. However, this query plan is usually based on estimates. For this reason it is likely that further improvements can be applied to

the query plan given additional information during the query execution. Hybrid join order optimization may also identify that the assumptions leading to the initial query plan were wrong. In this case the query execution may be restarted using a new query plan that is optimized accordingly.

Dimension 2: Should the query be optimized separately for each slave node?

Unified join order optimization determines a single optimized query execution tree for the given query. A copy of this query execution tree is transferred to all slave nodes. In this setting all slave nodes process an identical query execution tree. For this reason every slave node implicitly knows the query execution tree of all other slave nodes.

Distinct join order optimization optimizes an individual query execution tree for each slave node. As a consequence, a slave node only possesses knowledge about its own assigned query execution tree. To compensate for this lack of information, additional communication may be required to synchronize the query operations between the slave nodes.

Dimension 3: How should the optimized query plan be computed?

In order to find the optimal join order, a potentially huge amount of different query plans has to be evaluated in terms of their cost. *Centralized join order optimization* conducts the search for the optimal query plan on a single node. In the scope of this search it is often not possible to construct all permutations of the initial query execution tree. A heuristic can be used to determine promising query execution trees. Subsequently, the query execution tree with the lowest estimated cost out of all candidates is selected.

In contrast to that, *distributed join order optimization* makes use of the computational capabilities of all slave nodes. A subset of all potential query plans is searched by each slave node. During the optimization process every slave node determines the query execution tree with the lowest estimated cost of this subset. Afterwards, each slave node sends the found query execution tree to the master node. In a final step the master node chooses the query execution tree with minimal cost out of all received trees.

The computation of the optimized query plan is also dependant on which strategies have been selected in regard to dimension 1 and dimension 2. When combining dynamic (*dimension 1*) and centralized join order optimization, a single node can serve as an optimization coordinator. During the query execution the slave nodes contact the optimization coordinator by sending relevant information. Thereafter, the optimization coordinator determines possible optimizations and communicates the necessary modifications with all slave nodes. In case of distinct join order optimization (*dimension 2*) the optimization coordinator provides an individual set of modifications to each slave node.

Another possibility is the combination of dynamic (*dimension 1*) and distributed join order optimization. In this case the information obtained during the query ex-

ecution can be shared among all slave nodes. Based on the new information, the process of distributed join order optimization may be repeated in order to obtain a new optimized query execution tree. In terms of distinct join order optimization (*dimension* 2) an optimized query execution tree has to be determined for each slave node. Alternatively, each slave node may handle the optimization of its own query execution tree.

Dimension 4: Which cost function should be minimized?

The quality of join order optimization is dependant on the utilized cost function. In general minimizing the *number of intermediate results* is desirable as this measure strongly influences the query performance. However, focusing specifically on reducing the *network traffic* between the slave nodes or the *number of join computations* done during query processing may also be valid options. Even if all of the previously mentioned costs are low, one slave node might still pose to be a bottleneck for the entire query execution. This scenario occurs if the majority of join computations has to be computed by a single slave node, whereas the remaining slave nodes only compute few join computations. In this case minimizing the *workload imbalance* may be desired. If minimal workload imbalance were to be achieved, the number of join computations performed by each slave node would be equal.

3. Related Work

As part of the evaluation of the developed distributed optimization approach, a centralized optimization approach is used as a baseline. For this purpose, this section provides an overview of related research in the domain of SPARQL query optimization. Section 3.1 presents work related to query optimization approaches that are aimed at centralized RDF stores. Query optimization techniques employed by distributed RDF stores are introduced in section 3.2.

3.1. Query Optimization in Centralized RDF Stores

Schmidt et al. [15] present a set of algebraic equivalences for the complete SPARQL algebra. These algebraic equivalences are also termed *rewrite rules* as they allow for rewriting a SPARQL algebra expression to an equivalent expression. The authors provide basic equivalence rules regarding the join operator. These rules state that the order of consecutive join operators may be changed arbitrarily. Some of the rules only hold for a specific fragment of the SPARQL algebra. However, this fragment covers the subset of SPARQL introduced in section 2.2. In addition to the basic rules, equivalence rules related to the projection operator are introduced. The projection pushing rules allow moving projection operators to a different position in the query execution tree. Consequently, the number of variables that are bound in the intermediate results can be reduced as early as possible during the query execution.

Besides the equivalence rules, a *Semantic Query Optimization* approach is proposed. The optimization algorithm takes a set of semantic constraints that are satisfied by the data set. These constraints can either be specified by the user, be extracted from the data set automatically or be based on the semantics of RDF Schema (RDFS[3]). The algorithm iteratively rewrites a query by adjusting operators that violate the specified constraints. In the end the algorithm determines a semantically equivalent query that is minimal with respect to the number of query operations.

Loizou et al. [11] propose a set of heuristics that help with formulating more efficient queries. These heuristics identify certain query patterns that can be rewritten to obtain equivalent, but more performant, queries. The performance gain is achieved by reducing the number of complex graph patterns, namely those containing the `OPTIONAL` operator. Furthermore, the evaluation scope of certain graph patterns is restricted to named subsets of the graph. Thus, these graph patterns have to be evaluated against fewer RDF triples. Queries are also rewritten to avoid

[3] https://www.w3.org/TR/rdf-schema/ [Accessed: 29 June 2019]

the use of intermediate variables, which do not contribute to the final query result. The heuristics presented in [11] are not applicable to the RDF store Koral since they focus on SPARQL operators that are not supported by Koral.

Contrary to the approaches by Schmidt et al. [15] and Loizou et al. [11], the remainder of this section addresses work that specifically attempts to optimize the join order of SPARQL queries. For this purpose, some of the presented join order optimization approaches try to estimate the selectivity of graph patterns, since it is indicative of the size of the result set and thus also the corresponding workload.

Vidal et al. [18] propose a dynamic query optimization technique based on star-shaped triple pattern groups. In this case a star-shaped group consists of triple patterns that share a single common variable at the subject or object position. In order to explore the search space of all possible query plans, a probabilistic optimization algorithm, based on the *Simulated Annealing* algorithm, is used. The algorithm consists of several stages. At the start of each stage a random query plan is generated. Afterwards, several transformation steps are applied. In each transformation step the current query plan is modified according to a randomly chosen transformation rule. The newly produced query plan is accepted with a probability specified by an acceptance probability function. Generally, query plans are accepted if their cost is lower than that of the currently best query plan. However, the acceptance probability function may also accept query plans with higher costs in order to escape from local minima during exploration. In the end the algorithm returns an optimized bushy query plan.

In order to estimate the cost of star-shaped query plans, the authors propose an adaptive sampling method. The first triple pattern of the star-shaped group is evaluated and sample results are taken from the result set. For each sample the whole group is evaluated to determine the resulting number of intermediate results. Finally, the mean of the collected values is used as the estimated cost for the given query plan. Contrary to that, the cost estimation of query plans that are not star-shaped is conducted using techniques similar to those used in relational databases.

In [12], Neumann et al. introduce the RDF store *RDF-3X*. RDF-3X employs a join order optimization approach. This optimization algorithm uses dynamic programming to enumerate all potential query plans. The algorithm initially constructs partial query plans comprising only a few of the triple patterns of a query. Among the constructed query plans, the algorithm searches for optimal ones according to cost estimates. In subsequent steps, optimal, but partial, query plans are joined to form larger query plans until a complete optimized query plan has been determined.

The optimization approach utilizes a selectivity-based cost model to estimate the cost of query plans. The employed cost model relies on two kinds of precomputed statistics about the data set. Selectivity histograms constitute the first statistic. There are six possible orders in which the subject (S), predicate (P) and object (O) of a triple can be arranged: SPO, SOP, PSO, POS, OSP, OPS. An individual selectivity histogram is computed for each order to ensure that this statistic can be used to

estimate the selectivity of all possible triple patterns. To compute each histogram the components of all triples are first rearranged according to the corresponding order. Thereafter, the rearranged triples are grouped by their first two elements. For example, the histogram, which corresponds to the order PSO, places triples whose predicates as well as subjects are identical in the same group. Subsequently, groups of triples with similar components are merged until the histograms can be stored completely in main memory. The histograms summarize each of the resulting triple groups by storing the total number of contained triples, the number of distinct elements in the first position, and the number of distinct pairs of elements in the first two positions. These counts are used to estimate the selectivity of individual triple patterns.

As an example assume that the group of triples depicted in listing 3.1 is being summarized. Note that due to the order PSO being used, the first, second and third element of the shown triples coincides with the predicate, subject and object of the original triples, respectively. The total number of triples in this group is 4. The number of distinct elements in the first position is 1, since all triples share the predicate `rel:parentOf`. Lastly, the number of distinct pairs of elements in the first two positions is 2. In this case these two pairs are (`rel:parentOf`, `person:Craig`) and (`rel:parentOf`, `person:Mary`).

```
1. rel:parentOf     person:Craig    person:Jack .
2. rel:parentOf     person:Craig    person:Juliet .
3. rel:parentOf     person:Mary     person:Jack .
4. rel:parentOf     person:Mary     person:Juliet .
```

Listing 3.1: Example of a group of RDF triples rearranged using the order PSO.

To estimate the selectivity of triple pattern joins, the number of join partners of each triple group is stored. The number of join partners is the size of the result set obtained from joining the triple group itself with all triples in the data set according to a specific combination of positions of the join variable. The number of join partners is separately recorded for each possible combination.

The second statistic identifies the most frequent join patterns occurring in the data set and stores the exact size of their result sets. This statistic considers join patterns to be a group of triple patterns with different predicates that share a common subject (*star-shaped*) or a group of triple patterns where the object of a triple pattern is the subject of the subsequent triple pattern (*path-shaped*). If a frequent join pattern is found in a query plan, its selectivity is directly provided by the second statistic. In contrast, the selectivity of the remaining triple patterns is estimated using the selectivity histograms. The product of these selectivities constitutes the cost of a query plan.

The query performance of RDF-3X has been compared to the query performance of other RDF stores on three different data sets. During this comparison RDF-3X has proven to be faster than the other RDF stores. However, the presented results are not indicative of the quality of the employed optimization approach, since it is not clear which part of the performance gain can be attributed to the optimization itself.

Stocker et al. [16] present a framework for static selectivity-based optimization of SPARQL basic graph patterns (BGPs). BGPs are sets of triple patterns. The core optimization algorithm takes a set of triple patterns and constructs an optimized query plan by reordering them according to estimated selectivities of both triple patterns and joins. In the first step the algorithm searches two joined triple patterns whose estimated join selectivity is minimal. Both triple patterns are added to the optimized query plan in ascending order of their individual selectivity. Afterwards, the algorithm iteratively adds one of the remaining triple patterns that is joined to the already added triple patterns and whose corresponding join selectivity is minimal. The algorithm terminates once all triple patterns have been added to the optimized query plan. Additionally, if a BGP can be split into groups of triple patterns such that these groups are not joined with each other, then each triple pattern group is optimized separately by the algorithm.

Since the core optimization algorithm represents a framework for the optimization, the authors propose several selectivity estimation heuristics to be used in conjunction with the optimization algorithm. The heuristics *variable counting* and *variable counting predicates* estimate the selectivity of a triple pattern according to the number of contained variables and the positions in which the variables occur. Similarly, the selectivity of triple pattern joins is estimated based on the number and position of the join variables. These heuristics do not rely on precomputed statistics. The *graph statistics handler* heuristic utilizes statistics about the number of occurrences of each triple component at each triple position to estimate the selectivity of triple patterns. Since this heuristic does not support the estimation of the join selectivity, the variable counting heuristic is used for this purpose instead. The *probabilistic framework* heuristic and its variations rely on precomputed statistics. These statistics include the total number of triples and the number of distinct subjects. For each distinct predicate, the number of occurrences and a histogram representing the distribution of its objects is required. The selectivity of a triple pattern is estimated to be the product of the selectivity of its components. The subject selectivity is the average number of triples that the subject matches, the predicate selectivity is approximated by its number of occurrences, and the object selectivity is approximated using the predicate histograms. Additionally, full summaries of the result size are generated for all join pattern combinations of pairs of distinct predicates. These summaries are used to estimate the upper bound of the join selectivity.

The optimization quality of the heuristics has been evaluated on a data set containing 156 407 triples. For this purpose, the authors explore the performance of all possible query plans of 14 different queries and identify how well the query plans produced by the different heuristics perform in comparison. On average the query plans optimized by the probabilistic framework heuristic, the graph statistics handler heuristic and the variable counting heuristic are better than 97.7%, 89% and 50% of all query plans, respectively. Compared to that, the unoptimized query plans only perform better than 32% of all existing query plans.

Tsialiamanis et al. [17] introduce a heuristics-based query optimization approach for SPARQL. The authors implement a *heuristics-based SPARQL planner (HSP)* that optimizes queries by pursuing two main objectives. The first objective is to construct query plans that maximize the number of merge joins. This objective is achieved by grouping the triple patterns of a query according to shared variables. The determined triple pattern groups are sorted in descending order of their size and constitute the new query plan.

The second objective of the query planner is to minimize the number of intermediate results. For this purpose, the authors propose five independent heuristics that can be used to order triple patterns regarding their selectivity. The proposed heuristics are based on the syntactical form of the queries, as the authors argue that storing precomputed statistics is costly and that the statistics are likely to become outdated. HSP combines these heuristics for the optimization. The heuristics can be summarized as follows.

Heuristic 1 estimates the selectivity of a triple pattern according to the position and number of its variables. For example, the selectivity of a triple pattern with a variable in the subject position is estimated to be lower than that of a triple pattern with a variable in the object position.

Heuristic 2 estimates the selectivity of triple patterns depending on the position of their join variables in the whole graph pattern. This heuristic allows for avoiding joins with high selectivity.

Heuristic 3 states that the selectivity of a triple pattern decreases with the number of IRIs and literals.

Heuristic 4 indicates that IRIs in the object position of a triple pattern impose a higher selectivity than a literal in the same position.

Heuristic 5 attributes a lower selectivity to triple patterns that contain fewer of the projection variables of the query.

In case of queries that contain triple patterns with varying number and position of variables, the heuristics-based approach has proven to be able to produce near to optimal or identical query plans when compared to an optimization approach that relies on statistics. In contrast to that, the heuristics-based approach struggles to optimize star-shaped queries since the contained triple patterns are syntactically similar, and thus the heuristics become less effective. Additionally, due to the lack of statistics, there are cases in which the algorithm determines parts of the optimized query plan based on random choices, as more accurate estimates of the number of intermediate results are not possible. Among the presented heuristics, heuristic 3 achieves to determine the most efficient join orders for many queries.

All approaches presented in this section are aimed at centralized RDF stores. These approaches centrally optimize a single query execution tree. Therefore, these approaches differ from the distributed optimization approach developed in this thesis. The main differences are that these centralized optimization approaches do not distribute the optimization process across several nodes. In addition to that, the de-

veloped approach optimizes the query execution tree distinctly for each slave node. However, it seems plausible that these approaches can be employed in distributed RDF stores in order to centrally optimize a query during query planning.

3.2. Query Optimization in Distributed RDF Stores

In distributed RDF stores the graph data is shared across multiple nodes. Consequently, the processing of an incoming query request proves to be more difficult than in a centralized RDF store. To process the query in a distributed manner it is split into several subqueries such that each slave node can produce a set of intermediate results from its local RDF storage. Section 3.2.1 and section 3.2.2 present query optimization approaches that are used in federated and distributed RDF stores, respectively.

3.2.1. Query Optimization in Federated RDF Stores

In order to answer a query request, federated RDF stores query several remote RDF stores and combine the received results. For this purpose, the query federator component of the store forwards subqueries of the initial query to the individual RDF stores. The intermediate results corresponding to these subqueries are joined by the query federator. Afterwards, the joined results are delivered back to the user.

SPLENDID [4] is a query optimization approach that focuses on federated RDF stores. To avoid that all subqueries are sent to every known data source, this approach performs *triple pattern-wise source selection* (TPWSS). For each triple pattern, the approach identifies which data sources are relevant by using precomputed statistical information (VOID descriptions) provided by the remote RDF stores. The subsequent pruning step utilizes SPARQL ASK queries to identify data sources that were falsely categorized as relevant and removes them.

In addition to source selection, SPLENDID also optimizes the join order of basic graph patterns during query planning. An algorithm based on *Dynamic Programming* is used to iterate all potential query plans. The query performance of federated RDF stores is highly influenced by network traffic. Therefore, a cost function that approximates the network traffic cost is employed to determine the query plan with the lowest cost. The estimated cost is based on the cardinality of the result set of triple patterns and joined triple patterns. These cardinalities are approximated using the statistics provided by the VOID descriptions.

Wu et al. [20] propose a dynamic join order optimization approach designed for federated RDF stores. The federated RDF store presented in [20] evaluates the subqueries in sequence. Initially the first subquery is sent to the remote RDF stores. Once the results of a subquery are received, these results are bound to the variables of the next subquery (bind join), which is then sent to the remote RDF stores. This process is repeated until the results for the final subquery are obtained.

The optimization strategy by Wu et al. dynamically adapts the order of join operations. This approach utilizes the more accurate size of intermediate result sets obtained during the query execution, instead of relying on precomputed statistics. To determine the optimal join order, the remaining subqueries are constructed using a small set of samples. These subqueries are then sent to the remote RDF stores as COUNT queries. The result of these COUNT queries is a single number indicating the approximate result size of the subquery. In each step, the next subquery to be evaluated is then chosen according to the smallest result size.

Unlike the distributed RDF store Koral, all join operations, which are needed to combine the final query result, are executed by a single compute node, namely the query federator component [4, 20]. SPLENDID [4] statically optimizes the join order based on the estimated network traffic, whereas the approach by Wu et al. [20] optimizes the join order dynamically according to the intermediate results obtained during the query execution. In general these federated approaches differ from the query processing of distributed RDF stores and from the distributed optimization approach proposed in this thesis.

3.2.2. Query Optimization in Distributed RDF Stores

Graux et al. [5] propose a selectivity-based join order optimization approach that is implemented in *SPARQLGX*, a distributed RDF store based on Apache Spark. During query planning, the triple patterns of a query are ranked in regard to their number of variables and their estimated selectivity. In order to estimate the selectivity of a triple pattern, precomputed statistics about the RDF graph are used. The selectivity of a triple component at a specific triple position is equal to the number of occurrences of this triple component at the given triple position. If, however, the triple component is a variable, its selectivity is estimated to be the total number of triples contained in the RDF graph. The selectivity of a triple pattern is defined as the minimum selectivity of its subject, predicate and object. Finally, triple patterns are ranked in ascending order of their number of variables as well as their estimated selectivity. The optimized join order is determined based on this ranking of triple patterns. Initially, the first triple pattern is added to the optimized query plan. Step by step, the next triple pattern is selected by searching the remaining triple patterns in the order defined by the ranking. The first triple pattern that shares at least one common variable with the triple patterns of the query plan is chosen and added to the query plan. If such a triple pattern does not exist, the optimization approach selects the triple pattern with the lowest selectivity and adds it to the query plan instead. By first evaluating triple patterns with low selectivity, this optimization approach attempts to minimize the number of intermediate results.

SPARQLGX is compared to several state-of-the-art RDF stores, which are based on the Hadoop MapReduce framework. The comparison uses three data sets comprising 109 million, 134 million and 1.38 billion triples, respectively. 26 different queries are evaluated on all systems in order to observe their execution times. In

case of most queries SPARQLGX outperforms the other systems while yielding a comparable query execution time for the remaining queries.

The query chain algorithm presented by Liarou et al. [10] processes a query by passing it through a chain of nodes and maintaining a set of intermediate results throughout the process. Each node of the chain is responsible for evaluating the triple pattern at the corresponding position in the query. Upon receiving a query request, a node determines matches for the corresponding triple pattern and joins the found matches with the current set of intermediate results. Thereafter, the query request and the new set of intermediate results are passed to the next node in the chain. Once all triple patterns have been processed, the current set of intermediate results constitutes the query answer and is transferred back to the node that has issued the query request. The presented algorithm does not optimize the join order of incoming query requests. The authors note that each node would ideally select the next triple pattern to be evaluated according to its selectivity in order to reduce the network traffic between the nodes. However, such adjustments remain subject of future work. A single query execution tree is centrally constructed by the node that issues the query request. Since the algorithm is based on a query chain, all nodes process the same query execution tree in sequence. For this reason the algorithm differs from the distributed optimization approach proposed in this thesis, which intends to distribute the optimization process across all slave nodes and to process a distinctly optimized query execution tree on each slave node.

4. Distributed Query Optimization

In this section the distributed optimization approach developed in this thesis is presented. Section 4.1 introduces the join order optimization approach that is applied to optimize the query execution trees. Section 4.2 describes how the join order optimization process can be distributed among all slave nodes. Lastly, section 4.3 presents the adjustments that have been made to the query processing strategy of Koral in order to support the distributed join order optimization approach.

4.1. Join Order Optimization

This section is dedicated to presenting the join order optimization approach that is being implemented in Koral. For this purpose, one of the optimization approaches introduced in section 3 is chosen and described in detail. The two approaches by Wu et al. [20] and Vidal et al. [18] optimize the join order dynamically during the query execution. Since the join order optimization approach is supposed to optimize the query plan statically, these two approaches are not considered. Similarly, the optimization approach employed by *SPLENDID* [4] is not considered, as it is targeted at federated RDF stores. The query optimizer of *RDF-3X* [12] utilizes selectivity histograms and frequent join patterns. Since investigating how to compute, store and access these statistics efficiently is out of the scope of this thesis, the optimization strategy of RDF-3X will not be considered. Finally, among the remaining two approaches by Stocker et al. [16] and Graux et al. [5], the more recent optimization approach employed by *SPARQLGX* [5] is chosen.

This join order optimization approach is able to reuse the statistics that are already stored by Koral. However, a core difference between SPARQLGX and Koral is that SPARQLGX[4] only stores statistics about the most frequent[5] triple components, whereas Koral collects statistics for every triple component in the data set.

The approach utilizes these statistics to estimate the selectivity of triple patterns. Definition 4.2 introduces the estimated selectivity of a triple pattern, which is defined as the minimum number of occurrences of its components. The number of occurrences of a triple pattern component is defined in definition 4.1 and can directly be determined using the collected statistics. In the following, these definitions are used to determine an optimized join order.

[4] https://github.com/tyrex-team/sparqlgx [Accessed: 19 July 2019]

[5] By default only the 50 most frequent subjects, predicates and objects as well as the number of their occurrences are recorded.

Definition 4.1 Number of Occurrences of a Triple Pattern Component [5]**.**
Let $c \in (I \cup L \cup V)$ be a triple pattern component, let $k \in \{subject, predicate, object\}$ be a position in a triple and let G be an RDF graph that contains $|G|$ triples.
The function $occurrences^k_G(c)$ denotes the total number of occurrences of c at position k within graph G. In case that c is a variable, the number of occurrences is defined as $|G|$.

Definition 4.2 Selectivity Estimation of a Triple Pattern [5]**.**
Let (s, p, o) be a triple pattern and let G be an RDF graph.
The selectivity of a triple pattern is estimated as:

$$sel_G(s, p, o) := min(occurrences^{subject}_G(s), occurrences^{predicate}_G(p), occurrences^{object}_G(o))$$

The optimization approach sorts all triple patterns according to the number of contained variables as well as selectivity estimates. To be more precise, the triple patterns are primarily sorted in ascending order of their number of variables. Triple patterns sharing the same number of variables are subsequently sorted in ascending order of their estimated selectivity. Due to the difference between SPARQLGX and Koral described above, there is no need to cover the special case in which no statistics exist for a given triple pattern component when applying this approach to Koral. In example 4.1 the process of sorting the triple patterns of a query is clarified.

Example 4.1. Consider the query depicted in listing 2.2. Table 4.1 shows how the optimization algorithm orders the triple patterns of this query given the data set in listing 2.1, which contains 18 triples. In case of this query the resulting order is exactly the same as the order defined by the query. The first triple pattern contains the fewest variables, and therefore it remains in the first position. The second and the third triple pattern both have two variables. Consequently, these two triple patterns are ordered according to the estimated selectivity. Among these two triple patterns, the first one has the lower estimated selectivity of 4.

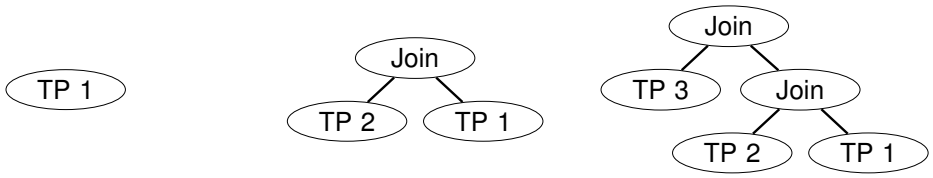
Triple Pattern	**Number of Variables**	**Estimated Selectivity**	**Resulting Order**
`?parent foaf:age "34"`	1	min(18, 5, 1) = 1	1
`?parent rel:parentOf ?child`	2	min(18, 4, 18) = 4	2
`?child foaf:firstName ?name`	2	min(18, 5, 18) = 5	3

Table 4.1.: Optimized order of the triple patterns of the query in listing 2.2.

After the triple patterns have been sorted, the final query plan is constructed based on this optimized order. The main focus of the construction algorithm is to avoid joining graph patterns that do not share common variables, since the result set of such joins corresponds to the Cartesian product of the intermediate result sets. Accordingly, the algorithm constructs a separate partial query execution tree for each group of triple patterns that can be joined while avoiding Cartesian products. The algorithm processes the triple patterns in the specified order and assembles the partial query execution trees incrementally. Whenever a triple pattern

is being processed, the algorithm attempts to join it with any of the partial query execution trees by checking for common variables. If no such query execution tree exists, a new partial query execution tree containing only this triple pattern is constructed. In a final step all partial query execution trees are combined to form the optimized query plan. Example 4.2 explains how the triple patterns of the query in listing 2.2 are processed.

Example 4.2. Consider the order of triple patterns defined in table 4.1. The algorithm starts by processing the first triple pattern. Since no partial query execution trees have been created yet, this triple pattern forms the first one. When processing the second triple pattern, the algorithm only has to check the previously constructed partial query execution tree and recognizes that the triple pattern shares the common variable `?parent` with it. Thus, the algorithm extends this partial query execution tree by joining it with the second triple pattern. In the next step the third triple pattern can successfully be joined with this partial query execution tree due to the common variable `?child`. Figures 4.1a, 4.1b and 4.1c depict the structure of the constructed partial query execution tree after processing the first, second and third triple pattern, respectively. In this example all triple patterns can properly be joined while avoiding Cartesian products, and as a consequence only one partial query execution tree has been assembled by the construction algorithm.



a) Partial query execution tree after adding the first triple pattern (TP 1)

b) Partial query execution tree after adding the second triple pattern (TP 2)

c) Partial query execution tree after adding the third triple pattern (TP 3)

Figure 4.1.: Structure of the partial query execution trees throughout the execution of the construction algorithm.

In order to provide a more detailed description of the algorithm, the corresponding pseudocode is depicted in listing 4.1. In lines 3 and 4, the partial query plans are iteratively constructed by adding the triple patterns in the defined order using the function `JoinPattern`, which is introduced in listing 4.2. The partial query execution trees are stored in the list `patternsList` and each of them is represented by a graph pattern. `JoinPattern` checks if the current triple pattern can be joined with any of the graph patterns in `patternsList`. The function `HaveCommonVariable` verifies that the triple pattern can be joined with a graph pattern by checking if they share at least one common variable. If no such graph pattern is found, the current triple pattern is added as a new graph pattern at the end of the list (line 9 of listing 4.2). However, if a graph pattern with a common variable has been found, then this graph pattern is removed from `patternsList` and a join operation, which

joins the current triple pattern and the found graph pattern, is constructed (lines 4-5 of listing 4.2). In this case `JoinPattern` is called recursively to add the newly constructed graph pattern to the list again. Note that this procedure also ensures that graph patterns, which could not be joined with each other previously, are subsequently joined together, as soon as new triple patterns have been included and the changed list of contained variables allows for it.

Once all triple patterns have been processed, `patternsList` may contain several graph patterns that could not be joined among each other. At this stage the algorithm can no longer avoid joining these graph patterns. As a result they are combined using the function `CombinePatterns`. In order to assemble the final query execution tree, this function starts by joining the last two graph patterns in `patternsList`. Afterwards, the resulting join operation is joined with the previous graph pattern in the list until all graph patterns have been included. As seen in example 4.2, it is also possible that `patternsList` only contains a single graph pattern. In this case this graph pattern already constitutes the final query execution tree and the function `CombinePatterns` does nothing.

```
Algorithm: BuildQueryExecutionTree(triplePatternList)
   Input:  triplePatternList - List of triple patterns ordered by number
                               of variables and estimated selectivity
   Result: Bushy query execution tree that comprises
           all triple patterns in triplePatternList

 1. Function BuildQueryExecutionTree(triplePatternList)
 2.   patternsList = []
 3.   foreach triplePattern in triplePatternList do
 4.     JoinPattern(patternsList, triplePattern)
 5.   end foreach
 6.   return CombinePatterns(patternsList)
```

Listing 4.1: Query Execution Tree Construction Algorithm [5].

```
Algorithm: JoinPattern(patternsList, newPattern)
   Input:  patternsList - A list of constructed graph patterns
           newPattern   - A graph pattern to be joined
   Result: List of graph patterns constructed by joining newPattern
           with a graph pattern in patternsList or appending
           newPattern to patternsList if no join is possible

 1. Function JoinPattern(patternsList, newPattern)
 2.   foreach pattern in patternsList do
 3.     if HaveCommonVariable(pattern, newPattern) do
 4.       patternsList.remove(pattern)
 5.       JoinPattern(patternsList, CreateJoin(newPattern, pattern))
 6.       return
 7.     end if
 8.   end foreach
 9.   patternsList.append(newPattern)
```

Listing 4.2: Function *JoinPattern* [5].

4.2. Distributed Join Order Optimization

To distribute the join order optimization each slave node optimizes its own query execution tree distinctly. Since the optimization algorithm relies on statistics, it is necessary that all slave nodes gather local statistics about their assigned graph chunk during the graph loading process. Just like the statistics gathered by the master node, the statistics that are collected by the slave nodes comprise the frequency of each triple component at each of the three triple positions. At the beginning of the query processing, all slave nodes receive an unoptimized query execution tree from the master node. Before the slave nodes start processing the query execution tree, they apply the join order optimization algorithm using their local statistics. Consequently, each slave node processes a query execution tree that is optimized according to the distribution of the RDF graph. In example 4.3 the distributed join order optimization is explained in detail.

Example 4.3. Consider the running example introduced in example 2.5 and assume that the RDF store Koral is running in the same setting as before. Tables 4.2 and 4.3 depict an excerpt from the local statistics that each slave node has gathered during the graph loading process. Note that these excerpts only cover the resources relevant to this example, whereas the full statistics cover all resources occurring in the graph chunks.

Resource	Frequency		
	Subject	Predicate	Object
foaf:firstName	0	2	0
foaf:age	0	2	0
rel:parentOf	0	4	0
ex:ownedBy	0	0	0
...	...	...	...

Table 4.2.: Statistics on slave node 1

Resource	Frequency		
	Subject	Predicate	Object
foaf:firstName	0	3	0
foaf:age	0	3	0
rel:parentOf	0	0	0
ex:ownedBy	0	2	0
...	...	...	...

Table 4.3.: Statistics on slave node 2

```
1. PREFIX ex:   <http://www.example.org/>
2. PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3. PREFIX rel:  <http://purl.org/vocab/relationship/>

4. SELECT ?name ?age
5. WHERE {
6.    ?dog   ex:ownedBy     ?owner .  // TP1
7.    ?owner rel:parentOf   ?child .  // TP2
8.    ?owner foaf:firstName ?name .   // TP3
9.    ?dog   foaf:age       ?age .    // TP4
10. }
```

Listing 4.3: Example of a SPARQL query.

The query in listing 4.3 retrieves the *name* of persons that are a parent and a dog owner as well as the *age* of the respective dog. Assume that this query has been sent to Koral and that both slave nodes have received the corresponding query execution tree. At this point both slave nodes start to optimize the received query execution

tree. For this purpose, each slave node computes the estimated selectivities of the four triple patterns and orders the triple patterns accordingly. The selectivity estimate and the number of variables of each triple pattern are shown in table 4.4. Thereafter, each slave node constructs an optimized query execution tree based on the new order of the triple patterns. Since the estimated selectivities vary between the two slave nodes, each of them constructs a different query execution tree. In figure 4.2 the resulting query execution trees are shown. Note that the estimated selectivity of `TP3` and `TP4` is identical on each slave node. Accordingly, the order of these two triple patterns depends on the exact implementation of the join order optimization algorithm. In this example `TP4` is joined before `TP3`.

Triple Pattern	**Number of Variables**	**Estimated Selectivity**	
		Slave node 1	Slave node 2
TP1 `?dog ex:ownedBy ?owner`	2	0	2
TP2 `?owner rel:parentOf ?child`	2	4	0
TP3 `?owner foaf:firstName ?name`	2	2	3
TP4 `?dog foaf:age ?age`	2	2	3

Table 4.4.: Selectivity estimates of the triple patterns in listing 4.3.

a) Optimized query execution tree on slave node 1.

b) Optimized query execution tree on slave node 2.

Figure 4.2.: Optimized query execution trees on both slave nodes.

4.3. Distributed Query Processing

The distributed join order optimization approach introduced in section 4.2 allows each slave node to optimize and process a distinct query execution tree. However, the default query processing strategy of Koral, as described in section 2.4.2, is not designed to handle a potentially different query execution tree on each slave node.

The default query processing strategy relies on the knowledge that each slave node processes an identical query execution tree. Based on this fact each slave node is implicitly aware of the structure of the query execution tree on all other slave

nodes. Thus, each slave node is able to determine to which operation a given variable mapping has to be transferred. Similarly, when receiving a variable mapping each slave node can use this knowledge to infer which operation of the query execution tree has to process it.

Given that some slave nodes may be processing different query execution trees, the basic assumption that each slave node processes the same query execution tree is no longer valid. Therefore, the slave nodes lack essential information in order to process the query. For this reason this section focuses on presenting the adjustments that have been made to support the processing of distinctly optimized query execution trees. In section 4.3.1 the extension of variable mappings that is required to trace them during the query processing is described. Section 4.3.2 explains how the query processing strategy of Koral has been adjusted and how the variable mappings are routed during query processing. Note that in the scope of this thesis, query execution trees which contain Cartesian products are not considered. Accordingly, these adjustments do not allow for the processing of such query execution trees. Lastly, section 4.3.3 covers the problem of determining whether a query has been processed completely.

4.3.1. Tracing of Variable Mappings

During the query processing the slave nodes communicate with each other by transferring variable mappings. To compensate for the lack of knowledge about the query execution trees of other slave nodes, these variable mappings are extended with additional information. In definition 4.3, *traceable variable mappings*, which replace regular variable mappings during the query execution, are introduced. Traceable variable mappings contain a regular variable mapping, a set of triple patterns whose results have already been joined, and a target variable. During the query processing the target variable indicates that the traceable variable mapping has been transferred based on the join responsibility of the resource that is assigned to this variable. Accordingly, the traceable variable mapping has to be joined on this variable. As long as a traceable variable mapping has not been transferred to another slave node, the target variable remains uninitialized.

Definition 4.3 Traceable Variable Mapping**.**
A traceable variable mapping *is a tuple* (μ, T, j) *that consists of a variable mapping* μ*, a set of triple patterns* $T \subseteq 2^{\mathscr{P}}$ *and the target variable* $j \in V$*.*

Example 4.4. The triple pattern `TP1` in table 4.4 matches two triples in the graph chunk of slave node 2. The resulting traceable variable mappings are:

$(\{(?\text{dog}, \texttt{dog:Merlin}), (?\text{owner}, \texttt{person:Craig})\}, \{\text{TP1}\}, -)$

$(\{(?\text{dog}, \texttt{dog:Merlin}), (?\text{owner}, \texttt{person:Mary})\}, \{\text{TP1}\}, -)$

The triple pattern set {TP1} indicates that so far only the results of triple pattern `TP1` are joined in these results. Additionally, the target variables of these traceable variable mappings are still uninitialized when they are created by the triple pattern match operation. This state is indicated by a dash (–) in place of the third element.

Definition 4.4 extends definition 2.10 of the `join` operator to sets of traceable variable mappings. For each pair of traceable variable mappings (μ_1, T_1, j_1) and (μ_2, T_2, j_2) whose variable mappings, μ_1 and μ_2, are compatible, a new traceable variable mapping is created by unioning the variable mappings μ_1 and μ_2, as well as the triple pattern sets T_1 and T_2. However, two traceable variable mappings are not joined if either of their triple pattern sets is a subset of the other. This constraint serves the purpose of avoiding unnecessary intermediate results. Example 4.5 illustrates the join of traceable variable mappings.

Definition 4.4 Join of Sets of Traceable Variable Mappings**.**
Let Ω_1, Ω_2 be sets of traceable variable mappings.
The `join` of Ω_1 and Ω_2 is defined as:

$$\Omega_1 \bowtie \Omega_2 := \{(\mu_1 \cup \mu_2, T_1 \cup T_2, -) \mid (\mu_1, T_1, j_1) \in \Omega_1 \wedge (\mu_2, T_2, j_2) \in \Omega_2 \wedge \mu_1 \sim \mu_2 \wedge T_1 \nsubseteq T_2 \wedge T_2 \nsubseteq T_1\}$$

Example 4.5. Consider the following sets of traceable variable mappings:
$\Omega_1 = \{(\{(?\text{dog}, \texttt{dog:Merlin}), (?\text{owner}, \texttt{person:Mary})\}, \{\text{TP1}\}, -)\}$
$\Omega_2 = \{(\{(?\text{owner}, \texttt{person:Mary}), (?\text{child}, \texttt{person:Juliet})\}, \{\text{TP2}\}, -)\}$

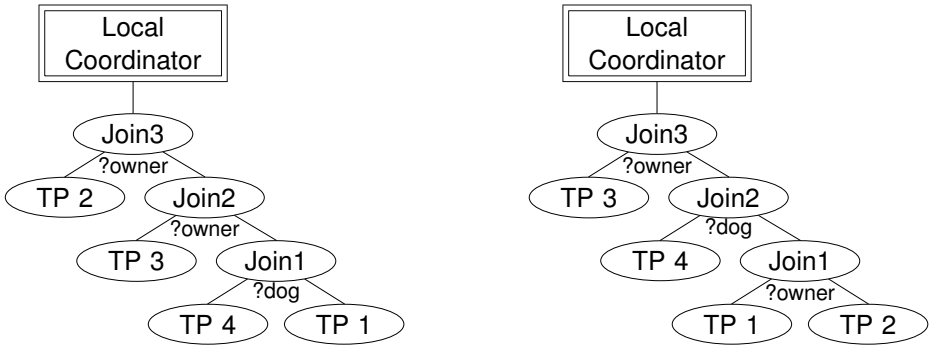
The result of $\Omega_1 \bowtie \Omega_2$ is Ω_3.
$\Omega_3 = \{(\{(?\text{dog}, \texttt{dog:Merlin}), (?\text{owner}, \texttt{person:Mary}), (?\text{child}, \texttt{person:Juliet})\}, \{\text{TP1,TP2}\}, -)\}$

The triple pattern set of the traceable variable mapping in Ω_3 provides the information that two traceable variable mappings, produced by matching `TP1` and `TP2` respectively, have been joined in order to create this result.

If $\Omega_1 \bowtie \Omega_3$ or $\Omega_2 \bowtie \Omega_3$ had been joined, the result would have been an empty set. In this case the traceable variable mappings are not joined since their triple pattern sets are subsets: $\{\text{TP1}\} \subseteq \{\text{TP1,TP2}\}$ and $\{\text{TP2}\} \subseteq \{\text{TP1,TP2}\}$. The traceable variable mapping in Ω_3 has been created by joining the results of `TP1` and `TP2`. Consequently, joining it with either of these result sets again would not provide any new information.

4.3.2. Routing of Traceable Variable Mappings

The default query processing strategy of Koral distributes the processing of the join operations among all slave nodes. Accordingly, the query operations transfer the intermediate results based on the join responsibilities of the contained resources to their parent query operation on either the same or another slave node. The introduction of traceable variable mappings allows the slave nodes to identify which triple patterns have already been processed. However, the communication between the slave nodes remains restricted due to potentially different query execution trees on the slave nodes. Each slave node is only aware of its own query execution tree. Thus, the slave nodes are unable to directly address variable mappings to a specific operation on any of the other slave nodes, since the knowledge, whether such an operation exists, is missing.



a) Query execution tree with local coordinator on slave node 1.

b) Query execution tree with local coordinator on slave node 2.

Figure 4.3.: Extended query execution trees.

In order to enable the exchange of variable mappings between the slave nodes, the query execution tree on each slave node is extended by a local coordinator. The extended query execution trees are depicted in figure 4.3. Each join operation in figure 4.3 is annotated with its join variables. In case a slave node identifies that another slave node is responsible for joining a traceable variable mapping, this mapping is transferred to the local coordinator of the respective slave node. Upon transferring a traceable variable mapping, its target variable is set to the join variable that has been used to determine the join responsibility. The local coordinator receives all traceable variable mappings that are sent by the other slave nodes. Moreover, the local coordinator knows the structure of the query execution tree on the respective slave node. Accordingly, the local coordinator is responsible for forwarding the received traceable variable mappings to the appropriate operations.

The local coordinator forwards traceable variable mappings to the join operations that have to process them. These join operations are determined based on the triple pattern set and the target variable of the traceable variable mappings (see below). Once the responsible join operations have been determined, all traceable variable mappings that are forwarded by the local coordinator are transferred as if they were sent by each of the child operations of the receiving join operation. As a consequence those traceable variable mappings are present in both sets that are being joined by the join operation. However, as defined in definition 4.4, a traceable variable mapping will not be joined with itself or other traceable variable mapping originating from the same triple pattern match operation. The reason for this is that traceable variable mappings are only joined if the resulting traceable variable mapping contains the results of more triple patterns than either of the ones being joined. This method guarantees that with each join step the results of at least one additional triple pattern are joined.

Forwarding of traceable variable mappings (General case): The local coordinator checks all join operations of the query execution tree. A join operation is a *valid target join operation*, if the traceable variable mapping has bindings for all join variables of the join operation and the target variable of the traceable variable mapping

is one of the join variables of the join operation. In the general case the traceable variable mappings are forwarded to all valid target join operations. Thus, a traceable variable mapping may be forwarded to more than one join operation.

Forwarding of traceable variable mappings of a single triple pattern (Special case): Traceable variable mappings, which are the result of a single triple pattern, are considered a special case. If a join operation exists which joins the results of the same triple pattern on the same join variable as the target variable indicated by the traceable variable mappings, then the traceable variable mapping is only forwarded to this exact join operation. If no such join operation exists, the traceable variable mapping is forwarded according to the general case instead.

Example 4.6. For the purpose of this example, assume that the join responsibility of the resources `dog:Merlin` and `person:Mary` has been assigned to slave node 1 and slave node 2, respectively. The graph cover with the adjusted join responsibilities[6] is shown in table 4.5.

Slave Node 1	**Slave Node 2**

`person:Craig|1 foaf:firstName|2 "Craig"|1`
`person:Craig|1 foaf:age|2 "33"|1`
`person:Craig|1 rel:parentOf|1 person:Juliet|2`
`person:Craig|1 rel:parentOf|1 person:Jack|2`
 	
`person:Mary|2 foaf:firstName|2 "Mary"|1`
`person:Mary|2 foaf:age|2 "34"|1`
`person:Mary|2 rel:parentOf|1 person:Juliet|2`
`person:Mary|2 rel:parentOf|1 person:Jack|2`
	`person:Juliet|2 foaf:firstName|2 "Juliet"|2`
	`person:Juliet|2 foaf:age|2 "8"|2`
	`person:Juliet|2 rel:siblingOf|2 person:Jack|2`
	
	`person:Jack|2 foaf:firstName|2 "Jack"|2`
	`person:Jack|2 foaf:age|2 "9"|2`
	`person:Jack|2 rel:siblingOf|2 person:Juliet|2`
	
	`dog:Merlin|1 foaf:firstName|2 "Merlin"|2`
	`dog:Merlin|1 foaf:age|2 "10"|2`
	`dog:Merlin|1 ex:ownedBy|2 person:Craig|1`
	`dog:Merlin|1 ex:ownedBy|2 person:Mary|2`

Table 4.5.: An example graph cover of the RDF graph in listing 2.1 with adjusted join responsibilities.

Assume that the slave nodes start processing the query execution trees shown in figure 4.3. During the query execution `TP1` produces the following traceable variable mappings on slave node 2:

$\mu_1 = (\{(?\text{dog}, \texttt{dog:Merlin|1}), (?\text{owner}, \texttt{person:Mary|2})\}, \{\text{TP1}\}, -)$

$\mu_2 = (\{(?\text{dog}, \texttt{dog:Merlin|1}), (?\text{owner}, \texttt{person:Craig|1})\}, \{\text{TP1}\}, -)$

Since the first join variable of `Join1` on slave node 2 is `?owner`, the traceable variable mappings μ_1 and μ_2 are transferred according to the join responsibility of the assigned resources `person:Mary|2` and `person:Craig|1`, respectively. Thus, μ_1 is not transferred to another slave node and is simply passed to the parent join operation `Join1` on slave 2. Meanwhile, μ_2 is transferred to the local coordinator on slave node 1. When transferring μ_2, the slave node sets the target variable of the traceable variable mapping to `?owner`. Consequently, the local coordinator on slave node 1 receives the traceable variable mapping in the following state:

$\mu_2 = (\{(?\text{dog}, \texttt{dog:Merlin|1}), (?\text{owner}, \texttt{person:Craig|1})\}, \{\text{TP1}\}, ?\text{owner})$

[6]The adjusted join responsibilities do not reflect the frequency of the corresponding resources on these slave nodes. Note however that it would be possible to establish a scenario with the given join responsibilities by adding additional triples about the two resources to the graph chunks.

The triple pattern set indicates that μ_2 is the result of a single triple pattern operation. Hence, the local coordinator on slave node 1 starts by checking the special case. As the query execution tree of slave 1 does not contain a join operation that joins the triple pattern `TP1` on the join variable `?owner`, μ_2 is forwarded according to the general case. The traceable variable mapping has bindings for the variables `?dog` and `?owner`, which means that all join variables of the join operations `Join1`, `Join2` and `Join3` on slave node 1 are bound. However, μ_2 is only forwarded to `Join2` and `Join3`, because the join variables of `Join1` do not contain the target variable `?owner`, which has been assigned to μ_2. Therefore, only `Join2` and `Join3` are valid target join operations.

The operation `TP2` on slave node 1 creates the traceable variable mappings:

$\mu_3 = (\{(?\text{owner}, \texttt{person:Craig|1}), (?\text{child}, \texttt{person:Jack|2})\}, \{\text{TP2}\}, -)$
$\mu_4 = (\{(?\text{owner}, \texttt{person:Craig|1}), (?\text{child}, \texttt{person:Juliet|2})\}, \{\text{TP2}\}, -)$
$\mu_5 = (\{(?\text{owner}, \texttt{person:Mary|2}), (?\text{child}, \texttt{person:Jack|2})\}, \{\text{TP2}\}, -)$
$\mu_6 = (\{(?\text{owner}, \texttt{person:Mary|2}), (?\text{child}, \texttt{person:Juliet|2})\}, \{\text{TP2}\}, -)$

In this case the first join variable of `Join3` on slave node 1 is `?owner`. Therefore, the join responsibility of the resources `person:Craig|1` and `person:Mary|2` is used to decide where these traceable variable mappings have to be joined. Accordingly, μ_3 and μ_4 are directly passed to the parent join operation `Join3` on slave 1. Since slave node 2 holds the join responsibility of `person:Mary|2`, μ_5 and μ_6 are transferred to the corresponding local coordinator on slave 2 and their target variables are set to `?owner`.

The traceable variable mappings μ_5 and μ_6 are the result of a single triple pattern operation as well. To forward these traceable variable mappings the local coordinator on slave node 2 checks the special case and searches for a join operation that joins the triple pattern `TP2` on the join variable `?owner`. The local coordinator finds the join operation `Join1`, which fulfills these criteria, and forwards μ_5 and μ_6 to `Join1` thereafter.

Forwarding of incomplete traceable variable mappings: Due to the strategy that the local coordinators use to forward traceable variable mappings, it is possible that a join operation produces an incomplete traceable variable mapping that does not have bindings for all join variables of the parent join operation. Alternatively, an incomplete traceable variable mapping may also be a traceable variable mapping that was produced by a join operation without a parent join operation and whose triple pattern set does not contain all triple patterns yet. To forward these incomplete traceable variable mappings, another adjustment has been made to the query processing strategy.

Incomplete traceable variable mappings are handled by the local coordinator on the same slave node. The local coordinator is responsible for finding join operation candidates that can process these incomplete traceable variable mappings instead. Once a join operation candidate has been found, it is used in place of the actual parent operation to transfer the corresponding incomplete traceable variable mapping

based on the default rule. Specifically, the join responsibility that is assigned to the first join variable of the join operation candidate is used to determine which slave node is responsible for joining the incomplete traceable variable mapping. In case that the current slave node is responsible for this join, the local coordinator forwards the traceable variable mapping directly to the join operation candidate. If, however, another slave node holds the join responsibility, the traceable variable mapping is transferred to the local coordinator on that slave node.

In order to find a join operation candidate the triple pattern set of the incomplete traceable variable mapping is used. The search for a join operation candidate starts at the triple pattern match operation that corresponds to the first triple pattern that is missing from the triple pattern set. All parent join operations of this triple pattern match operation are checked recursively. A join operation constitutes the join operation candidate if the incomplete traceable variable mapping has bindings for all of its join variables. In case no join operation candidate has been found when checking the first missing triple pattern, the process is repeated with the next triple pattern that is missing from the triple pattern set. Since query execution trees containing Cartesian products are not considered in this thesis, this method is guaranteed to find a missing triple pattern that shares a common join variable with the triple patterns that are already included in the traceable variable mapping. Additionally, each join variable has to be covered by at least one join operation of the query execution tree in order to produce the final results correctly. This method will eventually traverse the whole query execution tree and is therefore able to find a join operation candidate.

Example 4.7. Assume that the incomplete traceable variable mapping μ_1 has been created by the join operation `Join1` on slave node 2:

$\mu_1 = (\{(?\text{owner}, \texttt{person:Mary|2}), (?\text{child}, \texttt{person:Juliet|2}), (?\text{name}, \texttt{"Mary"})\}, \{\text{TP2,TP3}\}, -)$

The first triple pattern that is missing from the triple pattern set is `TP1`. Thus, the search is started at the triple pattern match operation `TP1`. The first parent join operation of `TP1` is `Join1`. The local coordinator detects that μ_1 has bindings for all join variables of the join operation `Join1`. The first join variable of `Join1` is `?owner`. Slave node 2 is responsible for joining the resource `person:Mary|2`, which is assigned to the variable `?owner`. As a consequence, the local coordinator forwards the traceable variable mapping μ_1 to the join operation `Join1` on slave node 2.

If the join operation `Join1` had not been a viable join operation candidate, the local coordinator would have proceeded to check its parent join operation `Join2`. Nevertheless, `Join2` would not have been a viable join operation candidate, as μ_1 does not have a binding for the join variable `?dog`. Afterwards, the local coordinator would have checked the join operation `Join3`, which is the parent join operation of `Join2`. The join operation `Join3` would have been a viable join operation candidate, however, the join operation `Join1` has been found prior to `Join3`.

Forwarding of finished traceable variable mappings: The last adjustment that has been made covers the forwarding of finished traceable variable mappings. Fin-

ished traceable variable mappings are traceable variable mappings whose triple pattern set indicates that the results of all triple patterns have been joined. Whenever a join operation creates a finished traceable variable mapping, this mapping is directly forwarded to the projection operation on the same slave node. Alternatively, if no projection operation exists in the query execution tree, the traceable variable mapping is transferred to the master node. Even if a parent join operation exists, it is not necessary to transfer finished traceable variable mappings there since they already constitute final results.

Example 4.8. Assume that the following traceable variable mapping has been created by the join operation `Join1` on slave node 2:

$$\mu_1 = (\{(?\text{dog}, \texttt{dog:Merlin|1}), (?\text{owner}, \texttt{person:Mary|2}), (?\text{child}, \texttt{person:Juliet|2}), \\ (?\text{name}, \texttt{"Mary"}), (?\text{age}, \texttt{"10"})\}, \{\text{TP1}, \text{TP2}, \text{TP3}, \text{TP4}\}, -)$$

The triple pattern set $\{\text{TP1}, \text{TP2}, \text{TP3}, \text{TP4}\}$ indicates that the results of all four triple patterns of the query have already been joined. Therefore, the traceable variable mapping μ_1 is forwarded to the projection operation on slave node 2.

The routing strategy for traceable variable mappings presented in this section may lead to the duplication of certain intermediate results. Traceable variable mappings may be forwarded to multiple join operations at once. Therefore, the join operations receiving these traceable variable mappings may produce duplicate intermediate results.

4.3.3. Finishing the Query Processing

Given the routing strategy for traceable variable mappings presented in section 4.3.2, it is no longer possible to detect whether the query execution is finished using the method described in section 2.4.2. Additionally, due to a potentially different query execution tree on each slave node, the operations of different slave nodes cannot directly address finish notifications to each other anymore.

To address this issue the local coordinators can either be in the *active* state or in the *idle* state. The active state indicates that at least one of the operations on the corresponding slave node is currently processing intermediate results. At the beginning of the query processing, the local coordinators start in the active state. A local coordinator switches into the idle state, if all triple pattern match operations have found every matching triple and all join operations have processed all of their received traceable variable mappings. The idle state indicates that all operations have temporarily finished the processing of intermediate results. Whenever a traceable variable mapping is received from another slave node while being in the idle state, the local coordinator returns to the active state.

Upon switching into the idle state, the local coordinator sends a *finish notification* to the local coordinator of all other slave nodes. Each finish notification contains (a) the number of traceable variable mappings that have been sent to the receiver of the finish notification and (b) the number of traceable variable mappings that have

been received from the receiver of the finish notification. For this purpose, each slave node has to keep track of the number of traceable variable mappings that have been sent to and received from the other slave nodes during the query processing.

If a local coordinator receives a finish notification while being in the active state, the finish notification is ignored. However, if the local coordinator is in the idle state, it compares the number of sent and received mappings contained in the notification with the number of mappings that have been received from and sent to the sender of the finish notification, respectively. In case that these numbers match, the local coordinator sends an *acknowledge notification* as a response. The acknowledge notification contains (a) the number of traceable variable mappings that have been sent to each individual slave node and (b) the number of traceable variable mappings that have been received from each individual slave node. In case that one of the comparisons fails, it implies that not all sent traceable variable mappings have been received yet. As a consequence, no acknowledge notification is sent since the query processing cannot be finished yet.

In the meantime, the local coordinator that had issued the initial batch of finish notifications awaits the response of the other slave nodes. Whenever a local coordinator returns to the active state, it will discard all acknowledge notifications that have been sent in response to the previously sent finish notification. If, on the contrary, an acknowledge notification has been received from every other slave node while being in the idle state, the local coordinator possesses complete knowledge about the number of sent and received traceable variable mappings of all slave nodes. In this situation, the local coordinator compares the number of sent and received traceable variable mappings for each pair of slave nodes. If all of the compared numbers match, it means that no intermediate results are currently being transferred over the network. Moreover, the fact that every slave node has responded with an acknowledge notification indicates that none of the slave nodes are currently processing intermediate results. Consequently, the local coordinator informs the master node as well as all other slave nodes that the query processing is finished.

Example 4.9. For the purpose of this example, assume that the RDF store Koral is running with three slave nodes. Consider figure 4.4 which depicts the number of traceable variable mappings that the three slave nodes have sent to and received from each other during the query processing.

Slave Node	**1**	**2**	**3**
Sent to	-	5	8
Received from	-	2	4

a) Slave node 1

Slave Node	**1**	**2**	**3**
Sent to	2	-	3
Received from	5	-	6

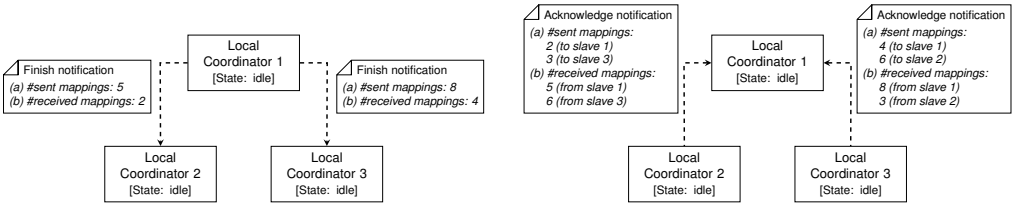
b) Slave node 2

Slave Node	**1**	**2**	**3**
Sent to	4	6	-
Received from	8	3	-

c) Slave node 3

Figure 4.4.: Example of the statistics of sent and received traceable variable mappings.

Once the local coordinator of slave node 1 changes into the idle state, it sends a finish notification to the local coordinators of slave 2 and slave 3. The finish noti-



a) Example of finish notifications. **b)** Example of acknowledge notifications.

Figure 4.5.: Example of finish and acknowledge notifications.

fications are shown in figure 4.5a. At this point, the local coordinators on slave 2 and slave 3 are already in the idle state. When the local coordinators receive the finish notifications, they compare the content of the finish notification with their local statistics about the number of sent and received traceable variable mappings.

The local coordinator on slave node 2 checks whether it has received 5 traceable variable mappings that have been sent by slave node 1 and whether slave node 1 has received 2 traceable variable mappings that have been sent by slave node 2. Similarly, the local coordinator on slave node 3 checks whether it has received 8 traceable variable mappings from slave node 1 and whether slave node 1 has received all 4 traceable variable mappings that have been sent by slave node 3. Since all of the compared numbers match, both local coordinators respond with an acknowledge notification, as shown in figure 4.5b.

Since the local coordinator on slave node 1 is still in the idle state and has not switched to the active state in the meantime, both acknowledge notifications are accepted. When the second acknowledge notification arrives, the local coordinator possesses all statistics shown in figure 4.4. Afterwards, the local coordinator on slave node 1 starts comparing these statistics for each pair of slave nodes. As soon as the local coordinator on slave node 1 has verified that no further traceable variable mappings are being transferred, it informs the other slave nodes and the master node that the query processing is finished.

5. Evaluation

In this section the distributed optimization approach presented in section 4 is evaluated, and the obtained results are presented. For the purpose of the evaluation, the distributed optimization approach has been implemented in Koral. The performance of the distributed optimization approach is compared to two baseline approaches. The default query processing strategy of Koral, as introduced in section 2.4.2, constitutes the first baseline. A centralized optimization approach, which has also been implemented in Koral, serves as the second baseline. The different systems are compared in terms of their query execution times. In section 5.1 the experimental setup that is used for the purpose of the evaluation is described. In section 5.2 the first and the second baseline are compared to each other in order to examine how efficient the centralized optimization approach is. In section 5.3 the distributed optimization approach is compared to the two baseline approaches. Section 5.4 focuses on discussing the results of the evaluation.

5.1. Experimental Setup

In the following the data set, the queries, the baseline approaches, the metrics and the setting which are used for the evaluation are described.
Data Set. For the evaluation a data set based on the Waterloo SPARQL Diversity Test Suite (WatDiv) [1] is used. The data set is created with the WatDiv data generator using the scale factor 100. The resulting data set is termed *WatDiv-10M* since it contains 10.9 million triples.
Queries. The WatDiv benchmark [1] provides a set of 17 query templates and 3 regular queries. The WatDiv query generator allows it to instantiate the query templates by replacing a placeholder term with a triple component occurring in a WatDiv data set. Accordingly, each query template has been instantiated once using the data set WatDiv-10M. In combination with the three regular queries, this amounts to 20 different test queries. A summary of the queries and their characteristics is provided in table 5.1. In this summary, *join variable count* denotes the number of variables that occur in at least two triple patterns. Consequently, these variables function as join variables. Moreover, *join variable degree* denotes the number of triple patterns that each individual join variable is involved in. The queries are categorized according to their shape as either *linear*, *star-shaped*, *snowflake-shaped* or *complex*. The linear queries are characterized by subject-object or object-subject joins, where a join variable occurs as the subject of one triple pattern and as the object of another triple pattern. The star-shaped queries contain one join variable

which appears in either the subject or object position of all triple patterns. The snowflake-shaped queries combine two or more star-shaped patterns in a single query. Lastly, the complex queries are combinations of the previous shapes while also containing triple patterns with high selectivity. All queries used for the evaluation are provided in appendix A.

Query	Category	Triple Pattern Count	Join Variable Count	Join Variable Degree
L1	linear	3	2	2, 2
L2	linear	3	2	2, 2
L3	linear	2	1	2
L4	linear	2	1	2
L5	linear	3	2	2, 2
S1	star-shaped	9	1	9
S2	star-shaped	4	1	4
S3	star-shaped	4	1	4
S4	star-shaped	4	1	4
S5	star-shaped	4	1	4
S6	star-shaped	3	1	3
S7	star-shaped	3	1	3
F1	snowflake-shaped	6	2	3, 4
F2	snowflake-shaped	8	2	6, 3
F3	snowflake-shaped	6	2	4, 3
F4	snowflake-shaped	9	2	6, 4
F5	snowflake-shaped	6	2	4, 3
C1	complex	8	4	4, 3, 2, 2
C2	complex	10	6	2, 3, 3, 3, 2, 2
C3	complex	6	1	6

Table 5.1.: Overview of the queries used for the evaluation.

Baselines. The *first baseline system* is the default query processing strategy of Koral, presented in section 2.4.2. The default query processing strategy of Koral does not perform any optimizations regarding the join order of the queries. It simply constructs and processes a left-linear query execution tree corresponding to each query.

The *second baseline system* is termed the *centralized optimization approach*. The centralized optimization approach utilizes the join order optimization described in section 4.1. However, instead of distributing the join order optimization among all slave nodes, the centralized optimization approach optimizes a single query execution tree on the master node while considering statistics about the whole RDF graph. Afterwards, the optimized query execution tree is processed by all slave nodes using the default query processing strategy of Koral.

Metrics. For each query that is being processed by one of the systems, the *query execution time* is measured. More specifically, query execution time is the time that passes between the query being received by Koral and the final result being transferred back to the query sender. In regard to the centralized and distributed optimization approach, the query execution time also includes the duration of the query optimization process.
Setting. All experiments are conducted on a server with 96GB of RAM and two Intel Xeon E5-2667 (2.90GHz) processors. Each processor has six physical cores, which amounts to a total of 24 cores due to hyper-threading being enabled on the server. The operating system of the server is Ubuntu 18.04.2 LTS with kernel 4.15.0-54. The RDF store Koral is written in Java. Therefore, Java version 1.8.0 is installed on the server in order to run it. The master node of Koral and four slave nodes are running on this server during the experiments.
Execution. A separate experiment is conducted for each of the three systems. For this purpose, the data set WatDiv-10M is loaded into the RDF store Koral using the *minimal edge-cut* graph cover strategy prior to the experiment. The minimal edge-cut graph cover strategy attempts to partition the triples of the data set into graph chunks of approximately the same size while minimizing the number of edges between the nodes of different graph chunks [9]. Once the store is ready to accept query requests, all 20 queries are sent to the store one after the other. Between two query requests the execution script pauses for 10 seconds. Additionally, a timeout of 15 minutes is set for all queries. In the case that one of the systems is unable to process a query within this time frame, the query will be skipped. This process is repeated a total of thirteen times while gathering measurements in each iteration. The first three iterations serve the purpose of warming the caches. Therefore, the corresponding measurements are discarded. The remaining ten measurements are used for the comparison of the systems in section 5.2 and section 5.3.

5.2. Evaluation of the Centralized Optimization Approach

In this section the results of the comparison between Koral's default query processing strategy (baseline 1) and the centralized optimization approach (baseline 2) are presented. Table 5.2 summarizes the results of the experiments on the data set WatDiv-10M. The table shows the number of unique results that each query produces as well as the average query execution time for both systems. If the execution time of a query exceeds the timeout of 15 minutes, the table indicates this by a dash. Boxplots corresponding to the results are shown in figure 5.1.

In regard to the linear queries, the centralized optimization approach is able to speed up the processing of the queries L1 and L2. Query L1 is processed about ten times faster, and the execution time of L2 is improved by about four seconds. The default query processing strategy constructs a left linear query execution tree for all queries. When this method is applied to L1, it results in a query execution tree that joins two triple patterns without a common variable. This join produces a large

Query	Number of unique results	Average query execution time (seconds)	
		Default Strategy (Baseline 1)	Centralized Optimization (Baseline 2)
L1	3	30.095	2.886
L2	103	25.131	20.862
L3	26	1.443	1.413
L4	61	1.432	1.453
L5	122	6.932	20.796
S1	13	7.971	7.944
S2	39	3.297	3.328
S3	0	3.265	3.295
S4	1	3.944	3.805
S5	26	3.296	3.255
S6	4	2.415	2.425
S7	0	2.414	2.343
F1	0	130.272	17.051
F2	11	7.530	7.191
F3	24	-	124.791
F4	0	98.321	98.347
F5	55	5.544	37.357
C1	16	13.226	46.153
C2	0	-	130.260
C3	434 169	73.768	30.576

Table 5.2.: Average query execution times on the data set WatDiv-10M.

set containing about 85 000 intermediate results, as it corresponds to the Cartesian product of the result sets of both triple patterns. All of the intermediate results have to be handled by the subsequent query operations. Also, query plans containing Cartesian products are more likely to lead to an imbalance of the workload of the slave nodes, since all join computations related to Cartesian products are processed by a single slave node. Consequently, this slave node may become a bottleneck during query processing, and thus subsequent query operations may be delayed. The centralized optimization achieves a speedup by avoiding Cartesian products when constructing the corresponding query execution tree. Even though the default query processing strategy constructs a similar query execution tree for the query L5, it does not impact the query execution time in a negative way, since one of the triple patterns involved in the Cartesian product only matches a single triple. On the contrary, due to the distributed setting of Koral the optimized version of L5 is slower since twice as many intermediate results have to be transferred between the

slave nodes during the query execution. The remaining linear queries, L3 and L4, only contain two triple patterns. As a consequence, both systems construct identical query execution trees for these queries, which is reflected by the similarity of the measured query execution times.

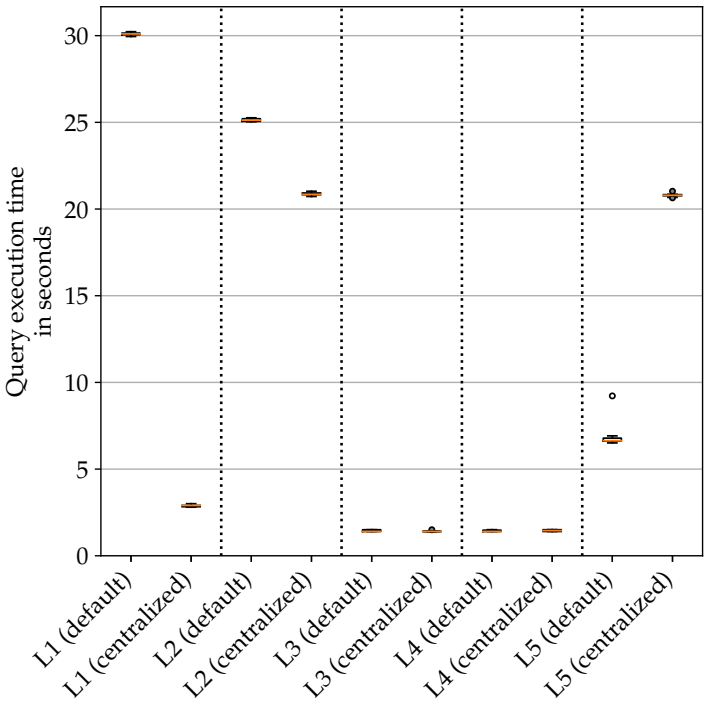
Due to the features of the star-shaped queries it is not possible to create a query plan that involves Cartesian products for queries S1 to S7. This fact benefits the default query processing strategy as it is prone to produce unfavorable query execution trees otherwise. Accordingly, table 5.2 indicates that both systems process the star-shaped queries in roughly the same time. The centralized optimization processes the queries S4 and S7 slightly faster.

In terms of the snowflake-shaped queries F1 and F3, the default query processing strategy constructs query plans involving Cartesian products once again. The intermediate result sets resulting from these Cartesian products are of size 142 378 and 4 649 938, respectively. Thus, the centralized optimization approach is capable of reducing the query execution time by creating query plans devoid of Cartesian products. The optimized version of F1 is more than seven times faster. Moreover, the default query processing strategy exceeds the timeout when executing query F3, whereas the centralized optimization processes this query within 124.791 seconds. Additionally, the centralized optimization approach increases the query performance of F2 by constructing a query plan whose join operations yield fewer intermediate results than those of the default query plan. In case of query F5 the opposite is observed. Here, the centralized optimization approach selects an unfavorable join that produces a large number of results. This choice leads to about a thousand times more intermediate results being transferred from slave node to slave node, which also shows in the increased query execution time.

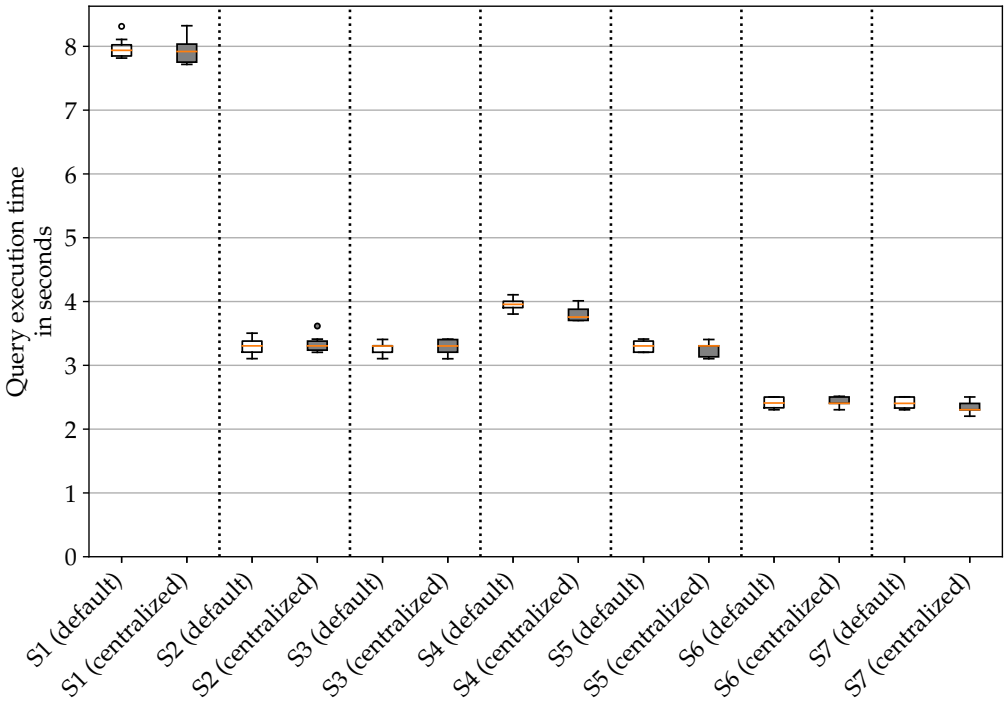
A similar scenario occurs in the case of query C1. One of the first join operations of the optimized query plan produces a larger amount of intermediate results in comparison to the query plan constructed by the default query processing strategy. The computational overhead, caused by the additional intermediate results, slows down the processing of the optimized version of C1. Contrary to that, the centralized optimization speeds up the queries C2 and C3. This is achieved by avoiding Cartesian products when constructing the query plan of C2. Consequently, the centralized optimization is able to process C2 in 130.26 seconds on average, while the default query processing strategy is unable to process the query request within the set time frame. Lastly, the centralized optimization approach is able to improve the query performance of C3 by considering the estimated selectivity during query planning. Due to the high selectivity of the triple patterns of C3, this procedure is beneficial in order to reduce the number of intermediate results that arise during query processing.

The evaluation of the two systems indicates that Cartesian products are often the cause of inferior query plans. In these cases the centralized optimization reduces the query execution time successfully by avoiding Cartesian products whenever possible. However, in some scenarios the centralized optimization approach constructs

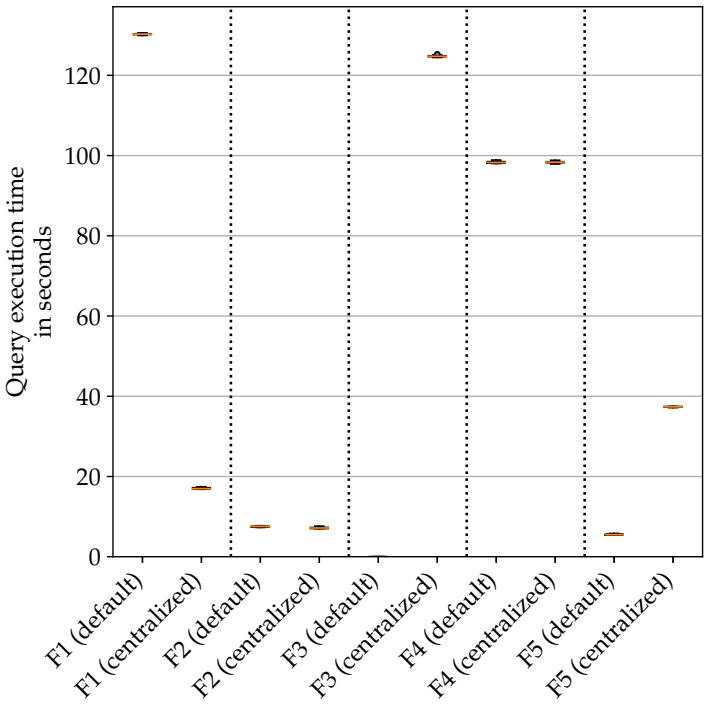
inferior query plans in comparison to the default query plans. The queries F5 and C1 are affected by such inferior query plans. The corresponding optimized query plans of these queries share a common pattern. These query plans are constructed in a specific way in order to avoid Cartesian products, which causes the query plans to contain two parallel join operations. While one of these join operations produces a small result set, the other one always yields a large set of intermediate results, and thus it slows down subsequent query operations considerably.



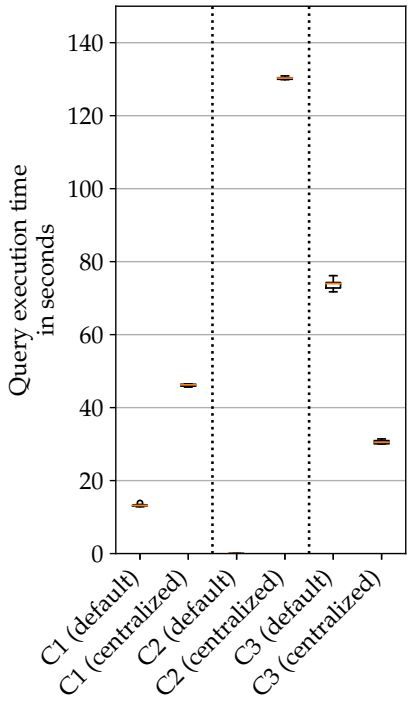
a) Linear Queries.



b) Star-shaped Queries.



c) Snowflake-shaped Queries.



d) Complex Queries.

Figure 5.1.: Comparison of the query execution times of the baseline systems on the data set WatDiv-10M.

5.3. Evaluation of the Distributed Optimization Approach

In this section the results of the distributed optimization approach are presented in comparison to the results of the default query processing strategy of Koral and the centralized optimization approach. The results of all experiments on the data set WatDiv-10M are summarized in table 5.3, and the corresponding boxplots are depicted in figure 5.2. Due to the distributed join order optimization that is employed by the distributed optimization approach, it possible that the slave nodes process different query execution trees. The queries which are affected by different optimized query execution trees on the slave nodes are marked with a star (*) in table 5.3. In case of the remaining queries, the optimized query execution trees are identical on all slave nodes and coincide with the query execution trees of the centralized optimization approach.

Query	**Number of unique results**	**Average query execution time (seconds)**		
		Default Strategy (Baseline 1)	Centralized Optimization (Baseline 2)	Distributed Optimization
L1	3	30.095	2.886	1.913
L2	103	25.131	20.862	20.782
L3	26	1.443	1.413	1.414
L4	61	1.432	1.453	0.752
L5	122	6.932	20.796	19.973
S1*	13	7.971	7.944	5.682
S2	39	3.297	3.328	1.824
S3	0	3.265	3.295	1.768
S4	1	3.944	3.805	2.042
S5*	26	3.296	3.255	1.590
S6	4	2.415	2.425	1.735
S7	0	2.414	2.343	1.563
F1*	0	130.272	17.051	10.390
F2*	11 (12)	7.530	7.191	11.165
F3*	24 (26)	-	124.791	140.198
F4*	0	98.321	98.347	117.580
F5*	55 (124-160)	5.544	37.357	35.201
C1*	16 (43)	13.226	46.153	52.090
C2*	0	-	130.260	287.014
C3	434 169	73.768	30.576	61.318

Table 5.3.: Average query execution times on the data set WatDiv-10M.

Result summary: The experiments indicate that in the case of most queries, the duplicate intermediate results produced by the distributed optimization approach seem to be responsible for an increased number of transferred intermediate results. Moreover, the general trend indicates that the number of join computations increases as well. As a consequence, the query execution times of most snowflake-shaped and complex queries are longer compared to the results of the centralized optimization approach.

Additionally, the observations indicate that the new method to detect finished queries employed by the distributed optimization approach may improve the query execution times. Linear and star-shaped queries with already short query execution times benefit from the new method, whereas such improvements could not be observed regarding the other queries. It is plausible that the additional computational overhead caused by the duplicate intermediate results outweighs the performance gain of the new method in case of the remaining queries.

Effect on the result sets: To ensure that the duplicate intermediate results do not influence the correctness of the query results, it has been verified that the distributed optimization approach yields all of the correct results for the queries in table 5.3. However, the final results of the queries F2, F3, F5 and C1 are directly affected by the duplication of intermediate results. In addition to the regular results, these queries return additional duplicate final results. The total number of results including duplicates is provided in brackets after the number of unique results in table 5.3. The query F2 returns 1 additional duplicate result, whereas the queries F3 and C1 return 2 and 27 additional duplicate results, respectively. Query F5 returns between 69 and 105 additional duplicate results. In this case the number of duplicate final results varies and depends on the exact time that the involved duplicate intermediate results are produced during the query processing. If the duplicate intermediate results are produced at an early stage of query processing, there are fewer intermediate results to join them with, and accordingly fewer duplicate final results will be returned. Nonetheless, these duplicate final results are still correct.

Effect on linear queries: The linear queries L2 and L3 are processed in the same time by the centralized as well as the distributed optimization approach. However, the execution times of the remaining linear queries L1, L4 and L5 are sped up by less than a second by the distributed optimization approach. To investigate the cause of these speedups, the results of the centralized optimization approach and the distributed optimization approach are compared in detail. The comparison of the results indicates that the number of intermediate results that are transferred between the slave nodes is the same when processing the queries L1, L4 and L5. Moreover, the number of join computations that the slave nodes have to process is identical for the queries L1 and L4, whereas the distributed optimization approach processes about twice as many join computations in case of query L5. Since these results do not explain the speedups, the speedups can be attributed to the adjustments that have been made to detect whether the query processing has been finished (see section 4.3.3).

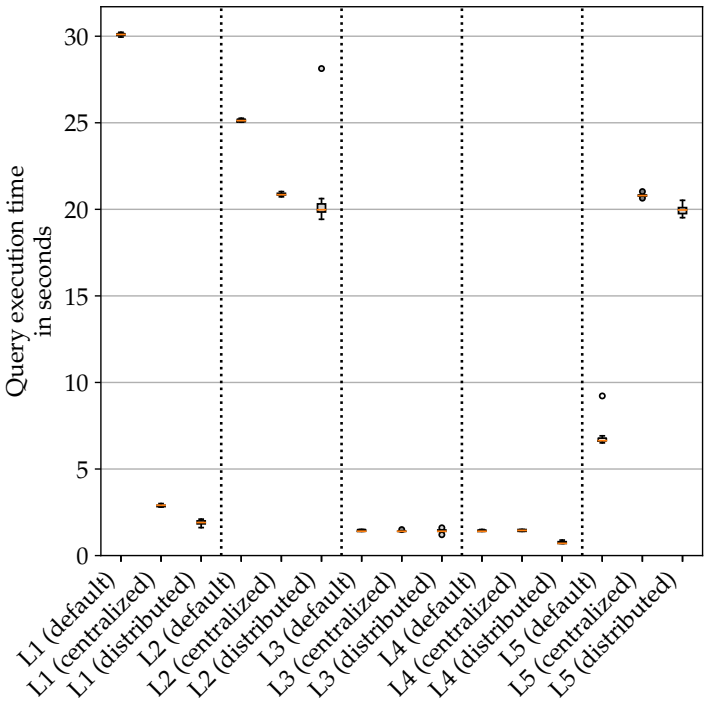
Effect on star-shaped queries: In terms of the star-shaped queries, the comparison of the results indicates that the distributed optimization approach processes these queries faster than the default query processing strategy as well as the centralized optimization approach. As a consequence of the distributed join order optimization, different query execution trees are processed by the slave nodes for the queries S1 and S5. In case of the query S5, the optimization does not influence the number of join computations during the query processing, whereas the distributed optimization approach was able to reduce the number of join computations for the query S1. Since improved query execution times have been observed for all star-shaped queries and since the reduction of join computations in the case of the query S5 is minor, the improvement of the query execution times likely stems from the new method to detect finished queries.

Effect on snowflake-shaped queries: Similar to the centralized optimization approach, the distributed optimization approach also benefits from avoiding the Cartesian products when processing the queries F1 and F3. During the processing of query F3, more intermediate results are transferred between the slave nodes in comparison to the centralized optimization approach. This causes the distributed optimization approach to be slower than the centralized optimization approach. Nevertheless, the distributed optimization approach still outperforms the default query processing strategy of Koral. In case of the query F1, the distributed optimization approach is able to reduce the number of intermediate results that are transferred between the slave nodes by more than half. Even though the number of join computations has increased from 332 to 22 894, this allows for query F1 to be processed in 10.390 seconds, which is 6.661 seconds faster than the centralized optimization approach. The opposite effect is observed for the queries F2 and F4. When processing these queries the distributed optimization approach transfers more intermediate results between the slave nodes than the centralized optimization approach. Additionally, the distributed optimization approach causes 9 and 33 times as many join computations in case of query F2 and F4, respectively. Consequently, the query execution times of these queries are increased. The results corresponding to query F5 are similar to those of query F1. In this case the distributed optimization approach manages to reduce the number of transferred intermediate results by about 25% while increasing the number of join computations by a factor of 8. On average the distributed optimization approach processes the query F5 in 2.156 seconds less than the centralized optimization approach.

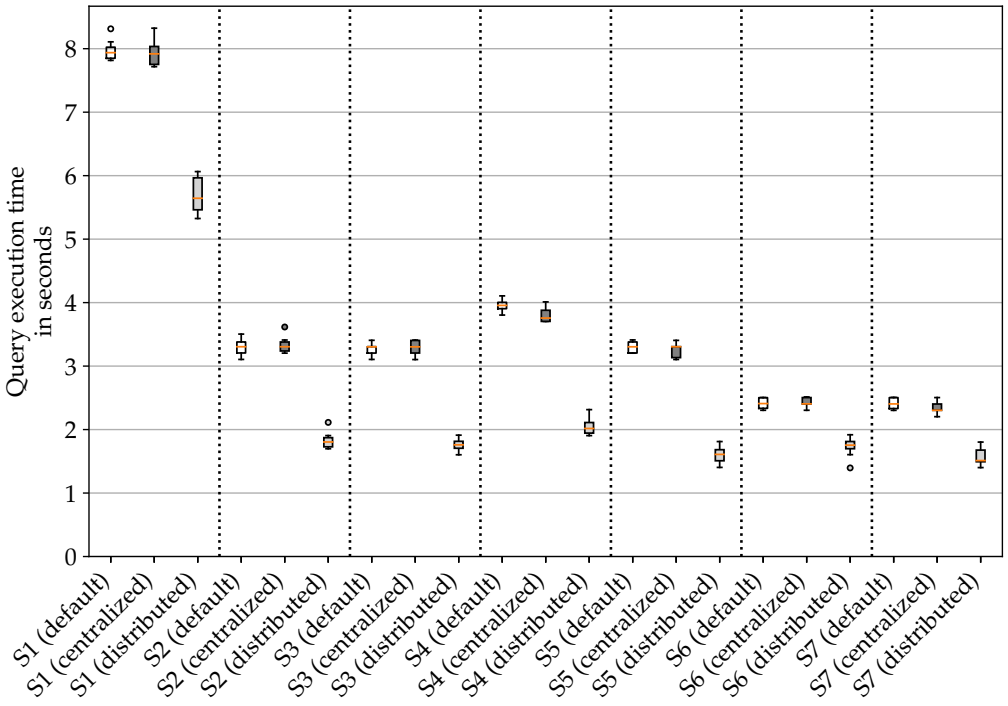
Effect on complex queries: The processing of the complex queries C1 and C2 is also affected by the additional duplicate intermediate results that are produced by the distributed optimization approach. As mentioned above, more intermediate results have to be transferred between the slave nodes as a consequence. In addition to that, the duplicate intermediate results also entail a higher number of join computations. Especially the query C2 is affected by the duplicate intermediate results. In this case twice as many intermediate results are transferred between the slave nodes and over 30 times as many join computations are being processed in comparison to the

centralized optimization approach.

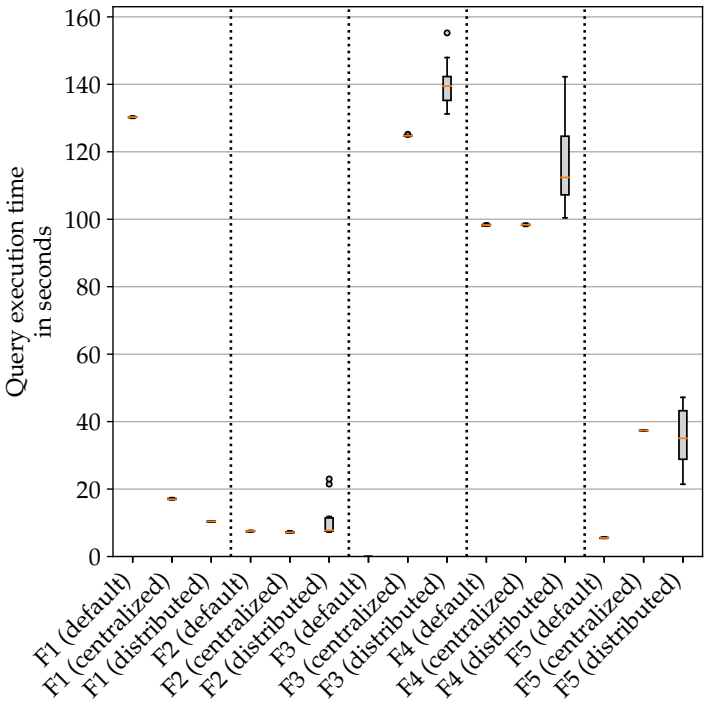
Lastly, in case of the query C3, the number of intermediate results that are transferred between the slave nodes as well as the number of join computations are identical when comparing the centralized and the distributed optimization approach. However, given the large number of intermediate results that are being processed in the case of the query C3, the increased query execution time can be attributed to the computational overhead that is caused by routing all of the traceable variable mappings according to their triple pattern set and their target variable.



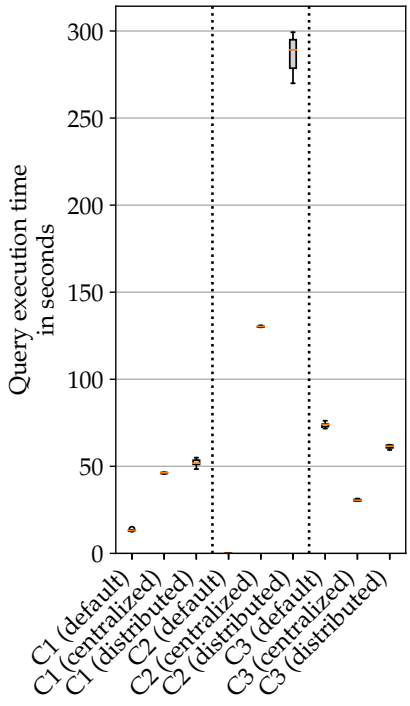
a) Linear Queries.



b) Star-shaped Queries.



c) Snowflake-shaped Queries.



d) Complex Queries.

Figure 5.2.: Comparison of the query execution times of all three systems on the data set WatDiv-10M.

5.4. Discussion

The evaluation of the results of the distributed optimization approach in section 5.3 has confirmed the expectation that the routing of traceable variable mappings causes a performance overhead. Especially queries that involve the processing of large amounts of intermediate results, such as query C3, are affected by this performance overhead. However, in the case of queries with short query execution times, the performance overhead seems to be negligible.

Another expectation has been that the distributed optimization approach is able to improve the query execution time of star-shaped queries. Star-shaped queries whose triple patterns share a common join variable in the subject position can be processed without exchanging intermediate results between the slave nodes. Thus, such queries can also be processed without producing any duplicate intermediate results. As a result, star-shaped queries should be able to benefit from the distinctly optimized query execution tree for each slave node, since the join order optimization aims to reduce the number of intermediate results that have to be processed by the individual slave nodes. While the results in section 5.3 have shown improved query execution times of star-shaped queries, these improvements could not be attributed to the optimization of the join order but rather to the new method to detect finished queries. Accordingly, the results of the evaluation were not able to confirm this expectation.

Contrary to the expectations, a general trend that indicates the reduction of join computations could not be observed. In case of most queries, the number of join computations has either risen or remained the same.

The observations suggest that the queries which have been used for the evaluation are not sufficient in order to judge the quality of the distributed optimization approach. None of the linear queries and only two of the five star-shaped queries resulted in different optimized query execution trees on the slave nodes during the experiment. Consequently, most of the results of the linear and star-shaped queries cannot be used to examine the effect of the distinct join order optimization. All of the linear and star-shaped queries are characterized by a short query execution time. Thus, the obtained results do not provide any insight into the processing of queries with an inherently higher workload.

Moreover, in its current state the distributed optimization approach produces duplicate intermediate results which impose an additional workload on the slave nodes during query processing. Thereby it is not possible to investigate the actual performance gain that is achieved by the distributed optimization approach since it is likely to be canceled out by the computational overhead that is induced by the duplication of intermediate results.

6. Conclusion

In order to improve the performance of query processing, distributed RDF stores usually employ a centralized query optimization strategy. Centralized query optimization strategies perform the optimization of the query execution tree on the master node and process the same optimized query execution tree on all slave nodes. However, in the settings of distributed RDF stores, the whole RDF graph is distributed among all slave nodes. As a consequence, a single optimized query execution tree may not coincide with the portion of the RDF graph that is stored on each individual slave node. Therefore, a centrally optimized query execution tree may only be beneficial for some slave nodes, whereas other slave nodes are disadvantaged by the centralized optimization.

To consider the distribution of the RDF graph during the optimization process, this thesis introduces a distributed query optimization approach that is suited for distributed RDF stores. The distributed optimization approach distinctly optimizes the join order of the query execution tree of each slave node according to statistics about its local RDF storage. This optimization attempts to reduce the number of intermediate results that have to be processed locally by the slave nodes. Consequently, it should allow the individual slave nodes to complete the processing of their query execution trees faster, and thus speed up the whole query processing. To support a potentially different query execution tree on each slave node, the transferred intermediate results have to be forwarded to the correct join operations on the corresponding slave node. A side effect of the routing strategy used to forward the intermediate results (see section 4.3.2) is that it may produce duplicate intermediate results during query processing. Lastly, the distributed optimization approach has been implemented in the distributed RDF store Koral to evaluate its performance.

The evaluation of the distributed optimization approach has revealed that especially queries with short query execution times benefit from the new method that is used to detect finished queries. However, the choice of queries used in the experiments and the duplicate intermediate results produced by the distributed optimization approach do not allow to investigate the effectiveness of the distributed optimization approach. While a performance gain may have been achieved by the distributed optimization approach, it is not noticeable in the results of the evaluation given the additional workload that is caused by the processing of duplicate intermediate results.

6.1. Future Work

The evaluation indicates that the duplication of intermediate results may severely affect the number of transferred intermediate results, the number of join computations and the query execution time. Therefore, one main aspect of future work is to investigate how the routing strategy of the distributed optimization approach can be adjusted to minimize the number of duplicate intermediate results. Ideally, the routing strategy can be changed such that traceable variable mappings are forwarded to as few join operations as necessary in an attempt to eliminate duplicate intermediate results. In case that it is not possible to eliminate duplicate intermediate results altogether, another approach would be to investigate methods that filter out duplicate intermediate results as early as possible during the query processing.

In the current state, the distributed optimization approach does not support the processing of query execution trees that involve Cartesian products. Expanding the routing and query processing strategy of the distributed optimization approach to support such join operations has to be addressed as part of future work.

Once the problem of duplicate intermediate results has been resolved, another aspect of future work would be to repeat the evaluation of the distributed optimization approach to assess its actual performance. For the purpose of this evaluation, the choice of queries should be adjusted to cover a wider variety of queries for each category. Each category should include queries with small and large result sets as well as queries with short and long query execution times. Furthermore, the choice of queries should predominantly consist of queries that result in different optimized query execution trees on the slave nodes.

Another interesting aspect would be to investigate how well the distributed optimization approach scales with the size of the data set as well as an increasing number of slave nodes. While the evaluation indicates that the new method to detect finished queries works in the setting of four slave nodes, it has to be further examined if it scales properly with an increasing number of slave nodes.

Appendix A: Evaluation Queries

```
1. SELECT ?v0 ?v2 ?v3 WHERE {
2.     ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes>
3.             <http://db.uwaterloo.ca/~galuc/wsdbm/Website2493> .
4.     ?v2 <http://schema.org/caption> ?v3 .
5.     ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v2 .
6. }
```

Listing A.1: Query L1 for WatDiv-10M.

```
1. SELECT ?v1 ?v2 WHERE {
2.     <http://db.uwaterloo.ca/~galuc/wsdbm/City182>
3.             <http://www.geonames.org/ontology#parentCountry> ?v1 .
4.     ?v2 <http://db.uwaterloo.ca/~galuc/wsdbm/likes>
5.             <http://db.uwaterloo.ca/~galuc/wsdbm/Product0> .
6.     ?v2 <http://schema.org/nationality> ?v1 .
7. }
```

Listing A.2: Query L2 for WatDiv-10M.

```
1. SELECT ?v0 ?v1 WHERE {
2.     ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
3.     ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes>
4.             <http://db.uwaterloo.ca/~galuc/wsdbm/Website3786> .
5. }
```

Listing A.3: Query L3 for WatDiv-10M.

```
1. SELECT ?v0 ?v2 WHERE {
2.     ?v0 <http://ogp.me/ns#tag>
3.             <http://db.uwaterloo.ca/~galuc/wsdbm/Topic189> .
4.     ?v0 <http://schema.org/caption> ?v2 .
5. }
```

Listing A.4: Query L4 for WatDiv-10M.

```
1. SELECT ?v0 ?v1 ?v3 WHERE {
2.     ?v0 <http://schema.org/jobTitle> ?v1 .
3.     <http://db.uwaterloo.ca/~galuc/wsdbm/City147>
4.         <http://www.geonames.org/ontology#parentCountry> ?v3 .
5.     ?v0 <http://schema.org/nationality> ?v3 .
6. }
```

Listing A.5: Query L5 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
    ?v0 <http://purl.org/goodrelations/includes> ?v1 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer887>
        <http://purl.org/goodrelations/offers> ?v0 .
    ?v0 <http://purl.org/goodrelations/price> ?v3 .
    ?v0 <http://purl.org/goodrelations/serialNumber> ?v4 .
    ?v0 <http://purl.org/goodrelations/validFrom> ?v5 .
    ?v0 <http://purl.org/goodrelations/validThrough> ?v6 .
    ?v0 <http://schema.org/eligibleQuantity> ?v7 .
    ?v0 <http://schema.org/eligibleRegion> ?v8 .
    ?v0 <http://schema.org/priceValidUntil> ?v9 .
}
```

Listing A.6: Query S1 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v3 WHERE {
    ?v0 <http://purl.org/dc/terms/Location> ?v1 .
    ?v0 <http://schema.org/nationality>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Country4> .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v3 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Role2> .
}
```

Listing A.7: Query S2 for WatDiv-10M.

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory2> .
    ?v0 <http://schema.org/caption> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v3 .
    ?v0 <http://schema.org/publisher> ?v4 .
}
```

Listing A.8: Query S3 for WatDiv-10M.

```
SELECT ?v0 ?v2 ?v3 WHERE {
    ?v0 <http://xmlns.com/foaf/age>
            <http://db.uwaterloo.ca/~galuc/wsdbm/AgeGroup1> .
    ?v0 <http://xmlns.com/foaf/familyName> ?v2 .
    ?v3 <http://purl.org/ontology/mo/artist> ?v0 .
    ?v0 <http://schema.org/nationality>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Country1> .
}
```

Listing A.9: Query S4 for WatDiv-10M.

```
SELECT ?v0 ?v2 ?v3 WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory2> .
    ?v0 <http://schema.org/description> ?v2 .
    ?v0 <http://schema.org/keywords> ?v3 .
    ?v0 <http://schema.org/language>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Language0> .
}
```

Listing A.10: Query S5 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v2 WHERE {
    ?v0 <http://purl.org/ontology/mo/conductor> ?v1 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre>
            <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre28> .
}
```

Listing A.11: Query S6 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v2 WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v1 .
    ?v0 <http://schema.org/text> ?v2 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/User19170>
        <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}
```

Listing A.12: Query S7 for WatDiv-10M.

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
    ?v0 <http://ogp.me/ns#tag>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Topic154> .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
    ?v3 <http://schema.org/trailer> ?v4 .
    ?v3 <http://schema.org/keywords> ?v5 .
    ?v3 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v0 .
    ?v3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory2> .
}
```

Listing A.13: Query F1 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
    ?v0 <http://ogp.me/ns#title> ?v2 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v3 .
    ?v0 <http://schema.org/caption> ?v4 .
    ?v0 <http://schema.org/description> ?v5 .
    ?v1 <http://schema.org/url> ?v6 .
    ?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v7 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre>
            <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre89> .
}
```

Listing A.14: Query F2 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
    ?v0 <http://schema.org/contentRating> ?v1 .
    ?v0 <http://schema.org/contentSize> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre>
            <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre89> .
    ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/makesPurchase> ?v5 .
    ?v5 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseDate> ?v6 .
    ?v5 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseFor> ?v0 .
}
```

Listing A.15: Query F3 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
    ?v2 <http://purl.org/goodrelations/includes> ?v0 .
    ?v0 <http://ogp.me/ns#tag>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Topic185> .
    ?v0 <http://schema.org/description> ?v4 .
    ?v0 <http://schema.org/contentSize> ?v8 .
    ?v1 <http://schema.org/url> ?v5 .
    ?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v6 .
    ?v1 <http://schema.org/language>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Language0> .
    ?v7 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}
```

Listing A.16: Query F4 for WatDiv-10M.

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
    ?v0 <http://purl.org/goodrelations/includes> ?v1 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer887>
        <http://purl.org/goodrelations/offers> ?v0 .
    ?v0 <http://purl.org/goodrelations/price> ?v3 .
    ?v0 <http://purl.org/goodrelations/validThrough> ?v4 .
    ?v1 <http://ogp.me/ns#title> ?v5 .
    ?v1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v6 .
}
```

Listing A.17: Query F5 for WatDiv-10M.

```
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
    ?v0 <http://schema.org/caption> ?v1 .
    ?v0 <http://schema.org/text> ?v2 .
    ?v0 <http://schema.org/contentRating> ?v3 .
    ?v0 <http://purl.org/stuff/rev#hasReview> ?v4 .
    ?v4 <http://purl.org/stuff/rev#title> ?v5 .
    ?v4 <http://purl.org/stuff/rev#reviewer> ?v6 .
    ?v7 <http://schema.org/actor> ?v6 .
    ?v7 <http://schema.org/language> ?v8 .
}
```

Listing A.18: Query C1 for WatDiv-10M.

```
SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
    ?v0 <http://schema.org/legalName> ?v1 .
    ?v0 <http://purl.org/goodrelations/offers> ?v2 .
    ?v2 <http://schema.org/eligibleRegion>
            <http://db.uwaterloo.ca/~galuc/wsdbm/Country5> .
    ?v2 <http://purl.org/goodrelations/includes> ?v3 .
    ?v4 <http://schema.org/jobTitle> ?v5 .
    ?v4 <http://xmlns.com/foaf/homepage> ?v6 .
    ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/makesPurchase> ?v7 .
    ?v7 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseFor> ?v3 .
    ?v3 <http://purl.org/stuff/rev#hasReview> ?v8 .
    ?v8 <http://purl.org/stuff/rev#totalVotes> ?v9 .
}
```

Listing A.19: Query C2 for WatDiv-10M.

```
SELECT ?v0 WHERE {
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v2 .
    ?v0 <http://purl.org/dc/terms/Location> ?v3 .
    ?v0 <http://xmlns.com/foaf/age> ?v4 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v5 .
    ?v0 <http://xmlns.com/foaf/givenName> ?v6 .
}
```

Listing A.20: Query C3 for WatDiv-10M.

References

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *Proceedings of The Semantic Web – ISWC 2014: 13th International Semantic Web Conference*, pages 197–212, Cham, 2014. Springer.

[2] M. Arenas, C. Gutierrez, and J. Pérez. On the Semantics of SPARQL. In *Semantic Web Information Management: A Model-Based Perspective*, pages 281–307. Springer, Berlin, Heidelberg, 2010.

[3] C. Bizer and R. Cyganiak. RDF 1.1 TriG. W3C Recommendation, W3C, 2014. http://www.w3.org/TR/2014/REC-trig-20140225/.

[4] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Conference on Consuming Linked Data*, volume 782 of *CEUR Workshop Proceedings*, Bonn, Germany, 2011.

[5] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, editors, *Proceedings of The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Part II*, volume 9982, pages 80–87, Cham, 2016. Springer.

[6] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, W3C, 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[7] D. Janke and S. Staab. Storing and Querying Semantic Data in the Cloud. In *Reasoning Web. Learning, Uncertainty, Streaming, and Scalability - 14th International Summer School 2018*, pages 173–222, Cham, 2018. Springer.

[8] D. Janke, S. Staab, and M. Thimm. Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In R. Usbeck, A. Ngonga, J.-D. Kim, K.-S. Choi, P. Cimiano, I. Fundulaki, and A. Krithara, editors, *Joint Proceedings of BLINK2017: Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data*, volume 1932 of *CEUR Workshop Proceedings*, Aachen, 2017.

[9] D. Janke, S. Staab, and M. Thimm. Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores. *Journal of Web Semantics*, 50:21–48, 2018.

[10] E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. M. Aroyo, editors, *Proceedings of The Semantic Web – ISWC 2006: 5th International Semantic Web Conference*, volume 4273, pages 399–413, Berlin, Heidelberg, 2006. Springer.

[11] A. Loizou, R. Angles, and P. T. Groth. On the Formulation of Performant SPARQL Queries. *Journal of Web Semantics*, 31:1–26, 2015.

[12] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. In *Proceedings of the VLDB Endowment*, volume 1, pages 647–659, 2008.

[13] A.-C. Ngonga Ngomo, S. Auer, J. Lehmann, and A. Zaveri. Introduction to Linked Data and Its Lifecycle on the Web. In *Reasoning Web. Reasoning on the Web in the Big Data Era*, volume 8714, pages 1–99. Springer, Cham, 2014.

[14] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. M. Aroyo, editors, *The Semantic Web – ISWC 2006: 5th International Semantic Web Conference*, volume 4273, pages 30–43, Berlin, Heidelberg, 2006. Springer.

[15] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[16] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[17] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based Query Optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 324–335, New York, NY, USA, 2012. ACM.

[18] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *The Semantic Web: Research and Applications*, volume 6088, pages 228–242. Springer, Berlin, Heidelberg, 2010.

[19] D. Wood, M. Lanthaler, and R. Cyganiak. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C, 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[20] H. Wu, A. Yamaguchi, and J.-D. Kim. Dynamic join order optimization for SPARQL endpoint federation. In *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems*, volume 1457 of *CEUR Workshop Proceedings*, Bethlehem, USA, 2015.