Study on Data Placement Strategies in Distributed RDF Stores

by
Daniel Dominik Janke

Approved Dissertation thesis for the partial fulfillment of the requirements for a
Doctor of Natural Sciences (Dr. rer. nat.)
Fachbereich 4: Informatik
Universität Koblenz-Landau

Chair of PhD Board: Prof. Dr. Jan Jürjens
Chair of PhD Commission: Prof. Dr. Susan Williams
Examiner and Supervisor: Prof. Dr. Steffen Staab
Further Examiner: Prof. Dr. Georg Lausen

Date of the doctoral viva: 31. January 2020

This thesis will be published in the book series Studies on the Semantic Web* at IOS Press.

*http://www.semantic-web-studies.net/

Abstract

The distributed setting of RDF stores in the cloud poses many challenges. One such challenge is how the data placement on the compute nodes can be optimized to improve the query performance. To address this challenge, several evaluations in the literature have investigated the effects of existing data placement strategies on the query performance. A common drawback in theses evaluations is that it is unclear whether the observed behaviors were caused by the data placement strategies (if different RDF stores were evaluated as a whole) or reflect the behavior in distributed RDF stores (if cloud processing frameworks like Hadoop MapReduce are used for the evaluation). To overcome these limitations, this thesis develops a novel benchmarking methodology for data placement strategies that uses a data-placement-strategy-independent distributed RDF store to analyze the effect of the data placement strategies on query performance.

With this evaluation methodology the frequently used data placement strategies have been evaluated. This evaluation challenged the commonly held belief that data placement strategies that emphasize local computation, such as minimal edge-cut cover, lead to faster query executions. The results indicate that queries with a high workload may be executed faster on hash-based data placement strategies than on, e.g., minimal edge-cut covers. The analysis of the additional measurements indicates that vertical parallelization (i.e., a well-distributed workload) may be more important than horizontal containment (i.e., minimal data transport) for efficient query processing.

Moreover, to find a data placement strategy with a high vertical parallelization, the thesis tests the hypothesis that collocating small connected triple sets on the same compute node while balancing the amount of triples stored on the different compute nodes leads to a high vertical parallelization. Specifically, the thesis proposes two such data placement strategies. The first strategy called overpartitioned minimal edge-cut cover was found in the literature and the second strategy is the newly developed molecule hash cover. The evaluation revealed a balanced query workload and a high horizontal containment, which lead to a high vertical parallelization. As a result these strategies showed a better query performance than the frequently used data placement strategies.

Zusammenfassung

Die Verteiltheit von RDF Cloud-Datenbanken beinhaltet viele Herausforderungen. Eine solche Herausforderung ist die optimale Verteilung der Daten auf die einzelnen Rechenknoten, so dass Anfragen möglichst schnell beantwortet werden können. Um eine solche optimale Verteilung zu finden, wurde in der Literatur bereits mehrfach untersucht, wie sich existierende Datenverteilungsstrategien auf die Anfrageperformanz auswirken. Bei vielen dieser Untersuchungen werden verschiedene RDF Cloud-Datenbanken miteinander verglichen. Dies hat den Nachteil, dass nicht klar ist, ob die gemachten Beobachtungen auf die Datenverteilungsstrategie zurückzuführen sind oder von anderen Faktoren verursacht werden. Bei anderen Untersuchungen wurden Cloud-Verarbeitungssysteme wie Hadoop MapReduce verwendet. In diesen Fällen ist es nicht klar, ob die Beobachtungen auf verteilte RDF Datenbanken übertragbar sind, die keine solchen Cloud-Verarbeitungssysteme nutzen. Um diese Nachteile zu überwinden, eine neue Methodik zum Benchmarking von Datenverteilungsstrategien entwickelt. Teil dieser Methodik ist eine datenverteilungsunabhängige verteilte RDF Datenbank, mit deren Hilfe die Wechselwirkungen zwischen der Datenverteilungsstrategie und der Anfrageabarbeitungsstrategie analysiert werden können.

Mit Hilfe dieser Benchmarking-Methodik wurden die häufig genutzten Datenverteilungsstrategien evaluiert. Diese Evaluation stellt die verbreitete Ansicht in Frage, dass Datenverteilungsstrategien, die sich wie die auf lokale Berechnungen fokussieren, Anfragen schneller abarbeiten können. Die Ergebnisse unserer Evaluation deuten darauf hin, dass Anfragen mit einer hohen Arbeitslast auf hashbasierten Datenverteilungsstrategien schneller ausgeführt werden können als z.B. bei der minimaler Kantenschnitt-Verteilung. Die Analyse der zusätzlichen Metriken weisen darauf hin, dass vertikale Parallelisierung (d.h., eine gut verteilte Arbeitslast) für eine effiziente Anfrageabarbeitung wichtiger sein kann als die horizontale Eindämmung (d.h., minimaler Datentransport).

Um eine Datenverteilungsstrategie mit einer hohen vertikalen Parallelisierung zu finden, wurde die Hypothese aufgestellt, dass wenn kleine zusammenhängende Tripelmengen auf demselben Rechenknoten gespeichert werden, jedoch gleichzeitig die Anzahl der auf den verschiedenen Rechenknoten gespeicherten Tripel balanciert wird, dies zu einer hohen vertikalen Parallelisierung führen kann. Um diese Hypothese zu untersuchen, wurden zwei solche Datenverteilungsstrategien vorgestellt und evaluiert: die in der Literatur gefundene überpartitionierte minimaler Kantenschnitt-Verteilung sowie die neu entwickelte Molekül-Hash-Verteilung. Bei ihrer Evaluation konnte eine balancierte Arbeitslast sowie eine hohe horizontale Eindämmung und somit eine hohe vertikale Parallelisierung festgestellt werden. Infolgedessen konnte eine bessere Anfrageabarbeitungsperformanz als bei den häufig verwendeten Datenverteilungsstrategien feststellt werden.

Publications and Disclaimer

Several parts of this thesis have been published at international journals or workshops as well as in book series.

- Chapter 2 and Chapter 3 that survey the related work of RDF stores in the cloud have been published as chapter 7 in the book "Reasoning Web 2018. Learning, Uncertainty, Streaming, and Scalability" [68].
- The methodology for benchmarking data placement strategies described in Chapter 4 as well as the evaluation of common graph cover strategies in Chapter 5 together with the additional evaluation details from Appendix C have been published in the Journal of Web Semantics [73] and the technical report [69]. The profiling system Koral has been published in the 2nd International Workshop on Benchmarking Linked Data 2017 (BLINK2017) [70] and as a poster at the 16th International Semantic Web Conference 2017 (ISWC2017) [71]. Initial evaluation results have been published in The International Workshop on Semantic Big Data 2017 (SBD2017) [72].
- The proof that the developed distributed query execution strategy is sound and complete, shown in Appendix B, has been published in the technical report [69].

The ideas presented in the publications are originated by the author of this thesis. The majority of the texts was also written by the author. The coauthors Steffen Staab and Matthias Thimm helped improving the ideas and the text quality.

Acknowledgments

First of all, I would like to thank Steffen Staab, who gave me the opportunity to obtain a PhD degree. During this process he continuously supported me by his scientific guidance,valuable discussions, critical feedback on how to improve my scientific approach, and provision of computational resources.

I am grateful to my postdoc supervisors especially Matthias Thimm for their advice and feedback. I would also like to thank my colleagues who helped by fruitful discussions and supported me with the everyday work. I want to thank Alex Baier and Guosong Xu for proof-reading my PhD thesis.

I am grateful to my parents, siblings and friends for their unrestricted encouragement.

Contents

CHAPTER 1

Introduction

The traditional web consists of documents written in natural language. These documents are easy to read and understand by humans but are difficult to be processed by computers since the natural language may cause problems. To simplify the automated processing of web documents and their meta data, the Resource Description Framework (RDF) was recommended by the World Wide Web Consortium (W3C) as a standardized format [34]. RDF is a directed graph whose edges and vertices are labeled. These labels serve as globally unique identifiers to address the entities described by the meta data as well as their relations.

Since the idea of the semantic web was raised, several organizations as well as companies have started to represent their data as RDF. For instance, the European Bioinformatics Institute (EMBL-EBI) has started to convert its datasets into RDF, resulting in a graph consisting of several trillions of triples once the conversion is finished [102].

In order to store and query RDF data efficiently, specialized databases called RDF stores have been developed. Traditionally, these RDF stores are running on a single computer. These centralized RDF stores satisfy the needs in many use cases. Nevertheless, for use cases in which analytical queries are performed on large data sets, the database needs to be fault tolerant, or a high number of queries have to be processed with a low response time, centralized RDF stores may not suffice [43]. Therefore, RDF stores in the cloud which use the computational and storage power of several computers have been designed to address these use cases. In cloud environments physical computers may be substituted by virtual computers to easily migrate them between physical computers.

When loading RDF data, RDF stores in the cloud have to decide on which compute nodes the individual data items are stored. During the distribution of the data items, the RDF stores in the cloud need to index where the data items were stored so that these data items can be located when they are requested by a user. When a user sends a data request in the form of a query, RDF stores in the cloud distribute the query operations over the different compute nodes. Each compute node applies the operations on its local data. Nevertheless, queries may require the combination of data from different compute nodes. In this case intermediate results have to be exchanged between compute nodes to process the overall results. Even though RDF stores in the cloud are already used by, for instance, BBC and Wikidata [84, 43], the cloud setting remains an area of research.

In order to speed up the query execution, researchers are investigating how the data placement strategies applied by the different RDF stores in the cloud to distribute the data items on the compute nodes affect the query execution time. For this purpose, some researchers evaluate the query execution times of several RDF stores in the cloud that use different data placement strategies (e.g., see [5], [29], [125] or [122]). Evaluating and comparing the query performances of different RDF stores in the cloud is helpful to judge the overall performances of RDF stores. Nevertheless, the compared RDF stores usually vary in more aspects than the data placement strategy such as the data indexing strategy, the distributed query execution strategy, etc. As a result, it is unclear which of the differences caused the varying observations.

Another way to find out data placement strategies that may lead to low query execution times is using a

single cloud processing framework such as Apache Spark[1]. This framework is used to create different graph covers of the same data set and execute queries on top of it afterwards. This type of evaluation is done by, e.g., [33]. One question that remains unclear is, to which extent the observations made with the specific cloud processing framework are also valid for RDF stores in the cloud that do not rely on a cloud processing framework.

In order to approximate the query execution times of RDF stores in the cloud that do not rely on cloud processing frameworks, evaluations in, e.g., [66] and [91], propose a hybrid RDF store that can use different data placement strategies to process queries. In order to efficiently process queries that do not need to exchange intermediate results between compute nodes, each compute node stores its graph chunk in a local centralized RDF store. Queries that need to combine intermediate results from different compute nodes are executed within the cloud processing framework Hadoop MapReduce[2]. The evaluations performed with this hybrid RDF store indicated that data placement strategies emphasizing local computation reduce the query execution time. However, since query processing on top of Hadoop MapReduce requires a potentially huge overhead of possibly several Hadoop jobs (see [74]), these results may differ from RDF stores in the cloud that do not rely on cloud computing frameworks at all. Nevertheless, for queries that do not need the exchange of intermediate results, the made observations reflect the behavior of RDF stores in the cloud that do not use cloud processing frameworks.

1.1. Research Questions

The main goal of this thesis is to *investigate the impact of data placment strategies on the performance of queries with a high workload executed on distributed master-slave RDF stores*. Thereby, distributed RDF stores are RDF stores in the cloud that allow for exchanging intermediate results between compute nodes during the query processing without the need of cloud processing frameworks or shared memory/storage. More specifically, the query performance of distributed RDF stores that follow a master-slave architecture as described in Section 2.2.2 are investigated. Examples of these distributed master-slave RDF stores are Clustered TDB [115], COSI [23], Partout [45], GraphDB [17], Blazegraph [2], SemStore [156], TriAD [55], DREAM [58], [118], DiploCoud [157] and [117].

Specifically, the following research questions will be addressed:

1. **How can the effect of data placement strategies on the query performance be evaluated?**
   To analyze the query performance of distributed RDF stores, no cloud computing frameworks is used. Instead the RDF graph is loaded and queried within a distributed RDF store. This ensures that the measurements made reflect the performance of queries in real, distributed RDF stores. In order to keep all evaluation variables (e.g., the indexing strategy) but the data placement strategy fixed, all data placement strategies are evaluated using the same distributed RDF store.

2. **How do frequently used data placement strategies impact the query performance?**
   A commonly held belief is that data placement strategies which reduce the number of transferred intermediate results show a better query performance. This belief is based on evaluations that used cloud computing frameworks to process queries. Therefore, this commonly held belief is challenged by analyzing the impact of data placement strategies on the query performance of distributed RDF stores.

[1] `https://spark.apache.org/`
[2] `https://hadoop.apache.org/`

3. **Which properties should data placements have to show a good query performance?**
   The analysis of the effects of frequently used data placement strategies on the query performance has revealed properties of data placement strategies that lead to faster or slower query execution times. Based on these observations, the properties of data placement strategies that allow for low query execution times are formulated, implemented and evaluated.

1.2. Research Contributions

To address the research questions described in the previous section, foundational knowledge about the working principles of RDF stores in the cloud has to be gained first. Therefore, *the first contribution of this thesis is to give an overview of the types of RDF stores in the cloud and the methodologies used to address the challenges of RDF stores in the cloud*. This survey is split into two parts. The first part in Chapter 2 describes the architecture type of RDF stores in the cloud and how queries are processed in the cloud setting, how compute nodes that contribute to a query can be identified and how fault tolerance is handled. The second part of the survey described in Chapter 3 presents different data placement strategies and how RDF stores in the cloud are evaluated.

The second contribution of the thesis is the development of a novel methodology for benchmarking data placement strategies, which is described in Chapter 4 to address research question 1. This methodology aims to make the interdependencies of the data placement strategies and the query processing understandable. This is achieved by comparing data placements in terms of query execution time. In addition it allows to investigate the following dimensions:

- *Load time* describes the time it takes to create a data placement. This is an indicator of how well a data placement strategy can scale by horizontal scaling of the cloud.
- *Storage balance* describes the extent to which each compute node stores a similar amount of data. This is an indicator that memory needs can be met with increasing data size by horizontal scaling of the cloud.
- *Horizontal containment* describes the extent to which computation of individual query results is local to the data stored on one (or few) compute node(s). This is an indicator that query processing is (to some extent) robust when the cloud is scaled horizontally.
- *Vertical parallelization* describes the extent to which different query results are computed in parallel on different compute nodes. This is an indicator that query processing can scale with growing result set sizes by horizontal scaling of the cloud.

Part of this methodology is an open source distributed RDF store (Koral) that serves as profiling system. Koral allows for measuring a large variety of metrics while executing queries on arbitrarily placed data.

Experiments based on this new methodology challenge the commonly held belief that data placement strategies emphasizing local computation lead to faster query executions. This is the *third contribution* of this thesis, which addresses research question 2. As described in Chapter 5, the results of the experiments indicate that, contrary to commonly held belief, *queries with a high workload may be executed faster on hash-based data distributions* than on graph-clustering-based data distributions. The analysis of the additional measurements indicates that *vertical parallelization* (i.e. a well-distributed workload) *may be more important than horizontal containment* (i.e. minimal data transport) for efficient query processing — even in a commodity network environment (1 GB/s). This analysis reveals that cloud computing frameworks as used in previous experiments such as [66] and [91] are a highly inefficient way of data transfer.

Another finding of the performed experiments is that none of the frequently used data placement strategies have a high vertical parallelization because they either balance the query workload among all compute nodes (as done by, e.g., the hash-based data placement) or provide a high horizontal containment by reducing the number of transferred intermediate results (as done by, e.g., graph clustering-based approaches). In order to achieve a good vertical parallelization and a fast query execution, a data placement strategy could split the graph into small sets of connected data items and then assign these sets to compute nodes such that the compute nodes store similarly amounts of data items.

To investigate this hypothesis and address research question 3, two such data placement strategies are described in Chapter 6. This is the *fourth contribution* of this thesis. The first strategy called *overpartitioned minimal edge-cut cover* is described in [55]. This strategy first performs a graph-clustering algorithm to split the graph into a high number of small partitions and thereafter assigns the partitions to compute nodes based on a greedy algorithm. Because its implementation did not scale well with the dataset size in the performed experiments, a novel second strategy called *molecule hash cover* has been developed. This is the *fifth contribution*. The new strategy assigns molecules[3] – a small set of connected triples – to compute nodes based on their hashes. Both strategies are implemented within Koral. The performed evaluation reveals that both data placement strategies have an even higher horizontal containment than the purely graph-clustering-based data placements and balance the query workload similarly to the hash-based data placements. As a result, *the vertical parallelization is higher and the query execution times are faster than in the frequently used data placement strategies*.

In short, the research contributions of this thesis are:

1. A survey of how RDF stores in the cloud address the challenges of the distributed setting.
2. A novel benchmarking methodology and its open source implementation that allow for a detailed understanding of the interdependencies of the data placement strategy and the query processing.
3. An evaluation of frequently used data placement strategies indicating that vertical parallelization is more important than horizontal containment and that hash-based data placements outperform graph-clustering-based data placements.
4. A novel data placement strategy called molecule hash cover that can be computed at scale and assigns connected triples to the same compute node while balancing the number of data items stored on the compute nodes.
5. An evaluation of the molecule hash cover and the overpartitioned minimal edge-cut cover, which indicates that data placement strategies that collocate closely connected triple sets while balancing the amount of data items stored on each compute node have a higher vertical parallelization and, as a result, lower query execution times than the frequently used data placement strategies.

[3] This strategy has been inspired by the definition of RDF molecules in [39], though it has been adapted for the needs of query processing.

CHAPTER 2

Foundations

In order to provide RDF stores that can scale to huge graph sizes, researchers have started to develop RDF stores in the cloud, where graph data is distributed over compute and storage nodes for scaling efforts of query processing and memory needs. The main challenges to be investigated for such development are: (i) strategies for data placement over compute and storage nodes, (ii) strategies for distributed query processing, and (iii) strategies for handling failure of compute and storage nodes. In this chapter (published in and taken from [68]), an overview of how these challenges have been addressed by scalable RDF stores in the cloud that have been developed in the last 15 years is given. This should give the required background to understand the remainder of this thesis.

In Section 2.1 the RDF and SPARQL is formalized as usual. Additionally, the data placement strategies are defined formally.

In Section 2.2, an overview of the different architectures of scalable RDF stores in the cloud is given. Basically there exist three types of architectures. The first type uses general cluster computing frameworks like Spark[1] or HBase[2] to perform queries on RDF graphs. The second type – so-called distributed RDF stores – splits the RDF graph into smaller parts that are then stored and queried on the compute and storage nodes. The last type are federated query processing systems. These systems do not have any influence on the data distribution over compute and storage nodes. Instead, they process queries over several RDF stores.

In case of cluster computing frameworks and distributed RDF stores, the RDF graph is distributed among several compute and storage nodes. The general procedure of data distribution strategies is to first split the graph into several not-necessarily disjoint triple sets. In relational and NoSQL databases this splitting is called sharding. Thereafter, the individual triple sets are assigned to compute and storage nodes. Since the topic of this thesis is the influence of the data distribution on the query performance, the reader is referred to the related work Chapter 3 in which an overview of how these two steps are performed by the existing RDF stores is given.

Querying a distributed data set is usually done by splitting the query into subqueries that can be answered locally without the need to exchange data over the network. The results of these subqueries are finally combined into the overall results of the query. In order to identify which parts of the queries can be answered locally on which compute nodes, an index is required that stores information about the data distribution. The different types of indices are described in Section 2.3. An overview of how distributed query processing is done by RDF stores in the cloud is given in Section 2.4.

Another challenge of scalable RDF stores in the cloud is that the failure of an individual compute or storage node should not lead to the failure of the complete RDF store. A brief overview of how this challenge is addressed is given in Section 2.5.

Due to the huge amount of RDF stores in the cloud that were developed in the last 15 years, distributed solutions for the handling of RDF streams or reasoning will not be presented. Interested readers are referred

[1] https://spark.apache.org/
[2] https://hbase.apache.org/

to [131]. Beside RDF stores in the cloud there also exist distributed graph databases like Sparksee[3] or Titan[4] as well as distributed graph processing frameworks like PGX.D [64] or PEGASUS [77]. They are not described in this chapter since they have not been presented as part of an RDF store, yet. Furthermore, this chapter gives an overview of how RDF stores in the cloud work. Readers interested in a performance comparison of RDF stores in the cloud are referred to, e.g., [51].

Section 2.1 was taken from [73]. The remaining sections were taken from [68].

2.1. Formalization of Graph Cover Strategies and SPARQL

2.1.1. Formalization of Graph Cover Strategies

RDF graphs are defined like in [56]. To illustrate their formalization, the graph shown in Figure 2.1 is used as running example. The graph represents the knows relationship between two employees of the university institute WeST and one employee of the Leibniz institute GESIS. Additionally, the graph includes the ownership of the dog Bello. The terms r:, e:, w:, g:, and f: abbreviate IRI prefixes.

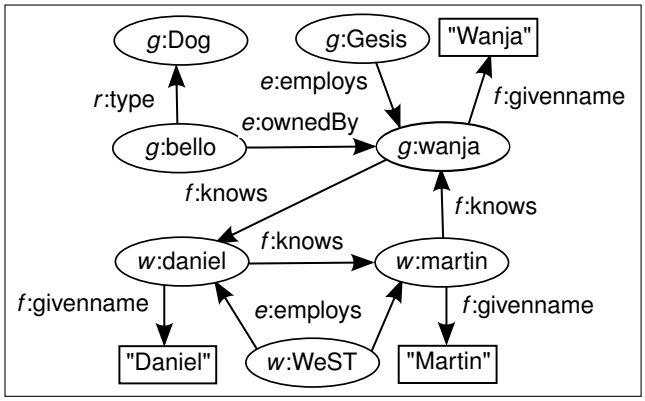


Figure 2.1.: Example graph describing the knows relationships between some employees of WeST and Gesis.

Assume a signature $\sigma = (I, B, L)$, where I, B and L are the pairwise disjoint infinite sets of IRIs, blank nodes and literals, respectively. The union of these sets is abbreviated as *IBL*.

Definition 1. The *set of all possible RDF triples* T for signature σ is defined by $T = (I \cup B) \times I \times (I \cup B \cup L)$. An *RDF graph* G or simply *graph* is defined as $G \subseteq T$. The set of all vertices contained in graph G is defined by $V_G = \{v | \exists s, p, o : (v, p, o) \in G \vee (s, p, v) \in G\}$.

$(s, p, o) \in T$ is also called a triple with *subject* s, *property* p and *object* o. To simplify later definitions, the functions subj(t), obj(t) and prop(t) return the subject, object or property of triple t, respectively. Likewise, subj(T), obj(T) and prop(T) are used to refer to the set of subjects, objects and properties in the triple set T.

In the context of distributed RDF stores, the triples of a graph have to be assigned to different compute nodes. The finite set of compute nodes is denoted as C in the rest of this thesis.

Definition 2. Let G denote an RDF graph. Then a *graph cover* is a function cover: $G \rightarrow 2^C$, that assigns each triple of a graph G to at least one compute node.

[3] http://www.sparsity-technologies.com/
[4] http://titan.thinkaurelius.com/

Definition 3. The function chunk returns the triples assigned to a specific compute node by a graph cover (*graph chunks*). It is defined as

$$\text{chunk}_{\text{cover}}: C \rightarrow 2^G$$
$$\text{chunk}_{\text{cover}}(c) := \{t | c \in \text{cover}(t)\} \quad .$$

For the description of the n-hop replication two additional definitions are required.

Definition 4. A graph cover of RDF graph G is *subject-complete*, if

$$\forall c \in C : \forall (s,p,o) \in \text{chunk}_{\text{cover}}(c) : \forall p', o' : \left(s,p',o'\right) \in G \Rightarrow \left(s,p',o'\right) \in \text{chunk}_{\text{cover}}(c) \quad .$$

Example 1. The graph cover shown in Figure 2.2 is subject-complete, since all triples with the same subject are located in the graph chunk of c_1 or c_2. A graph cover which is not subject-complete is shown in Figure 3.3. In this graph cover the triple (g:wanja, f:givenname, "Wanja") was assigned to c_1 whereas the triple (g:wanja, f:knows, w:daniel) was assigned to c_2.

Definition 5. A *(directed) path P* is a sequence $\langle t_0, t_1, ..., t_n \rangle$, if $\forall i \in [0,n] : t_i \in G$, $|\{t_0, t_1, ..., t_n\}| = n+1$ and

$$\forall j \in [1,n] : t_{j-1} = (s_{j-1}, p_{j-1}, s_j) \wedge t_j = (s_j, p_j, o_j) \quad .$$

The *length of path P* is $n+1$.

Example 2. In the example graph shown in Figure 2.1, $\langle$(w:daniel, f:knows, w:martin), (w:martin, f:knows, g:wanja)$\rangle$ is a path of length 2.

Definition 6. The *(directed) diameter* of a graph G is the length of the longest shortest path within G between two vertices of G.

Example 3. The example graph shown in Figure 2.1 has a diameter of 4 since the longest shortest paths within the example graph G are $\langle$(g:bello, e:ownedBy, g:wanja), (g:wanja, f:knows, w:daniel), (w:daniel, f:knows, w:martin), (w:martin, f:givenname, "Martin")$\rangle$ and the other path $\langle$(g:Gesis, e:employs, w:wanja), (g:wanja, f:knows, w:daniel), (w:daniel, f:knows, w:martin), (w:martin, f:givenname, "Martin")$\rangle$ with a length of 4.

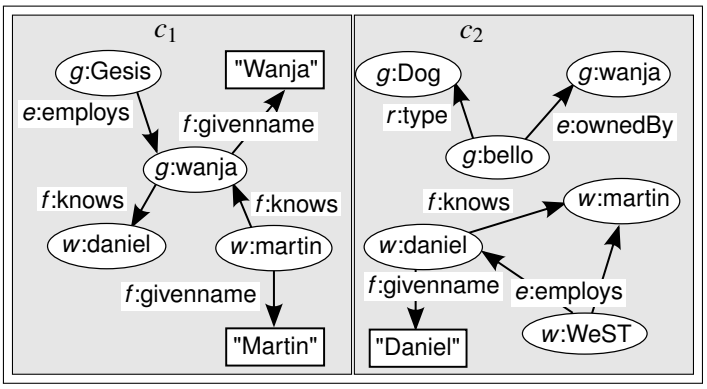


Figure 2.2.: An example graph cover of the example graph.

2.1.2. Formalization of SPARQL

The used SPARQL core is defined as done in [123], [120] and [16]. For this definition the infinite set of variables V that is disjoint from *IBL* is required. In order to distinguish the syntax of variables from other RDF terms, they are prefixed with ?. The syntax of SPARQL is defined as follows.

Definition 7. A *triple pattern* is a member of the set $TP = (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$.

Definition 8. A *basic graph pattern* (BGP) is a

1. triple pattern.
2. a conjunction $B_1.B_2$ of two BGPs B_1 and B_2.

Definition 9. A *SELECT query* is defined as `SELECT` W `WHERE` $\{B\}$ with $W \subseteq V$ and B a BGP.

Example 4. The following SELECT query returns the names of all persons known by employees of WeST that own the dog Bello. It contains a basic graph pattern that concatenates four triple patterns. In the following examples `?v1 <f:knows> ?v2` is abbreviated as tp_1, `?v2 <f:givenname> ?v3` as tp_2 and so on. All following examples in this section will refer to this query.

```
SELECT ?v3 WHERE {
    ?v1 <f:knows>  ?v2.
    ?v2 <f:givenname>  ?v3.
    <w:WeST> <e:employs> ?v1.
    <gs:bello> <e:ownedBy> ?v2
}
```

Before the semantics of a SPARQL query can be defined, some additional definitions are required. In the following $\mathscr{Q}$ represents the set of all SPARQL queries.

Definition 10. The function $\text{var} : \mathscr{Q} \to 2^V$ returns the set of variables occurring in a SPARQL query. It is defined as:

1. $\text{var}(tp)$ is the set of variables occurring in triple pattern tp.
2. $\text{var}(B_1.B_2) := \text{var}(B_1) \cup \text{var}(B_2)$ for the conjunction of the two BGPs B_1 and B_2.
3. $\text{var}(\texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}) := W \cap \text{var}(B)$ for $W \subseteq V$ and B a BGP.

Definition 11. A *variable binding* is a partial function $\mu : V \nrightarrow IBL$. The *set of all variable bindings* is $\mathscr{O}$.

The abbreviated notation $\mu(t)$ with $t \in TP$ means that the variables in t are substituted according to μ.

Example 5. The following three partial functions are variable bindings, that assign values to some variables. μ_1 would be an intermediate result produced by the first triple pattern of the example query in example 4 whereas μ_2 and μ_3 would be produced by the second triple pattern.

$$\mu_1 = \{(?v_1, \text{w:martin}), (?v_2, \text{g:wanja})\}$$
$$\mu_2 = \{(?v_2, \text{g:wanja}), (?v_3, \text{"Wanja"})\}$$
$$\mu_3 = \{(?v_2, \text{w:martin}), (?v_3, \text{"Martin"})\}$$

Definition 12. Two variable bindings μ_i and μ_j are *compatible*, denoted by $\mu_i \sim \mu_j$, if

$$\forall ?x \in \text{dom}(\mu_i) \cap \text{dom}(\mu_j) : \mu_i(?x) = \mu_j(?x).^5$$

Example 6. The variable bindings μ_1 and μ_2 from example 5 are compatible since in both variable bindings g:wanja is assigned to $?v_2$ which is the only variable occurring in the domains of both variable bindings. μ_1 and μ_3 as well as μ_2 and μ_3 are not compatible because they assign different values to common variables.

Definition 13. The *join* of two sets of variable bindings Ω_1 and Ω_2 is defined as

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

The variables contained in $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ are called *join variables*.

Example 7. The join of the two variable bindings sets $\{\mu_1\}$ and $\{\mu_2, \mu_3\}$ from example 5 produces a result set only containing the variable binding $\{(?v_1, \text{w:martin}), (?v_2, \text{g:wanja}), (?v_3, \text{"Wanja"})\}$ because only μ_1 and μ_2 are compatible.

[120] and [16] define the semantics of a SPARQL query as follows:

Definition 14. The *evaluation* of a SPARQL query Q over an RDF Graph G, denoted by $[\![Q]\!]_G$, is defined recursively as follows:

1. If $tp \in TP$ then $[\![tp]\!]_G = \{\mu | \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in G\}$.
2. If B_1 and B_2 are BGPs, then $[\![B_1.B_2]\!]_G = [\![B_1]\!]_G \bowtie [\![B_2]\!]_G$.
3. If $W \subseteq V$ and B is a BGP, then $[\![\texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}]\!]_G = \text{project}(W, [\![B]\!]_G) = \{\mu_{|W} | \mu \in [\![B]\!]_G\}$.[6]

The execution of a query requires the translation of a SPARQL query into a query execution tree. This tree defines the individual operations and their execution sequence. Thereby, each node of the query execution tree consists of three components: (i) the name of the operation to be executed, (ii) the set of variables that are bound in the resulting variable bindings and (iii) the set of child operations.

Definition 15. Let L_{node} be the set of node labels, then a *query execution tree* of a query Q, denoted as $\langle\!\langle Q \rangle\!\rangle$, is defined recursively as follows:

1. If $tp \in TP$ then $\langle\!\langle tp \rangle\!\rangle = (tp, var(tp), \varnothing)$.
2. If B_1 and B_2 are BGPs, then $\langle\!\langle B_1.B_2 \rangle\!\rangle = (\text{join}, var(B_1) \cup var(B_2), \{\langle\!\langle B_1 \rangle\!\rangle, \langle\!\langle B_2 \rangle\!\rangle\})$.
3. If $W \subseteq V$ and B is a BGP, then $\langle\!\langle \texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\} \rangle\!\rangle = (\text{project}, W, \langle\!\langle B \rangle\!\rangle)$.

The set of all query execution trees is called Υ.

Definition 16. The *variables common for all child trees* of a query execution tree are defined as follows.

$$\text{cVars} : \Upsilon \to 2^V$$
$$\text{cVars}((l, W, C)) := \bigcap_{(l', W', C') \in C} W' \ .$$

[5] $\text{dom}(\mu)$ refers to the set of variables of this binding.

[6] $\mu_{|W}$ means that the domain of μ is restricted to the variables in W.

Example 8. Figure 2.3 shows a graphical representation of one query execution tree for the example query from example 4. The following query execution tree represents the first join in its mathematical representation. It has the two child trees $(\text{tp}, \{?v_1, ?v_2\}, \varnothing)$ and $(\text{tp}, \{?v_2, ?v_3\}, \varnothing)$. Their common variables are $\{?v_2\}$.

$$
\begin{aligned}
&(\text{join}, \{?v_1, ?v_2, ?v_3\}, \{\\
&\qquad (\text{tp}, \{?v_1, ?v_2\}, \varnothing),\\
&\qquad (\text{tp}, \{?v_2, ?v_3\}, \varnothing)\\
&\})
\end{aligned}
$$

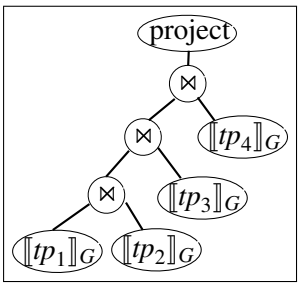


Figure 2.3.: An example query execution tree from the query in example 4.

2.2. Architectures

The RDF stores in the cloud can be categorized into three groups that characterize their architecture. The first type of RDF stores in the cloud makes use of cloud computing frameworks (see Section 2.4). These frameworks hide the complexity of distributed systems from the developers. This reduced developing complexity comes at the cost of limited influence on, e.g., the data placement. To overcome this limitation, developers of distributed RDF stores have to address the challenges of data placement, distributed query processing and fault tolerance on their own (see Section 2.2.2). In contrast to this, federated RDF stores aim to query data from several RDF stores that manage the stored data on their own (see Section 2.2.3). One application scenario would be querying several remote SPARQL endpoints that can be found in the linked open data cloud.

One RDF store in the cloud that caused a lot of attention after its launch is Neptune[7]. Due to a lack of descriptions that can be found, it is unclear which architecture it has.

2.2.1. RDF Stores Using Cloud Computing Frameworks

Implementing a distributed system is a challenging task. To reduce the complexity, several RDF stores in the cloud are realized on top of cloud computing frameworks. As shown in Fig. 2.4, these RDF stores need a master node that translates the RDF graph into some format that can be stored within the cloud computing frameworks. This is the job of the graph converter. Similarly, SPARQL queries need to be translated into queries or tasks that can be executed on the cloud computing framework to produce the query results that are then transferred back to the user. This is done by the query translator.

One of the first cloud computing frameworks that was used to build RDF stores in the cloud is Hadoop[8] [152]. RDF stores like SHARD [129], HadoopRDF [44] and CliqueSquare [46] transform the RDF graph into one

[7] `https://aws.amazon.com/neptune/`
[8] `https://hadoop.apache.org/`

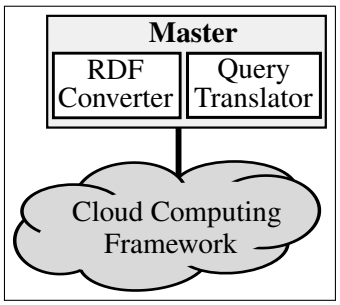


Figure 2.4.: Architecture of RDF stores using cloud computing frameworks.

or several files that are stored in the distributed file system of Hadoop [142]. SPARQL queries are translated into one or several jobs that are executed within Hadoop. Depending on how the RDF graph is separated into files, one or several files are processed during query execution.

To simplify the usage of Hadoop, the high-level query language and execution framework Pig[9] [112] was developed on top of Hadoop. Instead of translating SPARQL into Hadoop jobs directly, systems like PigSPARQL [136] and RAPID+ [83] translate SPARQL into the Pig query language. Pig then translate the code into Hadoop jobs and tries to optimize the orchestration of the individual jobs.

One of the main limitations of Hadoop is that the result of each individual task has to be written back into the distributed file system. To overcome this limitation, Spark[10] [161] was developed. Spark stores the result of each job in main memory. These results can be used by several other jobs before the final result is optionally persisted on disk. Spark is used by SPARQLGX [52], S2RDF [138], SPARQL-Spark [110] and PRoST [29].

On top of Spark the graph processing framework GraphX[11] [47] was developed. In this framework a graph can be loaded and algorithms can be performed on it. These algorithms are vertex-centric, i.e., each vertex is able to receive, process and send messages to its neighboured vertices. S2X [135] translates SPARQL queries into such vertex-centric algorithms to produce the query results. Similar to S2X TripleRush [146], Random Walk TripleRush [148] (both basing on Signal/Collect [147]) and [48] use vertex-centric graph processing frameworks.

Another type of cloud computing frameworks are NoSQL (Not only SQL) database systems. These systems usually scale well with a high number of compute nodes and are fault tolerant. Their drawback is that they usually do not provide ACID transactions. One type of NoSQL systems are key-value stores. These systems map a unique key to an arbitrary value. DynamoDB[12] [35] is such a distributed key-value store. It is used in AMADA [24] to index and store the triples of the RDF graph.

Column-family stores are another type of NoSQL database systems. These systems store tabular data like in relational databases. Instead of storing all data of a row physically together, column-family stores locate all entries of a set of columns (i.e., a column-family) physically together. Examples of these stores are HBase[13] (used by Jena-HBase [81], H_2RDF+ [116]), Cassandra[14] [89] (used by CumulusRDF [87]), Accumulo[15] (used by Rya [127] and RDF-4X [4]) and Impala[16] [130] (used by Sempala [137] and [124]). RDF stores relying on

[9] https://pig.apache.org/
[10] https://spark.apache.org/
[11] https://spark.apache.org/graphx/
[12] https://aws.amazon.com/de/dynamodb/
[13] https://hbase.apache.org/
[14] https://cassandra.apache.org/
[15] https://accumulo.apache.org/
[16] https://impala.apache.org/

these column-family stores usually vary in the way how they store the RDF graph in tables.

The last type of NoSQL database systems that are used in RDF stores are document stores. Document stores store the data in documents which are objects with arbitrary fields and values. Each of the fields can be used to index the data. The RDF store [32] uses Couchbase[17] and D-SPARQ [109] uses MongoDB[18].

2.2.2. Distributed RDF Stores

In contrast to RDF stores that use cloud computing frameworks, distributed RDF store have to address the challenges resulting from the distribution. To reduce the complexity of these challenges, most distributed RDF stores are realized with a master-slave architecture. In this architecture a dedicated compute node is the master. It is responsible for coordinating the individual slaves that store the RDF graph and process the queries. The disadvantage of this architecture is that the master node can easily become a bottleneck of the distributed RDF store. To overcome this limitation some distributed RDF stores are realized with a peer-to-peer architecture in which the design of every compute node is identical.

Distributed RDF Stores with Master-Slave Architecture

In distributed RDF stores that have a master-slave architecture there exists one master and several slaves. The master is a dedicate compute node that is responsible for the coordination of all slaves. The slaves are the compute nodes that store the graph and process the queries. The general architecture of such a distributed RDF store is shown in Figure 2.5.

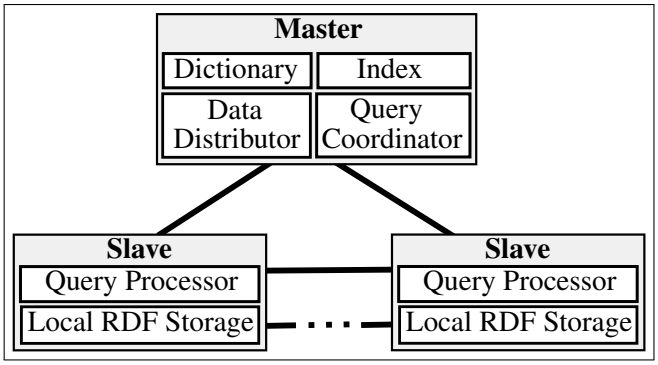


Figure 2.5.: Master-slave architecture used by distributed RDF stores.

When a graph is loaded, the master first replaces each string identifier of a resource by a shorter unique integer identifier. This replacement reduces the storage size of the graph that needs to be processed. The mapping between the string and integer identifiers are stored in the dictionary. Thereafter, the data distributor assigns the triples to the individual slaves. Thereby, an index is created which keeps track on which slave which part of the graph is stored. Each slaves stores the triples assigned to him in its local RDF storage.

In order to query the RDF store, a query is send to the master. The query coordinator first encodes all string identifier in the query with the help of the dictionary. The query is then translated into a query execution tree and optimized. With the help of the index, the query coordinator can decide which part of the query can be executed on which slave and initiates the query processing on the slaves. On each slave the query processor processes the (sub)query assigned to him on the local RDF storage. The intermediate results can then be

[17] https://www.couchbase.com/
[18] https://www.mongodb.com/

directly exchanged between all slaves. The final results are sent back to the query coordinator. With the help of the dictionary the query coordinator replaces the integer identifier of the results by their string identifiers and sends them back to the user.

The architectures of Custered TDB [115], COSI [23], Partout [45], GraphDB [17], Blazegraph [2], SemStore [156], TriAD [55], DREAM [58], [118], DiploCoud [157] and [117] are as described above. But there also exist variations of this architecture. For instance in Trinity.RDF [162], WARP [65], YARS2 [63] and 4store [61] the query coordinator also has to join intermediate results.

Some of the components of the master can be distributed among the slaves. For instance, in EAGRE [163] the data distribution is performed on all slaves in parallel and the index is distributed over all slaves. Furthermore, distributed RDF stores do not necessarily need all of the presented components. For instance, PHDStore [10], 2way [119] and [27] do not need a global index and/or dictionary. If distributed RDF stores adapt the data distribution during runtime based on the actual workload, the master also contains a redistribution controller as in AdPart [60], AdHash [9, 59] and Spartex [5].

Beside these pure distributed RDF stores there also exist hybrid RDF stores that also use some cloud computing infrastructure. In Sedge [159] a reimplementation of Pregel [97] is used to process distributed queries. A more common approach is, to use Hadoop to process only distributed joins as done in [66], [40], [160], SPA [93], VB-Partitioner [91], SHAPE [92] and [155].

Distributed RDF Stores with peer-to-peer Architecture

In distributed RDF stores that follow the peer-to-peer architecture all compute nodes – called peers – consist of the same components. This architecture has the advantage that no single compute node can become a bottleneck by design. The disadvantage is that usually no compute node knows how many compute nodes exist and how data are distributed among all compute nodes. Usually in systems like Edutella [38, 111], RDFPeers [25], PAGE [36], GridVine [6, 31], RDFCube [101], Atlas [78], 3RDF [12], [13] and [114], a distributed index is used that routes requests to the compute node storing the requested data. Since this distributed index is the central component of the architecture, the implementation choice of the index determines how the triples of the RDF graph are assigned to the compute nodes. The architecture of most peer-to-peer distributed RDF stores is shown in Figure 2.6.

To load an RDF graph, the complete graph can be sent to any compute node. The data distributor asks the distributed index on which compute node the individual triples should be stored. Based on this decision the triples are distributed over the compute nodes. When a compute node receives triples from a data distributor, it inserts them into its local RDF store and updates the distributed index. In order to speed up the graph loading procedure, the RDF graph can be split into several parts that are processed by the data distributors on several compute nodes in parallel.

When a user sends a query to any of the compute nodes, the query coordinator creates the query execution tree. Based on the index the query coordinator decides which part of the query execution tree is sent to which compute node. When a query processor receives some subquery from a query coordinator, it retrieves the requested information from its local RDF store. The intermediate results are sent to other compute nodes or back to the query coordinator where they will be further processed. Finally, the query coordinator computes the overall results and send them back to the user.

In contrast to the master-slave architecture, peer-to-peer distributed RDF stores usually do not have a global dictionary. As a consequence string identifiers are used when transferring triples during the loading of a graph or intermediate results during the query processing.

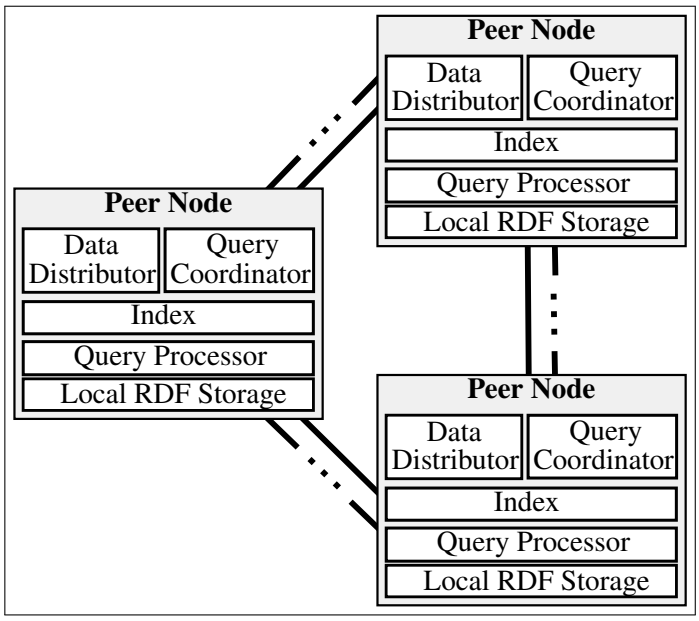


Figure 2.6.: Peer-to-peer architecture used by distributed RDF stores.

2.2.3. Federated RDF Stores

In the linked open data cloud[19] there exist many public data sets and SPARQL endpoints. In order to combine several data sets and query them together a naive approach would be to download all data sets and load them into a single RDF store. This naive approach has several disadvantages. First of all, it requires a lot of computational resources to store and process several data sets at once. Furthermore, when the data sets change, the system would need to keep track of these changes and update its local copies of the data sets.

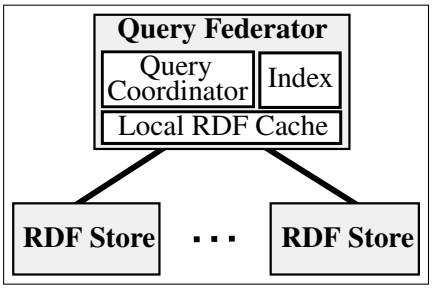


Figure 2.7.: Architecture of federated RDF stores.

To overcome these limitations, federated RDF stores have been developed. Their basic idea is to query remote RDF stores directly. The general architecture of federated RDF stores is shown in Figure 2.7. In order to query the remote RDF stores a global index is required that indicates which RDF stores contain data that are relevant for the query. This index is created by retrieving statistical information that are provided by the remote RDF stores (e.g., SPLENDID [50], WoDQA [8], LHD [151], DAW [134], SemaGrow [26], FEDRA [105], LILAC [106] and Odyssey [104]) or the user (e.g., DARQ [128]), by sending special queries to the

[19] `http://lod-cloud.net/`

remote RDF stores (e.g., FedX [141], ANAPSID [7, 107], Lusail [100]) or by observing the results that are returned during the processing of user queries (e.g., ADERIS [96]). Also combinations of these strategies are possible as in Avalanche [18]. Also the absence of the index is possible if the resource IRIs are dereferenced as proposed by SIHJoin [88].

When a user sends a query, the query coordinator transforms it into a query execution tree. With the help of the index it can decide which remote RDF store can contributed to the query. Based on this decision it forwards the query or parts of it to the corresponding remote RDF stores. The returned intermediate results are joined by the query coordinator and finally sent back to the user. In order to speed up the query execution, the query federator also contains a local RDF cache in which data retrieved by previous queries is cached. This cached data can be reused for future queries, which might reduce the number of queries that needs to be sent to the remote RDF stores.

2.3. Indices

RDF stores in the cloud distribute the triples of an RDF graph over several computers. When a query is sent to these RDF stores, they require an index which can tell on which compute nodes the data contributing to the query processing is located. These indices are either stored on a single compute node – i.e., the master or the query federator – (see Section 2.3.1) or they are distributed over several compute nodes (see Section 2.3.2). A centralized index has the advantage that it has knowledge about the whole RDF graph. To reduce the size of the index, some type of compression needs to be applied. In contrast to this distributed indices need fewer aggregation, since they are stored across all compute nodes. But a single index lookup may require the routing of the lookup via several compute nodes until the required information is found.

2.3.1. Centralized Indices

Hash-Based Index

In distributed RDF stores that distribute triples based on their hash values (for instance, Virtuoso Clustered Edition [41], 4store [61] or Trinity.RDF [162]), no explicit index is required. Based on the knowledge of all compute nodes, the hash function and the triple elements that were hashed, the compute node to which triples were assigned to based on a specific pattern can be identified.

Statistics-Based Index

Another type of centralized indices base on statistical information about the resources occurring in the data stored on the individual compute nodes. In RDF stores like DARQ [128], FedX, [150] and Sedge [159] the frequency of every subject, property and object occurring on each compute node is counted and stored. Since this information does not tell anything about the RDFS types stored on the compute nodes, systems like SPLENDID [128], WoDQA [128], LHD [128], SemaGrow [128] and Avalanche [128] bases on VoID descriptions [11] of the data stored on each compute node. These descriptions contain the occurences of URIs in the data set, the used RDFS types and the properties that occur in triples whose objects occur as subject in triples assigned to a different compute node. Since this information might be complicated to collect in a federated setting, if the remote RDF stores do not provide VoID descriptions, ANAPSID [128] restricts itself to only count the occurrences of properties and RDFS types.

If a triple with two specific constants is looked for, the indices described so far could only restrict the number of queried compute nodes by either of the two constants. To restrict the number of queried compute

nodes even further, LILAC [106] and SemStore [156] additionally count how frequently all subject-property, property-object and subject-object combinations occur.

Since not all subjects and objects occurring in a data set have an RDFS type, the RDF store Odyssey [104] defines the type of an subject s by the set of all properties occurring in at least one triple with subject s. Since the number of types may be very large, types with similar property sets are merged. Additionally, it counts how frequently instances of these types are connected by properties.

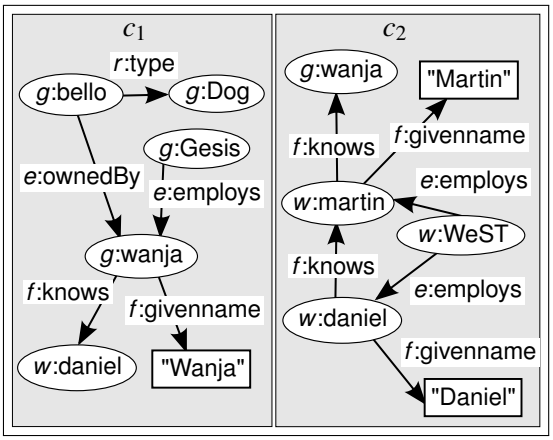
Summary-Graph-Based Index.

In some distributed RDF stores summary graphs are created. These summary graphs can be queried to identify compute nodes that store triples required for the processing of a query. The definitions of summary graphs differ between the different RDF stores.

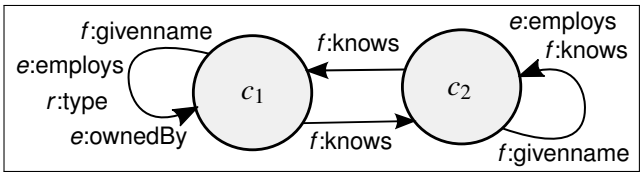
In TriAD [55] each compute node becomes a vertex in the summary graph. For each triple (s, p, o), an edge with label p is created in the summary graph that connects the vertices representing the compute nodes to which all triples with subject s and o were assigned to. To reduce the size of the summary graph, all edges connecting the same vertices and having the same label are represented by only a single edge.

Example 9. Figure 2.8b shows the summary graph created from the data distribution in Figure 2.8a. The vertices represent the compute nodes. For instance, c_1 represents all subject and objects occurring on compute node c_1. The self-loop at the vertices represent the properties occurring within these compute nodes. To simplify the graphic, all labels were attached to a single edge instead of creating an own edge for every label. The two edges connecting both vertices represent both triples whose subject and object were assigned to different compute nodes.

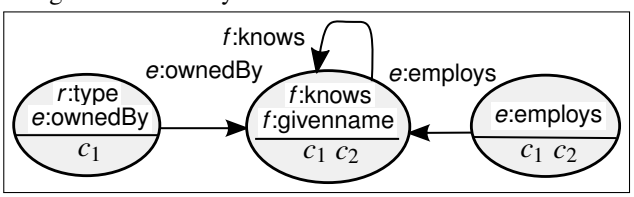
Another type of summary graph is used by EAGRE [163]. EAGRE splits the RDF graph into sets of triples with the same subject called molecules. The set of predicates in these molecules are called molecule type. These molecule types are the vertices in the summery graph. To reduce the number of molecule types, molecule types with similar properties are merged. Each triple t contained in a molecule of type T_1 whose object is the unique subject of all triples in another molecule of type T_2 will result in an edge with the label of the property that connects the vertices T_1 and T_2 of the summary graph.



(a) An example distribution of the example graph from Figure 2.1.



(b) The summary graph of the data distribution shown in Figure 2.8a used by TriAD.



(c) The summary graph of the data distribution shown in Figure 2.8a used by EAGRE.

Figure 2.8.: An example data distribution and the two summary graphs created by TriAD and EAGRE.

Example 10. Figure 2.8c shows the summary graph created by EAGRE for the data distribution shown in Figure 2.8a. The three vertices represent the three different molecule types. The properties occurring in molecules of that type are written in the upper part of the vertex. The compute nodes on which molecules of that type can be found are listed in the bottom part of each vertex. The leftmost vertex represents the dog molecule, the middle vertex represents the employee molecules and the right vertex represents the institute molecules. The e:employs edge connecting the institute molecule type with the employee molecule type represents the edges that connect the two institutes with their employees.

2.3.2. Distributed Indices

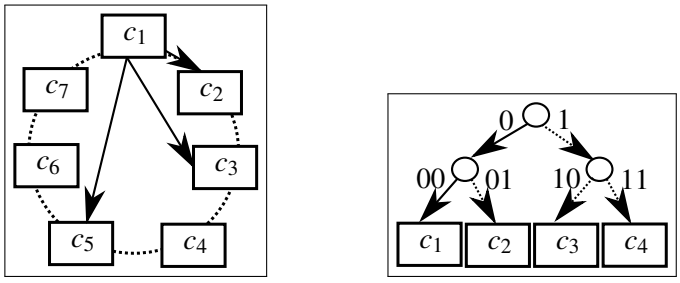
In distributed indices every compute node knows only a part of the complete index. In order to find every entry of the complete index, the compute nodes have to forward the index lookup requests to other compute nodes, until the compute node knowing the requested information is found. In order to route these index lookups, overlay networks are created that define to which compute node an index lookup should be forwarded.

Hash-Based Index

If a distributed RDF store distributes the triples based on their hash values, each compute node can determine on which compute node a triple can be found, by knowing the set of all available compute nodes. This approach is done, for instance, by HDRS [22] and Virtuoso Clustered Edition [42].

In peer-to-peer distributed RDF stores, the set of all compute nodes might be large and change over time. To prevent the replication of this set and keeping it up-to-date on all compute nodes, each compute node only stores a set of neighboured compute nodes, to which index lookups may be forwarded. The definition of the neighbourhood creates an overlay structure. In peer-to-peer distributed RDF stores the following overlay structures are used frequently:

Ring structure. In RDFPeers [25], PAGE [36], Atlas [78] and [95] the compute nodes are ordered, e.g., by their IP addresses. This order defines the direct neighbours of each compute node. To ensure that every compute node has exactly two neighbours, the first and last compute node are defined as neighbours. An example of the resulting ring is shown in Figure 2.9a. If only the ring structure would be given, finding a compute node that stores the requested data would take linear time. To reduce the lookup time, each compute nodes stores shortcuts to compute nodes at later positions in the ring. For instance, compute node c_1 knows compute nodes c_2, c_3 and c_5. If c_1 is asked whether it knows some information about resource r, it can compute with the hash function that triples with this resource would have been



(a) Ring overlay. (b) Tree overlay.

Figure 2.9.: The different types of overlay networks used in distributed hash indices.

assigned to, e.g., c_6. Since c_1 does not know c_6, it sends the request to c_5 since it is the closest compute node to c_6 that is known by c_1.

Tree structure. In GridVine [6, 31], UniStore [79], 3RDF [12], [14], [13] the overlay network is based on a prefix tree as shown in Figure 2.9b. Each vertex in this tree represents a prefix. The root has an empty prefix, the left child of the root node has the prefix 0 and the leaf c_1 stores all triples with resources that have a hash value starting with 00. Each compute node knows the path from the root to itself. For each node n in the path, the compute node knows one compute node contained in the subtree of the siblings of n. For instance c_1, would forward every hash with prefix 01 to compute node c_2 and every hash with prefix 1 to compute node c_3.

Combinations of both overlay structures can be found in [19] and [114]. The basic idea is that they initially use the ring overlay structure. If one compute nodes has to store too many entries, it redistributes it triples based on a tree structure.

Schema-Based Index

Instead of using hash-based indices, the RDFS types can be used to distribute data. The RDFS types contained in a data set usually build a type hierarchy. Similar to the tree overlay structure presented above, each compute node is responsible for the instances of the RDFS types assigned to him. If a compute node should retrieve instances of a given type T that is not assigned to him, it searches for a superclass of T for which he knows a responsible compute node and forwards the request to it. This so called semantic overlay network [30] is used, for instance, by SQPeer [85].

Data-Set-Integrated Index

The idea of data-set-integrated indices is that instead of having a dedicated index the index is stored within the RDF graph itself. In case of TriAD [55] the integer identifiers of the encoded vertex labels are prefixed by the identifier of the compute node that stores all triples with this vertex label as subject.

A completely different type of distributed index is presented in [121, 122]. It adapts the idea of the summary graph from TriAD as described in Section 2.3.1. Instead of realizing a separate index structure, it integrates the summary graph into its local RDF storage. With the help of these additional information a compute node can decide, whether another compute node has data that might lead to further query results.

Example 11. Figure 2.10 shows the data distribution from Figure 2.8a extended by the triples from the summary graph. On c_1 the graph chunk from c_2 is represented by a super vertex named c_2. All triples whose subject or object is stored on c_1 but the counterpart not, are represented by an edge connecting the vertex located on c_1 with the super vertex representing the other compute node that stores the other vertex. For instance, the subject of the triple (`w:daniel f:knows w:martin`) is stored on c_1 whereas its object is only stored on c_2. Therefore, the triple (`w:daniel f:knows` c_2) is added to compute node c_1. Furthermore, for each property p label occurring on c_2, a triple c_2 p c_2 is added to compute node c_1.

2.4. Distributed Query Processing Strategies

RDF stores in the cloud distribute RDF graphs over several compute nodes. One challenge which arises from this distribution is how to query the distributed graph. In general RDF stores in the cloud try to compute as much locally on a single compute node as possible without the need to exchange data. Therefore, the received

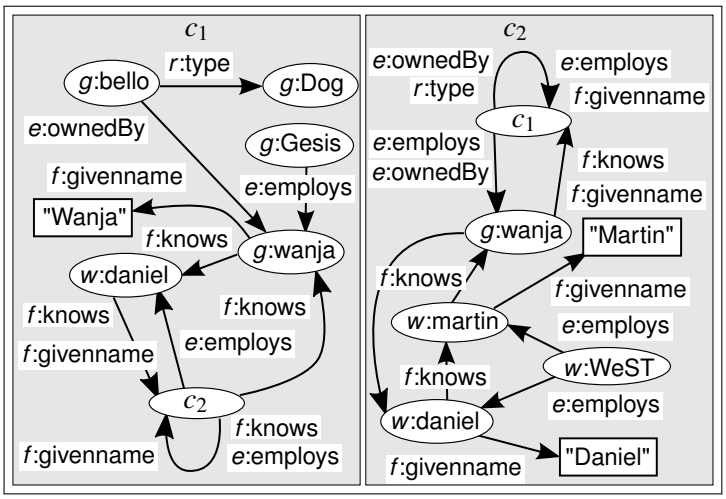


Figure 2.10.: The summary graph integrated into the graph cover shown in Figure 2.8a.

query is decomposed into subqueries that can be executed locally on a single compute node. In the simplest case these subqueries consist of a single triple pattern. Other systems can make use of some properties of the underlying data distribution. For instance, SHARD [116] uses a data distribution that assigns all triples with the same subject to one compute node. As a result, all star-shaped subqueries can be executed locally. Another example are RDF stores that make use of the n-hop replication. This replication ensures that all paths of length n can be queried locally. If the local RDF storage is able to return all partial matches of the query, the complete query can be executed by the local RDF storage. With the help of the indices, the number of compute nodes that can contribute to a subquery can be restricted.

The intermediate results of the subqueries needs to be joined in order to produce the overall query result. Several RDF stores transfer all intermediate results to a single compute node that is then responsible for joining it (see Section 2.4.1). If a huge number of intermediate results are produced, the joining compute node might be overloaded. Therefore, several RDF stores distribute the join computation over several compute nodes (see Section 2.4.2). An example how queries are executed in distributed graph processing frameworks is given in Section 2.4.3.

2.4.1. Centralized Join

Especially in federated RDF stores, the intermediate results of subqueries that were executed on remote RDF stores have to be joined on the query federator. Thereby, the join strategies of relational databases are applied. The following join strategies are used:

Nested loop join [103]: For each element in the intermediate result list of the first subquery, the intermediate result list of the second result list needs to be iterated completely to find all join candidates.

Merge join [103]: For this join the intermediate result lists must be ordered. One list is iterated and for each element the join candidates in the other list can be retrieved. Due to the ordering the other list does not need to be traversed from the beginning. Instead only the elements with the same value of the join variable needs to be reiterated. This join is performed by the distributed RDF store Partout [45].

Hash join [37]: The intermediate results of the subqueries are stored in separate hash tables. Each hash table distributes the results into several buckets. If both hash tables use the same number of buckets, only the intermediate results of two buckets need to be joined at once. When all buckets are joined, the join is finished.

Symmetric join [153]: This type of join is a non-blocking hash join. For every subquery a hash table is created that stores the already received intermediate results. When a new intermediate result is received, it is joined with all join candidates in the other hash table. The results are emitted and the intermediate result is inserted in the hash table of the subquery that produced it. This join strategy is performed by, e.g., ANAPSID [7] and LHD [151].

Bind join [57]: In order to perform a bind join, the first subquery is executed. For each returned distinct intermediate result μ, the second subquery is executed. Thereby, all variables already bound by μ are substituted in the second subquery by their bound values. This type of join is performed by, e.g., FedX [141], Avalanche [18] and SemaGrow [26].

Beside performing a single type of join operation, RDF stores like DARQ [128] and SPLENDID [50] choose between a bind join and a nested loop join or between a bind join and a hash join, respectively. The choice depends on the expected number of returned results.

Another join strategy is performed by [118]. In this distributed RDF store partial evaluation [75] is performed. This means that the complete query is executed on the local RDF stores of each compute node. These local RDF stores return the overall results and all intermediate results. For each intermediate result the subquery that created the result is returned. The intermediate results from all compute nodes are sent to a single compute node who finally joins the intermediate results and returns the overall results.

2.4.2. Decentralized Join

The subqueries into which a query is decomposed can easily produce a large number of intermediate results. Joining all intermediate results on a single compute node can overload the capacity of this compute node. To overcome this limitation several RDF stores in the cloud apply distributed joins.

Replication-Based Distributed Join

In order to distribute the number of join computations over the individual compute nodes, in SemStore [156] all intermediate results of a subquery are sent to all compute nodes on which the succeeding subquery is executed. This strategy increase the amount of transferred intermediate results but each compute node only joins its local results with the intermediate results produced by the other compute nodes.

Distributed Hash Join

In DiploCoud [157] the intermediate results of the subqueries are joined by a distributed hash join. Basically, the distributed hash join is similar to a centralized hash join. The only difference is that each compute node is a bucket of the used hash table.

The hypercube hash join was initially presented in [20] and was used in the distributed RDF store presented in [28]. The basic idea is that for each join variable one dimension is created. For instance, if a query has three join variables, three dimensions are created. Therefore, we need to arrange the compute nodes as a three-dimensional cube. Each compute node is responsible for one cubic region within this cube.

Thereafter, every triple pattern is executed in parallel producing intermediate results. If the intermediate result $\{(?v1, w:martin)\}$ is produced, it is forwarded to all compute nodes that are responsible for the value hash$(w:martin)$ in the $?v1$-dimension. Each compute node performs a local join of the intermediate results it has received from the different triple patterns. Depending on the selection of the regions each compute node is responsible for, the workload can be equally distributed among all compute nodes.

Distributed Merge Join

In a distributed merge join the intermediate result lists of the subqueries are sorted by the values of the join variables. Thereafter, each compute node receives all elements within a specific value range and joins them. This type of join is primarily performed in Hadoop-based RDF stores like H2RDF+ [116], SHARD [116], [66] and Spark-based RDF stores like SparkRDF [158] and SPARQLGX [52].

Distributed Bind Join

One way to realize a distributed bind join is implemented in AdHash [9]. The first triple pattern is executed on each compute node. Thereafter, each compute node performs a centralized bind join based on the intermediate results the first triple pattern has produced on this compute node.

In RDFPeers [25], GridVine [6, 31], Atlas [78] and 3RDF [12] each triple is stored on at most three compute nodes based on the hashes of its subject, property and object. As a consequence all triples in which one resource occurs are located on the same compute node. Since usually each triple pattern of a query contains at least one constant, all matches for this triple pattern can be found on a single compute node. In order to process a query, the query coordinator determines a sequence in which the compute nodes should be traversed to process all triple patterns. When moving from one compute node to another, all intermediate results are transferred. In order to join the intermediate results, bind joins are performed.

In order to generalize and parallelize the previously described strategy, [95] introduced the so called spread by value querying strategy[20]. The basic idea is that the first triple pattern is processed on the computed nodes on which matches occur. These compute nodes start a bind join processing with the second triple pattern. When a compute node identifies with the help of the global index that for one triple pattern there exist matches on a different compute node, it will fork the query processing on the other compute node that will continue with this branch of the query execution. The final query results are sent back to the query coordinator. This strategy was also used by, for instance, [14], [13], [121, 122] and TripleRush [146]. In order to speed up the query execution, Trinity.RDF [162] performs the spread by value strategy from the first and the last triple patterns in parallel.

Example 12. In this example the graph pattern `<w:WeST> <e:employs> ?v1. ?v1 <f:knows> ?v2. ?v2 <f:givenname> ?v3` should be executed on the data distribution with integrated summary graph index in figure 2.10. The first triple pattern creates the intermediate result $\{(?v1, w:martin)\}$ on compute node c_2. Based on this intermediate result the variable $?v1$ of the second triple pattern will be substituted by $w:martin$. When processing the triple pattern `<w:martin> <f:knows> ?v2`, the intermediate result $\{(?v1, w:martin), (?v2, g:wanja)\}$ is produced. Before processing the third triple pattern, $?v3$ will be substituted by $g:wanja$. As a result `<g:wanja> <f:givenname> ?v3` is executed. This time the only possible substitution for $?v3$ is the super vertex c_1. This match means that there exists a triple on compute node c_1 that would match with the triple pattern. Therefore, the query, the created variable binding,

[20] This idea is named differently in the literature. For instance, in Trinity.RDF [162] it is called graph exploration.

and the metadata that this intermediate result was created by the first two triple patterns is sent to compute node c_1. Now, c_1 executes `<g:wanja> <f:givenname> ?v3` and produces the intermediate result $\{(\texttt{?v1},\texttt{w:martin}),(\texttt{?v2},\texttt{g:wanja}),(\texttt{?v3},\texttt{"Wanja"})\}$. For the sake of simplicity, the other intermediate results produced by the triple patterns that were executed in parallel were skipped during the explanation of this example.

2.4.3. Distributed Query Processing in Graph Processing Frameworks

In S2X [158] a different type of distributed query processing is presented. First, each vertex checks whether it can be a substitution for some variable in the query by checking its incident edges, their labels and the adjacent vertices. Thereafter, it notifies the neighboured vertices by its intermediate results. If for one intermediate result no compatible join candidate can be found on the neighboured vertices, it is discarded. The notification of the neighboured vertices and the discard of local intermediate results is repeated until the intermediate result sets of each vertex in the graph do not change any more. The remaining intermediate results can be retrieved and joined as the final results.

2.5. Fault Tolerance

One problem of RDF stores in the cloud is that a single compute node may fail or become disconnected from the network. RDF stores that bases on cloud computing frameworks mainly rely on the fault tolerance of the used framework. In federated RDF stores the actual data is stored on remote RDF stores that are not under control of the system administrator. As a result the fault tolerance is not an urgent problem for both types of RDF stores.

In contrast to these RDF stores, the failure of compute nodes is an issue for distributed RDF stores. Since most of these RDF stores that can be found in the literature are proof-of-concept implementations, they do not address the problem of fault tolerance. The few systems that deal with fault tolerance, address this problem by replication. Systems like Virtuoso Clustered Edition [42] suggest to create identical copies of all compute nodes. If one compute node fails, it is replaced by one of its copies.

Another strategy to become fault tolerant is used by, e.g., 4store [61] and RDFPeers [25]. In these distributed RDF stores there exists a partial order of all compute nodes, for instance, created by the comparison of their IP addresses. To ensures that every compute node has a successor and predecessor, the successor of the last compute node is the first compute node. Based on this ordering, the triples assigned to one compute node are also assigned to the neighboured compute nodes. If one compute node fails, the index forwards the queries to one of the neighbours that store replicas of the data originally assigned to the failed compute node.

2.6. Further Challenges

To cope with the growing size of huge graphs, scalable RDF stores in the cloud are used, where the graph data is distributed among several compute nodes. From this distributed setting several challenges like (i) the data placement strategy, (ii) the distributed query processing, and (iii) the handling of failed compute nodes. In this chapter an overview of how most of these challenges are addressed by RDF stores in the cloud is given.

Due to the high number of RDF stores in the cloud, only an overview of the core challenges in distributed RDF stores have been given. Beside these core challenges there exist further features that are required during the practical usage of relational databases in the industry today. Realizing them in RDF stores is a challenging task so that they are only partly realized in RDF stores. In order to achieve a broader usage of RDF stores in

industry, further research is required to implement these features in RDF stores in the cloud. In the following we describe some of these features. As an example use case we assume an online retailer.

When two customers try to order a unique product at the same time, only one of the orders must be successful and the other must fail. To prevent the situation that both customers could successfully order the unique product, *transactional security* (i.e., atomicity, consistency, isolation and durability) is required. Realizing transactional security in a distributed setting where the data is separated among several compute nodes might require a lot of coordination between the compute nodes. This additional coordination increase the query execution time. To avoid this overhead while providing transactional security, most RDF stores in the cloud assume that the RDF graph is immutable after loading it. Only few RDF stores like Virtuoso Clustered Edition [41] allow for inserting or deleting triples after loading.

In order to identify which products were sold the most frequently in the last three month, the database is required to perform *online analytical processing (OLAP) queries*. This type of queries require a huge amount of data to be processed. In context of RDF stores in the cloud, processing OLAP queries cannot be done by sending all required data to a single compute node since a single compute node may be overloaded by the huge amount of data. Therefore, data placement strategies and distributed query execution strategies need to be developed supporting the parallel processing of OLAP queries with a few exchanged of network packets.

To simplify the search for a product, the online retailer has categorized its products in a category hierarchy. For instance, an orange lemonade can be categorized as lemonade, soft drink and drink. With the introduction of SPARQL 1.1 [123] *property paths* were introduced that allow for requesting all offered drinks independently of the subcategory they belong to. This type of query differs from the pure graph pattern matching done in SPARQL 1.0 since it can easily require the traversal of long paths. In a distributed setting the traversal of long paths may lead to a high network traffic reducing the query execution time. One challenge arising from these queries is, how to optimize the data placement for these queries. Alternatively to SPARQL, the retailer might want to use other query languages like GraphQL[21].

The retailer wants to prevent teenagers from buying alcohol. Therefore, he stores in the database the rules that customers younger than 20 are teenagers and teenagers should not be allowed to buy alcohol. These rules should be automatically applied to all customers. To realize this, RDF stores have to reason about RDF graphs. In this context *reasoning* means inferring logical consequences and checking the RDF graph consistency. Usually, reasoning is done during the graph loading and all logical consequences are stored as explicit triples in the graph. The challenge of distributed reasoning is that the reasoning of the complete graph may overload a single compute node. Therefore, distributed reasoning algorithms are required. A few RDF stores in the cloud like MaRVIN [113] and Rya [127] have addressed this challenge. Another problem is, if the RDF graph is mutable after loading. In this case the deletion or insertion of triples may produce inconsistencies, a lot of newly inferred triples need to be inserted, or formerly inferred triples need to be removed.

Finally, the retailer wants to advertise summer products like swimwear, portable fans, etc. more prominently if the temperature in the town where the customer lives is high. Therefore, the constantly streamed data from temperature sensors needs to be processed. This quickly arriving streamed data cause further challenges for RDF stores, like quickly combining the received data with static data, updating the database frequently or balancing the workload among all compute nodes. CQELS Cloud [90] is one example of a system that processes RDF streams in a distributed fashion.

[21] `https://graphql.org/`

CHAPTER 3

Related Work

RDF stores in the cloud aim to combine the storage and processing capabilities of several compute nodes. Especially for large RDF graphs, splitting the graph into smaller not-necessarily disjoint parts that are stored and processed by different compute nodes may have advantages. First of all, the size of the graph that can be managed by a distributed RDF store may be larger than the size of the graphs each of the compute nodes can manage on its own. Due to the smaller portions of the graph for which the individual compute nodes are responsible for, it needs to process only a small portion of the graph leading to potentially faster query execution times.

In the last 15 years, several different data distribution strategies were developed. Some of these strategies can efficiently create a data distribution or can simplify the indexing of the distributed RDF graph. Other strategies aim to improve the query performance by collocating closely connected data on the same compute node to reduce the amount of exchanged intermediate results during query processing. An overview of the most common data distribution strategies is given in Section 3.1.

The topic of this thesis is, how the different data distribution strategies influence the query performance. Therefore, their influences need to be evaluated. Section 3.2 gives an overview of the different evaluation methodologies that have been used in the literature.

Section 3.1 and Section 3.2 were mainly taken from [68].

3.1. Graph Cover Strategies

One core aspect of RDF stores in the cloud is, how the RDF graph is distributed among the compute nodes. Therefore, RDF stores in the cloud use different graph cover strategies. A common procedure to create a graph cover is to first split the RDF graph into small possibly overlapping subsets. Thereafter, these graph subsets are assigned to compute nodes. In RDF stores that base on cloud computing frameworks, the influence on how this assignment to compute nodes is done is usually limited. Therefore, in Section 3.1.1 is described how a graph is split into subsets that are then stored in the cloud computing framework. The graph cover strategies of distributed RDF stores can in general be separated into three categories: (i) hash-based graph cover strategies (see Section 3.1.2), (ii) graph-clustering-based graph cover strategies (see Section 3.1.3) and (iii) workload-aware graph cover strategies that distribute the graph based on a historic query workload (see Section 3.1.4). In order to reduce the amount of queries that need to combine data from different compute nodes, the n-hop replication was proposed that replicates triples at the border of the chunks of arbitrary graph covers (see Section 3.1.5).

When RDF stores are queried frequently, the initial distribution of the graph on the compute nodes may be suboptimal for the current query workload. To improve the query performance, some RDF stores have implemented a dynamic graph cover strategy (see Section 3.1.6). This strategy observes the current query workload and tries to optimize the data placement by moving or copying small triple sets from one compute node to another.

Since there is a huge number of graph cover strategies, this section focusses only on the most frequently graph cover strategies and gives only hints to a few variations that can be found. Graph cover strategies that were only used in a single RDF store are not presented.

To illustrate the different graph cover strategies, the graph from the running example introduced in the previous chapter will be used. It is shown in Figure 3.1.

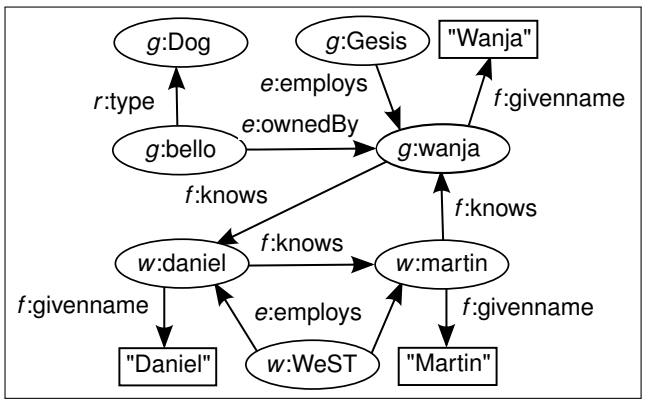


Figure 3.1.: Example graph describing the knows relationships between some employees of WeST and Gesis.

3.1.1. Graph Cover Strategies in Cloud-Computing-Framework-Based RDF Stores

RDF stores that build on cloud computing frameworks have usually limited influence how the data is placed on the individual RDF stores. Their influence is limited to the way how the RDF graph is split into subsets that are stored in files or tables. The goal of splitting the RDF graph into subsets is to reduce the amount of triples that need to be processed during the query execution. This is achieved by storing all triples with the same subject, object, property or combinations of them in the same file or table.

Molecule Graph Splits

In order to process star-shaped queries efficiently, the RDF graph is split into molecules of diameter 1. This means that all triples with the same subject are stored in one file as shown in Figure 3.2. D-SPARQ [109] follows this approach. The advantage of the molecule graph split is that for star shaped queries whose triple patterns are joined on a subject no distributed join needs to be processed. In case of a constant subject only a single file needs to be processed.

In order to reduce the number of required distributed joins, RAPID+ [83] proposes to store all triples that have the same resource identifier at a subject or object position in a single file. Additionally, RAPID+ reduces the number of distributed joins by increasing the molecule diameter in order to process path-shaped queries more efficiently.

In some RDF stores like Stratustore [144], Sempala [137] and SHARD [129] all molecules are stored in a single table. Each molecule is basically represented by a single row in this table with the subject as unique identifier. The properties are the column names and the object are the values stored in each cell. If the combination of subject and property occurs in several triples, these cells store a list of objects or several rows are created for this subject. This storage layout is called property table in the literature.

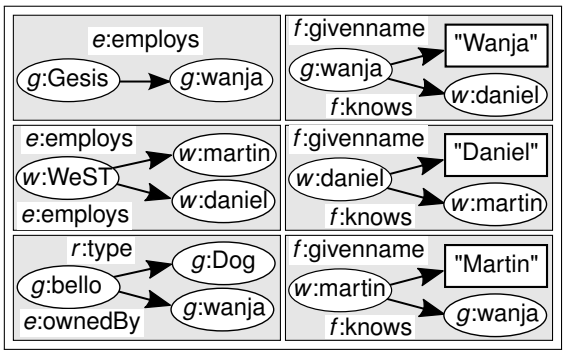


Figure 3.2.: The example graph split into molecules.

Vertical Graph Splits

The basic idea of the vertical cover originated in [3] is to store RDF data in a relational database such that for each property a table is created in which all triples with this property are stored. In the context of distributed RDF stores, approaches like Jena-HBase [82], PigSPARQL [136], [164] and SPARQLGX [138] store all triples with the same property in one file or table as shown in Figure 3.3. The advantage is that it is easy to compute but a query that matches with paths of length l will only match with triples on at most l compute nodes. Thus, this graph cover strategy is likely to result in an imbalanced workload and a high number of exchanged intermediate results.

One disadvantage of the vertical graph split as presented above is that frequently occurring properties like rdf:type lead to very large files. Therefore, the RDF store HadoopRDF [44] splits these tables based on combinations of properties and the RDFS types of the objects.

Another variant of the vertical graph split is realized in S2RDF [138]. In order to reduce the number of joins that need to be processed, they additionally create tables for all possible subject-subject and subject-object joins of triples.

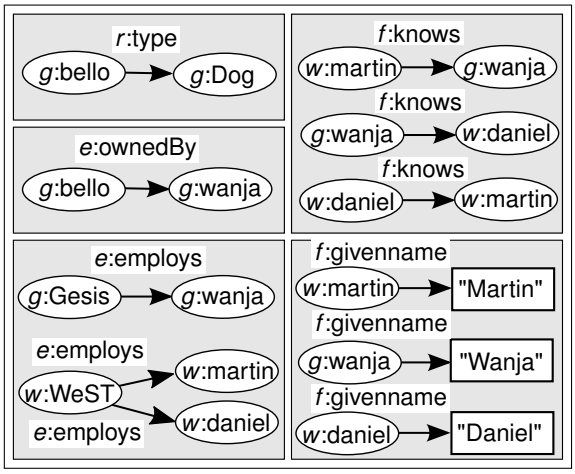


Figure 3.3.: An example vertical graph split of the example graph.

3.1.2. Hash-Based Graph Cover Strategies

Hash Cover

A hash cover assigns triples to chunks according to the hash values computed on their subjects modulo the number of compute nodes. Thus, all triples with the same subject – i.e., a molecule – are located in the same graph chunk. This graph cover strategy is used, for instance, by Virtuoso Clustered Edition [41], VB-Partitioner [91], SPA [93], 4store [61] and AdHash [9].

Example 13. The following hash function produces the graph cover shown in Figure 3.4.

$$\forall r \in \{\text{g:Gesis}, \text{g:wanja}, \text{w:martin}\}\text{: hash}(r) := 1$$
$$\forall r \in \{\text{g:bello}, \text{w:WeST}, \text{w:daniel}\}\text{: hash}(r) := 2 \ .$$

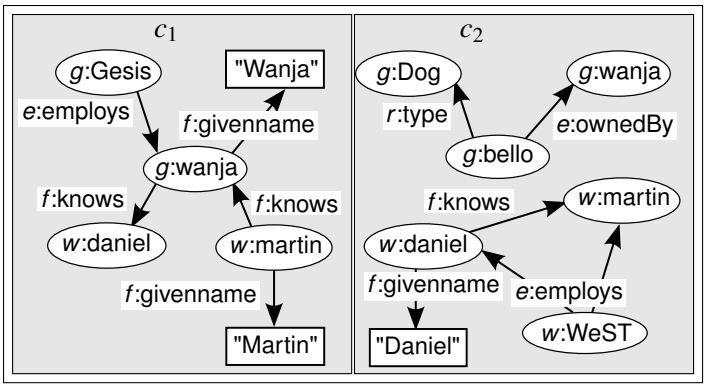


Figure 3.4.: An example hash cover of the example graph.

The advantages of the hash cover are that it is easy to compute and due to a relatively random assignment of triples to compute nodes the resulting graph chunks will have similar sizes. The disadvantages are that it may lead to a high number of exchanged intermediate results if a query matches with long paths. Since all hash covers are subject-contained, this graph cover strategy might be a good choice if the expected queries will only match with paths of a short length (ideally 1).

Beside the subject, distributed RDF stores like Trinity.RDF [162], Clustered TDB [115], YARS2 [62, 63] and RDFPeers [25] also use property and/or the object to assign each triple three times to the compute nodes.[1] Additionally, RDF stores like PAGE [36] and [13] append at least two elements of each triple and use the hash of the result to assign triples to compute nodes.

Hierarchical Hash Cover

Inspired by the observations that IRIs have a path hierarchy and IRIs with a common hierarchy prefix are often queried together, SHAPE [92] uses an improved hashing strategy to reduce the inter-chunk queries. First, it extracts the path hierarchies of all IRIs. For instance, the extracted path hierarchy of "`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`" is "`org/w3/www/1999/02/22-rdf-syntax-ns/type`". Then, for each level in the path hierarchy (e. g., "`org`", "`org/w3`", "`org/w3/www`", ...) it computes the percentage of triples sharing a hierarchy prefix. If the percentage exceeds an empirically defined threshold

[1] If the hash cover is only computed on the properties, the resulting graph cover would be similar to the vertical graph split.

and the number of prefixes is equal or greater to the number of compute nodes at any hierarchy level, then these prefixes are used for the hash cover.

Example 14. Assume the hash is computed on the prefixes g and w of the subject IRIs in the example graph. If the hash function returns 1 for g and 2 for w the resulting hierarchical hash cover is shown in Figure 3.5.

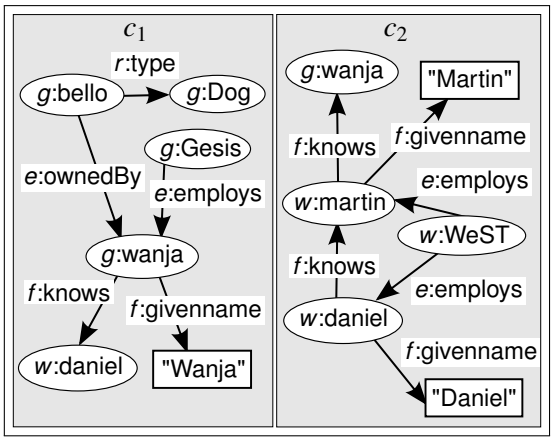


Figure 3.5.: An example hierarchical hash cover which is also a minimal edge-cut cover of the example graph.

In comparison to the hash cover the creation of a hierarchical hash cover requires a higher computational effort to determine the IRI prefixes on which the hash is computed. For queries that match with paths in which the subjects and objects have the same IRI prefix the number of exchanged intermediate results may be reduced. This reduction might come at the cost of a more imbalanced query workload since only a few chunks will contain these paths. Thus, the use of the hierarchical hash cover might be beneficial (i) if the network connecting the compute nodes is slow or (ii) if other functionality such as prefix matching benefits from the hierarchical hash cover.

3.1.3. Graph-Clustering-Based Graph Cover Strategies

Graph clustering considers splitting a graph into partitions, i.e., a graph cover with pairwise disjoint graph chunks. In this area a wide variety of algorithms was developed (for instance, see the survey [98]). The basic idea is that an RDF graph is partitioned by one of these algorithms. Since most of the graph clustering algorithms create an assignment from vertices to compute nodes, the triples are usually assigned to the compute nodes to which their subjects were assigned to. The most frequently applied graph clustering approach is the minimal edge-cut partitioning which is described below.

Another rarely used graph clustering algorithm tries to optimize the modularity. The modularity measures the difference between the actual number of edges within the partitions and the expected number of such edges. This strategy was applied for instance by MO+ [126].

Minimal Edge-Cut Cover

The minimal edge-cut cover is a vertex-centred partitioning which tries to solve the k-way graph partitioning problem as described in [80]. It aims at minimizing the number of edges between vertices of different partitions under the condition that each partition contains approximately $\frac{|V_G|}{k}$ many vertices. Details about the computation of k-way graph partitioning and the targeted approximation can, e.g., be found in [80]. RDF

stores like [66] and [118] convert the outcome of the minimal edge-cut algorithm, i.e., a split of the complete V_G into disjoint subsets, into a graph cover of G by assigning each triple to the compute node to which its subject has been assigned.

Example 15. A minimal edge-cut algorithm may assign the resources g:Dog, g:Gesis, g:bello, g:wanja and "Wanja" to compute node c_1 and all other resources to compute node c_2. For our specific running example the result of the minimal edge-cut cover strategy is identical to the results of the hierarchical hash cover strategy depicted in Figure 3.5.

In this example there exist two edges connecting vertices assigned to different chunks. One is the f:knows edge starting at g:wanja and ending at w:daniel. The other is the f:knows edge starting at w:martin and ending at g:wanja. Since the subject g:wanja of the first triple has been assigned to c_1, this triple has been assigned to c_1. The subject of the second triple w:martin has been assigned to c_2. Therefore, this triple has been assigned to c_2.

Since the minimal edge-cut cover considers the graph structure, the creation of the graph cover requires a high computational effort. The advantage of considering the graph structure may be a reduced number of exchanged intermediate results. This would make the minimal edge-cut cover a good choice if the network connection between compute nodes is slow.

In order to optimize the query performance, TriAD [55] creates an over-partitioning. For instance, to create a graph cover that assigns triples to 5 compute nodes, 100k-200k partitions are created. These partitions are then assigned to compute nodes. To improve the performance of queries that use RDFS schema information, in [121] the RDFS schema is replicated to all graph chunks. Since the minimal edge-cut cover strategy can lead to graph chunks whose cardinality varies a lot, [150] proposes to weight vertices by the number of triples in which they occur.

An alternative optimization is performed by EAGRE [163]. Instead of partitioning the original graph, a minimal edge-cut cover of the summary graph is created (see Section 2.3). Each vertex of the summary graph represents a set of molecules that have similar properties. They are weighted by the number of molecules they contain. This graph cover strategy ensures that molecules with similar properties are stored on the same compute node.

3.1.4. Workload-Aware Graph Cover Strategies

Another type of graph cover strategies assumes that the query workload does not change much over time. Therefore, they learn from a historic query workload which triples have been frequently queried together first. Based on this knowledge they try to find an optimal graph cover for future queries. These approaches are, for instance:

- The novel idea applied in WARP [65] is creating an initial minimal edge-cut cover and then replicate triples in a way such that all historic queries can be answered locally.
- In COSI [23] edges are weighted based on the frequency they are requested by the historic query workload. Thereafter, a weighted minimal edge-cut partitioning is performed leading to a reduced number of transferred intermediate results.
- In [17] the resulting graph cover aims to balance the overall workload of all queries equally among all compute nodes. Thereby, each query is processed by a single compute node in an ideal case. To reach this goal, the proposed algorithm assigns the triples required by the queries to compute nodes in a way that the number of replicated triples is reduced.

- In Partout [45] the queries contained in the historic query workload are first generalized by replacing rarely queried subject or object constants by variables. Thereafter, the matches of this generalized triple patterns are assigned to compute nodes in a way that (i) ideally each query can be answered by a single compute node without replicating triples and (ii) the query workload of all queries is distributed equally among all compute nodes.

- DiploCloud [157] generalizes the queries in the historic query workload by using schema information. After generalizing the queries, triple sets are computed that can produce a single query result. Finally, these triple sets are distributed equally among all compute nodes.

3.1.5. *n*-Hop Replication

Whenever a query combines data from different graph chunks, intermediate results need to be exchanged between different compute nodes. To reduce the number of exchanged intermediate results for a subject-complete graph cover of graph G, the n-hop replication strategy extends each of its chunks ch_i by replicating all triples contained in some path of length $\leq n$ in G starting at some subject or object occurring in ch_i. This way all queries that match with paths of length $\leq n$ could be processed without exchanging intermediate results. The n-hop replication is used by systems like [66] and VB-Partitioner [91].

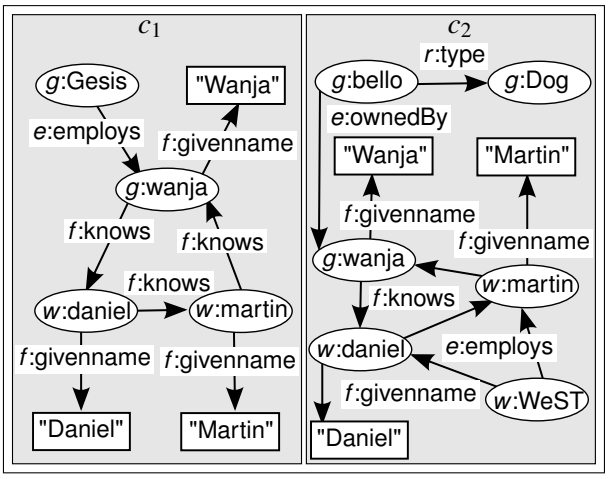


Figure 3.6.: The 2-hop extension of the hash cover in Figure 3.4.

Example 16. Applying the 2-hop replication extension on the hash cover in Figure 3.4 results in the 2-hop hash cover shown in Figure 3.6. In this cover a query could match with the path $\langle(\text{g:bello}, \text{e:ownedBy}, \text{g:wanja}), (\text{g:wanja}, \text{f:knows}, \text{w:daniel})\rangle$ on compute node c_2 without the need to exchange intermediate results.

The n-hop replication may reduce the number of transferred intermediate results at the cost of replicating triples. This replication will increase the effort to create the graph cover and increases the size of the graph chunks. Furthermore, the replication might cause a higher computational effort during the query processing since the replicated triples might lead to duplicate intermediate results. Thus, using the n-hop replication might be beneficial if the network connecting the compute nodes is slow and the number of replicated triples is low.

3.1.6. Dynamic Graph Cover Strategies

Graph covers created by one of the graph cover strategies above can lead to a high amount of data transfer between compute nodes, if the actual query workload needs to combine data stored on different compute nodes. In order to overcome this limitation, PHD-Store [9] and AdHash [9] keep track of basic graph patterns that are queried frequently. When the frequency exceeds a threshold, triples that match with these frequent triple patterns are replicated in a way that these basic graph patterns can be executed locally.

Instead of only trying to reduce the network communication, Sedge [9] tries to primarily distribute the query workload equally among the compute nodes. Therefore, Sedge keeps track how frequently the molecules are queried together. If a set of molecules is queried together with a high frequency, this set of molecules is replicated to a compute node with a low workload.

Another type of graph cover strategies assumes that during runtime new triples can be added to the RDF store. In this setting it may happen that a single compute node stores many more triples than other compute nodes. To prevent the compute node from being overloaded, the triples of that compute node can be redistributed based on the prefix of some hash values (as done in [114]). Another strategy is performed by [19]. The triples are sorted lexicographically and one half of them is sent to another compute node. In both cases the systems keep track to which compute node they have moved the triples.

3.2. Evaluation Methodologies

In order to evaluate the performance of RDF stores several benchmarks were proposed. In general, benchmarks consist of a data set, a set of queries and several performance metrics. In order to test the RDF stores with differently-sized RDF graphs, benchmarks usually use a data set generator that generated RDF stores based on a schema and/or specific characteristics. Some benchmarks provide fixed queries or query patterns that contain special variables that are substituted by constants after data set generation (see Section 3.2.1). Instead of providing query patterns, benchmark generators generate queries based on query characteristics (see Section 3.2.2). In Section 3.2.3, we elaborate how benchmarks are used to evaluate distributed RDF stores.

3.2.1. Benchmarks

Lehigh University Benchmark

LUBM [54] was developed to test the query optimizer performance. It generates an RDF graph based on its Univ-Bench ontology. This ontology describes universities, their departments, employees, courses, students and related activities. In order to provide more realistic data sets, several constraints are applied during the data generation. For instance, a university can have between 15 and 25 departments and the ratio of undergraduate students to faculties is between 8 and 14. The 14 provided SPARQL queries are designed to test how well the query optimizer can improve the join ordering. The characteristics of the queries are shown in table 3.1a. LUBM proposes the following performance metrics:

Load time is the time that the RDF store needs to parse and load the RDF graph.

Repository size is the size of all files that are required by the RDF store to store the data set including dictionary and indices.

Query execution time is the average time to execute a query ten times.

Query completeness and soundness measures which percentage of all results were retrieved and the percentage of correct results.

Query	# Triple Patterns	Query Diameter
Q1	2	1
Q2	6	2
Q3	2	1
Q4	5	1
Q5	2	1
Q6	1	1
Q7	4	2
Q8	5	2
Q9	6	2
Q10	2	1
Q11	2	1
Q12	4	2
Q13	2	2
Q14	1	1

(a) LUBM query characteristics.

Query	# Triple Patterns	Query Diameter
Q1	5	1
Q2	10	1
Q3a	2	1
Q3b	2	1
Q3c	2	1
Q4	8	2
Q5a	6	2
Q5b	6	2
Q6	9	2
Q7	13	5
Q8	8	2
Q9	4	2
Q10	1	1
Q11	1	1
Q12a	6	2
Q12b	8	2
Q12c	1	1

(b) SP^{2}Bench query characteristics.

Table 3.1.: Query characteristics of LUBM and SP^{2}Bench.

SP^{2}Bench

SP^{2}Bench [140] was designed to test the most common SPARQL constructs and how they are applied in realistic queries. It provides a data set generator that creates an RDF graph that follows the DBLP schema. This schema describes publications like articles and inproceedings with their bibliographic information. The generated graph mimics the characteristics of the real DBLP graph. SP^{2}Bench provides 17 queries. They mainly focus on testing the capabilities of the query optimizer to optimize the join order but also the optimization of complex filters and duplicate elimination. The characteristics of the queries are given in table 3.1b. The proposed performance metrics are:

Load time is the time that the RDF store needs to parse and load the RDF graph.

Query execution time is the the arithmetic mean of all execution times of a query.

Global query execution times are the arithmetic and geometric means of all 17 queries. The geometric mean is computed by multiplying the execution times of all 17 queries and then computing the 17th root of the result. If a query could not be processed, it is punished with 3600 seconds.

Memory consumption is measured by (i) the maximum amount of memory that was allocated during the processing of each individual query and (ii) the average memory consumption of all queries.

Berlin SPARQL Benchmark (BSBM)

The BSBM[2] [21] focuses on the use case of an e-commerce platform. It aims to simulate the search and navigation patterns of multiple concurrently acting customers. The data set is generated based on a relational schema. This schema represents products, their offers and the custom reviews of the products. BSBM provides 12 query patterns whose characteristics are given in table 3.2. These queries mainly test the ability to optimize the join ordering and the early application of filters. In BSBM a query pattern refers to a query in which some constants are replaced by a special type of variable. During the runtime of the benchmark these special variables are replaced with varying constants occurring in the data set. A set of queries in which each query pattern is instantiated is called a query mix. Several of these query mixes are executed in parallel in order to measure the performance of the RDF stores. The proposed performance metrics are:

Load time is the time that the RDF store needs to parse and load the RDF graph.

Query mixes per hour is the number of query mixes that can be completely processed within one hour.

Queries per second is the number of queries, which have been instantiated from a single query pattern, that can be answered within one second.

Query	# Triple Patterns	Query Diameter
Q1	5	1
Q2	15	2
Q3	7	1
Q4	10	1
Q5	7	1
Q6	2	1
Q7	14	2
Q8	10	2
Q9	1	1
Q10	7	2
Q11	2	2
Q12	9	2

Table 3.2.: BSBM query characteristics.

Semantic Publishing Benchmark (SPB)

The SPB[3] [86] is a benchmark motivated by the industry. The use case is a publisher organization that provides metadata about its published work. Many journalists search for data and perform insertions and deletions concurrently. SPB provides a data set generator that is designed to create data sets with several billions of triples that mimic the BBC data sets. Similar to BSBM it provides query templates that contain special variables that will be instantiated before query execution. SPB defines two set of query templates. The basic query set focuses on join ordering, duplicate elimination and filtering whereas the advanced query set additionally contains, e.g., analytical queries. The query characteristics are given in table 3.3a and in table 3.3b. The proposed performance metrics are:

[2] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
[3] http://ldbcouncil.org/developer/spb

Query	# Triple Patterns	Query Diameter
Q1	26	3
Q2	9	2
Q3	8	1
Q4	4	1
Q5	6	2
Q6	5	2
Q7	6	1
Q8	11	2
Q9	9	1
Q10	7	3
Q11	8	2

(a) Basic query set characteristics.

Query	# Triple Patterns	Query Diameter
Q1	26	3
Q2	9	2
Q3	1	1
Q4	3	1
Q5	3	2
Q6	4	2
Q7	2	1
Q8	4	1
Q9	1	1
Q10	5	1
Q11	11	1
Q12	9	1
Q13	5	1
Q14	12	2
Q15	10	2
Q16	9	1
Q17	5	2
Q18	6	1
Q19	4	1
Q20	11	2
Q21	9	1
Q22	9	1
Q23	9	1
Q24	4	1
Q25	4	1

(b) Advanced query set characteristics.

Table 3.3.: SPB query characteristics.

Minimum, maximum and average query execution times for each executed query.

Average execution rate per second measures how many queries could be finished per second on average.

FedBench

FedBench [139] is designed as a benchmark for federated RDF stores. It provides three different data set collections: (i) a general linked data collection containing DBPedia, GeoNames, Jamendo, Linked_MDB, New York Times and Semantic Web Dog Food, (ii) a life science data collection containing KEGG, ChEBI and DrugBank as well as (iii) a data set of 10M triples generated with the data set generator of SP^{2}Bench. FedBench provides two self-made query sets as well as the queries from SP^{2}Bench for the three data collections. The first two query sets focus on the number of data sources involved, the join ordering and the query results set sizes. Since the actual benchmark cannot be found online any more, the characteristics of the queries cannot be examined. The proposed performance metrics are:

Query execution time for each executed query.

Number of requests to remote RDF stores during the processing of each query.

3.2.2. Benchmark Generators

DBPedia SPARQL Benchmark (DBSB)

The general ideas of DBSB [108] are to scale the DBPedia data set to the required size and to create queries based on a historic query log of DBPedia SPARQL endpoints. In order to generate the data set a DBPedia dump is taken. To increase its size, triples are replicated and their namespaces are changed. To shrink the data set size, triples are removed in a way that characteristics like the indegree and the outdegree of vertices is not changed. In order to generate queries, DBSB clusters all queries of a historic query log. Out of each cluster the most frequent queries are picked as well as queries that cover most SPARQL features. Based on the selected queries, new queries are generated by replacing the constants with resources of the generated data set during the benchmark generation process.

Waterloo SPARQL Diversity Test Suite (WatDiv)

WatDiv [15] was designed to create benchmarks that are able to test the performance changes of RDF stores under varying data sets and query characteristics. Therefore, the data set generator is able to create data sets with variations of (a) the entity types, (b) the graph topology, (c) the well-structuredness of entities (i.e., which portion of the defined edges are usually present at an entity), (d) the probability of edges connecting two entities and (e) the cardinality of properties. In order to generate queries based on a data set, the following characteristics are defined:

Triple Pattern Count defines the number of triple patterns occurring in the generated query.

Join Vertex Count counts the number of resources or variables that occur in multiple triple patterns.

Join Vertex Degree determines in how many triple patterns each join vertex occurs.

Join Vertex Type defines whether a subject-subject, subject-object or object-object join is performed.

Result Cardinality is the number of results.

Filter Triple Pattern Selectivity defines with witch portion of the graph a triple pattern matches.

BGP-Restricted f-TP Selectivity determines to which extent a triple pattern contributes to the overall selectivity of a query.

Join-Restricted f-TP Selectivity determines to which extent a triple pattern contributes to the overall selectivity of a join.

FEASIBLE

FEASIBLE [133] does not provide a data set generator. Instead it can use an arbitrary data set for which a historic query log exists. FEASIBLE aims to generate queries that have characteristics similar to the queries in a query log. Therefore, in a first step all syntactical incorrect queries and queries with no results are removed. Each query is transformed into a vector based on the following query characteristics:

SPARQL features defines which SPARQL features like `SELECT`, `ASK`, `UNION`, etc. occur in the query.

Triple Pattern Count defines the number of triple patterns occurring in the generated query.

Join Vertex Count counts the number of resources or variables that occur in multiple triple patterns.

Join Vertex Degree determines in how many triple patterns each join vertex occurs.

Join Vertex Type defines whether a subject-subject, subject-object or object-object join is performed.

Triple Pattern Selectivity defines with which portion of the graph a triple pattern matches.

From the resulting set of query vectors, the requested number of queries are selected in a way that their vectors are as far away as possible from each other.

SPLODGE

The idea of SPLODGE [49] is to generated queries with a given set of characteristics from an arbitrary data set. Thereby, it uses the following query characteristics:

Query Type defines whether a `SELECT`, `CONSTRUCT`, `ASK` or `DESCRIBE` query should be generated.

Join Type defines whether a conjunctive join (`.`), disjunctive join (`UNION`) or left-join (`OPTIONAL`) should be performed.

Result Modifiers defines whether the result set should be altered by `DISTINCT`, `LIMIT`, `OFFSET` or `ORDER_BY` operators.

Variable Patterns defines at which positions of the triple pattern variables should occur.

Join Patterns defines whether a subject-subject, subject-object or object-object join is performed.

Cross Products defines whether a conjunctive join without join variables should be performed.

Number of Sources defines from how many different data sources triples should be combined to answer the query.

Number of Joins defines how many joins should occur in the query.

Query Selectivity defines with which portion of the graph all triple patterns of a query match.

3.2.3. Performed Evaluations

The before mentioned benchmarks are usually used to evaluate and compare the performance of RDF stores as a whole. Table 3.4 summarizes the evaluations published from the beginning of 2016. All of them use at least one of the benchmarks described above. Beside the generated data sets they usually also use a few realistic data sets. The maximal data set size is in most cases approximately 1 billion triples. In two cases a data set with up to 4.2 billion triples was used. Most RDF stores in the cloud were deployed on 10 compute nodes. In one evaluation 19 compute nodes were used. Nevertheless, evaluating an RDF store as a whole may not allow for identifying to which extent the observed performance was caused by the underlying hardware or the RDF stores itself.

The evaluations in these papers use rather small data sets. To give a better overview of the capabilities of current RDF stores, [1] reports RDF stores running on a single server or in the cloud that could store RDF graphs consisting of several billions or even one trillion triples (see the summary in table 3.5).

Paper	Benchmark	Max. Data Set Size	# compute nodes	compute node size
[29]	WatDiv	~100M triples	10	6 cores, 32 GB RAM, 4 TB disk
[125]	WatDiv	~1,000M triples	10	6 cores, 32 GB RAM, 4 TB disk
[110]	LUBM	~1,330M triples	18	12 cores, 50 GB RAM
	WatDiv			
[106]	WatDiv	~10M triples	11	4 cores, 24 GB RAM
[100]	LUBM	~35M triples	18 slaves	16 cores, 28 GB RAM
			1 federator	16 cores, 56 GB RAM
[5]	LUBM	~4,200M triples	12	24 cores, 148 GB RAM
[157]	LUBM	~220M triples	4-16 slaves	4 cores, 8 GB RAM, 500 GB disk
			1 master	4 cores, 16 GB RAM, 500 GB disk
[138]	WatDiv	~1,000M triples	10	6 cores, 32 GB RAM, 4 TB disk
[135]	WatDiv	~100M triples	10	6 cores, 32 GB RAM, 4 TB disk
[122]	WatDiv	~1,382M triples	10	8 cores, 32 GB RAM
	LUBM			
[119]	BSBM	~5M triples	4-12	2 cores, 8 GB RAM
[118]	WatDiv	~1,099M triples	10	4 cores, 16 GB RAM, 500 GB disk
	LUBM			
	FedBench			
[117]	WatDiv	~250M triples	10	4 cores, 16 GB RAM, 150 GB disk
[60]	WatDiv	~4,288M triples	5-12	24 cores, 148 GB RAM
	LUBM			
[52]	WatDiv	~1,380M triples	10	4 cores, 17 GB RAM
	LUBM			
[4]	LUBM	~3,100M triples	11	8 cores, 16 GB RAM, 3TB disk

Table 3.4.: Evaluations of RDF stores in the cloud published since 2016.

In order to compare the influence of alternative graph cover strategies or different query execution strategies, all but the examined component of the distributed RDF store need to stay the same. This was done, for instance in [110] to compare different query execution strategies on top of Spark or in [33], [66], [91] and [163] to compare different graph cover strategies. But these evaluations used Hadoop or its distributed file system to exchange data during the query processing. As a result, it remains unclear whether the made observations are similar to the observations that could be made in distributed RDF stores in which the data is exchanged directly between the compute nodes. To the best of my knowledge, no paper evaluated the influence of the graph cover strategy on the query performance using the same distributed RDF store.

RDF Store	Max. Data Set Size	# compute nodes	compute node size
Oracle Database 12c[a]	~1 trillion triples	1	360 cores, 2 TB RAM, 45 TB disk
AllegroGraph[b]	~1 trillion triples	1	?
Stardog[c]	~50,000M triples	1	32 cores, 256 GB RAM
Virtuoso Clustered Edition [42]	~37,000M triples	8	8 cores, 16 GB RAM, 4 TB disk
GraphDB[d]	~17,000M triples	1	16 cores, 512 GB RAM
4store [61]	~15,000M triples	9	?
Blazegraph [2]	~12,700M triples	?	?
YARS2 [2]	~7,000M triples	?	?
Jena TDB[e]	~1,700M triples	1	2 cores, 10 GB RAM
RDFox[f]	~1,700M triples	1	16 cores, 50 GB RAM

[a] http://www.oracle.com/us/corporate/features/database-12c/index.html
[b] https://franz.com/agraph/allegrograph/
[c] https://www.stardog.com/
[d] www.ontotext.com/products/ontotext-graphdb/
[e] https://jena.apache.org/
[f] http://www.cs.ox.ac.uk/isg/tools/RDFox/

Table 3.5.: Evaluations of RDF stores reported by [1].

CHAPTER 4

Methodology for Benchmarking Graph Cover Strategies

When defining a methodology for investigating the effects of graph cover strategies on distributed RDF stores, several challenges arise. Beyond the overall performance for the processing of SPARQL queries [123], observing indications that contribute to understanding how graph cover strategies may relate to scalability is useful. Two examples of such indicators are the horizontal containment and the vertical parallelization. The horizontal containment describes to which extent computation of individual query results is local to one (or few) graph chunk(s). This is an indicator that query processing is (to some extent) robust when the cloud is scaled horizontally. The vertical parallelization describes to which extent different query results may be computed in parallel on different compute nodes. This is an indicator that query processing can scale with growing result set sizes by horizontal scaling of the cloud. All used high-level indicators are formally defined in Section 4.1.

Ideally, the graph cover strategy would be the only independent input variable based on which to pursue evaluation and to obtain values for dependent variables. Performance observations of graph cover strategies, however, are tightly interwoven with several factors. The first two factors are the data sets and the specific queries that are processed as part of the benchmark (cf. Section 4.2). Furthermore, actual query execution constitutes a highly influential factor, too, for which the execution strategies (cf. Section 4.3) as well as execution operation (cf. Section 4.4) are defined. For these two factors, the described methodology aims at experimenting with a diverse set of inputs in order to allow for recognizing the patterns of influence between graph cover strategies and performance measures. This chapter is mainly taken from [73].

4.1. Evaluation Measures

In this subsection, the measures that seem to be most useful to characterize different graph cover strategies are defined. Experiments with further measurement and statistics functions, e. g. standard deviation instead of Gini coefficient, have been made but they seemed to be correlated so that the measures that seem to be the most intuitive to interpret were chosen.

Load times

Loading a data set typically involves at least seven steps, some of which may be interleaved and/or parallelized:

1. Initial dictionary encoding of nodes and labels unused during graph cover creation (see Section 4.4) for faster access and memory savings.

2. Computation of the graph cover.

3. Final dictionary encoding of nodes and labels used during graph cover creation.
4. Collection of statistical information.
5. Setting join responsibility of resources.
6. Transfer of data chunks to compute nodes.
7. Indexing of data chunks at local compute nodes.

Given a data set and a graph cover strategy, the overall load time comprises these 7 steps. Since the computation of the graph cover is directly related to the graph cover strategy, the load time L is defined as the time required for the computation of the graph cover. This measurement is a weak indicator for setting up a data set in a cloud RDF store. However, all other steps by themselves are complex enough to warrant deeper investigation in the future.

Storage imbalance

Scaling the cloud for handling growing memory needs may be jeopardized by graph cover strategies aiming for a low number of transferred intermediate results. They might generate a skewed distribution delegating expensive tasks on few compute nodes. Therefore, the quality of the storage distribution resulting from a graph cover strategy with the storage imbalance b are evaluated.

Definition 17. For a given cover, *storage imbalance* b is defined by the Gini coefficient

$$b := \frac{2 * \sum_{i=1}^{|C|} i * \text{volSeq}(i)}{(|C|-1) * \sum_{c \in C} \text{vol}(c)} - \frac{|C|+1}{|C|-1}, \ 0 \leq b \leq 1$$

whereby $\text{vol}(c) := |\text{chunk}_{\text{cover}}(c)|$ describes the number of triples on a compute node c and $\text{volSeq}(i)$ returns the size of the ith chunk in the ascending sequence of all $\text{vol}(c)$.

Thus, storage imbalance b is defined by the Gini coefficient of the distribution of triple occurrences with $b = 1$ indicating maximal imbalance and $b = 0$ maximal balance.

Beside the Gini coefficient, experiments with the standard deviation and entropy were also performed but the Gini coefficient was chosen, since the produced values are within a fixed range between 0 and 1 independent of the actual chunk sizes and thus better comparable. Furthermore, the experiments showed that the storage imbalance between different graph cover strategies is better visible than using entropy.

Storage redundancy

When handling with triple replication the number of triples in the graph chunks will be larger than the original number of triples in the graph. Therefore, the storage redundancy is defined as the blow-up factor, where $r = 1$ indicates no redundancy and $r = |C| \cdot |G|$ indicates maximal replication of triples on all compute nodes.

Definition 18. For a given cover of a graph G, the *storage redundancy* is defined as

$$r := \frac{\sum_{c \in C} \text{vol}(c)}{|G|}, \ 1 \leq r \leq |C| \cdot |G|$$

whereby $\text{vol}(c) := |\text{chunk}_{\text{cover}}(c)|$ describes the number of triples on a compute node c.

Overall query performance

Depending on the target use case, different overall performance characteristics of an RDF store may be desirable. While the time to delivery of the complete result is crucial, e.g., for statistical reports, in a fact-finding mission one may be more interested in only few top-k results being returned quickly. Hence, different kinds of performance characteristics are provided. Characteristics depend on measuring the time interval between issuing the query q (more precisely the query execution tree as elaborated on in Section 4.3) at time t_0^q and the time when the i-th result is returned at t_i^q with K^q representing the overall number of query results for query q. The superscript ^{q} is dropped when it is clear from context as in the following definitions.

Definition 19. Overall query performance is evaluated by the following functions, with K being the overall number of query results:

Query time to completion: $exTime := t_K - t_0$

Result curve function: $\chi(t) := \frac{|\{t_i | t_i - t_0 \leq t\}|}{K}$

$\chi(t)$ allows us to plot the percentage of returned results on a time axis between 0 and $t_K - t_0$. A curve that is strictly below another one will indicate that results are returned more slowly.

Horizontal containment

Time-based measurements such as *exTime* depend on the exact configuration of the system such as network bandwidth and latency. In a distributed system, often the most time-consuming operation is data transfer. Graph cover strategies that lead to massive data transfer indicate that computation of individual query results is not contained on one or few compute nodes, and hence suggests that it will not allow the cloud to be scaled horizontally. Hence, an abstract level of data transfer is measured:

Definition 20. For a join operation op, $\left|\mathrm{dom}(\mu_i^{op})\right|$ are the number of variables of a variable binding and m are the number of variable bindings $m = \left|\{\mu_i^{op}\}\right|$. Each join operation that leads to the sending of variable bindings from one compute node to another transfers data of size $m^{op} \cdot \left|\mathrm{dom}(\mu_i^{op})\right|$. The size of the data transferred from one compute node c is measured as $T_c := \sum_{op} m^{op} \cdot \left|\mathrm{dom}(\mu_i^{op})\right|$. The overall data transfer is then defined as $T := \sum_{c \in C} T_c$ for a given cover and a given query execution tree.

Additionally to the data transfer which measures the amount of transferred data, the amount of transferred packets is measured:

Definition 21. For a given cover and a given query execution tree, the number of transferred packets is defined as $P := \sum_{c \in C} P_c$, where P_c is the number of packets sent from c to any other compute node $c' \neq c$.

The data transfer is sometimes also used as the preferred measurement for overall query efforts in the cloud, as in standard cloud architecture the processor-to-remote-memory gap by far excels the processor-to-local-memory gap. In newer hardware architectures that natively support remote direct memory access large differences between these gaps cannot be taken for granted anymore. Thus, the data transfer and the workload imbalance are measured.

Vertical parallelization

In order to measure workload independently of time, the number of join comparisons to be performed are observed. Given a query execution tree the overall workload will be identical for all graph cover strategies. The vertical parallelization expresses how many join comparisons might be executed by different compute

nodes in parallel. This number is very difficult to obtain as it would require the definition and implementation of complex concepts in a distributed system such as 'simultaneous' or 'nearly simultaneous'. A simple, but effective strategy is pursued here, by simply measuring how the workload is distributed over different compute nodes using the Gini coefficient.

Definition 22. For a cover and a query execution tree, *workload imbalance W* is the Gini coefficient:

$$W := \frac{2 * \sum_{i=1}^{|C|} i * \mathrm{wSeq}(i)}{(|C|-1) * \mathrm{w}(C)} - \frac{|C|+1}{|C|-1}, \ 0 \leq W \leq 1$$

where the workload of a compute node $\mathrm{w}(c)$ is defined by the number of join comparisons on c, $\mathrm{wSeq}(i)$ denotes the ith workload in the ascending workload sequence of all compute nodes, and $\mathrm{w}(C) = \sum_{c \in C} \mathrm{w}(c)$ is the total computational effort on all slaves.

In the strict sense, workload imbalance does not measure vertical parallelization, because an actual query might involve many compute nodes in a strictly sequential manner. However, each sequential processing of a query requires data transfer. Thus, in combination with horizontal containment the following interpretation table can be created that derives a comprehensive picture when jointly considering workload imbalance and the packet transfer as a measure for horizontal containment (see Table 4.1). Based on the evaluation the workload imbalance may be seen as low for a $W < 0.1$ and the horizontal containment may be seen as high if less than 0.01 packets would be transferred to produce a single query result.

	P high	P low
W low	high vertical parallelization	low to medium vertical parallelization
W high	low vertical parallelization	low vertical parallelization (unlikely situation)

Table 4.1.: Measurement of vertical parallelization.

4.2. Data Set and Queries

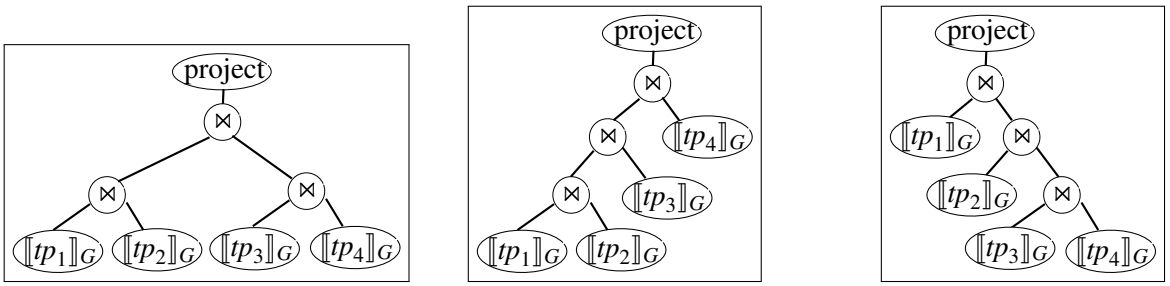
Since the core functionality of SPARQL is provided by matching basic graph patterns, the proposed evaluation methodology follows the strategy of most other benchmarks, performing evaluations with varied basic graph pattern structures. In particular, the strategy of SPLODGE [49] is adopted, which varies the query characteristics given arbitrary real-world data sets:

Number of joins: controls the number of triple patterns in the basic graph pattern.

Selectivity: controls the number of triples involved in answering the query.

Join pattern: controls the branching factor that shapes the basic graph pattern to a smaller or larger extent into a path–shaped query or star–shaped query.

Number of sources controls for the number of data sources that need to be involved to answer a query (e.g., DBPedia and GeoNames would be two).



(a) Bushy query execution tree. (b) Left-linear query execution tree. (c) Right-linear query execution tree.

Figure 4.1.: The three different query execution strategies for the query from example 4.

While the first three are common to most benchmarks, the last one has been specifically added to SPLODGE for benchmarking federated stores. Varying this parameter between 1 and several units is important in this context, as several graph cover strategies may easily collocate data from a single data source on a single compute node. When testing the limits of graph cover strategies, it must be ensured that also 'hard' test cases are created. Since some queries might produce accidentally huge result sets, the number of results is limited to 1 million.

To better identify the influence of data-set-specific characteristics on the performance of the graph cover strategies, the frequently used Waterloo SPARQL Diversity Test Suite (WatDiv) [15] is used to generate additional data sets. For each data set 20 queries based on the basic testing query templates[1] are generated. These generated queries consists of star-shaped queries (S1-S7), path-shaped queries (L1-L5) and combinations of both shapes (C1-C3 and F1-F5).

4.3. Query Execution Strategies

In order to find out about weaknesses and strengths of graph cover strategies, it has to be determined how far our evaluation measures are influenced by the graph cover strategies themselves and how far they are influenced by interfering aspects of the overall RDF store. Query planning and execution are so intrinsically interwoven that it is rather impossible to come up with one (or several) query optimizers and planners that fit all challenges.This issue is remedied in a similar way as done for data set and queries: The suitability of the different graph cover strategies under variations of query executions are systematically explored. Thus, the performance of "the best run", which would be hard to achieve anyway, is not measured but the robustness and susceptibility of graph cover strategies vs. execution strategies are characterized.

Specifically, (i) a bushy query execution tree with minimal height, (ii) a left-linear query execution tree, in which the triple patterns are joined in the sequence they are defined and (iii) a right-linear query execution tree are used. Thus, trees of different heights and topological sorting are used. To evaluate the performance of graph cover strategies under variations of query execution trees, an operative environment that can handle different graph cover strategies and such variations of query execution trees have been devised. This environment is described next.

Example 17. Figure 4.1 shows the three different query execution trees generated for the query from example 4. The bushy query execution tree joins all consecutive triple patterns pairwise as shown in Figure 4.1a. The resulting intermediate results are joined pairwise again until all joins are performed. The left-linear query

[1] `https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml`

execution tree joins the triple patterns in the sequence they were defined in the query (see Figure 4.1b) whereas the right-linear query execution tree joins them in the reverse sequence as shown in Figure 4.1c.

4.4. Distributed RDF Store for Arbitrary Graph Covers (Koral)

The distributed RDF store for arbitrary graph covers (Koral)[2] implements a query execution mechanism that receives a data set, a graph cover, a query and a query execution strategy and computes the corresponding query result set. Its formal definition is given in Section 4.4.2 and the proofs of soundness and completeness are given in Appendix B.

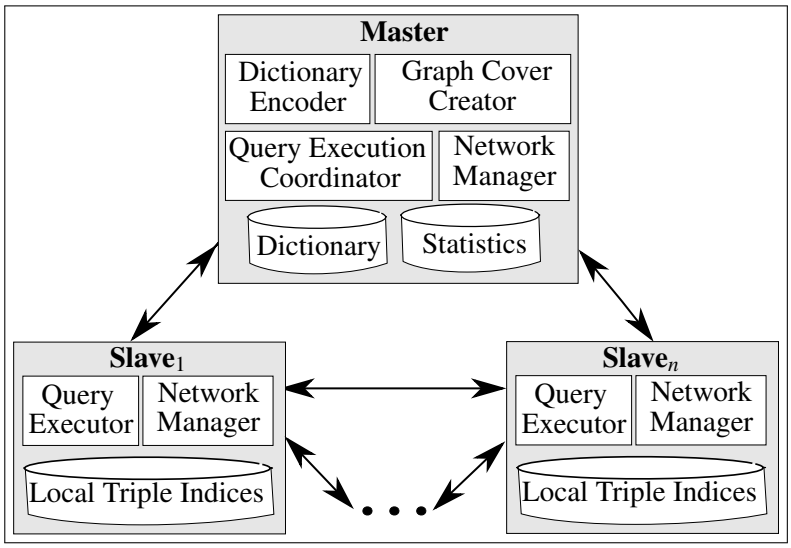


Figure 4.2.: Architecture of Koral.

Koral is an extension of state-of-the-art asynchronous execution mechanisms such as realised in TriAD [55]. The extensions render the query execution mechanism independent from the underlying graph cover. The architecture of Koral is depicted in Figure 4.2. Koral consists of one master node and $|C|$ slave nodes. The network managers maintain peer-to-peer network connections and manage the network communication.

4.4.1. Graph Loading

At loading, the huge size of the input graph needs to be reduced as early as possible. Therefore, the contained textual resources are replaced by numerical ids. The creation of the ids as well as storing the mapping between the textual and the numerical representation is done by the dictionary encoder. If a minimal edge-cut cover should be created, the subject, property and object of the triples can be encoded. In the cases of the hierarchical cover the subjects are kept in their textual representation since the IRI hierarchy is required for the creation of this cover. The encoded graph is then used by the graph cover creator to create the requested graph chunks. In case of the hierarchical cover the textual subject resources are encoded afterwards in order to reduce the size of the graph chunks. During the creation of the graph cover, for each triple the graph chunks in which it occurs is stored.

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

As described in Section 4.3 each resource is uniquely assigned to a slave that is responsible for joining it during the query processing. In order to increase the *horizontal containment*, the assignment of a resource is based on the frequency with which it occurs in the different graph chunks. Therefore, the frequency of the different resources in the different chunks is counted and stored in a statistics database. In the current implementation the statistics database is a single huge file which is randomly accessed. Each resource stores its statistical data at a dedicated region of the file. When the statistical data have been completely collected, the loading process iterates over all graph chunks again. During the iteration, the slaves responsible for joining the individual resources are determined. The resource id is then prefixed by the id of the responsible slave and written to disk again.

After adjusting the join responsibility, the graph chunks are sent to the slaves. The slaves create local index structures (SPO, OSP, and POS indices as described in [154]) that also includes the information on which slaves the individual triples are stored. While the multi-pass strategy has the disadvantage that it iterates the data files several times, it has the advantage that it prevents to run out of memory and is thus highly scalable for very large files. In order to reduce the cost of disk I/O all components except the statistics database access the data files linearly.

4.4.2. Query Execution During Run-time

At run-time, a query execution coordinator is instantiated for each received query. During the initial parsing step the query execution tree is created that specifies the query execution strategy (bushy, left-linear, right-linear). Thereby, all constants are encoded using the dictionary and the join responsibility is adjusted using the statistics.

The created query execution tree is serialized and submitted to all slaves. Each slave deserializes the tree, prepares all query operations for execution and finally sends a ready-to-start notification to the query execution coordinator. When all slaves are ready to start, the coordinator instructs all slaves to start the execution of the query operators. This synchronization step has the advantage that the receiver of an intermediate result sent from one query operation to another on a different slave node is guaranteed to exist.

When the slaves execute the individual triple operations, the match operations use the local triple indices to find matches for the corresponding triple pattern. The resulting variable bindings are transferred to the succeeding join operation on the slave responsible for the join of the resource, i.e. $\mu(v)$ (where v is the join variable) aiming at *horizontal containment*. The data transfer of the intermediate variable bindings is formally defined in Section 4.3. In order to make better use of the network bandwidth, several intermediate results are bundled together and sent to the receiving slave within one packet.

Whenever the join operation receives a variable binding, it is joined with the cached variable bindings. The join results are directly sent. Since the number of received variable bindings can exceed the memory of a slave, up to 32k variable bindings are cached in memory. If more variable bindings needs to be cached, they are inserted into a persistent B-tree.

When all child operations in the query execution tree of a query operation o are finished and no further input needs to be processed, it sends a finish notifications to all o operations on the other slaves. If o has received the finish notifications from all other o operations, it declares itself as finished. This synchronization step is required to guarantee that all results are found.

If a query operation has no succeeding operation, i.e. it is the root operation in the query execution tree, it sends its results to the query coordinator. The coordinator decodes the ids using the dictionary and sends the decoded variable bindings to the sender of the query. When the coordinator receives the finish notification of the root operations from all slaves, it sends a finish notification to the sender of the query and terminates itself.

In the case that the query contains a limit for the number of results, the query coordinator counts the number

of variable bindings it has sent to the sender of the query. If the limit is reached, it instructs all slaves to abort the query execution.

Formalization of Distributed Query Execution Strategy

To improve the comprehensibility, the distributed query execution strategy of Koral is formalized in two steps. First, the initial strategy is defined that ignores replicated triples. Thereafter, the initial strategy is extended to benefit of replicated triples. In Koral only the extended query execution strategy has been implemented.

Distributed Query Execution Strategy Ignoring Triple Replication

The distributed query execution strategy aims to benefit from the vertical parallelization and horizontal containment capabilities provided by the underlying graph cover. Therefore, query results that can be produced by triples contained in only one graph chunk should be computed on the compute node storing the chunk without causing additional data transfer. In order to approximate this goal, each compute node executes all operations of the query execution tree. Whenever a query operation produces an intermediate variable binding, it has to be decided whether the succeeding join can be processed locally or on a different compute node. This decision is defined by the join responsibility of a resource which is assigned to a join variable of the succeeding join.

Definition 23. The *join responsibility of a resource* is a function $\text{jResp} : IBL \rightarrow C$ that assigns each resource to a compute node.

In order to benefit from the horizontal containment of a graph chunk, the actual join responsibility assignment is based on the occurrence of resources in the different graph chunks. A resource r is assigned to the compute node who stores the graph chunk containing r at the subject position most frequently. If r does not occur at the subject position, the occurrence at the object and predicate position defines its join responsibility.

Example 18. Assume the example hash cover from Figure 3.4. The join responsibilities for the different resources are:

$$r \in \{\text{g:Gesis}, \text{g:wanja}, \text{w:martin}, \text{"Wanja"}, \text{"Martin"}, \text{f:knows}, \text{f:givenname}\} \Rightarrow \text{jResp}(r) := c_1$$

$$r \in \{\text{g:bello}, \text{w:WeST}, \text{w:daniel}, \text{r:type}, \text{e:employs}, \text{g:Dog}, \text{e:ownedBy}, \text{"Daniel"}\} \Rightarrow \text{jResp}(r) := c_2 \ .$$

Since a join operation may consist of more than one join variable, one of these variables needs to be selected deterministically in order to determine the compute node responsible for the join processing. Therefore, an arbitrary but fixed strict total order $<_V$ is defined on V (e.g., a lexicographic order on the variable names). With its help the least variable out of the set of all join variables can be extracted.

In order to simplify the definition of the distributed query execution strategy, a function is defined that extracts from all intermediate results produced by a query operation on a compute node the set of all variable bindings that should be transferred to a dedicated compute node c. This decision is based on the join responsibility of the resource bound to the join variable but there exists two corner cases: (i) empty variable bindings are transferred to all compute nodes and (ii) if the join is a Cartesian product (i.e., no join variable exists), then the variable binding is sent to the first compute node. Therefore, an arbitrary but fixed strict total order $<_C$ is defined on C (e.g., a lexicographic order on the IP addresses of the compute nodes).

Definition 24. Let $<_S$ be a strict total order defined on a set S, then the *minimum* is defined as follows.

$$\begin{aligned} &\min_{<_S} : 2^S \rightarrow S \\ &\min_{<_S}(S') := s, \text{ iff } S' \subseteq S \wedge S' \neq \varnothing \wedge s \in S' \wedge \forall s_i \in S' : s_i \neq s \Rightarrow s < s_i \ . \end{aligned}$$

Definition 25. The *route function* extracts all variable bindings from a set of variable bindings $\hat{\Omega}$ that will be produced by a query Q processed on a compute node c. For an arbitrary but fixed join responsibility function jResp, it is defined as follows.

$$\begin{aligned} \text{route} &: C \times \Upsilon \times \mathscr{O} \to \mathscr{O} \\ \text{route}(c, \langle\!\langle Q \rangle\!\rangle, \hat{\Omega}) &:= \{\mu \in \hat{\Omega} | \mu = \varnothing \\ &\quad \vee (\text{cVars}(\langle\!\langle Q \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\ &\quad \vee (\text{cVars}(\langle\!\langle Q \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu(\min_{<_V}(\text{cVars}(\langle\!\langle Q \rangle\!\rangle))) = c)\} \quad . \end{aligned}$$

Example 19. If the triple pattern `?v1 <f:knows> ?v2` from the query in example 4 on page 8 is executed on compute node c_1 of the example hash cover in Figure 3.4 on page 28, the variable binding $\mu_1 = \{(\texttt{?v1}, \text{g:wanja}), (\texttt{?v2}, \text{w:daniel})\}$ is produced as shown in Figure 4.3. The succeeding join operation will join on variable `?v2`. μ_1 maps `?v2` on the IRI w:daniel. The join responsibility for this IRI was assigned to $\text{jResp}(\text{w:daniel}) = c_2$. Therefore, $\text{route}(c_2, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_1\}) = \{\mu_1\}$ will identify μ_1 to be transferred to compute node c_2.

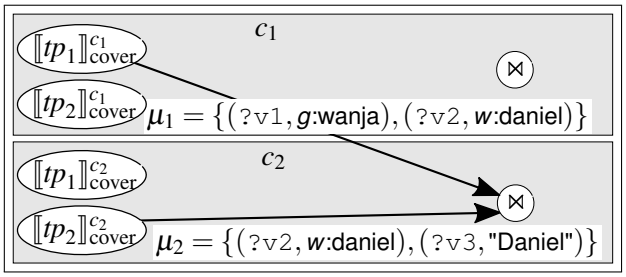


Figure 4.3.: An illustration of the distributed join.

On compute node c_2, the triple pattern `?v2 <f:givenname> ?v3` will lead to the creation of the variable binding $\mu_2 = \{(\texttt{?v2}, \text{w:daniel}), (\texttt{?v3}, \text{"Daniel"})\}$. The succeeding join operation will join on variable `?v2`. μ_2 maps this variable on the same IRI as μ_1. Thus, $\text{route}(c_2, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_2\}) = \{\mu_2\}$ will identify μ_2 to be joined on the same compute node c_2 on which it was produced. Since now, both compatible mappings μ_1 and μ_2 are assigned to the same join operation on the same compute node, they can be joined.

Definition 26. For an arbitrary but fixed jResp the *evaluation* of a SPARQL query Q over a graph cover called cover *on a computer* c, denoted by $[\![Q]\!]^c_{\text{cover}}$, is defined recursively as follows:

1. If $tp \in TP$ then $[\![tp]\!]^c_{cover} = \{\mu | \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in \text{chunk}_{\text{cover}}(c)\}$.
2. If B_1 and B_2 are BGPs, then
$[\![B_1.B_2]\!]^c_{\text{cover}} = \left(\bigcup_{c_i \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_i}_{\text{cover}}) \right) \bowtie \left(\bigcup_{c_i \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_i}_{\text{cover}}) \right)$.
3. If $W \subseteq V$ and B is a BGP, then $[\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^c_{\text{cover}} = \text{project}(W, [\![B]\!]^c_{\text{cover}}) = \{\mu_{|W} | \mu \in [\![B]\!]^c_{\text{cover}}\}$.

Definition 27. The *distributed evaluation* of a SPARQL query Q over an arbitrary graph cover called cover that assigns triples of an arbitrary RDF graph G to compute nodes C, denoted by $[\![Q]\!]_{\text{cover}}$, is defined as

$$[\![Q]\!]_{\text{cover}} := \bigcup_{c \in C} [\![Q]\!]^c_{\text{cover}}.$$

With the help of these definitions, it is possible to prove that the defined distributed execution mechanism is semantically correct and complete.

Theorem 1. *The centralized evaluation of query Q produces exact the same results as its distributed evaluation, i.e.*

$$[\![Q]\!]_{\text{cover}} = [\![Q]\!]_G \ .$$

The proof of Theorem 1 is shown in Appendix B. A full example of the distributed query execution strategy without triple replication is given in Section A.1.

The presented distributed query execution strategy is similar to the shuffle join of Apache Spark[3]. Whereas the shuffle join uses the hash of the join key to identify the compute node on which two data items are joined, the presented query strategy joins the data items on the compute node where the join node occurs the most frequently.

Distributed Query Execution Strategy With Triple Replication

The distributed query execution strategy described in the previous section has the disadvantage that in the presence of replicated triples the query would match with each replica, transfer them all to one compute node and then computes joins for each of them. Thus, triple replication would lead to a increased number of transferred intermediate results and a higher computational effort. In order to benefit from triple replication, systems like [92] try to avoid the transfer of duplicate intermediate results by using the replicated triple to increase the amount of intermediate results that can be computed locally on the individual compute nodes.

Systems like [92] benefit of triple replication by avoiding the exchange of intermediate results. To reduce the exchange of intermediate results within Koral, the distributed query execution strategy is extended differently from [92]. This extension keeps track on which compute nodes a produced intermediate result is already known. Only if the compute node on which it should be processed next does not know it already, it is transferred to it. In order to do so, each triple is annotated with the compute nodes to which it was assigned during the creation of the graph cover.

Example 20. Figure 4.4 shows the 2-hop replication extension of a hash cover from Figure 3.6 on page 31. The information to which compute nodes each triple was assigned to is annotated behind the edge labels. In the following, this cover will be used to explain how the extension of the query execution mechanism works. Since the occurrences of the resources in the different chunks have changed, the new join responsibilities are:

$$r \in \{\text{g:Gesis}, \text{g:wanja}, \text{"Wanja"}, \text{"Martin"}, \text{"Daniel"}, \text{f:knows}, \text{f:givenname}\} \Rightarrow \text{jResp}(r) := c_1$$

$$r \in \{\text{g:bello}, \text{w:WeST}, \text{w:daniel}, \text{w:martin}, \text{r:type}, \text{e:employs}, \text{g:Dog}, \text{e:ownedBy}\} \Rightarrow \text{jResp}(r) := c_2 \ .$$

When matching a triple pattern with these annotated triples, the resulting variable binding can be extended by the set of compute nodes to which the original triple was assigned to. By doing so, the resulting localized variable binding – in the following named as μ^l – know on which compute nodes they could be created.

Definition 28. A *localized variable binding* is a tuple (μ, C') with the set of compute nodes $C' \subseteq C$ on which the variable binding μ is known. The set of localized variable bindings denoted as Ω^{loc}.

[3] https://www.waitingforcode.com/apache-spark-sql/shuffle-join-spark-sql/read

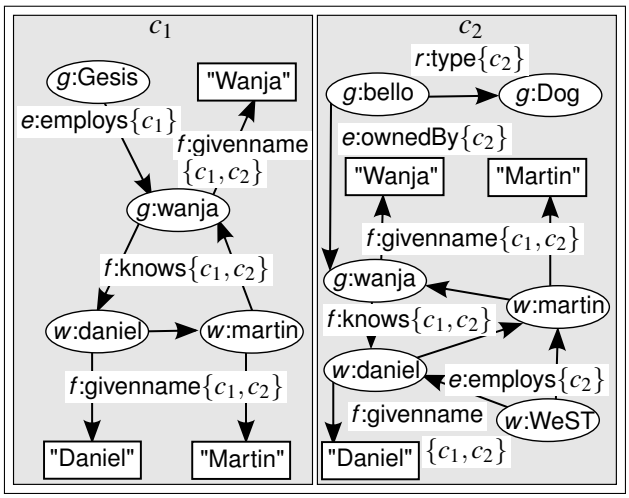


Figure 4.4.: The 2-hop replication extension of the hash cover in Figure 3.6 on page 31 with compute nodes annotated to the triples.

Example 21. When finding matches for the triple pattern $tp_1 =$ `?v1 <f:kknows> ?v2` from the example query in example 4 on page 8 with the graph chunk stored on compute node c_2 the localized variable binding $\mu_1^l = (\{(\texttt{?v1}, \text{w:martin}), (\texttt{?v2}, \text{g:wanja})\}, \{c_1, c_2\})$ is created. Since the underlying triple was assigned to both compute nodes, also μ_1^l was created on both compute nodes.

On compute node c_2 the triple pattern $tp_e =$ `<w:WeST> <e:employs> ?v1` creates the localized variable binding $\mu_2^l = (\{(\texttt{?v1}, \text{w:martin})\}, \{c_2\})$. Since the underlying triple was only assigned to compute node c_2, μ_2^l is only known by c_2.

In order to join two localized variable binding, definition 13 on page 9 is extended. A variable binding that is created by joining two localized variable bindings μ_1^l and μ_2^l will be created and known on all compute nodes that know μ_1^l and μ_2^l.[4]

Definition 29. The *join* of two sets of localized variable bindings Ω_1^{loc} and Ω_2^{loc} is defined as

$$\Omega_1^{loc} \bowtie \Omega_2^{loc} = \left\{ \left(\mu_1 \cup \mu_2, C_1' \cap C_2'\right) \mid \left(\mu_1, C_1'\right) \in \Omega_1^{loc} \wedge \left(\mu_2, C_2'\right) \in \Omega_2^{loc} \wedge \mu_1 \sim \mu_2 \right\} \ .$$

Thereby, $\mu_1 \sim \mu_2$ means that the variable bindings μ_1 and μ_2 are compatible as defined by definition 12 on page 9.

Example 22. The join of μ_1^l and μ_2^l from the previous example will result in the localized variable binding $\mu_3^l = (\{(\texttt{?v1}, \text{w:martin}), (\texttt{?v2}, \text{g:wanja})\}, \{c_2\})$ since μ_2^l was only known on compute node c_2.

Crucial for reducing the number of transferred intermediate results is the decision whether a localized variable binding should be transferred or not. This decision is defined in the route$^{\text{loc}}$ function in definition 30. Three different cases can be distinguished: (i) an empty mapping should be transferred, (ii) the following join operation has no join variable, i.e., it is a Cartesian product, or (iii) the following join operation is a join on at least one join variable. In the first case the empty variable binding needs to be transferred to all compute

[4] In the distributed query execution strategy with triple replication only localized variable bindings are produced. Therefore, the join is only defined on localized variable bindings.

nodes. Thus, after transferring it, it will be known by all compute nodes. In order to prevent sending duplicates of the empty variable binding, only the first compute node knowing it will send it.

In case of a Cartesian product, all variable bindings need to be transferred to the first computer who will then process them. After sending the localized variable binding, only the receiving compute node will know it. In order to prevent the transfer of duplicates, only the first compute node knowing it will send it. One special case is, when the compute node to which the variable binding should be transferred already knows it. In this case, the variable binding will not be transferred over network.

In case of a join operation with at least one join variable, it is checked first, whether the compute node responsible for performing the join already knows the variable binding or not. If the compute node does not know it, it is transferred by the first compute node knowing it. As a consequence only the compute node responsible for the join will know it. If the compute know knows the variable binding already, then every compute node who knows the variable binding will forward it only to the succeeding local join operation. Thus, query processing can benefit from replicated triples, since joins can be processed on the local replicas of triples.

Definition 30. The *replication-aware route function* defines which localized variable bindings $\hat{\Omega}^{loc}$ are transferred from one compute node c_s to the parent query operation $\langle\langle Q \rangle\rangle$ on another compute node c_t. Thereby, C is the set of compute nodes, Υ is the set of query execution trees and $\mathscr{O}^{loc}$ the set of all localized variable binding sets. For an arbitrary but fixed join responsibility function jResp, it is defined as follows.

$$\text{route}^{\text{loc}} : C \times C \times \Upsilon \times \mathscr{O}^{loc} \to \mathscr{O}^{loc}$$

$$\begin{aligned}
\text{route}^{\text{loc}}(c_s, c_t, \langle\langle Q \rangle\rangle, \hat{\Omega}^{loc}) := \Big\{ (\mu, C'_r) \big| \exists\, (\mu, C') \in \hat{\Omega}^{loc} : \big(&\mu = \varnothing \wedge c_s = \min_{<_C}(C') \wedge C'_r = C\big) \\
\vee \big(&\text{cVars}(\langle\langle Q \rangle\rangle) = \varnothing \wedge c_t = \min_{<_C}(C) \wedge c_t \notin C' \wedge c_s = \min_{<_C}(C') \wedge C'_r = \{c_t\}\big) \\
\vee \big(&\text{cVars}(\langle\langle Q \rangle\rangle) = \varnothing \wedge c_t = \min_{<_C}(C) \wedge c_t \in C' \wedge c_s = c_t \wedge C'_r = \{c_t\}\big) \\
\vee \big(&\text{cVars}(\langle\langle Q \rangle\rangle) \neq \varnothing \wedge c_s = \min_{<_C}(C') \wedge \text{jResp}(\mu(\min_{<_V}(\text{cVars}(\langle\langle Q \rangle\rangle))) = c_t \wedge c_t \notin C' \wedge C'_r = \{c_t\}\big) \\
\vee \big(&\text{cVars}(\langle\langle Q \rangle\rangle) \neq \varnothing \wedge c_s = c_t \wedge \text{jResp}(\mu(\min_{<_V}(\text{cVars}(\langle\langle Q \rangle\rangle))) = c_u \wedge c_u \in C' \wedge C'_r = C'\big) \Big\} \quad .
\end{aligned}$$

Example 23. When executing the example query from example 4 on the 2-hop hash cover from Figure 4.4, the first triple pattern will create the localized variable binding μ_1^l on both compute nodes as shown in Figure 4.5. The following join on `?v2`. Based on the join responsibility of $\mu_1^l(\texttt{?v2}) = \text{g:wanja}$ compute node c_1

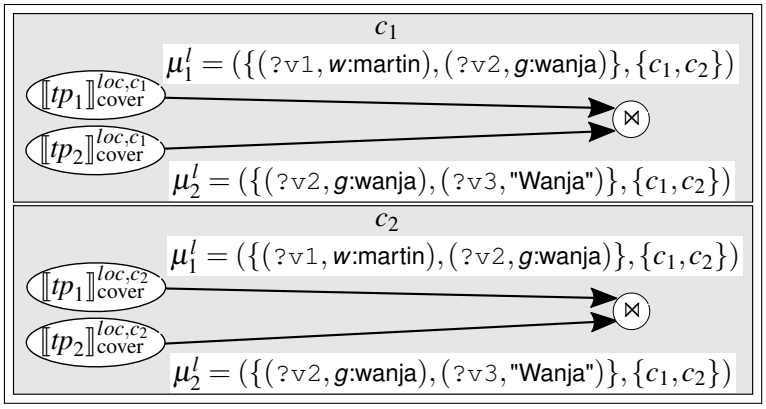


Figure 4.5.: Forwarding of duplicate localized variable bindings at the presence of replicated triples.

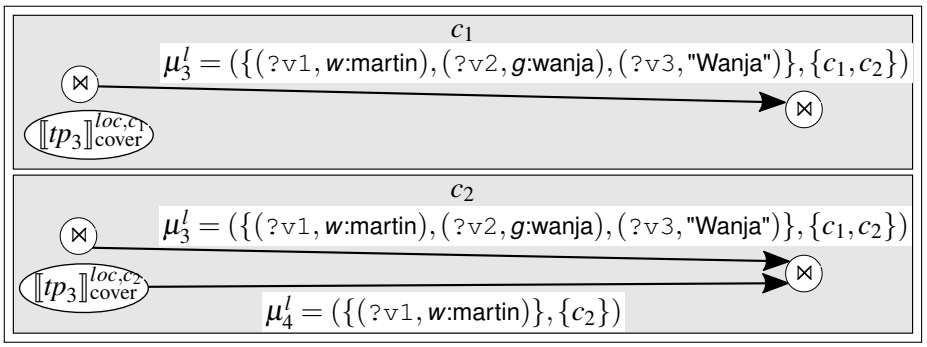


Figure 4.6.: Forwarding of unique localized variable bindings at the presence of replicated triples.

would be responsible for joining it. Since c_1 knows the variable binding already, route$^{\text{loc}}$ determines both localized variable bindings to be transferred to the local join operations, i.e., route$^{\text{loc}}(c_1, c_1, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_1^l\}) =$ route$^{\text{loc}}(c_2, c_2, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_1^l\}) = \{\mu_1^l\}$. The second triple pattern produces the localized variable binding μ_2^l on both compute nodes. Also this variable binding will be forwarded to the local succeeding join operations.

Both join operations will compute the resulting localized variable binding μ_3^l that is also known by both compute nodes. As shown in Figure 4.6 this variable binding is forwarded to the succeeding local join operations. The third triple pattern will produce the localized variable binding μ_4^l only on compute node c_2. Since the join responsibility of w:martin is c_2 and μ_4^l is already known on c_2 it is forwarded to the succeeding join operation on c_2.

When joining the localized variable bindings μ_3^l and μ_4^l on compute node c_2 the resulting localized variable binding $\mu_5^l = (\{(?v1, w{:}martin), (?v2, g{:}wanja), (?v3, "Wanja")\}, \{c_2\})$ is only known on compute node c_2, since μ_4^l is only known on c_2. The succeeding join operation would have the join variable ?v2. μ_5^l assigns g:wanja to this variable. Since compute node c_1 is responsible for joining it and c_1 does not know the localized variable binding yet, it is transferred to c_1, i.e. route$^{\text{loc}}(c_2, c_1, \langle\!\langle\langle\!\langle\langle\!\langle tp_1.tp_2 \rangle\!\rangle.tp_3 \rangle\!\rangle.tp_4 \rangle\!\rangle, \{\mu_5^l\}) = \{\mu_5^l\}$.

With the help of the previous definitions the replication aware evaluation of a SPARQL query can be defined.

Definition 31. For an arbitrary but fixed jResp the *replication-aware evaluation* of a SPARQL query Q over a graph cover called cover *on a computer* c, denoted by $[\![Q]\!]^{loc,c}_{\text{cover}}$, is defined recursively as follows:

1. If $tp \in TP$ then

$$[\![tp]\!]^{loc,c}_{cover} = \{(\mu, \text{cover}(\mu(tp))) | \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in \text{chunk}_{\text{cover}}(c)\}.$$

2. If B_1 and B_2 are BGPs, then

$$[\![B_1.B_2]\!]^{loc,c}_{\text{cover}} = \left(\bigcup_{c_i \in C} \text{route}^{\text{loc}}(c_i, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_i}_{\text{cover}}) \right) \bowtie \left(\bigcup_{c_i \in C} \text{route}^{\text{loc}}(c_i, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_i}_{\text{cover}}) \right).$$

3. If $W \subseteq V$ and B is a BGP, then

$$[\![\texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}]\!]^{loc,c}_{\text{cover}} = \text{project}(W, [\![B]\!]^{loc,c}_{\text{cover}}) = \left\{ (\mu_{|_W}, C') \mid (\mu, C') \in [\![B]\!]^{loc,c}_{\text{cover}} \right\}.$$

Definition 32. The *replication-aware distributed evaluation* of a SPARQL query Q over an arbitrary graph cover called cover that assigns triples of an arbitrary RDF graph G to compute nodes C, denoted by $[\![Q]\!]^{loc}_{\text{cover}}$, is defined as

$$[\![Q]\!]^{loc}_{\text{cover}} := \left\{ \mu \,\middle|\, (\mu, C') \in \bigcup_{c \in C} [\![Q]\!]^{loc,c}_{\text{cover}} \right\}.$$

With the help of these definitions, it is possible to prove that the defined replication-aware distributed execution mechanism is semantically correct and complete.

Theorem 2. *The centralized evaluation of query Q produces exact the same results as its replication-aware distributed evaluation, i.e.*

$$[\![Q]\!]^{loc}_{\text{cover}} = [\![Q]\!]_G \ .$$

It can be proven that in the case of graph cover strategies that do not replicate triples the distributed query execution strategy and the replication-aware distributed query execution strategy work identically.

Theorem 3. *In case of a graph cover without triple replication, i.e.,* $\forall t \in G : |\text{cover}(t)| = 1$*, the distributed query execution strategy and the replication-aware distributed query execution strategy evaluate the BGP B identically:*

$$\forall c \in C : \left\{ \mu \,\middle|\, (\mu, C') \in [\![B]\!]^{loc,c}_{cover} \right\} = [\![B]\!]^{c}_{cover} \ .$$

The proofs of Theorems 2 and 3 are shown in Appendix B. A full example of the distributed query execution with triple replication is given in Section A.2.

4.4.3. Limitations

A single graph cover strategy might have a property that can be used to speed up the query processing. For instance, in the case of a hash cover, the compute node storing all triples with subject s can easily be identified by computing $\text{hash}(s) \bmod |C|$ where C is the number of all compute nodes. This knowledge of triple placement can be used to optimize query processing. The resulting optimized query processing strategy might not work for minimal edge-cut covers since they do not have this property. Thus, starting to consider properties of specific graph cover strategies to improve the distributed query processing would lead to several query execution strategies. This would limit the comparability of graph cover strategies that will use different query execution strategies.

In order to have comparable evaluation results of different graph cover strategies, the distributed query execution strategy must not consider any graph cover strategy-specific properties. In [92] and [66] hash-based graph cover strategies and the minimal edge-cut cover strategy with and without n-hop replication were compared. In their systems they decided based on the length of the paths with which the query matches whether a query needs to exchange intermediate results or not. If no intermediate results need to be exchanged, the complete query is executed on all compute nodes. If intermediate results need to be exchanged, the query is first decomposed into subqueries that only match with paths with a length of at most n. The created intermediate results are then combined by executing potentially several MapReduce jobs. In case of triple replication this strategy reduces the number of exchanged intermediate results but does not avoid the creation and processing of duplicate intermediate results caused by triple replication.

The decomposition of queries into subqueries that match with paths that have a length of at most n as done by [92] and [66] could not be realized in Koral, since this decomposition strategy uses a property that is specific

for the n-hop replication. To become more graph cover-independent Koral keeps track on which compute nodes the individual triples and intermediate results are known, instead. With this information, transferring intermediate results to compute nodes that know these results already can be avoided. Similar to [92], the distributed query execution strategy does not prevent duplicate results being produced. This might lead to a poorer performance of graph cover strategies with a huge portion of replicated triples.

For graph cover strategies without triple replication the main difference between the presented approach and the systems [92] and [66] is how the join of intermediate results from different compute nodes is handled. [92] and [66] use MapReduce to join them with the consequence that these joins via MapReduce punish network traffic with an overhead. To avoid this punishment the approach of TriAD [55] was adapted in which the joins of intermediate results are assigned to computed nodes based on the occurrence of resources as subjects in the locally stored graph chunks. This strategy aims to reduce the network traffic for subject-object and subject-subject joins. Nevertheless, the assignment of join responsibilities based on the subject occurrences might lead to a poorer performance of graph cover strategies that assign triples with the same subject to different compute nodes.

CHAPTER 5

Evaluation of Common Graph Cover Strategies

With the evaluation methodology presented in the previous chapter, the impact of different graph cover strategies on the query execution effort can be analyzed. The used experimental setup is explained in Section 5.1. The results are described in Section 5.2.[1] This chapter was mainly taken from [73].

5.1. Experimental Setup

The set of configurations in the benchmark results from the multiplicative combination of (i) the set of different graph cover strategies, (ii) the set of different query-data-set combinations, and (iii) the set of different query execution strategies.

Compared Graph Cover Strategies

During the evaluation, a hash cover, a hierarchical hash cover, a minimal edge-cut cover and a vertical cover are compared. Both hash covers and the vertical cover are reimplemented following the descriptions in [92]. For both hash covers, the hash is computed only on the subject of each triple. For the creation of the minimal edge-cut cover METIS [80] is used as done by other distributed RDF stores like [121], [66], D-SPARQ [109] and WARP [65].

Additionally, the effect of the n-hop replication is examined. [66] and [91] identified a value of $n = 2$ to be a good balance between storage redundancy and gained query performance. Since it is to be expected that the random distribution of a hash cover will lead to a high number of exchanged intermediate results, a 2-hop hash cover is expected to have the greatest benefits from the reduced network traffic.

Data Sets and Queries

For the evaluations of the common graph cover strategies, three real-world data sets and two generated data sets are used. The focus of the performed analysis will be on the real-world data sets, since they avoid effects that may occur due to the generation process of the synthetic data sets. Nevertheless, generated data sets are used to check whether the observations made with the real-world data sets can also be made with the generated data sets. If similar observations can be made for all data sets, then it is an indicator that the made observations are true for most data sets.

As real-world data sets 500M, 1000M and 2000M triples subsets of the real-world billion triple challenge data set from 2014 (BTC2014) [76] are used. These data sets are referred to as BTC500M, BTC1000M and BTC2000M, respectively. The BTC2014 data set has been generated by crawling data from several data

[1] A script to generate the used data sets, the generated queries, the raw and preprocessed measurements of all experiments as well as many additional diagrams can be found at `https://github.com/Institute-Web-Science-and-Technologies/graphCoverStrategyEvaluationData`.

sources of the linked open data cloud. The used subsets contain the first 500 million, one billion and two billion syntactically correct triples.

Additionally, two datasets were generated with the Waterloo SPARQL Diversity Test Suite v0.6 [15]: the WatDiv1000M data set containing 1,099,208,068 triples and the WatDiv100M data set containing 109,786,094 triples. Some characteristics of all used data sets are shown in C.1.

In comparison to evaluations as described in [162] that store 1 billion triples per compute node, the data sets used in the evaluation of this thesis are relatively small. Due to limited computational resources, experiments with 40 compute nodes that are capable to process 1 billion triples each could not be performed. In order to compensate this, smaller data sets but also much smaller compute nodes were used. Whereas [162] used compute nodes with 6 CPU cores and 96 GB RAM, for the experiments in this thesis, the used compute nodes had only 1 CPU core and 2 GB RAM as described below.

Following the strategy explained in Section 4.2, basic graph patterns are generated with SPLODGE varying the query characteristics. In order to measure the effect of an increasing number of joins, the generated queries have one join involving two triple patterns or seven joins involving eight triple patterns. These joins can be subject-subject joins or subject-object joins leading to star-shaped and path-shaped queries, respectively. Since the horizontal containment of graph cover strategies might depend on the join patterns, queries for both join pattern types are generated.

Some graph cover strategies might locate triples from the same data set within one graph chunk. This might lead to a better result for queries requesting triples from only one data source but worse for queries that require triples from different data sources. In order to examine this effect, queries that require triples only from one data source and queries that combine triples from three different data sources are generated.

In order to generate queries that produce different amounts of intermediate results, queries with a selectivity between 0.001% and 0.01% are generated for the 1000M triples subset. Thus, the match operations alone guarantee that there will be between 1 million and 10 million intermediate results. Experiments with a selectivity rate of 0.1% were performed but, since no query optimization strategy is applied, the join operations of these queries produced such an amount of intermediate results that it exceeded the available resources. Summarized, the selected query characteristics are:

Various number of joins: 2 and 8 triple patterns.

Varying selectivity: 0.001% and 0.01% involving between 1 million and 10 million triples.

Varying join patterns: path-shaped (subject-object join) and star-shaped (subject-subject join).

Varying number of data sources: 1 and 3 source data sets.

For each of the WatDiv data sets 20 queries are generated based on the basic testing query templates[2]. These generated queries consists of star-shaped queries (S1-S7), path-shaped queries (L1-L5) and combinations of both shapes (C1-C3 and F1-F5). The generated queries for all the data sets are given in C.2.

Evaluation Setup using the Graph Cover Evaluation Platform (CEP)

Using the extensible evaluation platform for graph cover strategies (CEP)[3], the evaluation is set up as follows. CEP downloads the BTC2014 data set, removes all syntactically incorrect triples and creates the 500M, 1000M and 2000M triples data set. The resulting 1000M data set is used by SPLODGE [49] configured as

[2] https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

[3] https://github.com/Institute-Web-Science-and-Technologies/cep

described above to generate the query set for the benchmark. Thereafter, it downloads the WatDiv generator and generates the WatDiv1000M and WatDiv100M data sets together with the queries.

For each graph cover strategy Koral is initialized, the data set is loaded and the list of configured queries is executed 10 times. Thus, the effect of operating system-dependent caches storing the results of the previously executed query is reduced, because no query is immediately reexecuted after it has finished. In order to prevent the effect of outliers caused by, e.g. garbage collection, from all 10 executions of a query, the best and the worst execution time are ignored and the arithmetic mean is used for *exTime*. CEP collects all measurements during graph loading and query execution and creates tables and corresponding diagrams.

In order to evaluate the scalability of the different graph cover strategies, the BTC1000M data set and 11, 21, and 41 virtual machines (VMs) are used to evaluate graph covers with 10, 20 and 40 graph chunks, respectively. Thereafter, 21 virtual machines are used to evaluate the graph cover strategies with the BTC500M, BTC1000M and BTC2000M triples data sets. Finally, the WatDiv1000M and WatDiv100M data sets are evaluated using 11 virtual machines.

Computer and Software Environment

The graph cover evaluation platform CEP is executed on a VM with 4 cores and 8 GB RAM. Koral is executed on 11, 21 and 41 VMs. The master has 4 cores and 64 GB RAM and the 10 to 40 slaves have 1 core and 2 GB RAM each. Since the CEP and the Koral master VM need to store the complete data set, they have a 1 TB hard disk. The slaves have 300 GB hard disks. The physical computers on which the VMs run are connected via a 1 Gigabit Ethernet network.

The operating system of each VM is a 64 bit Ubuntu 14.04.4 with the Linux kernel 3.13.0-96. The Oracle JDK in version 1.8.0_101 is used to execute CEP in version 0.0.1 and Koral in version 0.0.1. In order to create the minimal edge-cut cover, METIS is used in version 5.1.0.dfsg-2. To generate the WatDiv data sets and the corresponding queries the WatDiv generator in version 0.6 is used.

Summary of Evaluation Setup

Table 5.1 summarizes the setup of the performed evaluations. The hash cover with the 2-hop hash cover, the hierarchical hash cover, the minimal edge-cut cover and the vertical cover were evaluated with the BTC1000M data set distributed among 10 slaves. Additionally, the master and one slave were executed on the master VM to measure the performance of a centralized RDF store that runs on only a single compute node. In order to measure the effect of an increasing number of slaves on the hash cover, the hierarchical cover and the minimal edge-cut cover, the BTC1000M data set was distributed among 10, 20 and 40 slaves. Additionally, the effect of scaling the data set size from 500 million, 1 billion up to 2 billion triples on the hash cover, the hierarchical hash cover and the minimal edge-cut cover was evaluated on 20 slaves. Finally, the WatDiv100M and WatDiv1000M data sets were used to check whether the made observations could also be made for the generated data sets.

5.2. Results

When investigating the effect of the graph cover strategy on the query execution effort, the possible configurations of independent variables (configuration settings) and dependent variables (evaluation measures) is staggering. In order to shrink the number of configurations first the overall query performance of all graph cover strategies with the query performance of a centralized setting in which the queries are executed on a single compute node is compared in Section 5.2.1. For graph cover strategies that can process the queries

Data set	Number of slaves		
	10	20	40
BTC500M	-	hash hierarchical edge-cut	-
BTC1000M	hash hierarchical edge-cut vertical 2-hop hash	hash hierarchical edge-cut	hash hierarchical edge-cut
BTC2000M	-	hash hierarchical edge-cut	-

(a) Summary of the evaluation setup with the BTC2014 data sets.

Data set	Number of slaves 10
WatDiv100M	hash hierarchical edge-cut
WatDiv1000M	hash hierarchical

(b) Summary of the evaluation setup with the WatDiv data sets.

Table 5.1.: Summary of the evaluation setup.

faster than in the centralized setting, first the analysis of measurements that do not depend on queries will be presented in Section 5.2.2. These measurements comprise the loading time of the graph covers on the one hand side. On the other side the size and the structure of the resulting graph chunks are analysed since they influence whether a graph cover will have a good overall performance. The overall query performance under a larger variation of independent variables is depicted in Section 5.2.3. The observed performance is caused by the horizontal containment and vertical parallelization of the different graph cover strategies. Therefore, in Section 5.2.4 indicators for horizontal containment and vertical parallelization based on few selected independent variables will be analyzed. Additionally, the change of the observed results will be investigated, when scaling the number of virtual machines with a fixed data set size as well as scaling the data set size with a fixed number of virtual machines. The observed effects are described as separate paragraphs in each section.

In order to improve the comprehensibility of the diagrams, the queries are named based on their characteristics. For instance, the query `so #tp=8 #ds=3 sel=0.01` describes a query containing 8 subject-object joined triple patterns which match triples from 3 data sources and the sum of the selectivities of all triple patterns is 0.01. Table 5.2 shows the number of results returned by the queries for the different BTC2014 data sets. For all queries the number of results increased while scaling up the data set size. In the following the queries that were aborted after one million results are called the aborted queries. All the other queries are called finished queries. For both WatDiv data sets, none of the 20 queries were aborted.

5.2.1. Comparison with Centralized Execution

One reason to use a distributed RDF store is that queries might be executed faster than on a single compute node. The results described in this section can be summarized as:

- Queries can be executed on the hash cover, hierarchical hash cover and minimal edge-cut cover faster than on a single compute node.
- The vertical cover has a slower query execution time since the matches for triple patterns can only be found on a few compute nodes.
- The 2-hop hash cover tends to have a slower query execution time since the high amount of replicated triples cause many intermediate results being computed several times.

Query	#Results for BTC500M	#Results for BTC1000M	#Results for BTC2000M
q_1: `so #tp=2 #ds=1 sel=0.001`	855	1k	3k
q_2: `so #tp=2 #ds=1 sel=0.01`	86k	127k	173k
q_3: `so #tp=8 #ds=1 sel=0.001`	6k	61k	597k
q_4: `so #tp=8 #ds=1 sel=0.01`	241	1k	144k
q_5: `so #tp=8 #ds=3 sel=0.001`	1,000k	1,000k	1,000k
q_6: `so #tp=8 #ds=3 sel=0.01`	328k	754k	1,000k
q_7: `ss #tp=2 #ds=1 sel=0.001`	1,000k	1,000k	1,000k
q_8: `ss #tp=2 #ds=1 sel=0.01`	65k	104k	148k
q_9: `ss #tp=8 #ds=1 sel=0.001`	1,000k	1,000k	1,000k
q_{10}: `ss #tp=8 #ds=1 sel=0.01`	1,000k	1,000k	1,000k
q_{11}: `ss #tp=8 #ds=3 sel=0.001`	1,000k	1,000k	1,000k
q_{12}: `ss #tp=8 #ds=3 sel=0.01`	4	12	60

Table 5.2.: Number of query results for the BTC2014 data sets.

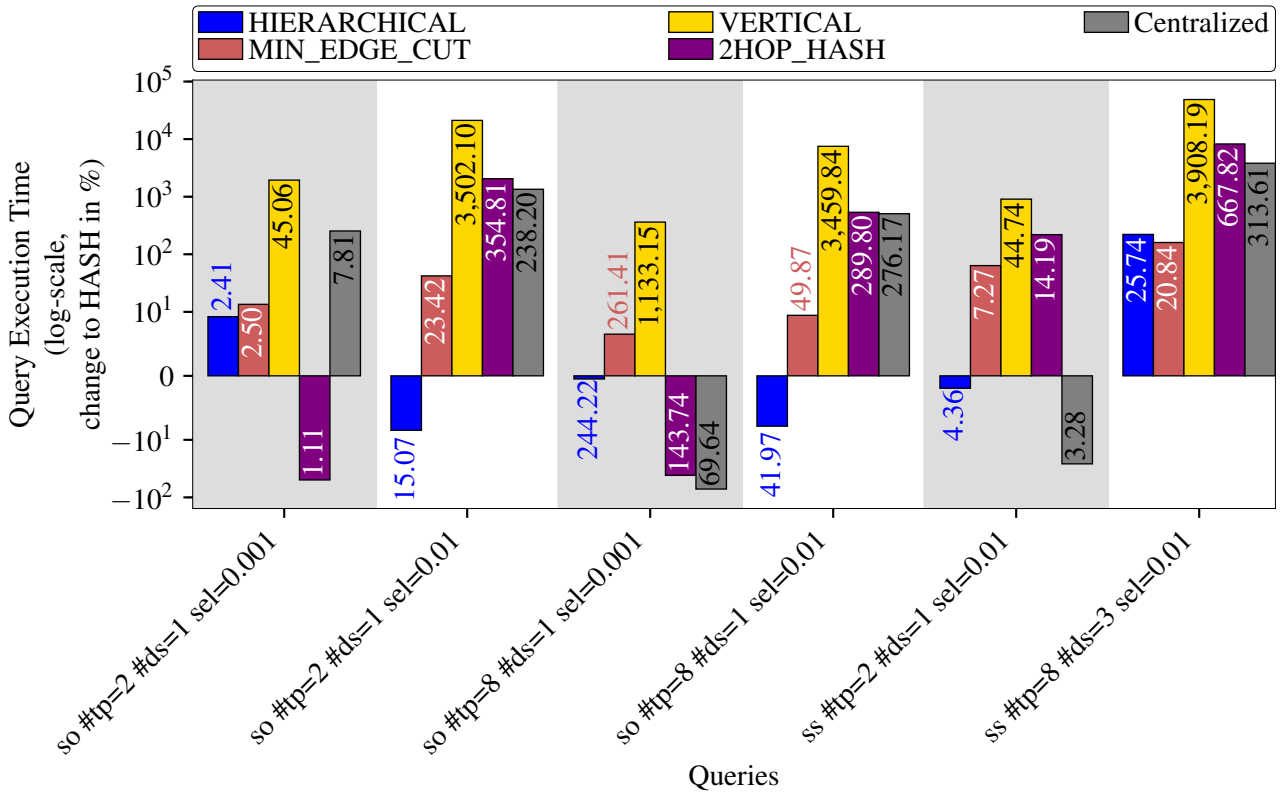


Figure 5.1.: Change of the *exTime*s of all finished queries relative to the hash cover using bushy query execution with 10 slaves using the BTC1000M data set. The numbers within the bars are the absolute query execution times in seconds.

To check whether the examined graph cover strategies could reduce the query execution time, one Koral master and one slave were executed on the master VM with 64 GB main memory. It loaded the BTC1000M data set and executed the queries on it. Then, the measured execution time are compared with the times measured for the different graph covers on 10 slaves running on VMs with 2 GB of main memory each. The resulting query execution times of the finished queries are shown in Figure 5.1. Since the execution times vary strongly, the change is shown in comparison to the hash cover.

Comparison with Hash, Hierarchical and Minimal Edge-cut Cover

For two queries the centralized setting executed them more than 10% faster than all the other graph cover strategies. In both cases the total computational effort of the queries is 8 till 31 times higher than for the other queries. This huge amount of computational effort indicates that the 2 GB of main memory are too small to cache all intermediate results in main memory and thus have to be stored on disk. In contrast to this, the single master node has enough main memory to cache the intermediate results without accessing the disk. For the other queries, the centralized query execution strategy needs more than 3 times longer to execute the queries than the hash cover, the hierarchical hash cover or the minimal edge-cut cover.

Comparison with Vertical Cover

When focussing on the vertical cover, the query execution time is more than 5 times slower than in the centralized case. The cause for this poor performance is that triples were assigned to compute nodes based on their properties. When executing queries each triple pattern that has a resource at the property position it will only find matches in at most one graph chunk. Figure 5.2 shows that for the queries with two triple patterns matches on only two slaves where found and for queries with 8 triple patterns matches on only 4 till 5 slaves were found. For all other evaluated graph cover strategies each query found matches on all slaves and, e.g., for the hash cover the chunk with the fewest matches has only less than 2% fewer matches than the chunk with the most matches for almost all queries. Since the vertical cover has matches only on a few slaves its workload

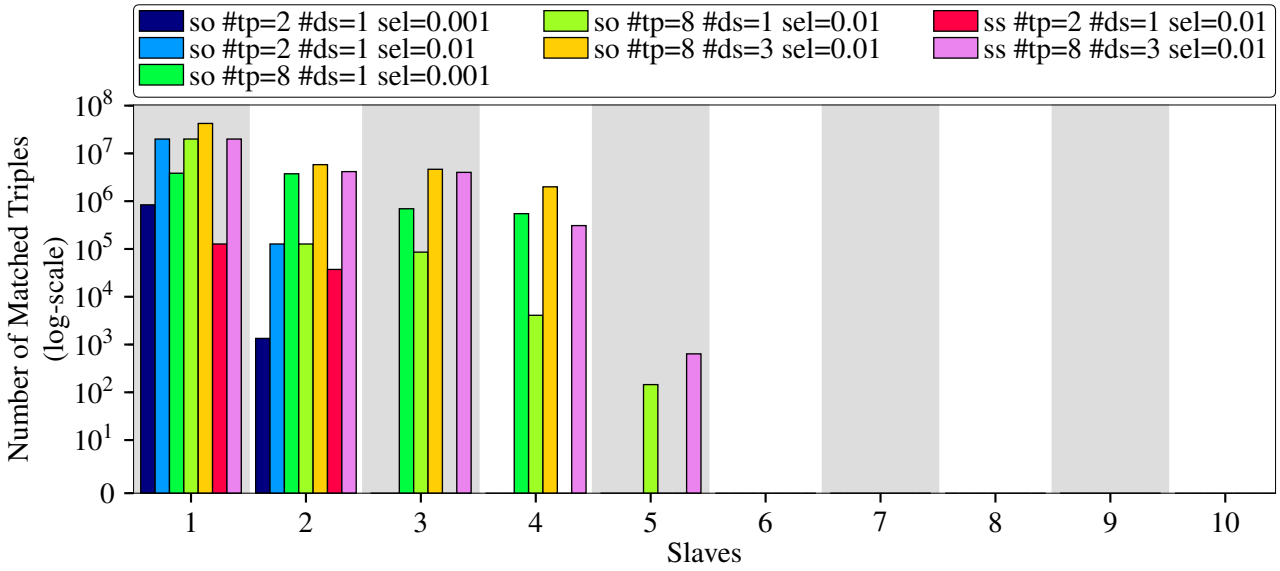


Figure 5.2.: Number of triples that were matched on the individual graph chunks of the vertical cover.

imbalance is between 0.63 and 1 whereas the highest workload imbalance of the other graph cover strategies is 0.43. Also the number of transferred intermediate results is more than 70% higher than for the other graph cover strategies.

Comparison with 2-Hop Hash Cover

The 2-hop hash cover leads to at least 5% slower query execution times than in the centralized setting for most queries. In comparison to the hash cover, the 2-hop hash cover reduces the number of transferred packets by 90% till 100% and also the workload imbalance is reduced by more than 50% for almost all queries. The reason for the poor performance of the 2-hop hash cover is that the storage redundancy r is 4.19. This high number of replicated triples leads to a total computational effort that is 4 till 10 times higher than for the graph cover strategies without replication. Thus, the high number of duplicate computations lead to the slow query execution times of the 2-hop hash cover. Only for two queries the 2-hop hash cover was faster than the hash cover. In these cases the computation effort was only 4 times higher while the workload imbalance was below 0.01 and the number of transferred packages was reduced by 98% till 100% in comparison to the hash cover.

Due to the poor performance of the vertical cover and the 2-hop hash cover the focus of the remaining chapter will rely on the hash cover, the hierarchical hash cover and the minimal edge-cut cover.

5.2.2. Query Independent Measurements

In this section the required time to load the graph covers and analyse the size and structure of the created graph covers is examined. The main findings are:

- The minimal edge-cut cover takes the longest time to be created and produces the most imbalanced graph chunks.
- The graph chunks of the minimal edge-cut cover have the largest diameters.

Load Time

Even if the loading time has not a direct influence on the query performance, it is helpful to know, whether a graph cover strategy can be computed in a reasonable amount of time for even large data sets. The loading time L itself consists of several different steps like the dictionary encoding and the statistics collection. The most interesting step is the graph cover creation, on which this section will focus here.

As shown in Figure 5.3, the hash cover is created the fasted. It takes around one hour to iterate the data set and assign triples to the corresponding compute nodes. The hierarchical hash cover requires between seven and eight hours. The longer cover creation time is caused by iterating the complete data set twice and the additional computation to find the optimal IRI hierarchy level for creating the graph cover. With 30 to 32 hours, the minimal edge-cut cover creation takes the longest time.

Scalability. The effect of an increasing number of graph chunks on the creation time of the minimal edge-cut cover and the hierarchical hash cover is negligible since the difference is only less than 10%. For the hash cover the creation time was almost stable when scaling from 10 to 20 chunks. But when scaling to 40 chunks the creation time increased by 35 minutes. This increase may be caused by a much higher number of switches between the 40 chunk files when assigning the triples to the chunks. When the size of the data set was scaled, the creation time increased to the same extent as the data set size grew for both hash-based covers. Only the creation time of the minimal edge-cut cover became three times higher when doubling the data set size.

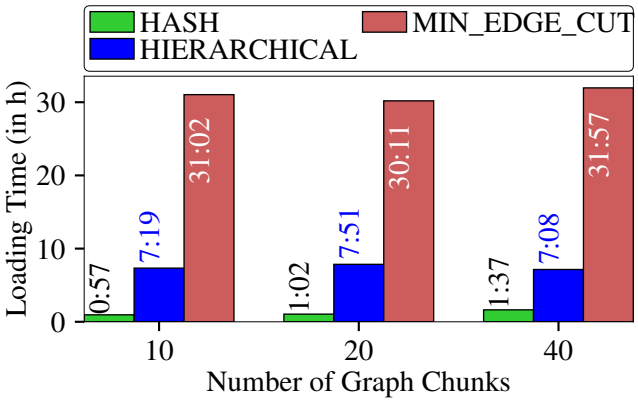


Figure 5.3.: Creation times of different graph covers for different slave numbers and the BTC1000M data set.

WatDiv Data Sets. For the WatDiv1000M data set, the hash cover and the hierarchical hash cover were created in almost the same time as for the BTC1000M data set. Since METIS tried to allocate more than one terabyte of main memory, it could not be crated. For the WatDiv100M data set, the hash, hierarchical and minimal edge-cut covers could be created in 16 minutes, 36 minutes and 7:44 hours, respectively. Thus, the hierarchical hash cover could reduce its creation time the most.

In the current implementation the focus is on the graph cover creation step. The loading steps that are the same for all graph cover strategies have not been optimized since they do not give any insights which graph cover strategy is created faster. Therefore, the complete loading procedure consumes much more time. The central bottleneck in the current implementation is the statistics database which needs more than a week for collecting the statistics data. It requires optimization for practical use, but not for the purpose of this evaluation.

Storage Imbalance

If a graph cover strategy produces some graph chunks that are much larger than the others, then the compute nodes storing these large chunks may have a higher query workload than the compute nodes storing the smaller chunks. In order to visualize the storage imbalance, Figure 5.4 shows, how many triples are contained in the different graph chunks sorted in descending order for 20 graph chunks[4]. As shown in Table 5.3 the storage imbalance of both hash-based covers is similar. When scaling up from 10 to 40 graph chunks, the storage imbalance almost doubles but still have a very low storage imbalance. Both graph cover strategies distribute triples based on subject hashes leading to a nearly optimal storage balance. Thus, the number of triples per chunk decreases with an increasing number of graph chunks. For the WatDiv1000M data set, no storage imbalance is given for the minimal edge-cut cover since it could not be created.

The storage imbalance value of the minimal edge-cut cover is at least 10 times higher than for the other graph covers. As shown in Figure 5.4 the minimal edge-cut cover has one graph chunk that contains more than twice the number of triples than each the other chunks has. Even when the number of chunks is increased from 10 to 40 the number of its triples is only reduced by roughly 36%. Furthermore, there exists some graph chunks that contain much fewer triples than the average graph chunk size. If the number of graph chunks is increased, the number of these small chunks also increases leading to a higher workload imbalance value.

[4]The figures for 10 and 40 graph chunks and for the BTC500M and BTC2000M data sets as well as both WatDiv data sets do not provide additional insights and are therefore omitted.

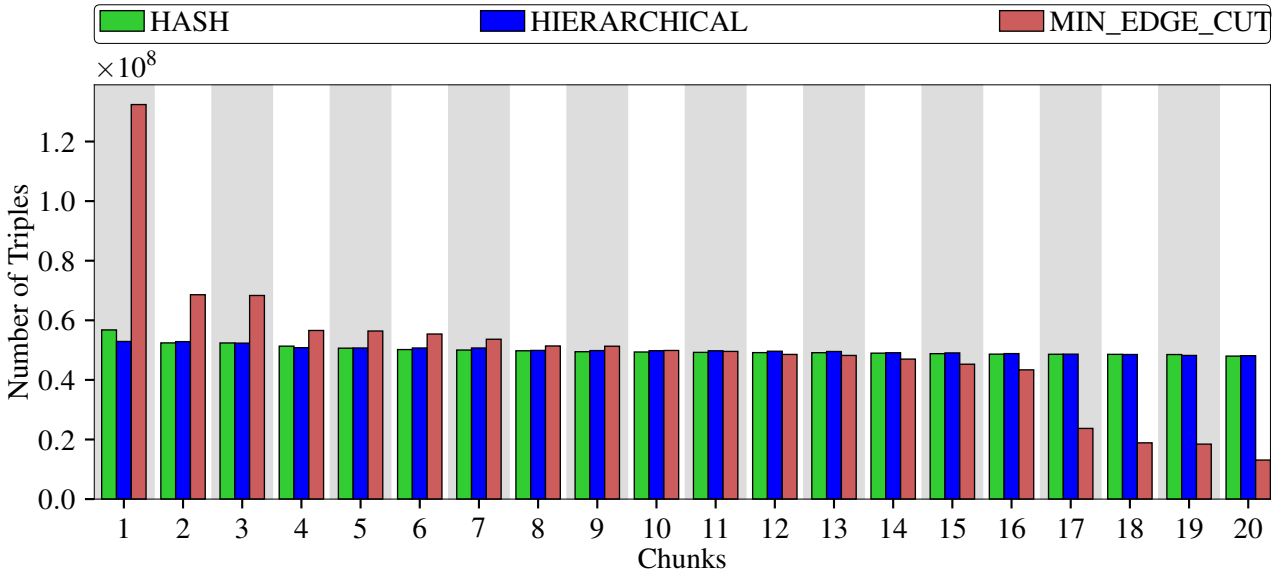


Figure 5.4.: Number of triples contained in each of the 20 graph chunks.

Data set	BTC2014			WatDiv100M	WatDiv1000M
# chunks	10	20	40	10	10
Hash cover	0.0167	0.0196	0.0275	0.0016	0.0006
Hierarchical hash cover	0.0119	0.0160	0.0240	0.0010	0.0006
Minimal edge-cut cover	0.1787	0.2418	0.2441	0.4355	-

Table 5.3.: The storage imbalance b of the different graph covers at different number of graph chunks.

Cause for High Storage Imbalance of Minimal Edge-cut Cover. When investigating the cause for the high storage imbalance of the minimal edge-cut cover, the output of METIS stated that the number of vertices per graph chunk only vary by at most 3% from the average number of vertices per chunk (i.e., $\frac{|V|}{|C|}$). Since in the evaluation the size of a graph chunk is determined by the number of triples, the imbalanced chunk sizes are caused by different number of incident edges that are assigned to the different graph chunks.

Structure of Graph Chunks. In order to measure the number of cut edges, let a cut edge be defined as a triple whose subject and object are owned by different graph chunks. A resource r is owned by a graph chunk, if all triples with r as subject are assigned to this graph chunk. When measuring the number of cut edges for the different graph cover strategies, it could be observed that all graph cover strategies cut between 42% and 54% of all triples. Thereby, the hierarchical hash cover cuts at most 0.5% fewer triples than the hash cover. The minimal edge-cut cover cuts 4% fewer triples than the hash-based covers.[5]

In order to understand, why nearly every second triple is a cut edge, 1k to 30k triples subsets of the BTC2014 data set were plotted. Similar to [94], the BTC2014 graph consists of one huge, densely-connected core and

[5]During the creation of the minimal edge-cut cover, the rdfs:type triples were removed and added to the chunk which owned their subject later on. Since only 4% of the cut edges had a rdfs:type label, the high number of cut edges for the minimal edge-cut cover was not caused by the used procedure.

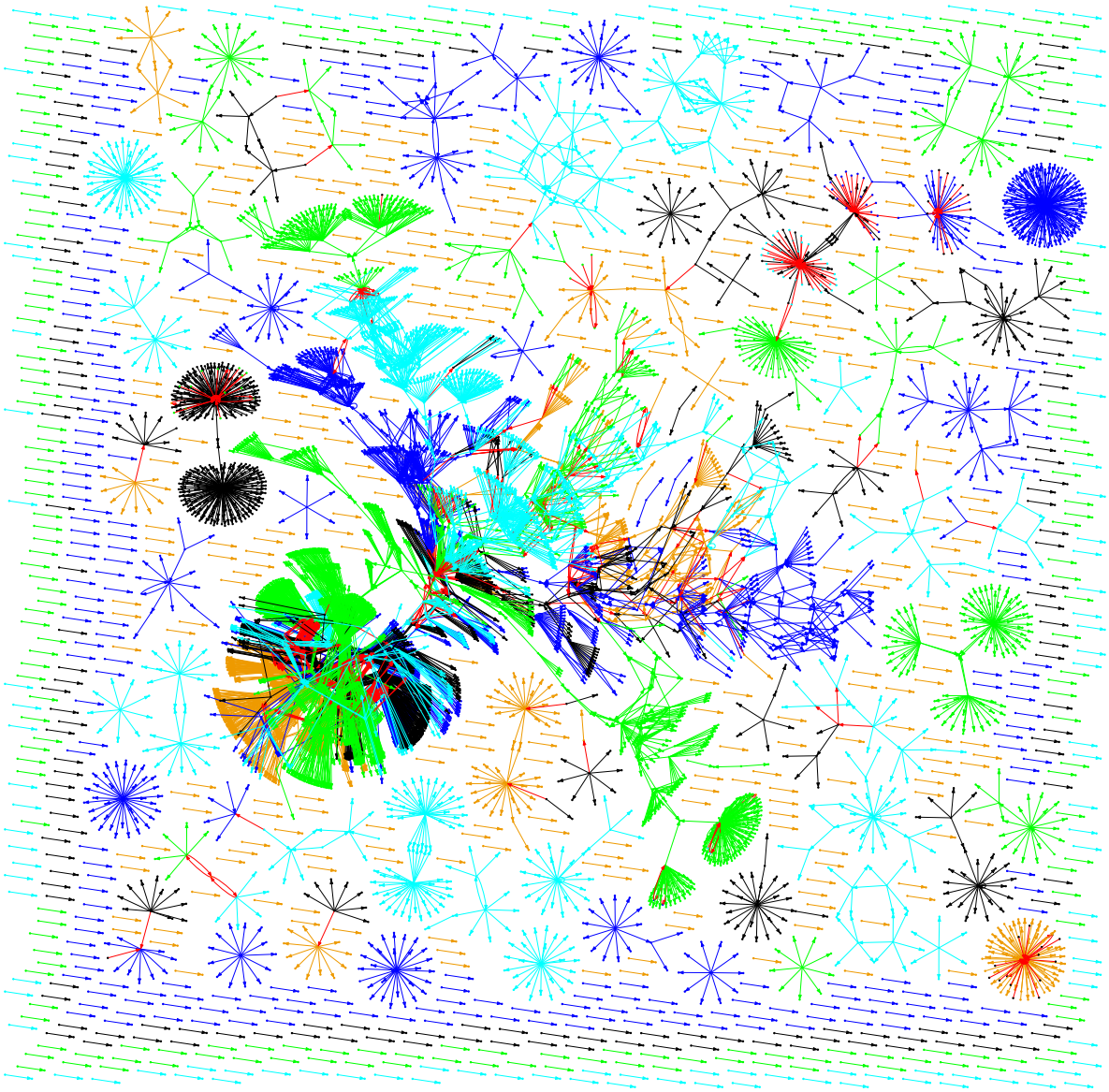


Figure 5.5.: A plot of a minimal edge-cut cover of a 10k triples subset consisting of five chunks. Each chunk is drawn by a different colour. Cut edges are indicated by red arrows.

several small sets of vertices which are densely connected with each other, but are only loosely or not at all connected to the huge core. Additionally, around 20% of all triples used a subject and object that were not used by any other triple. When coloring the triples by the chunks to which they were assigned to, it became visible that each of the examined graph cover strategies cuts the huge densely connected core, leading to a high number of cut edges.

Coloring the different chunks in the plots lead to another observation that might affect the query performance. The triple assignment in both hash-based covers is more or less random, leading to graph chunks with a low diameter (i.e., the longest shortest path within a graph chunk). In contrast to this, the diameters of the

minimal edge-cut chunks are higher, as recognizable at, e.g., the green chunk in Figure 5.5. Thus, it is more likely that path-shaped queries will require less data transfer. Furthermore, most triples of the huge core are contained by the green, orange and black chunks whereas the cyan and the blue chunks mainly contain triples not contained in the core. Since especially the path-shaped queries will have only a few matches outside the core, the chunks containing portions of the core will have a higher workload than the other chunks.

Scalability. The scalability experiments showed that the effects of scaling the number of graph chunks and the data set size on the storage imbalance is negligible.

5.2.3. Measuring Overall Query Performance under Varying Independent Variables

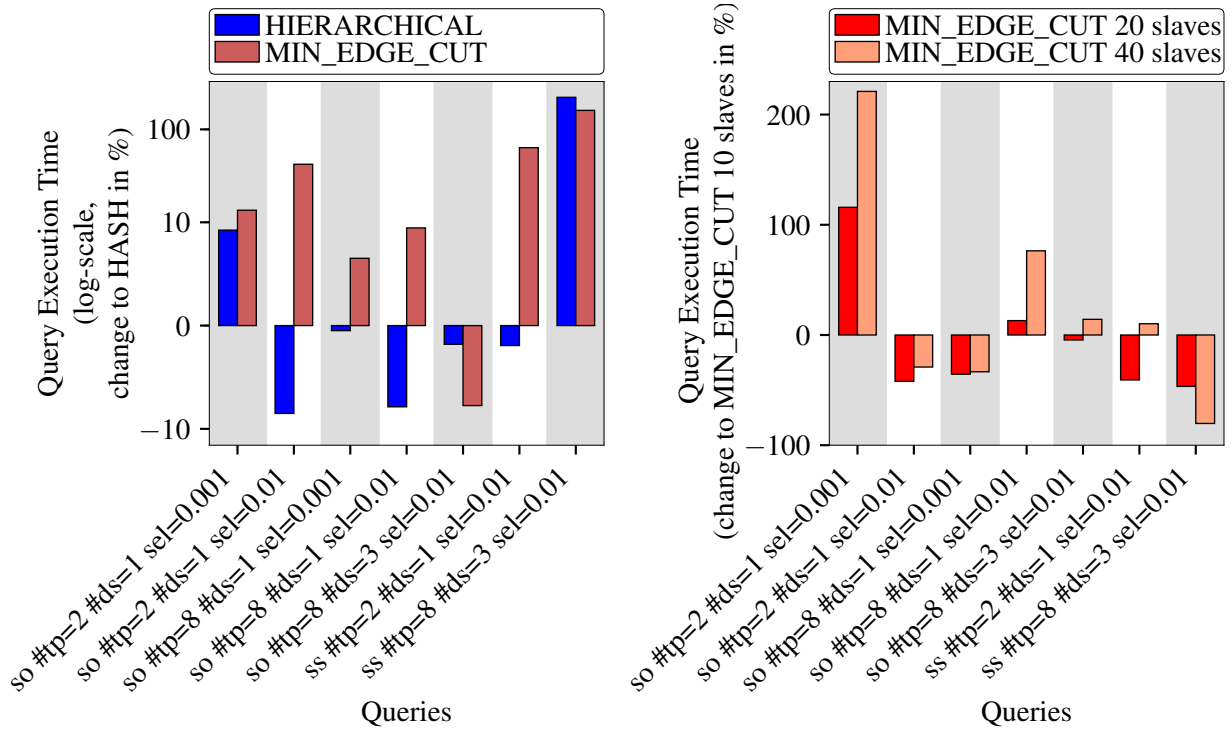
The investigation of the overall query performance of the different graph cover strategies described in this section resulted in the following core findings:

- The hash-based covers have a better overall performance than the minimal edge-cut cover.
- Both hash-based covers perform nearly the same. Only for small data sets, the hierarchical hash cover may have a slightly better overall performance than the hash cover.
- Star-shaped queries are executed faster than path-shaped queries. Path-shaped queries with a few triple patterns are executed faster than path-shaped queries with many triple patterns.

Comparison of Query Execution Strategies. For the study of the effect of the graph cover strategy on the query performance, a graph cover-independent query optimizer is simulated by using three different query execution strategies for each query (see Section 4.3). When investigating the effect of the different strategies, the bushy query execution strategy produces the least query execution times in 75 of 180 cases and the longest query execution times in only 16 cases. This better performance of the bushy query execution strategy is independent of the used graph cover. Since similarly to [149], the bushy query execution strategy is faster than the other strategies in the performed evaluation, this chapter focuses on this strategy in the following.

Query Performance for 10 Slaves. In order to examine the overall query performance, the queries that were finished completely are investigated. The corresponding query execution times are given in C.3. Since the execution times of the different queries vary so strongly that even with a log-scaled y-axis the differences between the different graph cover strategies would not be visible for some queries, Figure 5.6a shows the differences of the hierarchical hash cover and the minimal edge-cut cover relative to the runtime of the hash cover for each query with 10 slaves. For 5 of 7 queries the minimal edge-cut cover produces the longest query execution times. This is caused by the imbalanced workload of the minimal edge-cut cover (see Figure 5.11). In case of query `so #tp=8 #ds=3 sel=0.01` the minimal edge-cut cover is the fastest. In this case the difference of the workload imbalance between the minimal edge-cut cover and both hash-based covers is only less than 0.1 whereas the minimal edge-cut cover requires more than 40% fewer packets to be transferred (see Figure 5.10a). When comparing both hash-based queries, the hierarchical hash cover is slightly (i.e., $< 10\%$) faster for 5 of 7 queries. In the cases of `so #tp=2 #ds=1 sel=0.01` and `so #tp=8 #ds=1 sel=0.001` it is caused by a lower workload balance. In the other three cases the lower number of transferred packets explains the faster query execution. In case of query `so #tp=2 #ds=1 sel=0.001` the hierarchical hash cover is slower even if the number of transferred packets is equal to the hash cover and the workload is better balanced. Since the difference is only 204 msec, the delay might be caused by other effects not part of the evaluation like the usage of the network by other services.

Query Performance for the WatDiv Data Sets. To check whether the query performances observed for the BTC1000M data set is caused by some data-set-specific characteristics, 20 queries where executed on the



(a) Change of the *exTime*s relative to the hash cover using 10 slaves.

(b) Change of the *exTime*s relative to 10 slaves for the minimal edge-cut cover.

Figure 5.6.: Change of the *exTime*s of all finished queries using bushy query execution and the BTC1000M data set.

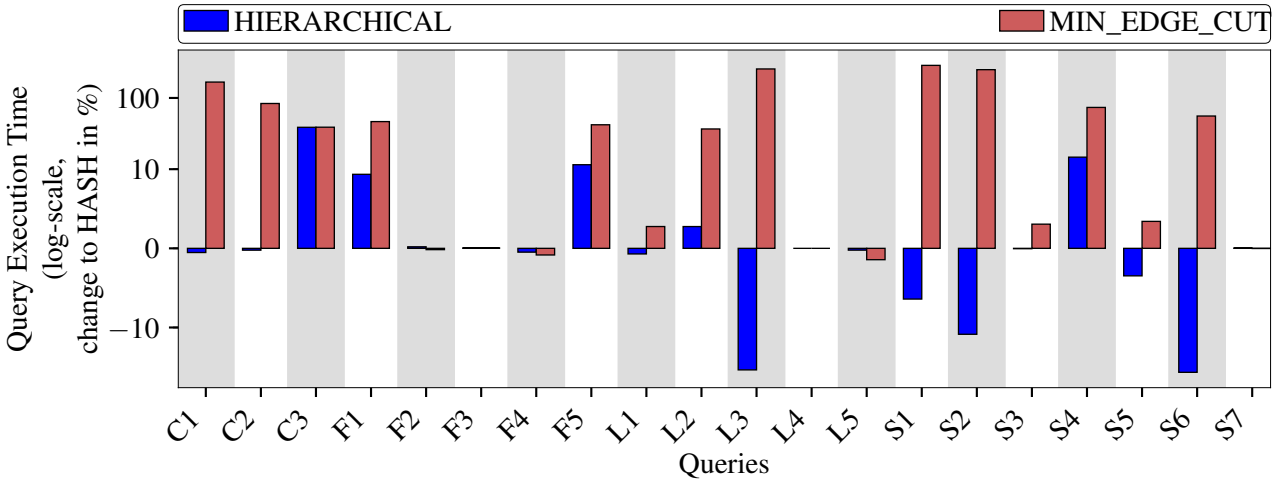


Figure 5.7.: Change of the *exTime*s relative to the hash cover using bushy query execution for the WatDiv100M data set.

WatDiv1000M and WatDiv100M data sets. For the WatDiv1000M data set only the query execution times of both hash-based covers could be compared. For half of all queries the differences of both query execution strategies are almost the same (i.e., within a range of 3% difference). For the remaining queries, 7 queries were executed faster with the hash cover and 3 queries with the hierarchical hash cover. So none of both strategies is faster in general.

For the WatDiv100M data set, the queries could be executed also on the minimal edge-cut cover. As shown in Figure 5.7, 13 out of 20 queries are executed 3%-287% slower with the minimal edge-cut cover while the remaining queries were executed almost as fast as for the hash cover. With none of both hash-based graph covers the queries are executed faster in general.

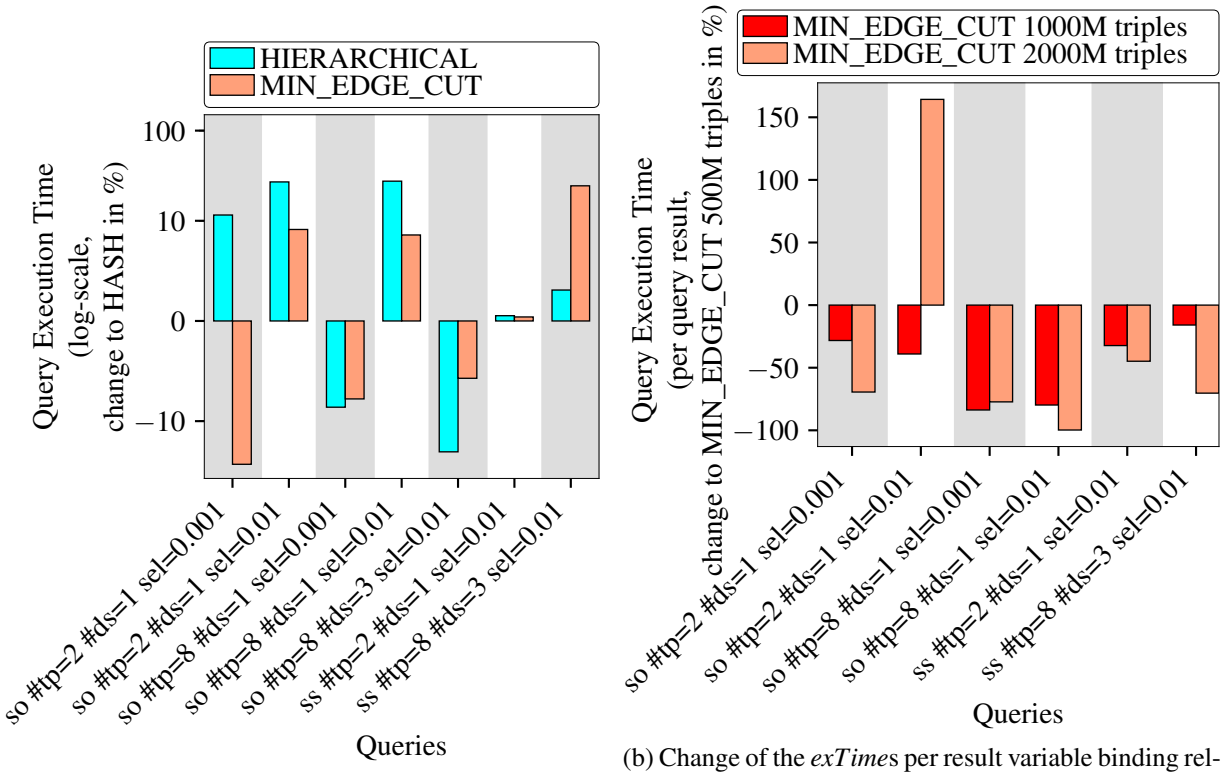
Scaling Number of Slaves

When scaling up the number of slaves, the effect on the query execution times is similar for the different graph covers. Therefore, the effect of scaling up the number of slaves is discussed here only for the minimal edge-cut cover. Figure 5.6b shows the change of the query execution times is given relative to the execution times for 10 slaves, so that the effect of scaling up the number of slaves is better visible. The assumption that the execution time will be reduced by roughly 50% when scaling up to 20 slaves and by roughly 75% when scaling up to 40 slaves could only be observed for query `ss #tp=8 #ds=3 sel=0.01`. In this case the workload imbalance is nearly constant and no network traffic occurs. For most other queries, the query execution time increases. For some queries, the execution time increases when already scaling to 20 slaves. For other queries, the execution time decreases when scaling to 20 slaves but increases when scaling to 40 slaves. This observation is independent of the graph cover strategy. This increasing query execution time is mainly caused by the hugely increasing number of transferred packets between the slaves. Therefore, the network latency seems to be a factor limiting the scalability in the experimental setting, as already identified in [145], chapter 24.3 for gigabit networks in general. The performed experiments show that the speed-up when scaling with the number of slaves is limited, as described by rules like the Universal Scalability Law [53] which says that in distributed settings the performance gain by adding further compute nodes to the system is limited by, e.g., an increasing number of communications.

Query Performance for 40 Slaves. All graph cover strategies are affected by the increased query execution time to a different extent, when scaling up the number of slaves. This leads to the query execution times for 40 slaves shown in Figure 5.8a. Now, the differences between the different graph cover strategies became smaller. The maximal difference has decreased from 220% to 52%. The minimal edge-cut cover is the slowest cover strategy for only one query. As described in Section 5.2.4, this is caused by the smaller differences in the workload imbalance as well as in the number of transferred packets between the minimal edge-cut cover and both hash-based covers. Since the number of transferred packets has increased faster for the hierarchical hash cover than for the plain hash cover, the latter seems to be faster for more queries than the other graph cover strategies.

Scaling Data Set Size

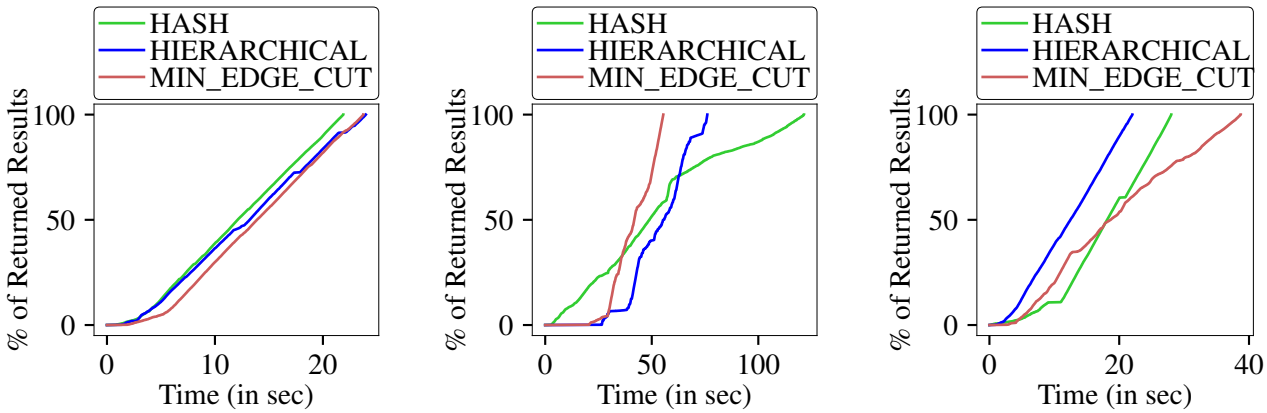
When scaling up the data set size, the query execution time increases for all graph cover strategies since all queries produce more results. In order to deal with the higher number of query results, *exTime* was divided by the number of query results leading to the execution time per result variable binding shown in Figure 5.8b for the minimal edge-cut cover. For almost all queries the execution time per result variable binding decreases when the data set size increases. As described in Section 5.2.4 this speed up is caused by a better balanced query workload that can even compensate the increased number of transferred packets. A special case is



(a) Change of the *exTime*s relative to the hash cover using 40 slaves and the BTC1000M data set.

(b) Change of the *exTime*s per result variable binding relative to the BTC500M data set for the minimal edge-cut cover.

Figure 5.8.: Change of the *exTime*s of all finished queries using bushy query execution.



(a) `ss #tp=8 #ds=1 sel=0.01.` (b) `so #tp=8 #ds=3 sel=0.001.` (c) `ss #tp=8 #ds=3 sel=0.001.`

Figure 5.9.: χ for some queries at scale 10 for the 1000M triples data set.

query `so #tp=8 #ds=1 sel=0.01` for which the highest speed up of 99% was observed. This high speed up cannot be explained by the workload imbalance and the packet transfer since both increase. Instead, the total computational effort per result variable binding dropped by more than 99%. Thus, there were much less intermediate results without join partners produced. Due to this, this query could produce 60,000% more results requiring only 42% more time. The only query which required more time per result variable binding was `so #tp=2 #ds=1 sel=0.01`. In this case the increased packet transfer could not be compensated by the decreased workload imbalance. These observations are independent of the graph cover strategy.

Comparison of Graph Cover Strategies at the Different Data Set Sizes. Since the query execution time speed ups affect the different graph cover strategies to a different extent, the different strategies are compared for every data set size. For the BTC500M data set the minimal edge-cut cover required the longest execution times for 4 out of 7 finished queries, whereas the hierarchical hash cover was the fastest for 5 out of 7 finished queries. When observing the query execution times for the BTC2000M data set, the minimal edge-cut cover is still the slowest for 4 out of 6 finished queries whereas now, the hash cover is the fastest for 4 out of 6 finished queries. This indicates that the hierarchical hash cover seems to be better for smaller graph chunks whereas hash produces better results for larger graph chunks. This conclusion is strengthened by the observation that the hierarchical hash cover has a slightly (i.e. <10%) reduced number of transferred packets and improved workload balance than the hash cover for the BTC500M data set whereas the opposite observation can be found for the BTC2000M data set.

Results Over Time

When investigating how fast the different graph cover strategies produce their results over time, most queries have result curve functions χ like the ones shown in Figure 5.9a for query `ss #tp=8 #ds=1 sel=0.01`. It takes some time until the first result is produced but thereafter the results are arriving continuously. For queries with only one join (not shown in Figure 5.9), the time until the first result is returned is shorter. For most of these queries, the hash cover produces the query results faster and the minimal edge-cut cover slower than the other graph cover strategies. Figure 5.9b shows the χ for query `so #tp=8 #ds=3 sel=0.001`. In this case the hash cover produces the initial results faster but thereafter the results are returned more slowly than for the other graph cover strategies. This slower return rate of the results is caused by a higher number of transferred packets and a higher total computation effort that is more imbalanced among all slaves. The hierarchical hash cover is slower than the minimal edge-cut cover, since it needs to exchange more packets between the slaves. Since no data transfer exists for queries `ss #tp=8 #ds=3 sel=0.0001` and `ss #tp=2 #ds=1 sel=0.01`, the different result return speeds of the different graph cover strategies (see Figure 5.9c for the first of both queries) are caused by the workload imbalance of the three strategies. Initially, the minimal edge-cut cover is faster than the hash cover. This may be caused by a more balanced workload in the beginning of the query execution.

Susceptibility to Query Size and Shape

The measurements indicate that queries with only two triple patterns are executed faster than queries with 8 triple patterns in most cases. This effect affects the hash-based graph covers more than the minimal edge-cut cover. This may be caused by the larger chunk diameters for the minimal edge-cut cover (see Section 5.2.2). When focusing on the query shape, star-shaped queries tend to be faster than path-shaped queries. The explanation is that in the evaluation their results are produced without data transfer as described in the following section. Since star-shaped queries have no data transfer their horizontal containment is optimal for all graph

cover strategies. This leads to faster execution times at all evaluated scale levels in the term of slave numbers for this type of queries.

Susceptibility to Number of Sources

Based on the evaluation the number of data sources does not seem to have an effect on the execution time. The only observation that can be made is that the hierarchical hash cover is faster than the hash cover for most queries using data from several data sources.

5.2.4. Measuring Dependent Variables

In order to find the reasons for the findings of the previous section, the indicators for horizontal containment and vertical parallelization are analyzed, now. The core findings are:

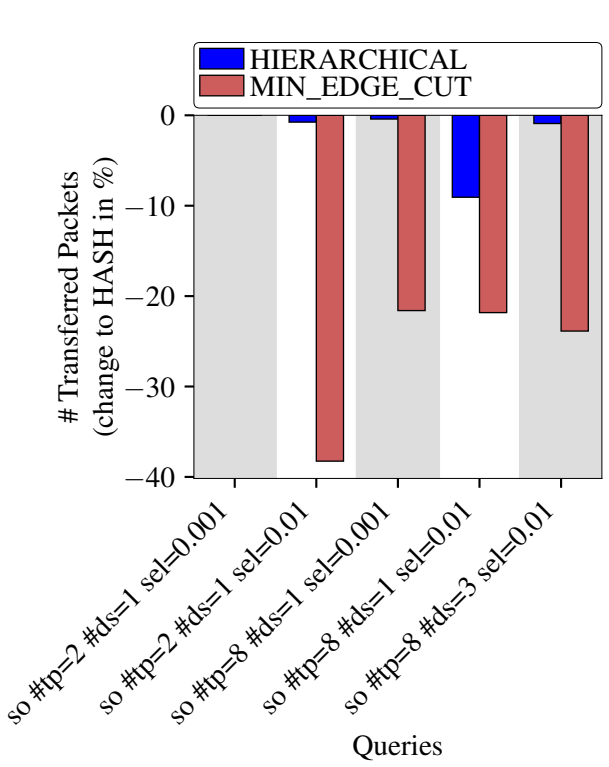
- The minimal edge-cut cover has the best horizontal containment but the highest workload imbalance.
- Scaling up the number of slaves or the data set sizes reduces the horizontal containment for all graph cover strategies.
- The query workload becomes more imbalanced, when the number of slaves is scaled up but it becomes more balanced, when the data set size is increased.

Horizontal Containment

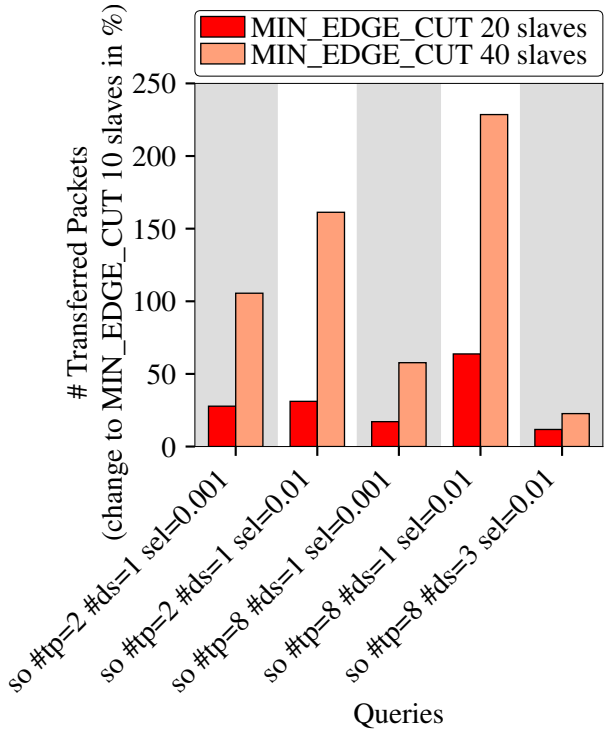
Horizontal Containment of Star-shaped Queries. One factor influencing the overall query performance is the horizontal containment. A first observation is that the examined graph cover strategies assign triples with the same subject to the same chunk. Therefore, all triples required to produce one result of a star-shaped query are located in the same graph chunk. Since the query execution strategy performs the required joins on the slave storing the original triples, no data transfer or packet transport could be observed. Thus, all graph cover strategies result in a perfect horizontal containment for star-shaped queries.

Horizontal Containment of Path-shaped Queries. When investigating the path-shaped queries, the data transfer T and the number of transferred packets P increase for all graph cover strategies, if the number of triple patterns included in the path-shaped query increases. Thus, the likelihood to leave a graph chunk during query processing increases for all graph cover strategies when the length of the queried path increases. In order to examine the horizontal containment of the different graph cover strategies, Figure 5.10a shows how the number of transferred packets changes relative to the hash cover using 10 slaves for the different graph cover strategies. The minimal edge-cut cover requires 20%-43% fewer packets to be transferred than the hash cover. As described in Section 5.2.2, this reduction is not caused by the fewer cut edges. Instead, it is caused by the higher number of connections within one graph chunk leading to higher graph chunk diameters. Only for query `so #tp=2 #ds=1 sel=0.001` the number of transferred packets is almost identical. The hierarchical hash cover reduces the number of transferred packets by only less than 10% for all queries. Thus, the minimal edge-cut cover has the best horizontal containment whereas the horizontal containment of the hierarchical hash cover is only slightly better than the one of the hash cover. Similar observations are made when investigating the data transfer.

Effect of Other Query Characteristics on Horizontal Containment. Since queries with different characteristics were used, it has been investigated which of theses characteristics lead to an increased data transfer and number of transferred packets. The findings are that long path-shaped queries have the highest data transfer



(a) Comparison of the different graph covers at 10 slaves.



(b) Comparison of the different number of slaves for the minimal edge-cut cover.

Figure 5.10.: The relative change in the number of transferred packets P of the bushy query execution.

and star-shaped queries the smallest. The overall query selectivity cannot be used to estimate the data transfer. In the performed evaluation, the impact of the number of data sources cannot be separated from the influence of the number of results, since both queries with `#ds=3` produce also more results than the other queries.

Horizontal Containment for the WatDiv Data Sets. In order to check whether the made observations are specific for the BTC1000M data set, the packet transfer of the 20 queries executed on both WatDiv data sets was investigated. For queries whose triple patterns are joined only on their subject, no packets were transferred. The number of transferred packets of both hash-based covers vary by less than 5% for each query and both data sets. The minimal edge-cut cover reduces the number of transferred packets only for 3 out of 20 queries by 8%-38% using the WatDiv100M data set. For the remaining queries the observed numbers of transferred packets are similar to the ones observed for the hash-based covers.

Scaling Number of Slaves. The effect of scaling up the number of slaves on the packet transport and the data transfer of the finished queries is not so huge. As shown in Figure 5.10b the number of transferred packets increases by 12%-64% when scaling from 10 to 20 slaves and by 23%-229% when scaling from 10 to 40 slaves using the minimal edge-cut cover. In case of the hierarchical cover the increases are 6%-67% and 12%-206%. For the hash cover the increases are 6%-51% and 16%-177%. In contrast to the high impact on the number of transferred packets, the data transfer increases only slightly (i.e., by at most 16%). Thus, the horizontal containment decreases for all graph cover strategies when the number of slaves is increased. Since in the evaluation scaling up the number of slaves affects the number of transferred packets more strongly than the data transfer, a network with a low latency seems to be more important than a network with a high bandwidth, to achieve low execution times for a high number of slaves.

Scaling Data Set Size. Scaling up the data set size while keeping the number of slaves constant leads to increased graph chunk sizes. Thus, one may assume that the number of transferred packets per result variable binding decreases since more query results might be computed with the data of a single chunk. In contrast to this assumption, the number of transferred packets per result variable binding increases by up to 100% for all queries when scaling up to the 1000M triples data set. When scaling up to the 2000M triples data set the number of transferred packets per result variable binding increases between 50% and 450% for all queries. These increases are independent of the graph cover strategy. Thus, the horizontal containment becomes worse when the data set size increases. Since all graph cover strategies are affected by the increased number of transferred packets to almost the same extent, the changes in the number of transferred packets between the different graph cover strategies stays nearly the same for most queries.

Vertical Parallelization

Total Computational Effort. The second factor influencing the overall query performance is the vertical parallelization, which combines the data transfer presented in the previous section with the workload imbalance. Before analysing the workload imbalance, it is examined how the total computational effort $w(C)$ changes. As expected, in the evaluation the total computational effort stays the same independent of the graph cover strategy and the number of slaves for the finished queries. When increasing the data set size, the computational effort increases for all graph cover strategies equally, since the queries produce more results.

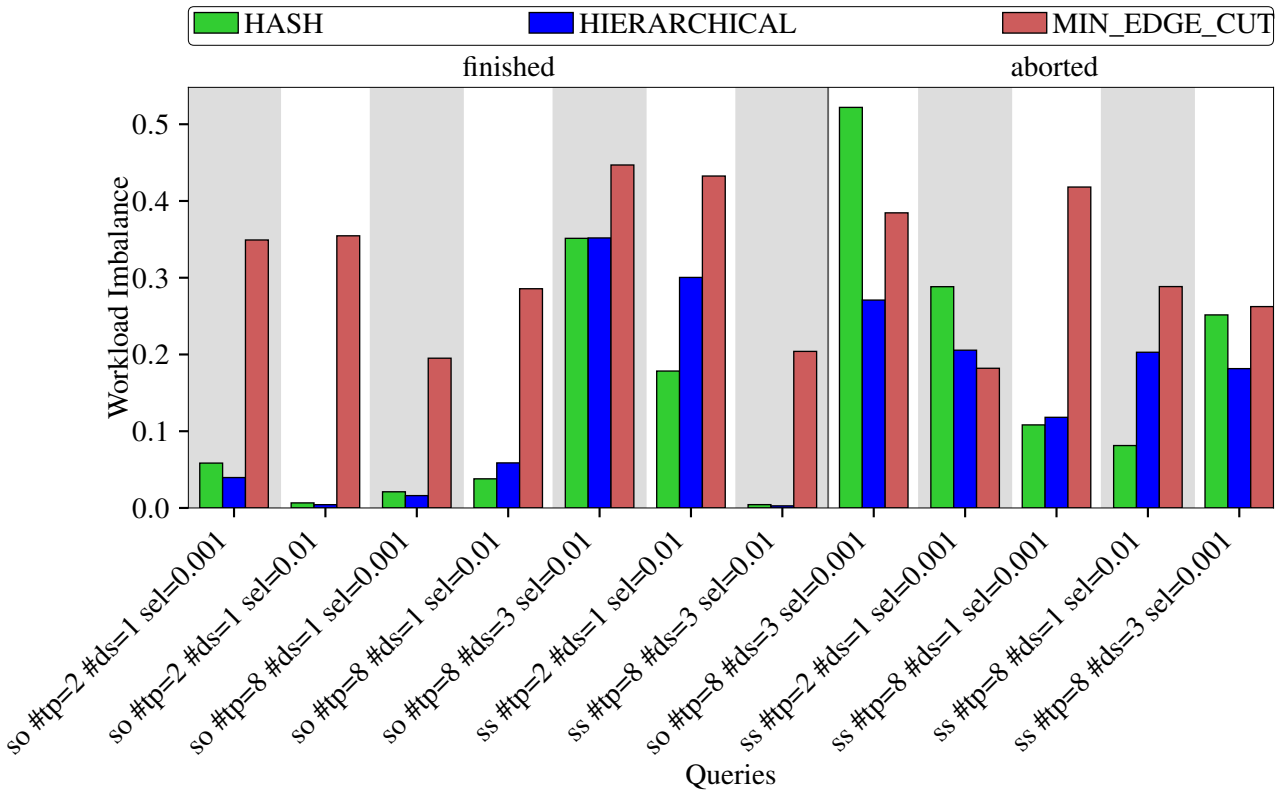


Figure 5.11.: Workload imbalance W of the bushy query execution. Comparison of the different graph covers at 10 slaves.

Workload Imbalance. When analysing the workload imbalance W as shown in Figure 5.11, the minimal edge-cut cover has the most unbalanced workload of all graph covers except for queries aborted after one million results. This is caused by the small graph chunks that contain only small portions of the huge densely connected core of the data set (see Section 5.2.2). Since these chunks contain fewer matches for the triple patterns in the queries, they have a much smaller workload than the other graph chunks, whereas the single huge graph chunk does not produce a higher workload. The workload of the hash and the hierarchical graph cover is similarly balanced for all queries that were not aborted. Also the storage imbalance is similar for both graph covers. In case of the queries that were aborted after one million results, none of the graph cover strategies balances its workload better than the others in general.

Combining the high workload imbalance W with the high horizontal containment, the minimal edge-cut graph cover only allows a low vertical parallelization for all types of queries. The vertical parallelization of both hash-based covers depends on the type of query. Long path-shaped queries that combine triples from several sources lead to a low vertical parallelization whereas short path-shaped queries lead to a medium vertical parallelization.

Workload Imbalance for the WatDiv Data Sets. To check whether the observed workload imbalances are independent of the used data set, both WatDiv data sets are used. The workload imbalance of the hash-based graph covers varies by less than 0.02 for almost all queries. For the WatDiv100M data set, the workload of the minimal edge-cut cover was the most imbalanced. For 12 out of 20 queries, its workload imbalance was 0.41 to 0.61 higher than the workload imbalance of both hash-based covers.

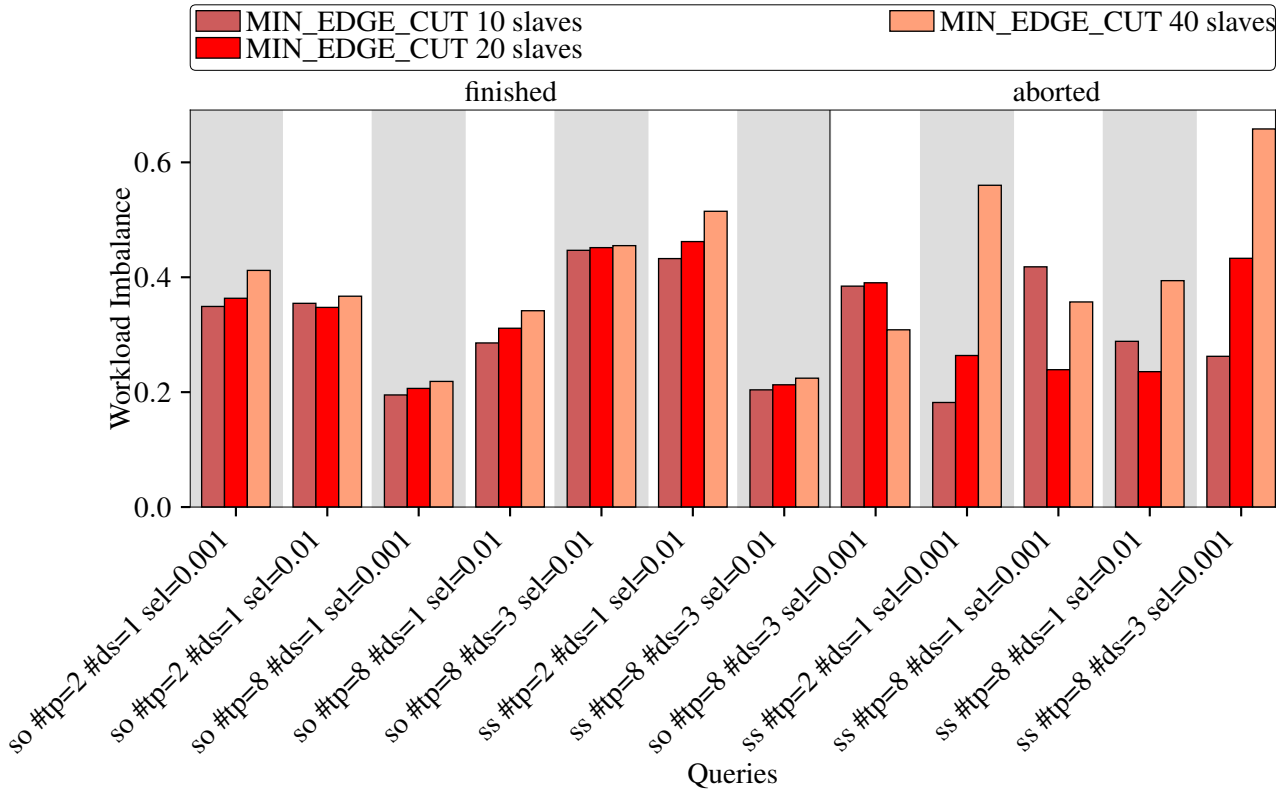


Figure 5.12.: Workload imbalance W of the bushy query execution. Comparison of the different number of slaves for the minimal edge-cut cover.

Scaling Number of Slaves. In order to visualize the effect of the number of slaves on the workload imbalance better, Figure 5.12 shows the workload imbalances for the minimal edge-cut cover at the different numbers of slaves. For all finished queries the workload becomes more imbalanced, when the number of slaves increases. Even for the aborted queries this is true for most queries. While the median workload imbalance of all queries increases by 4% and 15% for the minimal edge-cut cover when scaling to 20 and 40 slaves, the median workload imbalance increases by 27% and 125% for the hash cover, and 68% and 144% for the hierarchical hash cover. Thus, when scaling horizontally, the workload imbalance of the hash-based covers increases faster than for the minimal edge-cut cover. Nevertheless, the minimal edge-cut cover has the most imbalanced workloads for every examined number of slaves.

Scaling Data Set Size. As shown in Figure 5.13, the workload imbalance decreases for the minimal edge-cut cover for most queries, when the data set size increases. When scaling from 500M to 1000M triples the median workload imbalance decreases by 8% and by 42% when scaling from 500M to 2000M triples. For the hash covers the median workload decreases by 25% and 32%, and for the hierarchical hash cover it decreases by 24% for both data sets.

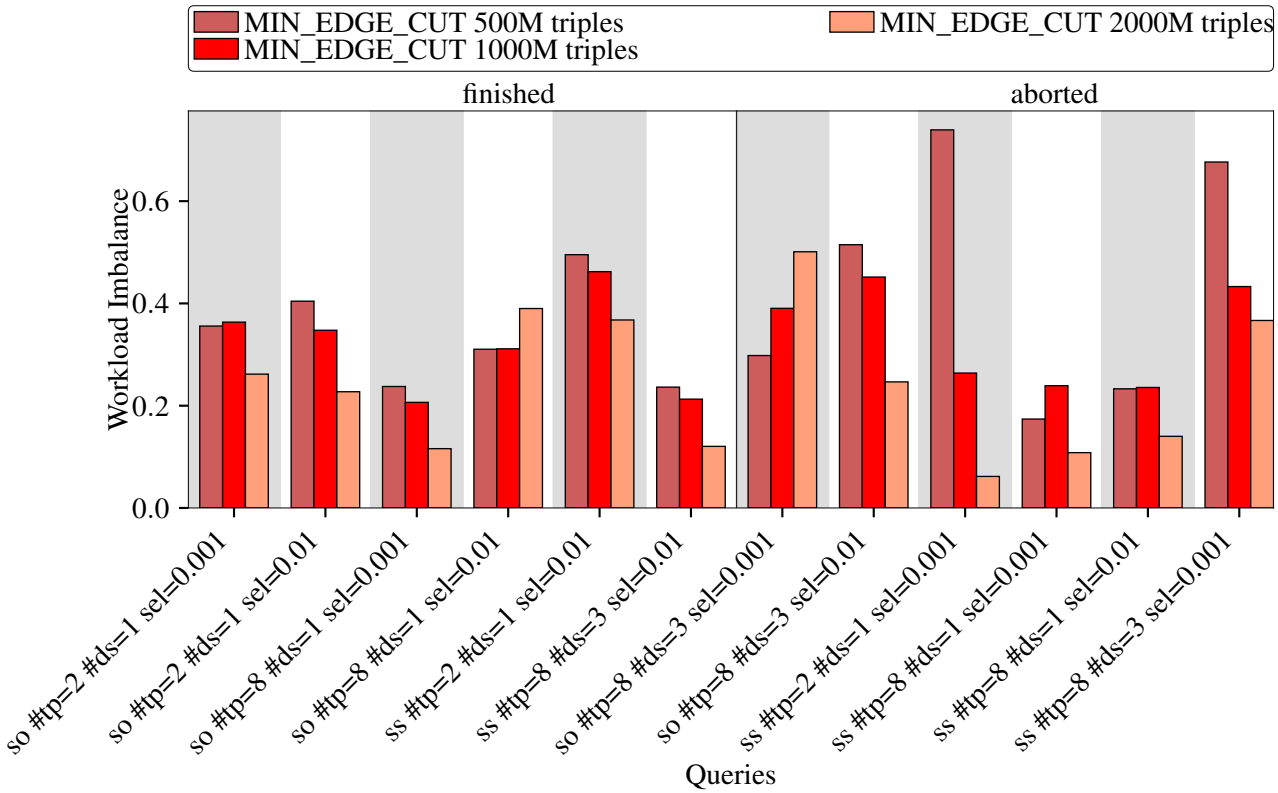


Figure 5.13.: Workload imbalance W of the bushy query execution. Comparison of the different data set sizes for the minimal edge-cut cover.

5.3. Lessons Learned

When analyzing the results of the experiments, the observations has confirmed several expectations:

- The distributed query execution speeds up the query execution for the hash-based and the minimal edge-cut cover since they can benefit of the parallelization.
- Star-shaped queries are executed faster than path-shaped queries. Path-shaped queries with a few triple patterns are executed faster than path-shaped queries with many triple patterns.
- Scaling up the number of slaves reduces the horizontal containment since the number of triples in each chunk is reduced.
- The vertical cover needs to transfer the most packets and has the highest workload imbalance since the triple patterns match only with triples of a few graph chunks. This leads to the longest query execution times in our experiments.
- The minimal edge-cut cover takes the longest time to be created and produces the most unbalanced graph chunk sizes, but has the best horizontal containment.

Beside the expected outcomes, the evaluation has shown several surprising results:

- The good horizontal containment of the minimal edge-cut cover is caused by the higher graph chunk diameter and not by the marginal reduced number of cut edges in comparison to the hash-based covers.
- Scaling up the data set sizes reduces the horizontal containment for all graph cover strategies.
- The query workload becomes more imbalanced, when the number of slaves is scaled up but it becomes more balanced, when the data set size is increased.
- The hash-based covers have a better overall performance than the minimal edge-cut cover, even if the latter has less data transfer. Thus, the workload imbalance may be more important than the data transfer.
- Both hash-based covers perform nearly the same.
- The n-hop replication reduces the number of transferred packets drastically but the overall query performance is decreased since duplicate intermediate results increase the total computational effort strongly.

A few aspects that were planned to be investigated could not be analyzed by the experiments:

- Since most of the generated queries that combine triples from three data sources were aborted after one million results, the impact of the data source number on the query execution time could not be examined.
- None of the investigated graph cover strategies without replication has a high vertical parallelization in general. Thus, the question remains how large the effect of the vertical parallelization on the query execution time is.

5.4. Discussion

In this study the impact of frequently used graph cover strategies have been examined. These strategies consist of two hash-based graph covers, the minimal edge-cut cover, the vertical cover and the 2-hop extension of the hash cover. The minimal edge-cut cover strategy takes more effort to be prepared than the hash-based covers and the vertical cover but due to the reduced number of cut edges, one might expect that queries can be processed locally with less data transfer.

Commonly, papers like [92, 121, 156] make the assumption that a graph cover strategy with minimal data transfer implies low query execution time. However, our results suggest that while minimal edge-cut reduces the number of transferred packets up to 43% in comparison to hash-based strategies (see Figure 5.10a), due to a more unbalanced workload (see Figure 5.11), the query execution time of minimal edge-cut is effectively slower (see Figure 5.6a).

The vertical cover has an even 100 times slower query execution time than the minimal edge-cut cover or the hash-based covers, since matches for the triple patterns can only be found on a few slaves whereas for the other graph cover strategies the matches were found on all slaves. Thus, it is to be expected that the vertical cover will lead to a more imbalanced query execution strategy and/or a higher amount of transferred intermediate results independent of the query execution strategy.

The 2-hop extension could reduce the number of transferred intermediate results by more than 90% but due to a 4 to 10 times higher total computation effort the queries took up to 82 times longer to finish. Thus, graph cover strategies that replicate a high portion of triples need a distributed query execution strategy that avoids the computation of duplicate results in order to benefit from the triple replication.

The performed investigation suggests that in the used setting the minimal edge-cut cover takes the most effort to be prepared (see Figure 5.3) but does not perform better over all (see Figure 5.6a). Since both hash-based covers perform similarly, the simpler hash cover implementation may be preferred, if other functionality such as prefix matching does not benefit from the hierarchical hash cover.

CHAPTER 6

Combining the Benefits of Graph Clustering and Hash Partitioning

The evaluation in Chapter 5 has revealed that none of the frequently used graph cover strategies has a high vertical parallelization. Hash-based graph cover strategies balance the query workload over all compute nodes. This good workload balance comes at the cost of a high number of intermediate results that need to be exchanged between the compute nodes during query processing. Graph-clustering-based graph cover strategies reduce the amount of transferred intermediate data at the cost of an imbalanced query workload.

Given these results, the main hypothesis of this chapter is that a graph cover strategy that combines the balanced query workload of the hash-based graph cover strategies with the reduced data transfer of the graph clustering approaches should lead to a high vertical parallelization and thus, allow for faster query executions. To reduce the data transfer, all data items required to produce a single query result should be stored on the same compute node. To balance the query workload among all compute nodes the various data item sets that produce different individual query results should be equally distributed among all compute nodes. Therefore, it is expected that graph cover strategies that (i) collocate the individuals of small sets of closely connected data items on compute nodes and (ii) store similar amounts of data items on the different compute nodes to execute queries faster than the frequently used data placement strategies.

In this chapter this hypothesis is investigated by evaluating two graph cover strategies that fulfil both properties (see Section 6.1). The first strategy called *overpartitioned minimal edge-cut cover* is described in [55]. It first performs a graph-clustering algorithm to split the graph into a high number of small partitions and thereafter assigns the partitions to compute nodes based on a greedy algorithm. Since its implementation did not scale well with the data set size in the performed experiments, a novel second strategy called *molecule hash cover* has been developed. It assigns molecules[1] – a small set of connected triples – to compute nodes based on their hashes. The implementations of both strategies have been integrated into Koral and are made open source[2]. Both strategies were evaluated (see Section 6.2) with the same extensive set of experiments that were used for the evaluation in Chapter 5. The results are presented in Section 6.3.

6.1. Proposed Graph Cover Strategies

To combine the balanced workload of hash-based graph covers with the low number of transferred intermediate query results of graph-clustering-based graph covers, graph cover strategies may collocate closely connected triples on the same compute node while balancing the number of stored triples over the compute nodes. In the following, two graph cover strategies that fulfill these properties are presented: the novel molecule hash cover (see Section 6.1.1) and the overpartitioned minimal edge-cut cover (see Section 6.1.2) originally presented

[1] This strategy has been inspired by the definition of RDF molecules in [39]. It has been adapted for the needs of query processing.
[2] `https://github.com/Institute-Web-Science-and-Technologies/koral`

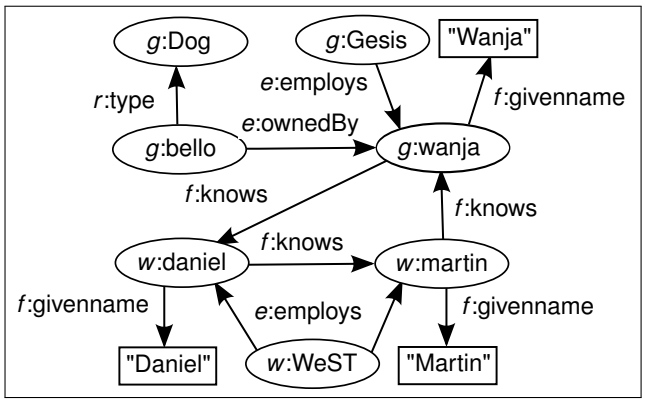


Figure 6.1.: Example graph describing the knows relationships between some employees of WeST and Gesis.

in [55]. Both strategies were implemented within the profiling system Koral. Their source code is made open source as part of Koral.

6.1.1. Molecule Hash Cover

The molecule hash cover aims to reduce the number of transferred intermediate results of the hash cover while maintaining its balanced query workload. The basic idea is that instead of randomly distributing individual triples a set of connected triples – so called molecules – are arbitrarily distributed among the compute nodes. Thereby, no triple should be replicated in order to keep the data set size fixed and to avoid the computation of duplicate intermediate results. Before the algorithm can be described, some additional definitions are required.

Definition 33. A subset $M_v \subseteq G$ of an RDF graph G is called *molecule*[3] *of vertex* v if

1. for all triples $t \in M_v$ there is a path $\langle t_0, t_1, ..., t_n, t \rangle$ such that $t_0 = (v, p_0, o_0)$ and $t_0, t_1, ..., t_n \in M_v$ and
2. if s is a subject of some triple in M_v, then
   $\forall (s, p, o) \in G : (s, p, o) \in M_v$.

The vertex v is called anchor vertex.[4]

Based on this definition, a molecule is a set of triples that are reachable from its anchor vertex. This property of molecules aims for an efficient processing of path-shaped queries. Furthermore, star-shaped queries, in which the triple patterns are joined on the subject, will be processed efficiently. Nevertheless, the definition of a molecule does not require all triples of the graph that are reachable from the anchor vertex to be contained in the molecule. This allows a decomposition of the graph into disjoint molecules.

Definition 34. The *(directed) molecule diameter* of a molecule M_v is the length of the longest of all shortest paths in the graph M_v that start at anchor vertex v.

Example 24. The molecule with diameter 1 of the anchor vertex g:gesis in the example graph shown in Figure 6.1 only contains the triple (g:Gesis, e:employs, g:wanja). A molecule with diameter 2 of the same anchor vertex would additionally contain (g:wanja, f:knows, w:daniel) and (g:wanja, f:givenname, "Wanja").

[3] The definition of an RDF molecule origninated in [39] has been adapted to allow molecule diameters ≥ 1.

[4] The term anchor vertex was taken from [91].

```
1  Input: RDF graph G, set of compute nodes C,
2         maximal molecule diameter d,
3         target working queue size w
4  Output: graph cover named cover_molHash
5  ---------------------------------------------------------------
6  incidenceMap = createIncidenceMap(G)
7  V = incidenceMap.getKeySet()
8  W = ∅
9  // initialize with vertices that have indegree 0
10 for (v ∈ V):
11    if (δ⁻(v) == 0):
12       W = W ∪ {(v,0,v)}
13       V = V \ {v}
14 // perform breadth-first search
15 while (W ≠ ∅ ∨ V ≠ ∅):
16    while (|W| < w ∧ V ≠ ∅):
17       // ensure target working queue size
18       v = V.getVertex();
19       W = W ∪ {(v,0,v)}
20       V = V \ {v}
21    W' = ∅
22    for ((v,l,a) ∈ W):
23       for ((v,p,o) ∈ incidenceMap.get(v)):
24          cover_molHash = cover_molHash ∪ {((v,p,o), {hash(a) mod |C|})}
25          if (l+1 == d):
26             W' = W' ∪ {(o,0,o)}
27          else:
28             W' = W' ∪ {(o,l+1,a)}
29          V = V \ {o}
30       incidenceMap.remove(v)
31    W = W'
```

Listing 6.1: The molecule hash cover creation algorithm.

The idea of the molecule hash cover is to decompose the RDF graph into disjoint molecules with a fixed maximal diameter. The resulting set of molecules is then equally distributed among all compute nodes. This idea is rather simple but effective as indicated by the evaluation described in Section 6.3.

The algorithm that creates a molecule hash cover bases on a breadth-first search. Thereby, the graph is decomposed into molecules that are distributed among the compute nodes. The actual algorithm is presented in Listing 6.1. It receives an RDF graph G, the set of compute nodes C and the maximal diameter d of the molecules that should be distributed as input. In order to speed up the algorithm, a target size w of the working queue – the queue that contains the vertices to be visited next by the breadth-first search – is given as additional input. The result of the algorithm is a graph cover according to definition 2 since the breadth first search iterates over all triples and assigns them to the compute node to which the anchor vertex of the corresponding molecule belongs.

In line 6 the graph is converted into a map, which maps a vertex v to the set of all triples that have v as subject. Thereafter, the working queue W is initialized with all vertices that have an indegree of 0 (lines 10-13). The elements of the working queue are triples of the form (v, l, a), where v is the vertex to be visited next, l the length of the path from anchor vertex a to v and a the anchor vertex of the current molecule.

Example 25. In the graph of the running example the working queue would be initialized with the elements $(\text{w:WeST}, 0, \text{w:WeST})$, $(\text{g:Gesis}, 0, \text{g:Gesis})$ and $(\text{g:bello}, 0, \text{g:bello})$.

In lines 15-31 a breadth-first search is performed, starting at the vertices with indegree 0. These vertices are selected as starting points, since the breadth-first search cannot find them otherwise. In the lines 23-29 all triples whose subject is contained in the working queue are traversed. Each of these triples are assigned to the compute node to which the anchor vertex of the current molecule belongs (line 24). If the diameter of the molecule reaches the maximal diameter size d, the object of the traversed triple becomes the anchor vertex of a new molecule (line 26). Otherwise, the object is inserted into the working queue with an incremented molecule diameter and the original anchor vertex. After all triples with the same vertex v as subject are traversed, the entry of v is removed from the incidence map (line 30). This ensures that no triple is assigned to more than one compute node.

Example 26. In the running example $(\text{w:WeST}, 0, \text{w:WeST})$ is extracted from the working queue first. Based on the hash function shown in example 13 both triple with w:WeST as subject are assigned to compute node c_2. Thereby, the elements $(\text{w:daniel}, 1, \text{w:WeST})$ and $(\text{w:martin}, 1, \text{w:WeST})$ are appended to the working queue. Then, $(\text{g:Gesis}, 0, \text{g:Gesis})$ is dequeued from the working queue, the corresponding triples are assigned to compute node c_1 and $(\text{g:wanja}, 1, \text{g:Gesis})$ is appended to the working queue. When $(\text{g:bello}, 0, \text{g:bello})$ is dequeued, the corresponding triples are added to compute node c_2 and $(\text{g:Dog}, 1, \text{g:bello})$ as well as $(\text{g:wanja}, 1, \text{g:bello})$ are added to the working queue.

Now, the working queue contains two entries for g:wanja: $(\text{g:wanja}, 1, \text{g:Gesis})$ and $(\text{g:wanja}, 1, \text{g:bello})$. Since the former one was added to the working queue first, the outgoing edges of g:wanja will be assigned only to compute node $\text{hash}(\text{g:Gesis}) = 1$. The final molecule hash cover is shown in Figure 6.2. Each molecule is colored in a different color.

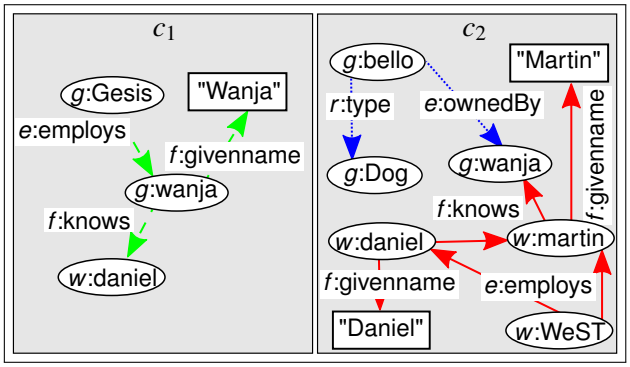


Figure 6.2.: An example molecule hash cover of the example graph from Figure 6.1.

Since there might be cycles in the graph that are not reachable from any vertex with indegree 0, one of these vertices is arbitrarily selected and added to the working queue. This happens in lines 16-20. During initial experiments the working queue only contained a single element after a few iterations. This small working queue size caused a long execution time of the algorithm. In order to speed up the execution, new random

anchor vertices, that have not been visited yet, are added to the working queue whenever the size of the working queue is smaller than a target size w (lines 16-20).

Theorem 4. *For RDF graphs G in which the number of edges m is larger than the number of vertices, the molecule hash cover can be produced in the worst time complexity $\mathcal{O}(m \log m)$.*

Proof. Let $m = |G|$ denote the number of triples of the graph G and $n = |V_G|$ denote the number of vertices in G. The input of the algorithm is an RDF graph stored as a list of triples. Its conversion into an incidence map, mapping a vertex v to the set of triples with v as subject, in line 6 of Listing 6.1 can be done with a single merge sort. Thereby, the list of triples is sorted by their subjects first. Thereafter, the sorted list is iterated and all triples with the same subject are aggregated. This has the time complexity $\mathcal{O}(m \log m)$. During this evaluation it is also possible to compute the indegree of each vertex without changing the complexity class. Thus, the code within the for loop in the lines 10-13 is executed n times. If V is implemented as a balanced search tree, the operation in line 13 has logarithmic complexity. The complexity of the for loop is $\mathcal{O}(n \log n)$.

The code within the while loop starting in line 16 is executed at most n times. If V is implemented as a balanced search tree, the operations in line 18 and 20 can be performed in logarithmic time. This leads to an effort of $\mathcal{O}(n \log n)$.

Since each edge is traversed exactly once, the code in lines 24-29 is executed m times. The operation in line 29 has logarithmic complexity, while the other operations can be performed in constant time. Thus, this code has the complexity $\mathcal{O}(m \log n)$. The lines 23 and 30 are executed $n+m$ times. Each of both operations have logarithmic complexity leading to the complexity $\mathcal{O}((m+n) \log n)$.

When combining the complexities of the different code segments, the overall time complexity of the algorithm is $\mathcal{O}((m+n) \log n + m \log m)$. Since in realistic RDF graphs the number of edges m is much larger than the number of vertices n, the time complexity can be simplified to $\mathcal{O}(m \log m)$. □

6.1.2. Overpartitioned Minimal Edge-Cut Cover

Another graph cover strategy that collocates closely connected triples on the same compute node while balancing the number of stored triples over the compute nodes can be found in [55]. In order to identify the closely connected triples, it performs a minimal edge-cut partitioning. But instead of creating a single partition per compute node it creates a much higher number of partitions. Therefore, this strategy is called overpartitioned minimal edge-cut cover in the context of this thesis. As motivated in [55], the number of partitions to create can be determined by $\sqrt{\frac{\lambda * |E|}{d * |C|}}$ with $d = \frac{|E|}{|V|}$ the average degree of a vertex, V the set of vertices, E the set of edges and C the set of compute nodes. λ is a parameter with which the number of partitions can be adjusted to get the fastest query execution times. The authors of [55] stated that in their experiments $\lambda = 187$ resulted in the fastest query execution times. After the partitions have been created, they are combined to graph chunks, one for each compute node. This assignment is done by a greedy algorithm that assigns a partition to the smallest chunk.

6.2. Experimental Setup

To investigate how the molecule hash cover and the overpartitioned minimal edge-cut cover perform in comparison with the frequently used graph cover strategies, several experiments are performed. The set of configurations in the experiments results from combining the different graph cover strategies with the different data sets and the corresponding queries. This experimental setting follows the evaluation methodology described in Chapter 4 and is identical to the one used for the evaluation described in Chapter 5.

Compared Graph Cover Strategies

In order to evaluate the molecule hash cover and the overpartitioned minimal edge-cut cover (with $\lambda = 187$), both have been implemented within the glass box profiling system Koral. Their performances are compared with the performances of the hash cover, hierarchical hash cover and the minimal edge-cut cover. Since these frequently used graph cover strategies showed a better performance than the 2-hop hash cover in the evaluation described in Chapter 5, it is not used for the comparison.

Data Sets and Queries

The first used data sets consist of the 500 million, 1 billion and 2 billion triples subsets of the real-world billion triple challenge data set from 2014 (BTC2014) [76]. They are called BTC500M, BTC1000M and BTC2000M within this chapter. Additionally, two data sets have been generated with the Waterloo SPARQL Diversity Test Suite v0.6 [15]: the WatDiv1000M data set containing 1,099,208,068 triples and the WatDiv100M data set containing 109,786,094 triples.

For the BTC2014 data set, the seven queries were used that do not abort after 1 million results. These queries consists of 2-8 triple patterns that are subject-object joined (path-shaped) or subject-subject joined (star-shaped). For each of the WatDiv data sets 20 queries have been generated based on the basic testing query templates[5]. These generated queries consist of star-shaped queries (S1-S7), path-shaped queries (L1-L5) and combinations of both shapes (C1-C3 and F1-F5).

Experiment Execution

For each graph cover strategy and data set, Koral is cleared. Then the data set is loaded and the list of corresponding queries is executed 10 times. Thus, the effect of operating system-dependent caches storing the results of the previously executed query is reduced, because no query is immediately reexecuted after it has finished. In order to prevent the effect of outliers caused by, e.g. garbage collection, from all 10 executions of a query, the best and the worst execution time are ignored and the arithmetic mean is used for *exTime*.

Computer and Software Environment

Koral is executed on 11, 21 and 41 VMs distributed over 4, 6 and 8 physical computers, respectively. The master has 4 cores and 64 GB RAM and the slaves have 1 core and 2 GB RAM each. Since the Koral master VM needs to store the complete data set, it has a 1 TB hard disk. The slaves have 300 GB hard disks. The physical computers on which the VMs run are connected via a 1 Gigabit Ethernet network.

The operating system of each VM is a 64 bit Ubuntu 14.04.4 with the Linux kernel 3.13.0-96. The Oracle JDK 1.8.0_101 is used to execute Koral in version 0.0.1. In order to create the minimal edge-cut cover, METIS is used in version 5.1.0.dfsg-2. To generate the WatDiv data sets and the corresponding queries the WatDiv generator in version 0.6 is used.

6.3. Results

The molecule hash cover strategy allows for configuring the diameter of the molecules in which the graph is split. In order to reduce the number of possible configurations for the experiments, the query execution

[5] https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

times for several molecule hash covers with varying molecule diameters were performed. As described in Section 6.3.1 the molecule hash cover with a diameter of 3 lead to the fastest query execution times.

The performed evaluations aim to investigate, whether the molecule hash cover and the overpartitioned minimal edge-cut cover are able to combine the balanced workload of hash-based graph cover strategies with the low number of transferred intermediate results of graph-clustering based graph cover strategies and thus, lead to a high vertical parallelization. First, the analysis of measurements that are independent of the queries are described in Section 6.3.2. Beside the loading time, this includes how balanced the chunk sizes are since this influences the query execution time as already described in Chapter 5. The overall performances of all compared graph cover strategies are presented in Section 6.3.3. This performance relies on the horizontal containment and vertical parallelization. Thus, in Section 6.3.4 indicators for the horizontal containment and the vertical parallelization are analyzed.

6.3.1. Effect of Molecule Diameter for the Molecule Hash Cover

The molecule hash cover has the direct molecule diameter as a parameter. To analyze its impact on the query performance the diameter was varied between 2 and 7 for the BTC1000M data set. It was observed that with a higher molecule diameter the number of transferred packets decreases for most queries. When increasing the diameter from 2 to 3, the packet transfer drops by up to 26% whereas the change from a diameter 3 to 7 is below 6%. In contrast to the packet transfer, the workload imbalance increases slightly for most queries. When the molecule diameter is increased from 2 to 3 the workload imbalance increases by up to 0.12. When increasing the molecule diameter from 3 to 7 the workload imbalance increases by up to 0.36. As a result the average query execution time drops by 11% when increasing the diameter from 2 to 3 and increases by up to 32% when increasing the diameter from 3 to 7. Therefore, only the molecule hash cover with a diameter of 3 will be compared with the other graph cover strategies in the remainder of this section.

6.3.2. Query Independent Measurements

In this section, the required time to create the graph covers is analyzed. Furthermore, it is investigated how balanced the sized of the created graph chunks are. The main findings are:

- The molecule hash cover needs a bit more time than the hierachical hash cover to be created but needs much less time than the minimal edge-cut cover. The overpartitioned minimal edge-cut cover needs the most time to be created.
- The overpartitioned minimal edge-cut cover has the most balanced graph chunk sizes. The sizes of the molecule hash chunks are less balanced than the hash based graph chunks but more balanced than the sizes of the minimal-edge-cut-based chunks.

Load Time

In the evaluation of the BTC2014 and the WatDiv1000M data sets, the hash-based graph cover strategies, which do not consider the graph structure, took 1 hour and the hierarchical hash cover took 7 hours to be created (see Figure 6.3). In contrast, the minimal edge-cut cover and the overpartitioned minimal edge-cut cover focus on the graph structure to create large connected graph chunks. They required 31 hours and 79 hours to be created for the BTC2014 data set. For the WatDiv1000M data set, none of both strategies could be created since METIS, which was used to create the minimal edge-cut partitioning, tried to occupy more than 1TB main memory. A similar observation has been made by [60]. In contrast to this, the molecule hash cover only uses

the graph structure for the computation of the indegrees and the breadth-first search. As a result, it was able to load all data sets and required 11 hours for the BTC2014 and 7 hours for the WatDiv1000M data set (which is only 10 minutes slower than the hierarchical hash cover). For the latter synthetic data set, the molecule hash cover was faster since the breadth-first-search needed fewer iterations over the disk-based working queue. The relation between the loading times of the different graph cover strategies is similar for the WatDiv100M data set.

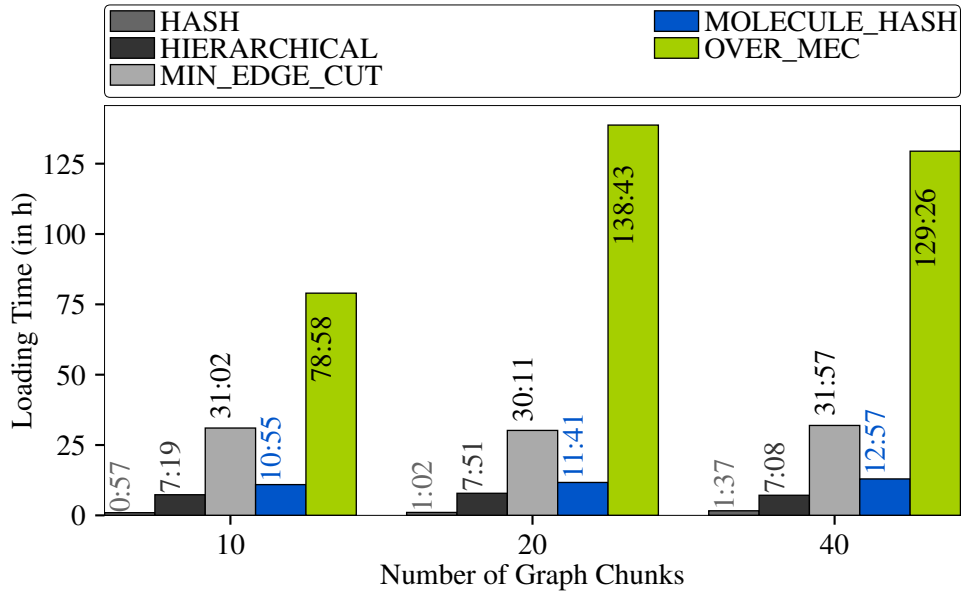


Figure 6.3.: Creation times of different graph covers for different slave numbers and the BTC1000M data set.

Scalability. When scaling up the number of slaves from 10 to 20 and from 20 to 40, the load time T of the molecule hash cover varied by less than 10% and hence is negligible for the BTC1000M data set. When scaling from 10 to 20 slaves, the overpartitioned minimal edge-cut cover required 75% more time to be created. The cause is that METIS required 58% more time to create the graph partitioning and the combination of the resulting partitions to a higher number of graph chunks took another 66% longer. When scaling from 20 to 40 slaves the overpartitioned minimal edge-cut cover, the loading time decreased by 7%. When doubling the data set size the loading time of the molecule hash cover also doubles, similar to the loading times of the hash and the hierarchical hash covers. In contrast to this the loading times of the overpartitioned minimal edge cut cover and the minimal edge-cut cover increase by a factor of 3.

Since the focus of the evaluation is the query execution, only a proof-of-concept non-optimized implementation has been provided for loading. Nevertheless, the overpartitioned minimal edge-cut cover has originally been implemented in the RDF store TriAD [55]. As reported in [60], with the more sophisticated implementation of TriAD it took only 12 hours to load a data set of ~1.4 billion triples. In use cases, in which a lot of long-running analytical queries are issued against a static data set, the savings of the shorter query execution time may overcome the longer loading times.

Storage Imbalance

As shown in table 6.1, the minimal edge-cut cover strategy produces the least balanced graph chunks and the overpartitioned minimal edge-cut cover strategy the most balanced graph chunks for 10 chunks. For both

Data set	BTC1000M			WatDiv100M	WatDiv1000M
# chunks	10	20	40	10	10
Hash cover	0.0167	0.0196	0.0275	0.0016	0.0006
Hierarchical hash cover	0.0119	0.0160	0.0240	0.0010	0.0006
Minimal edge-cut cover	0.1787	0.2418	0.2441	0.4355	-
Molecule hash cover	0.0145	0.0235	0.0361	0.0114	0.0049
Overpartitioned minimal edge-cut cover	0.0001	0.0013	0.0020	0.00004	-

Table 6.1.: The storage imbalance b of the different graph covers at different number of graph chunks.

graph cover strategies, no storage imbalance could be given for the WatDiv1000M data set since they could not be load it. The low storage imbalance of the overpartitioned minimal edge-cut covers is achieved by the greedy algorithm that tries to optimize the storage balance while assigning partitions to compute nodes. For all graph covers created by the hash, hierarchical hash and molecule hash cover strategies, b is below 0.02 for 10 slaves. None of these three graph cover strategies is more balanced then the others for the BTC1000M, WatDiv100M and WatDiv1000M data sets. The best storage imbalance with at most 0.0001 is achieved by the overpartitioned minimal edge-cut cover. This good storage balance is achieved by the greedy algorithm that tries to optimize the storage balance while assigning partitions to compute nodes.

Scalability. Similar to the frequently used graph cover strategies, scaling up the number of slaves and the data set size have only a negligible effect on the storage imbalance.

6.3.3. Measuring Overall Query Performance

In this section it is investigated, whether the molecule hash cover and the overpartitioned minimal edge-cut cover execute queries faster or slower than the frequently used graph cover strategies. The main findings are:

- The molecule hash cover and the overpartitioned minimal edge-cut cover execute queries faster than the frequently used graph cover strategies. Thereby, the overpartitioned minimal edge-cut cover tends to be faster than the molecule hash cover.
- The molecule hash cover and the overpartitioned minimal edge-cut cover show the highest execution time reductions for queries that have a mixture of several different join patterns and path-shaped queries with a high diameter.
- For purely star-shaped queries that do not need to transfer intermediate results, the molecule hash cover may increase the query execution times in comparison with the hash-based covers but is still faster than the minimal edge-cut cover.

As described in Chapter 5 bushy query execution trees produced the lowest query execution times for the frequently used graph cover strategies. Therefore, the queries were executed using the bushy query execution trees for the molecule hash cover and the overpartitioned minimal edge-cut cover. Since with both graph covers the queries could be executed faster than with the frequently used graph cover strategies, the evaluations with the other query execution tree types were omitted. Since the query execution times vary a lot between the queries, the execution times are presented relative to the hash cover. The absolute query execution times can be found in Appendix C.3.

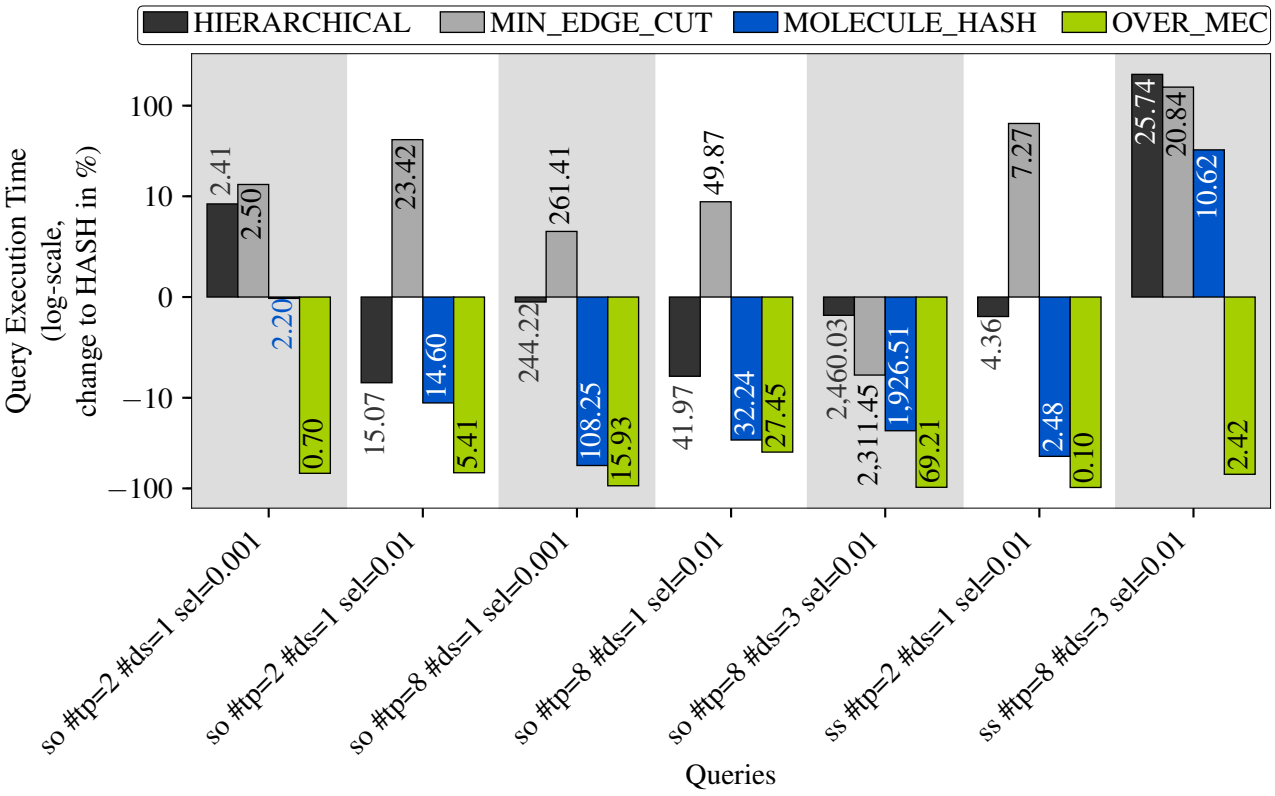


Figure 6.4.: *exTime* of all queries relative to the hash cover for BTC1000M. The numbers within the bars are the absolute query execution times in seconds.

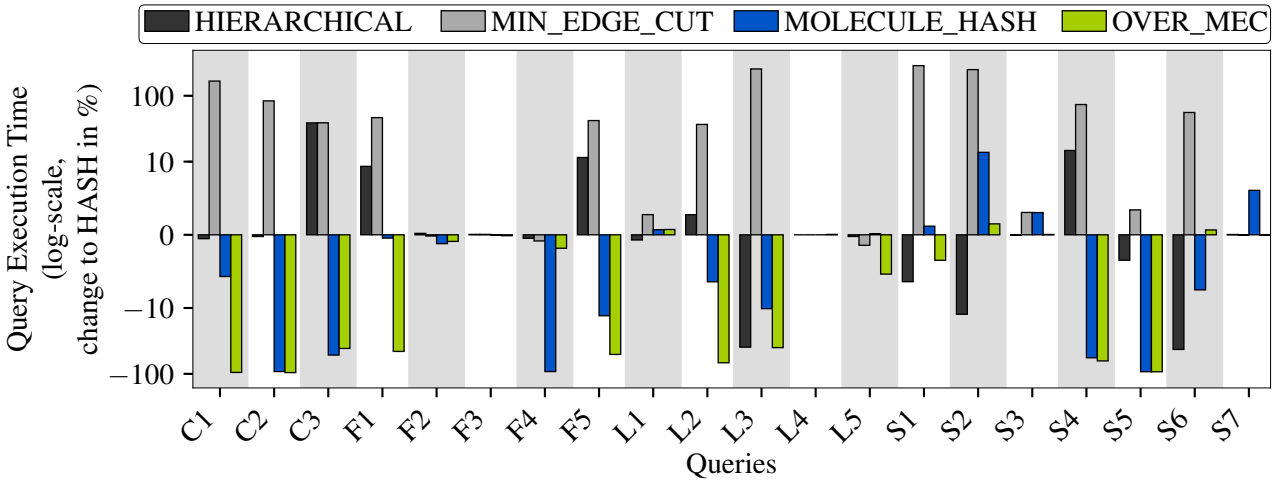


Figure 6.5.: *exTime* of all queries relative to the hash cover for WatDiv100M.

Query Performance for BTC1000M. Figure 6.4 shows the execution times *exTime* of the queries for all graph covers using the BTC1000M data set. For almost all queries the molecule hash cover reduces the query execution time between 11% and 56%. This reduction is caused by 38%-62% less packet transfer P that could overcome the workload imbalance W increase of below 0.05 for most queries. Only for query `ss #tp=8 #ds=3 sel=0.01`, the molecule hash cover increased the query execution time. In this case, the increased workload imbalance of the molecule hash cover could not be compensated since for this query no packets were transferred between compute nodes.

The overpartitioned minimal edge-cut cover reduces the query execution times between 40% and 98%. These high reductions are caused by an almost perfectly balanced query workload ($W \leq 0.03$) and almost no transferred packets ($P <$ -99%). For `so #tp=8 #ds=3 sel=0.01` and `ss #tp=2 #ds=1 sel=0.01`, the high reductions of more than 90% is mainly caused by a decreased workload imbalance of 99% and 89% for both queries.

Query Performance for WatDiv100M. For the synthetic WatDiv100M data set, the observations are similar to the ones of the BTC1000M data set. The query execution times are shown in Figure 6.5. In 13 out of 20 cases the molecule hash cover improved the query execution speed in comparison with the hash cover. In 10 of these cases the reduction amounted to 5%-92% (43% on average). In these cases the workload imbalances of both graph cover strategies were similar but the number of transferred packets reduced by 6%-95%. Most of these queries consisted of a mixture of different join types with at least 6 triple patterns. Only queries S2 and S7 were executed slower than in case of the hash cover. For both queries no packets were transferred between compute nodes but the workload imbalance almost doubled leading to the slower query execution.

The overpartitioned minimal edge-cut cover could reduce the query execution times by 5%-96% (60% on average) or had a similar execution time for all queries. The workload imbalance was up to 0.14 lower than the one of the hash cover. The number of transferred packets could be reduced by 23%-100%. Similar to the molecule hash cover, the overpartitioned minimal edge-cut cover performed the best for queries consisting of a mixture of different join types with at least 6 triple patterns.

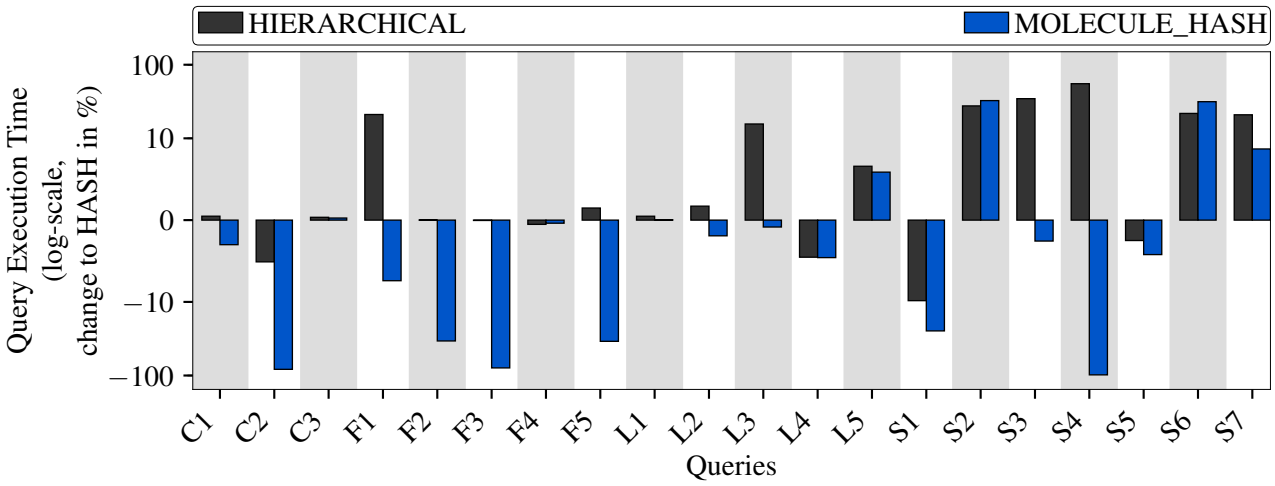


Figure 6.6.: *exTime* of all queries relative to the hash cover for WatDiv1000M.

Query Performance for WatDiv1000M. When scaling up the size of the data set, similar observations have been made (see Figure 6.6). As described in the previous section, the minimal edge-cut cover and the overpartitioned minimal edge-cut cover could not be created for this data set. For 9 out of 20 queries, the molecule hash cover is able to reduce the query execution time by 4%-98% (41% on average) and for 7 out of 20 queries

it has similar execution times as the hash cover. The execution time reduction may be caused by a 7%-96% reduced packet transfer while the workload imbalance staid almost constant. Most of these queries consisted of a mixture of different join types with at least 6 triple patterns. In 4 cases the execution time increased by more than 5%. Most of these queries were star-shaped queries with a low number of transferred packets. Thus, the reduction of the packet transfer could not compensate the workload imbalance increase.

Scaling Number of Slaves

When scaling up the number of slaves the molecule hash cover showed a similar performance as the hash covers (see Chapter 5). Only for query `ss #tp=8 #ds=3 sel=0.01` the query execution times almost halved when scaling from 10 to 20 slaves and from 20 to 40 slaves. For most queries the query execution times increase when increasing the number of slaves. This is caused due to the higher number of transferred intermediate results since the chunk sizes became smaller. Only for two queries the query execution times dropped when scaling to 20 slaves but increased when scaling to 40 slaves. Nevertheless, the molecule hash cover executes 6 out of 7 queries faster than the frequently used graph cover strategies for 40 slaves.

When increasing the number of slaves, the overpartitioned minimal edge-cut cover splits the initial RDF graph into fewer but larger partitions. These larger partitions lead to an up to 3 times higher workload imbalance during query execution. As a consequence the query execution times increase for almost all queries. As a result the overpartitioned minimal edge-cut cover executes only 4 out of 7 queries the fastest. Especially for the star-shaped queries in which no packet transfer could be observed, the increased workload imbalance lead to slower query execution times. This effect may be compensated by setting λ to a higher value when increasing the number of slaves.

Scaling Data Set Size

Scaling the size of the data set affects the molecule hash cover and the overpartitioned minimal edge-cut cover similar as the frequently used graph cover strategies. For almost all queries the query execution time per query result decreases. Only for query `so #tp=2 #ds=1 sel=0.01` the query execution times per result increase, as discussed in Chapter 5.

For the BTC500M data set the molecule hash cover executes 6 out of 7 queries faster than the hash cover, similar to the BTC1000M data set. For the BTC2000M data set the molecule hash cover is still faster than the hash cover for 5 out of 7 queries. In both cases the molecule hash cover could not reduce the number of transferred packets enough to compensate the increased workload imbalance. Since the data set scalability experiments were conducted with 20 slaves, the overpartitioned minimal edge-cut cover produced larger partitions leading to higher workload imbalances. As a consequence, for all data set sizes the overpartitioned minimal edge-cut cover could execute only 4 out of 7 queries the fastest. Especially the two star shaped queries were executed slower, since they did not cause any packet transfer.

6.3.4. Measuring Dependent Variables

In order to identify the reasons for the overall performance of the proposed graph cover strategies described in the previous section, the indicators of the horizontal containment and vertical parallelization are analyzed now. The main findings are:

- The molecule hash cover and the overpartitioned minimal edge-cut cover have a better horizontal containment than the frequently used graph cover strategies. Scaling up the number of slaves reduces the horizontal containment whereas scaling up the data set size improves the horizontal containment.

- The molecule hash cover has a slightly higher workload imbalance than the hash-based covers, independent of the number of slaves and the data set size.

- The overpartitioned minimal edge-cut cover has the most balanced query workload for 10 slaves. Scaling up the number of slaves decreases the workload imbalance, since the initial graph is split into fewer but larger partitions.

Horizontal Containment

All examined graph cover strategies assign triples with the same subject to the same chunk. Therefore, all triples required to produce one result of a subject-subject-joined star-shaped query are located in the same graph chunk. Since Koral performs joins on the slave storing the original triples, no data transfer was observed.

Horizontal Containment for BTC1000M. For the path-shaped queries, both graph cover strategies that collocate items of small closely connected triple sets on the same compute node are able to reduce the number of transferred packets *P* between compute nodes even more than the best frequently used graph cover strategy (see Figure 6.7). The molecule hash cover is able to reduce the packet transfer *P* for most path-shaped queries by 38%-62% in comparison with the hash cover. The overpartitioned minimal edge-cut cover can reduce *P* by 99%-100%.

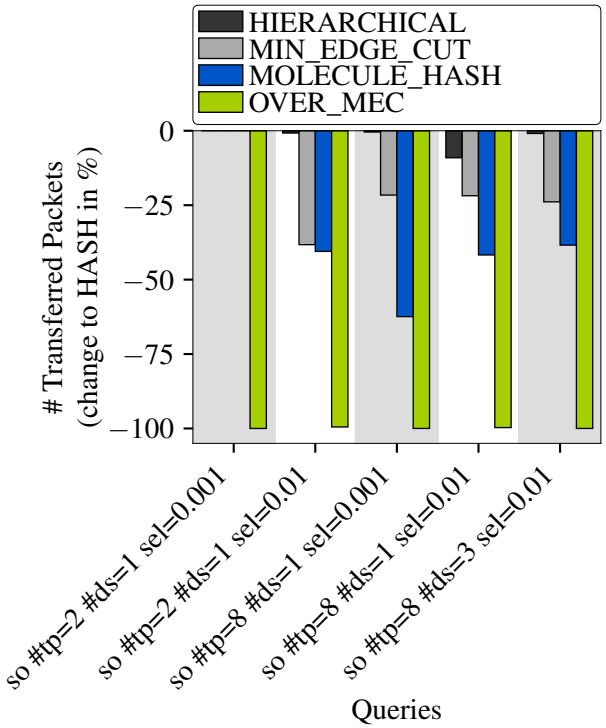


Figure 6.7.: Packet transfer *P* relative to hash cover for the BTC1000M data set using 10 slaves.

As described in Chapter 5 real-world data sets consist of a densely-connected huge core and most queries match with triples within this core. The traditional minimal edge-cut cover cuts this core into 10 partitions whereas the overpartitioned minimal edge-cut cover cuts it into 60k partitions. Due to the high reduction of transferred packets these smaller partitions seem to fit better with the subgraphs with which the queries match.

Horizontal Containment for WatDiv Data Sets. For the WatDiv100M data set the molecule hash cover could reduce the packet transfer for 13 out of 20 queries by 6%-95% (26% on average). For the remaining queries the number of transferred packets is similar to the ones of the hash cover. The overpartitioned minimal edge-cut cover could reduce the number of transferred packets by 23%-100% (84% on average) for 15 queries. Only for 5 queries the number of transferred packets were similar to the ones of the hash cover. For the WatDiv1000M data set, the molecule hash cover reduced the number of transferred packets by 7%-96% (28% on average) for 9 out of 20 queries. For the remaining queries the packet transfer was similar as for the hash cover.

Scaling Number of Slaves. When scaling up the number of slaves while keeping the data set size fixed, the horizontal containment is reduced for all graph cover strategies. As described in Chapter 5, the lower the packet transfer for 10 slaves the more it will increase. Since both presented graph cover strategies have a lower packet transfer than the frequently used graph cover strategies, the packet transfer increases by an average of 136% for the molecule hash cover and by an average of 266% for the overpartitioned minimal edge-cut cover when scaling from 10 slaves to 40 slaves. Nevertheless, the molecule hash cover and the overpartitioned minimal edge-cut cover are still able to reduce the packet transfer by 18%-54% and 83%-99% in comparison to the hash cover for 40 slaves, respectively. Thus, they still reduce the packet transfer more than the frequently used graph cover strategies.

Scaling Data Set Size. For the frequently used graph cover strategies the number of transferred packets per query result increases when the data set size is scaled up. In contrast to this, the packet transfer per query result reduces by 32%-99% when scaling up to 2 billion triples for all queries using the molecule hash cover. For the overpartitioned minimal edge cut cover, the packet transfer per query result increases by 24% for one query but reduces by 97%-100% for the remaining queries when scaling up to 2 billion triples. Thus, the horizontal containment of both graph cover strategies improves when scaling up the data set size.

Vertical Parallelization

Workload Imbalance for BTC1000M. When analyzing the workload imbalance W in Fig. 6.8, the molecule hash cover tends to increase the workload imbalance slightly by up to 0.05 in comparison with the hash cover for 5 out of 7 queries. Nevertheless, the workload imbalance of the molecule hash cover is still 22%-90% lower than the workload imbalance of the minimal edge-cut cover. In contrast to the molecule hash cover, the overpartitioned minimal edge-cut cover could reduce the workload imbalance to a maximum of 0.03 for all queries. Thus, both examined graph cover strategies that collocate items of small closely connected triple sets on the same compute node tend to balance the query workload similar to hash-based graph cover strategies or even better

Workload Imbalance for WatDiv Data Sets. For the WatDiv100M data set, the workload imbalance of the molecule hash cover is similar to the one of the hash cover for 18 out of 20 queries. The difference is less than 0.02. Only for queries S2 and S7 the workload imbalance of the molecule hash cover is twice as high as the workload imbalance of the hash cover. The overpartitioned minimal edge-cut cover balanced the workload similar to the hash cover for 14 out of 20 queries. For the remaining 6 queries, it could reduce the workload imbalance by up to 0.14 in comparison with the hash cover. When using the WatDiv1000M data set the workload imbalance of the molecule hash cover varies by less than 0.1 from the workload imbalance of the hash cover for all queries.

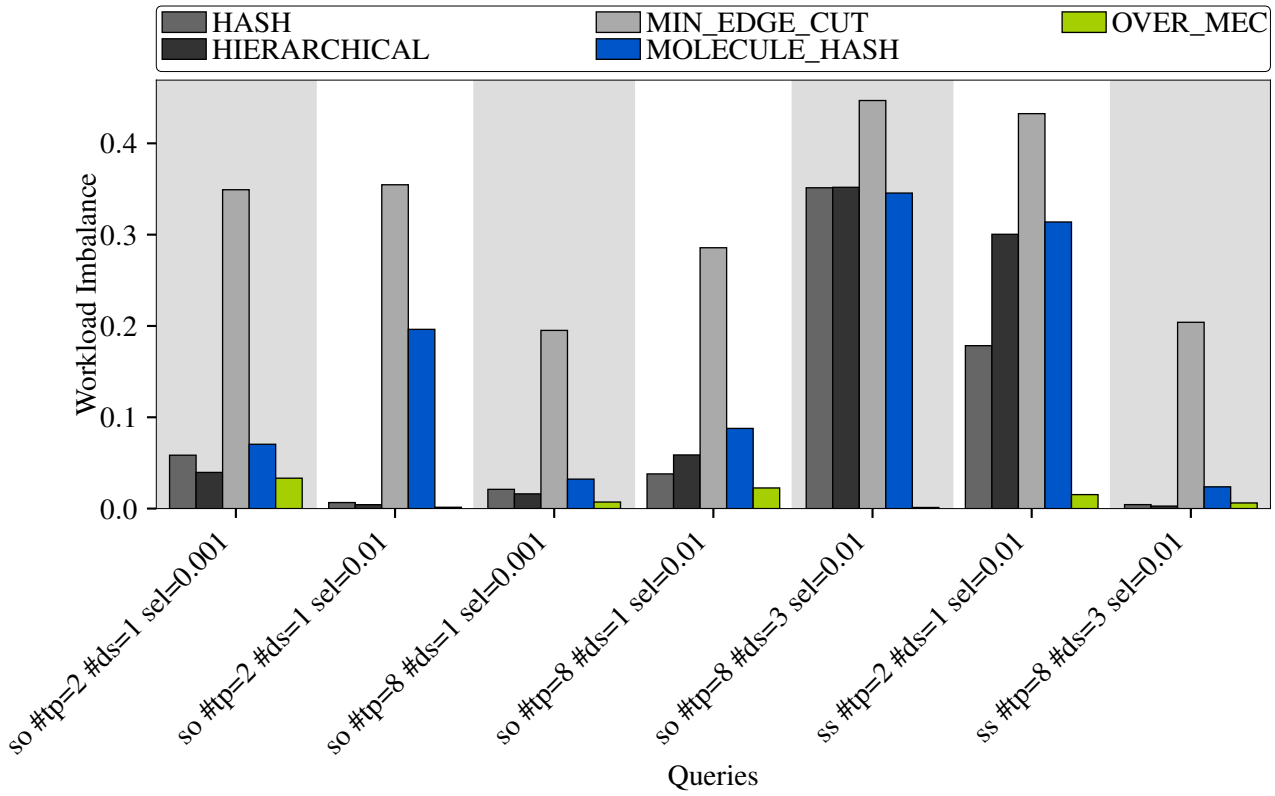


Figure 6.8.: Workload imbalance W for the BTC100M data set using 10 slaves.

Scaling Number of Slaves. As already described in Chapter 5, the workload imbalance increases when scaling up the number of slaves. For the molecule hash cover, the median increase is 29% when scaling to 20 slaves and 50% when scaling to 40 slaves. Nevertheless, for 40 slaves, the workload imbalance remains only slightly higher than the workload imbalance of the hash cover for 5 out of 7 queries. Only for two queries the workload imbalance is 0.08 and 1.8 higher.

For the overpartitioned minimal edge-cut cover, the median increase is 98% when scaling to 20 slaves and 267% when scaling to 40 slaves. As described earlier, this large workload imbalance increase is caused by the lower number of partitions into which the graph is split resulting in larger partitions. As a result, the workload imbalance of the overpartitioned minimal edge-cut becomes similar to the workload imbalance of the hash cover for 4 out of 7 queries. For 3 queries including both star-shaped queries the workload imbalance is even higher than the one of the hash cover.

Scaling Data Set Size. Similar to the behavior of the frequently used graph cover strategies, the workload imbalance of both proposed graph cover strategies decrease when scaling up the data set size. The median workload imbalance decrease of the molecule hash cover is 17% when scaling from BTC500M to BTC1000M and 10% when scaling from BTC1000M to BTC2000M. For the overpartitioned minimal edge-cut cover, the median workload imbalance decreases by 40% when scaling from BTC500M to BTC1000M and by 33% when scaling from BTC1000M to BTC2000M.

6.4. Lessons Learned

When analyzing the results, several expectations could be confirmed:

- The molecule hash cover needs a bit more time than the hierachical hash cover to be created but needs much less time than the minimal edge-cut cover. The overpartitioned minimal edge-cut cover needs the most time to be created.
- The overpartitioned minimal edge-cut cover has the most balanced graph chunk sizes. The sizes of the molecule hash chunks are less balanced than the hash based graph chunks but more balanced than the sizes of the minimal-edge-cut-based chunks.
- The molecule hash cover and the overpartitioned minimal edge-cut cover execute queries faster than the frequently used graph cover strategies. Thereby, the overpartitioned minimal edge-cut cover tends to be faster than the molecule hash cover.
- For purely star-shaped queries that do not need to transfer intermediate results, the molecule hash cover may increase the query execution times in comparison with the hash-based covers but is still faster than the minimal edge-cut cover.
- Scaling up the number of slaves reduces the horizontal containment of both proposed graph covers.
- The molecule hash cover has a slightly higher workload imbalance than the hash-based covers, independent of the number of slaves and the data set size.
- For both proposed graph cover strategies that collocate closely connected triples on the same compute node while balancing the number of stored triples over the compute nodes, the vertical parallelization seems to be better than for the frequently used graph cover strategies.

Beside the expected results, some surprising results could be observed:

- The molecule hash cover and the overpartitioned minimal edge-cut cover show the highest execution time reductions for queries that have a mixture of several different join patterns and path-shaped queries with a high diameter.
- The molecule hash cover and the overpartitioned minimal edge-cut cover have an even better horizontal containment than the minimal edge-cut cover.
- Scaling up the data set size improves the horizontal containment for both proposed graph cover.
- The overpartitioned minimal edge-cut cover has the most balanced query workload for 10 slaves. Scaling up the number of slaves decreases the workload imbalance, since the initial graph is split into fewer but larger partitions.

6.5. Discussion

In the performed evaluation, the performances of the molecule hash cover strategy and the overpartitioned minimal edge-cut cover strategy on a real-world and two synthetic data sets were investigated. Both strategies (i) split the graph into small sets of connected triples that are collocated on the same compute node and (ii) try to balance the number of stored triples on the different compute nodes. Their performances are compared with

the hash-based graph cover strategies that balance the number of stored triples without considering the graph structure. Additionally, their performances were also compared with the one of a minimal edge-cut cover that focuses on the graph structure to create large sets of connected triples.

In contrast to the hash cover, the minimal edge-cut cover has the disadvantage that the differently sized graph chunks lead to an unbalanced query workload. To overcome this drawback, the molecule hash cover relies on the equal distribution of the hash function and the overpartitioned minimal edge-cut cover uses a greedy algorithm to group the small partitions to equally-sized chunks. As a result, the query workloads are almost perfectly balanced among all compute nodes.

For most queries, the molecule hash cover could reduce the query execution times by an average of 41% in comparison to the hash cover. An even higher reduction of the query execution time by an average of 67% can be achieved by the overpartitioned minimal edge-cut cover for most queries. These speed-ups are caused by the up to 100% lower number of transferred packets. Both graph cover strategies showed their best performances for queries consisting of a mixture of different join types with at least 6 triple patterns. This indicates that the small sets of connected triples better align to the triple sets with which more complex queries match.

Based on the query execution times, the overpartitioned minimal edge-cut cover outperforms the molecule hash cover. Nevertheless, the drawback of the former graph cover strategy is its up to 12.6 times longer loading time in the evaluation. Furthermore, the implementation relies on METIS that could not create the minimal edge-cut partitioning for the WatDiv1000M data set with approximately 1.1 billion triples due to a too high main memory consumption. Since the used implementation of the molecule hash-cover relies on a disk-based algorithms, it was able to load all data sets used in the evaluation.

CHAPTER 7

Conclusion

This thesis starts with a survey of RDF stores in the cloud. Thereby, different types of RDF stores are identified and a summary is provided with regard to how researchers address the challenges of the cloud setting such as data placement, distributed query execution and fault tolerance. One type of RDF stores in the cloud are distributed RDF stores. This type of RDF stores is of special interests to this thesis because it allows for full control of the data placement and the distributed query processing. In order to speed up the query execution, the interdependencies of the data placement and the distributed query processing have to be understood.

To gain a deeper understanding of these interdependencies a methodology for benchmarking data placement strategies is developed in this thesis. With this methodology the frequently used data placement strategies are evaluated. This evaluation reveals that graph-clustering-based data placement strategies have a high horizontal containment — i.e., the individual query results can be created locally without the need to exchange intermediate results between compute nodes — but the overall query workload is not balanced across all compute nodes, leading to a low vertical parallelization. In contrast to these data placement strategies, distributing triples arbitrarily over the compute nodes as done by the hash-based data placement strategies tends to have a balanced query workload at the cost of a low horizontal containment, leading to a low to medium vertical parallelization. As a result the hash-based data placement strategies show a better overall query performance than the graph-clustering-based strategies. None of the frequently used data placement strategies have a high vertical parallelization in the performed experiments.

To find a data placement strategy with a high vertical parallelization, the thesis tests the hypothesis that collocate small connected triple sets on the same compute node while balancing the amount of triples stored on the different compute nodes can lead to a high vertical parallelization. Specifically, two data placement strategies are employed, namely, the overpartitioned minimal edge-cut cover and the molecule hash cover (the latter being a newly developed data placement strategy). The evaluation shows a balanced query workload and a high horizontal containment, which leads to a high vertical parallelization. As a result these strategies show a better query performance than the frequently used data placement strategies.

Outlook

As described in the previous section, data placement strategies that split the data into small sets of connected data items and subsequently group these sets to similarly sized data chunks may lead to faster query executions than the frequently used data placement strategies. The newly developed data placement strategy, namely, molecule hash cover, splits the data set arbitrary into small sets of connected data items and then distributes them arbitrarily over different compute nodes. Future work could investigate whether the performance of the data placement strategy can be improved by, e.g., collocating connected small data item sets on the same compute node instead of distributing them arbitrarily.

Furthermore, the number of data placement strategies that optimize the data placement based on historic query workloads is increasing. One such workload-aware data placement strategy is implemented in the dis-

tributed RDF store DiploCloud [157]. It splits the data set into small sets of data items that are required to create the individual query result of the historic queries. In the evaluation DiploCloud shows similar performances to TriAD [55] which uses the overpartitioned minimal edge-cut cover. Therefore, a potential extension of Koral is to further support the creation and evaluation of these workload-aware data placement strategies.

In the benchmarking methodology presented in this thesis, the used queries are generated to test different query characteristics. Nevertheless, the extent to which these queries reflect real query workloads is unclear. Therefore, future work is required to improve the benchmarking methodology by investigating real query logs such as the one of Wikidata [99] or the LSQ data set [132].

Finally, the presented benchmarking methodology focuses on the execution of individual queries with high workloads such as in use cases where analytical queries are requested against a data set. Nevertheless, RDF stores in the cloud are also used in settings in which many queries with mostly small workloads are requested against a system in parallel. For these settings, the query throughput is an important measure. To address these settings, the developed methodology can be adapted to additionally measure the query throughput and thereafter be applied to reevaluate the frequently used data placement strategies.

Bibliography

[1] Largetriplestores. https://www.w3.org/wiki/LargeTripleStores, accessed: 2018-07-10

[2] The bigdata® RDF Database. Retrieved: 29.10.2014, http://www.bigdata.com/whitepapers/bigdata_architecture_whitepaper.pdf

[3] Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases. pp. 411–422. VLDB '07, VLDB Endowment (2007), http://dl.acm.org/citation.cfm?id=1325851.1325900

[4] Abbassi, S., Faiz, R.: RDF-4X: A Scalable Solution for RDF Quads Store in the Cloud. In: Proceedings of the 8th International Conference on Management of Digital EcoSystems. pp. 231–236. MEDES, ACM, New York, NY, USA (2016). doi: 10.1145/3012071.3012104

[5] Abdelaziz, I., Harbi, R., Salihoglu, S., Kalnis, P.: Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics. IEEE Transactions on Parallel and Distributed Systems **28**(12), 3374–3388 (2017). doi: 10.1109/TPDS.2017.2720174

[6] Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) The Semantic Web – ISWC 2004. pp. 107–121. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[7] Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 18–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[8] Akar, Z., Halaç, T.G., Ekinci, E.E., Dikenelli, O.: Querying the Web of Interlinked Datasets using VOID Descriptions. In: WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012 (2012), http://ceur-ws.org/Vol-937/ldow2012-paper-06.pdf

[9] Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Adaptive Partitioning for Very Large RDF Data. CoRR **abs/1505.0** (2015), http://arxiv.org/abs/1505.02728

[10] Al-Harbi, R., Ebrahim, Y., Kalnis, P.: PHD-Store: An Adaptive SPARQL Engine with Dynamic Partitioning for Distributed RDF Repositories. CoRR **abs/1405.4** (2014), http://arxiv.org/abs/1405.4979

[11] Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing Linked Datasets with the VoID Vocabulary. W3C interest group note, W3C (2011), http://www.w3.org/TR/2011/NOTE-void-20110303/

[12] Ali, L., Janson, T., Lausen, G.: 3rdf: Storing and Querying RDF Data on Top of the 3nuts Overlay Network. In: 2011 22nd International Workshop on Database and Expert Systems Applications. pp. 257–261 (aug 2011). doi: 10.1109/DEXA.2011.1

[13] Ali, L., Janson, T., Schindelhauer, C.: Towards Load Balancing and Parallelizing of RDF Query Processing in P2P Based Distributed RDF Data Stores. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 307–311 (feb 2014). doi: 10.1109/PDP.2014.79

[14] Ali, L., Janson, T., Lausen, G., Schindelhauer, C.: Effects of Network Structure Improvement on Distributed RDF Querying. In: Hameurlain, A., Rahayu, W., Taniar, D. (eds.) Data Management in Cloud, Grid and P2P Systems. pp. 63–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[15] Aluç, G., Hartig, O., Özsu, M., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) The Semantic Web – ISWC 2014, Lecture Notes in Computer Science, vol. 8796, pp. 197–212. Springer International Publishing (2014). doi: 10.1007/978-3-319-11964-9_13

[16] Arenas, M., Pérez, J.: Federation and Navigation in SPARQL 1.1. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web. Semantic Technologies for Advanced Query Answering, Lecture Notes in Computer Science, vol. 7487, pp. 78–111. Springer Berlin Heidelberg (2012). doi: 10.1007/978-3-642-33158-9_3

[17] Basca, C., Bernstein, A.: Distributed SPARQL Throughput Increase: On the effectiveness of Workload-driven RDF partitioning. In: ISWC2013 (2013)

[18] Basca, C., Bernstein, A.: Querying a Messy Web of data with AVALANCHE. Web Semantics: Science, Services and Agents on the World Wide Web **26**(0) (2014), `http://www.websemanticsjournal.org/index.php/ps/article/view/361`

[19] Battré, D., Heine, F., Höing, A., Kao, O.: On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT Based RDF Stores. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.H., Ouksel, A.M. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 343–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

[20] Beame, P., Koutris, P., Suciu, D.: Skew in Parallel Query Processing. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. pp. 212–223. PODS '14, ACM, New York, NY, USA (2014). doi: 10.1145/2594538.2594558

[21] Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Syst. **5**(2), 1–24 (2009). doi: 10.4018/jswis.2009040101

[22] Böhm, C., Hefenbrock, D., Naumann, F.: Scalable Peer-to-peer-based RDF Management. In: Proceedings of the 8th International Conference on Semantic Systems. pp. 165–168. I-SEMANTICS '12, ACM, New York, NY, USA (2012). doi: 10.1145/2362499.2362523

[23] Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In: Advances in Social Networks Analysis and Mining (ASONAM). pp. 248–255 (Aug 2010). doi: 10.1109/ASONAM.2010.80

[24] Bugiotti, F., Camacho-Rodríguez, J., Goasdoué, F., Kaoudi, Z., Manolescu, I., Zampetakis, S.: SPARQL Query Processing in the Cloud. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management. Emerging Directions in Database Systems and Applications, Chapman and Hall/CRC (Apr 2014)

[25] Cai, M., Frank, M.: RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. Proceedings of the 13th International Conference on World Wide Web pp. 650–657 (2004), http://dl.acm.org/citation.cfm?id=988760

[26] Charalambidis, A., Troumpoukis, A., Konstantopoulos, S.: SemaGrow: Optimizing Federated SPARQL Queries. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 121–128. SEMANTICS '15, ACM, New York, NY, USA (2015). doi: 10.1145/2814864.2814886

[27] Cheng, L., Kotoulas, S.: Scale-Out Processing of Large RDF Datasets. IEEE Transactions on Big Data **1**(4), 138–150 (2015). doi: 10.1109/TBDATA.2015.2505719

[28] Chu, S., Balazinska, M., Suciu, D.: From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 63–78. SIGMOD '15, ACM, New York, NY, USA (2015). doi: 10.1145/2723372.2750545

[29] Cossu, M., Färber, M., Lausen, G.: PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies. In: Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018. pp. 469–472 (2018). doi: 10.5441/002/edbt.2018.49

[30] Crespo, A., Garcia-Molina, H.: Semantic Overlay Networks for P2P Systems. In: Moro, G., Bergamaschi, S., Aberer, K. (eds.) Agents and Peer-to-Peer Computing. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[31] Cudre-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An Infrastructure for Peer Information Management. IEEE Internet Computing **11**(5), 36–44 (sep 2007). doi: 10.1109/MIC.2007.108

[32] Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., Keppmann, F., Miranker, D., Sequeda, J., Wylot, M.: NoSQL Databases for RDF: An Empirical Evaluation. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013, Lecture Notes in Computer Science, vol. 8219, pp. 310–325. Springer Berlin Heidelberg (2013). doi: 10.1007/978-3-642-41338-4_20

[33] Curé, O., Naacke, H., Baazizi, M.A., Amann, B.: On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark. In: Proc. of the 11th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (at ISWC-2015). pp. 16–31 (2015)

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

[35] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. pp. 205–220. SOSP '07, ACM, New York, NY, USA (2007). doi: 10.1145/1294261.1294281

[36] Della Valle, E., Turati, A., Ghioni, A.: PAGE: A Distributed Infrastructure for Fostering RDF-Based Interoperability. In: Eliassen, F., Montresor, A. (eds.) Distributed Applications and Interoperable Systems. pp. 347–353. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[37] DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation Techniques for Main Memory Database Systems. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. pp. 1–8. SIGMOD '84, ACM, New York, NY, USA (1984). doi: 10.1145/602259.602261

[38] Dhraief, H., Kemper, A., Nejdl, W., Wiesner, C.: Processing and Optimization of Complex Queries in Schema-Based P2P-Networks. In: Ng, W.S., Ooi, B.C., Ouksel, A.M., Sartori, C. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[39] Ding, L., Peng, Y., da Silva, P.P., McGuinness, D.L.: Tracking RDF Graph Provenance using RDF Molecules. Tech. rep., UMBC (2005), https://ebiquity.umbc.edu/paper/html/id/240/Tracking-RDF-Graph-Provenance-using-RDF-Molecules

[40] Du, F., Bian, H., Chen, Y., Du, X.: Efficient SPARQL Query Evaluation in a Database Cluster. IEEE Int. Congress on Big Data pp. 165–172 (2013). doi: 10.1109/BigData.Congress.2013.30

[41] Erling, O., Mikhailov, I.: Towards Web Scale RDF. In: 4th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008) (2008)

[42] Erling, O., Mikhailov, I.: Virtuoso: RDF Support in a Native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer Berlin Heidelberg (2010). doi: 10.1007/978-3-642-04329-1_21

[43] Everett, N.: [wikidata-tech] wikidata query backend update (take two!). https://lists.wikimedia.org/pipermail/wikidata-tech/2015-March/000740.html (March 2015)

[44] Farhan Husain, M., McGlothlin, J., Masud, M.M., Khan, L., Thuraisingham, B.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. Knowledge and Data Engineering, IEEE Transactions on **23**(9), 1312–1327 (Sep 2011). doi: 10.1109/TKDE.2011.103

[45] Galarraga, L., Hose, K., Schenkel, R.: Partout: A Distributed Engine for Efficient RDF Processing. CoRR **abs/1212.5** (2012), http://arxiv.org/abs/1212.5636

[46] Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J.A., Zampetakis, S.: CliqueSquare: Flat plans for massively parallel RDF queries. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 771–782 (apr 2015). doi: 10.1109/ICDE.2015.7113332

[47] Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 599–613. OSDI'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2685048.2685096

[48] Goodman, E.L., Grunwald, D.: Using Vertex-centric Programming Platforms to Implement SPARQL Queries on Large Graphs. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. pp. 25–32. IA^3 '14, IEEE Press, Piscataway, NJ, USA (2014). doi: 10.1109/IA3.2014.10

[49] Görlitz, O., Thimm, M., Staab, S.: SPLODGE: Systematic generation of SPARQL benchmark queries for linked open data. The Semantic Web–ISWC 2012 pp. 116–132 (2012). doi: 10.1007/978-3-642-35176-1_8

[50] Görlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data - Volume 782. pp. 13–24. COLD'11, CEUR-WS.org, Aachen, Germany, Germany (2010), http://dl.acm.org/citation.cfm?id=2887352.2887354

[51] Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: A Multi-Criteria Experimental Ranking of Distributed SPARQL Evaluators (2016), https://hal.inria.fr/hal-01381781

[52] Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part II, pp. 80–87. Springer International Publishing, Cham (2016). doi: 10.1007/978-3-319-46547-0_9

[53] Gunther, N.J.: A simple capacity model of massively parallel transaction systems. In: 19. International Computer Measurement Group Conference, San Diego, CA, USA, December 5-10, 1993 (1993)

[54] Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2-3) (2005), http://www.websemanticsjournal.org/index.php/ps/article/view/70

[55] Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: SIGMOD. pp. 289–300 (2014). doi: 10.1145/2588555.2610511

[56] Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of Semantic Web Databases. In: PODS. pp. 95–106. ACM (2004). doi: 10.1145/1055558.1055573

[57] Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries Across Diverse Data Sources. In: Vldb. VLDB '97, vol. Athens, Gr, pp. 276–285. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)

[58] Hammoud, M., Rabbou, D.A., Nouri, R., Beheshti, S.M.R., Sakr, S.: DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. Proc. VLDB Endow. **8**(6), 654–665 (2015). doi: 10.14778/2735703.2735705

[59] Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N.: Evaluating SPARQL Queries on Massive RDF Datasets. PVLDB **8**(12), 1848–1851 (2015), http://www.vldb.org/pvldb/vol8/p1848-harbi.pdf

[60] Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. The VLDB Journal **25**(3), 355–380 (jun 2016). doi: 10.1007/s00778-016-0420-y

[61] Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Scalable Semantic Web Knowledge Base Systems - SSWS2009. pp. 94–109 (2009)

[62] Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: Proc. of LA-WEB '05. pp. 71—-. IEEE (2005). doi: 10.1109/LAWEB.2005.25

[63] Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC-2007, vol. 4825, pp. 211–224. Springer Berlin Heidelberg (2007). doi: 10.1007/978-3-540-76298-0_16

[64] Hong, S., Depner, S., Manhardt, T., Van Der Lugt, J., Verstraaten, M., Chafi, H.: PGX.D: A Fast Distributed Graph Processing Engine. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 58:1—-58:12. SC '15, ACM, New York, NY, USA (2015). doi: 10.1145/2807591.2807620

[65] Hose, K., Schenkel, R.: WARP: Workload-aware replication and partitioning for RDF. In: Data Engineering Workshops (ICDEW). pp. 1–6 (Apr 2013). doi: 10.1109/ICDEW.2013.6547414

[66] Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB **4**(11), 1123–1134 (2011)

[67] Janke, D., Skubella, A., Staab, S.: Evaluating sparql 1.1 property path support. In: Usbeck, R., Ngonga, A., Kim, J.D., Choi, K.S., Cimiano, P., Fundulaki, I., Krithara, A. (eds.) Joint Proceedings of BLINK2017: Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data (BLINK2017-NLIWoD3). No. 1932 in CEUR Workshop Proceedings, Aachen (2017), http://ceur-ws.org/Vol-1932/paper-04.pdf

[68] Janke, D., Staab, S.: Storing and Querying Semantic Data in the Cloud. In: D'Amato, C., Theobald, M. (eds.) Reasoning Web. Learning, Uncertainty, Streaming, and Scalability: 14th International Summer School 2018, Esch-sur-Alzette, Luxembourg, September 22–26, 2018, Tutorial Lectures. pp. 173–222. Springer International Publishing, Cham (2018). doi: 10.1007/978-3-030-00338-8_7

[69] Janke, D., Staab, S., Thimm, M.: Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores. Tech. rep., WeST (2016), https://owncloud.uni-koblenz-landau.de/owncloud/s/5c25skgCkkDgxyQ

[70] Janke, D., Staab, S., Thimm, M.: Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In: Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017. (2017), http://ceur-ws.org/Vol-1932/paper-05.pdf

[71] Janke, D., Staab, S., Thimm, M.: Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In: Nikitina, N., Song, D., Fokoue, A., Haase, P. (eds.) ISWC 2017 Posters & Demonstrations and Industry Tracks. No. 2963 in CEUR Workshop Proceedings, Aachen (2017), http://ceur-ws.org/Vol-1963/paper489.pdf

[72] Janke, D., Staab, S., Thimm, M.: On Data Placement Strategies in Distributed RDF Stores. In: Proceedings of The International Workshop on Semantic Big Data. pp. 1:1–1:6. SBD '17, ACM, New York, NY, USA (2017). doi: 10.1145/3066911.3066915

[73] Janke, D., Staab, S., Thimm, M.: Impact analysis of data placement strategies on query efforts in distributed RDF stores. Journal of Web Semantics **50**, 21 – 48 (2018). doi: 10.1016/j.websem.2018.02.002, http://www.websemanticsjournal.org/index.php/ps/article/view/516

[74] Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The performance of mapreduce: An in-depth study. PVLDB **3**(1), 472–483 (2010), http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/E03.pdf

[75] Jones, N.D.: An Introduction to Partial Evaluation. ACM Comput. Surv. **28**(3), 480–503 (sep 1996). doi: 10.1145/243439.243447

[76] Käfer, T., Harth, A.: Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2014/ (2014)

[77] Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In: 2009 Ninth IEEE International Conference on Data Mining. pp. 229–238 (2009). doi: 10.1109/ICDM.2009.14

[78] Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, updating and querying RDF(S) data on top of DHTs. Web Semantics: Science, Services and Agents on the World Wide Web **8**(4) (2010), http://www.websemanticsjournal.org/index.php/ps/article/view/250

[79] Karnstedt, M., Sattler, K.U., Richtarsky, M., Muller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: Querying a DHT-based Universal Storage. In: 2007 IEEE 23rd International Conference on Data Engineering. pp. 1503–1504 (apr 2007). doi: 10.1109/ICDE.2007.369054

[80] Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998). doi: 10.1137/S1064827595287997

[81] Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.M., Castagna, P.: Jena-HBase: A Distributed, Scalable and Effcient RDF Triple Store. Tech. rep., Department of Computer Science at The University of Texas at Dallas (2012)

[82] Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.M., Castagna, P.: Jena-HBase: A Distributed, Scalable and Effcient RDF Triple Store. In: Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012 (2012), http://ceur-ws.org/Vol-914/paper_14.pdf

[83] Kim, H., Ravindra, P., Anyanwu, K.: From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. PVLDB **4**(12), 1426–1429 (2011), http://www.vldb.org/pvldb/vol4/p1426-kim.pdf

[84] Kiryakov, A., Bishop, B., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: The Features of BigOWLIM that Enabled the BBC's World Cup Website. In: Workshop on Semantic Data Management (SemData@VLDB 2010) (Sep 2010)

[85] Kokkinidis, G., Christophides, V.: Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In: Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) Current Trends in Database Technology - EDBT 2004 Workshops. pp. 486–495. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[86] Kotsev, V., Kiryakov, A., Fundulaki, I., Alexiev, V.: LDBC Semantic Publishing Benchmark (SPB) - v2.0 First Public Draft Release. Tech. rep., The Linked Data Benchmark Council (June 2014), `https://github.com/ldbc/ldbc_spb_bm_2.0/blob/master/doc/LDBC_SPB_v2.0.docx?raw=true`

[87] Ladwig, G., Harth, A.: CumulusRDF: Linked Data Management on Nested Key-Value Stores. In: Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC2011) (Oct 2011)

[88] Ladwig, G., Tran, T.: SIHJoin: Querying Remote and Local Linked Data. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) The Semantic Web: Research and Applications. pp. 139–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[89] Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (apr 2010). doi: 10.1145/1773912.1773922

[90] Le-Phuoc, D., Nguyen Mau Quoc, H., Le Van, C., Hauswirth, M.: Elastic and Scalable Processing of Linked Stream Data in the Cloud. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013. pp. 280–297. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[91] Lee, K., Liu, L.: Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud. In: Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis. pp. 46:1—-46:12. ACM (2013). doi: 10.1145/2503210.2503302

[92] Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB **6**(14), 1894–1905 (Sep 2013). doi: 10.14778/2556549.2556571

[93] Lee, K., Liu, L., Tang, Y., Zhang, Q., Zhou, Y.: Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud. In: IEEE CLOUD '13. pp. 327–334 (2013). doi: 10.1109/CLOUD.2013.63

[94] Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Statistical Properties of Community Structure in Large Social and Information Networks. In: Proceedings of the 17th International Conference on World Wide Web. pp. 695–704. WWW '08, ACM, New York, NY, USA (2008). doi: 10.1145/1367497.1367591

[95] Liarou, E., Idreos, S., Koubarakis, M.: Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) The Semantic Web - ISWC 2006. pp. 399–413. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[96] Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints. In: Meersman, R., Dillon, T., Herrero, P., Kumar, A., Reichert, M., Qing, L., Ooi, B.C., Damiani, E., Schmidt, D.C., White, J., Hauswirth, M., Hitzler, P., Mohania, M. (eds.) On the Move to Meaningful Internet Systems: OTM 2011. pp. 808–817. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[97] Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International

Conference on Management of Data. pp. 135–146. SIGMOD '10, ACM, New York, NY, USA (2010). doi: 10.1145/1807167.1807184

[98] Malliaros, F.D., Vazirgiannis, M.: Clustering and community detection in directed networks: A survey. Physics Reports **533**(4), 95–142 (2013). doi: 10.1016/j.physrep.2013.08.002

[99] Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In: Vrandečić, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L.A., Simperl, E. (eds.) Proceedings of the 17th International Semantic Web Conference (ISWC'18). LNCS, vol. 11137, pp. 376–394. Springer (2018)

[100] Mansour, E., Abdelaziz, I., Ouzzani, M., Aboulnaga, A., Kalnis, P.: A Demonstration of Lusail: Querying Linked Data at Scale. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1603–1606. SIGMOD '17, ACM, New York, NY, USA (2017). doi: 10.1145/3035918.3058731

[101] Matono, A., Pahlevi, S.M., Kojima, I.: RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.H., Ouksel, A.M. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 323–330. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

[102] McMurry, J., Jupp, S., Malone, J., Burdett, T., Jenkinson, A., Parkinson, H., Davies, M., Brandizi, M., et al.: Report on the scalability of semantic web integration in biomedbridges (2015). doi: 10.5281/zenodo.14071

[103] Mishra, P., Eich, M.H.: Join Processing in Relational Databases. ACM Comput. Surv. **24**(1), 63–113 (1992). doi: 10.1145/128762.128764

[104] Montoya, G., Skaf-Molli, H., Hose, K.: The Odyssey Approach for Optimizing Federated SPARQL Queries. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I. pp. 471–489 (2017). doi: 10.1007/978-3-319-68288-4_28

[105] Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Federated SPARQL Queries Processing with Replicated Fragments. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., D'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015, pp. 36–51. Springer International Publishing, Cham (2015)

[106] Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Decomposing Federated Queries in presence of Replicated Fragments. Web Semantics: Science, Services and Agents on the World Wide Web **42**(1) (2017), http://www.websemanticsjournal.org/index.php/ps/article/view/486

[107] Montoya, G., Vidal, M.E., Acosta, M.: A Heuristic-based Approach for Planning Federated SPARQL Queries. In: Proceedings of the Third International Conference on Consuming Linked Data - Volume 905. pp. 63–74. COLD'12, CEUR-WS.org, Aachen, Germany, Germany (2012), http://dl.acm.org/citation.cfm?id=2887367.2887373

[108] Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 454–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[109] Mutharaju, R., Sakr, S., Sala, A., Hitzler, P.: D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine. In: ISWC (Posters & Demos)'13. pp. 261–264 (2013)

[110] Naacke, H., Amann, B., Curé, O.: SPARQL Graph Pattern Processing with Apache Spark. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems. pp. 1:1—-1:7. GRADES'17, ACM, New York, NY, USA (2017). doi: 10.1145/3078447.3078448

[111] Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Löser, A.: Super-peer-based Routing and Clustering Strategies for RDF-based Peer-to-peer Networks. In: Proceedings of the 12th International Conference on World Wide Web. pp. 536–543. WWW '03, ACM, New York, NY, USA (2003). doi: 10.1145/775152.775229

[112] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-so-foreign Language for Data Processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 1099–1110. SIGMOD '08, ACM, New York, NY, USA (2008). doi: 10.1145/1376616.1376726

[113] Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(4) (2009), `http://www.websemanticsjournal.org/index.php/ps/article/view/173`

[114] Osorio, M., Aranda, C.B.: Storage Balancing in P2P Based Distributed RDF Data Stores. In: Proceedings of the Workshop on Decentralizing the Semantic Web 2017 co-located with 16th International Semantic Web Conference (ISWC 2017) (2017), `http://ceur-ws.org/Vol-1934/contribution-04.pdf`

[115] Owens, A., Seaborne, A., Gibbins, N., McSchraefel: Clustered TDB: A Clustered Triple Store for Jena (Nov 2008), `http://eprints.soton.ac.uk/266974/`, `http://eprints.soton.ac.uk/266974/`

[116] Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H2RDF+: An Efficient Data Management System for Big RDF Graphs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 909–912. SIGMOD '14, ACM, New York, NY, USA (2014). doi: 10.1145/2588555.2594535

[117] Peng, P., Zou, L., Chen, L., Zhao, D.: Query Workload-based RDF Graph Fragmentation and Allocation. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016. pp. 377–388 (2016). doi: 10.5441/002/edbt.2016.35

[118] Peng, P., Zou, L., Özsu, M.T., Chen, L., Zhao, D.: Processing SPARQL Queries over Distributed RDF Graphs. The VLDB Journal **25**(2), 243–268 (apr 2016). doi: 10.1007/s00778-015-0415-0

[119] Penteado, R.R.M., Scroeder, R., Hara, C.S.: Exploring Controlled RDF Distribution. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 160–167 (2016). doi: 10.1109/CloudCom.2016.0038

[120] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1—-16:45 (Sep 2009). doi: 10.1145/1567274.1567278

[121] Potter, A., Motik, B., Horrocks, I.: Querying Distributed RDF Graphs: The Effects of Partitioning. In: Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014). pp. 29–44 (2014)

[122] Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Distributed RDF Query Answering with Dynamic Data Exchange. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I, pp. 480–497. Springer International Publishing, Cham (2016). doi: 10.1007/978-3-319-46523-4_29

[123] Prud'hommeaux, E., Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation, W3C (2013), http://www.w3.org/TR/sparql11-query/

[124] Przyjaciel-Zablocki, M., Schätzle, A., Lausen, G.: TriAL-QL: Distributed Processing of Navigational Queries. In: Proceedings of the 18th International Workshop on Web and Databases. pp. 48–54. WebDB'15, ACM, New York, NY, USA (2015). doi: 10.1145/2767109.2767115

[125] Przyjaciel-Zablocki, M., Schätzle, A., Lausen, G.: Querying Semantic Knowledge Bases with SQL-on-Hadoop. In: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. pp. 4:1—-4:10. BeyondMR'17, ACM, New York, NY, USA (2017). doi: 10.1145/3070607.3070610

[126] Pujol, J.M., Erramilli, V., Rodriguez, P.: Divide and Conquer: Partitioning Online Social Networks. CoRR **abs/0905.4** (2009), http://arxiv.org/abs/0905.4918

[127] Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: A Scalable RDF Triple Store for the Clouds. In: 1st Int. Workshop on Cloud Intelligence. pp. 4:1–4:8. ACM (2012). doi: 10.1145/2347673.2347677

[128] Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) The Semantic Web: Research and Applications. pp. 524–538. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[129] Rohloff, K., Schantz, R.E.: High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In: Programming Support Innovations for Emerging Distributed Applications. pp. 4:1—-4:5. PSI EtA '10, ACM, New York, NY, USA (2010). doi: 10.1145/1940747.1940751

[130] Russell, J.: Getting Started with Impala: Interactive SQL for Apache Hadoop. O'Reilly Media (2014), http://shop.oreilly.com/product/0636920033936.do

[131] Sakr, S., Wylot, M., Mutharaju, R., Le Phuoc, D., Fundulaki, I.: Linked Data: Storing, Querying, and Reasoning. Springer International Publishing, Cham, 1 edn. (2018). doi: 10.1007/978-3-319-73515-3

[132] Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.C.N.: LSQ: The Linked SPARQL Queries Dataset. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015. pp. 261–269. Springer International Publishing, Cham (2015)

[133] Saleem, M., Mehmood, Q., Ngonga Ngomo, A.C.: FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., D'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015. pp. 52–69. Springer International Publishing, Cham (2015)

[134] Saleem, M., Ngonga Ngomo, A.C., Xavier Parreira, J., Deus, H.F., Hauswirth, M.: DAW: Duplicate-AWare Federated Query Processing over the Web of Data. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I, pp. 574–590. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). doi: 10.1007/978-3-642-41335-3_36

[135] Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., Lausen, G.: S2X: Graph-Parallel Querying of RDF with GraphX. In: Wang, F., Luo, G., Weng, C., Khan, A., Mitra, P., Yu, C. (eds.) Biomedical Data Management and Graph Online Querying. pp. 155–168. Springer International Publishing, Cham (2016)

[136] Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management. pp. 4:1—-4:8. SWIM '11, ACM, New York, NY, USA (2011). doi: 10.1145/1999299.1999303

[137] Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) The Semantic Web – ISWC 2014, Lecture Notes in Computer Science, vol. 8796, pp. 164–179. Springer International Publishing (2014). doi: 10.1007/978-3-319-11964-9_11

[138] Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF Querying with SPARQL on Spark. PVLDB **9**(10), 804–815 (2016), `http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf`

[139] Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 585–600. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[140] Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP2Bench: A SPARQL Performance Benchmark. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management: A Model-Based Perspective, pp. 371–393. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). doi: 10.1007/978-3-642-04329-1_16

[141] Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 601–616. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[142] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10 (2010). doi: 10.1109/MSST.2010.5496972

[143] Skubella, A., Janke, D., Staab, S.: Beseppi: Semantic-based benchmarking of property path implementations. In: The Semantic Web - 15th International Conference (06/06/19) (June 2019), https://eprints.soton.ac.uk/429356/

[144] Stein, R., Zacharias, V.: RDF on Cloud Number Nine. In: Ceri, S., Valle, E.D., Hendler, J., Huang, Z. (eds.) Proceedings of the 4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable & Dynamic. CEUR Workshop Proceedings (2010)

[145] Stevens, W.R.: TCP/IP Illustrated (Vol. 1): The Protocols. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1993), http://www.pcvr.nl/tcpip/

[146] Stutz, P., Verman, M., Fischer, L., Bernstein, A.: TripleRush: a fast and scalable triple store. In: 9th International Workshop on Scalable Semantic Web Knowledge Base Systems. CEUR Workshop Proceedings, http://ceur-ws.org, Aachen, Germany (2013)

[147] Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) The Semantic Web – ISWC 2010. pp. 764–780. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[148] Stutz, P., Paudel, B., Verman, M., Bernstein, A.: Random Walk TripleRush: Asynchronous Graph Querying and Sampling. In: Proceedings of the 24th International Conference on World Wide Web. pp. 1034–1044. WWW '15, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2015). doi: 10.1145/2736277.2741687

[149] Vidal, M.E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries, pp. 228–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). doi: 10.1007/978-3-642-13486-9_16

[150] Wang, R., Chiu, K.: Optimizing Distributed RDF Triplestores via a Locally Indexed Graph Partitioning. In: Parallel Processing (ICPP), 2012 41st International Conference on. pp. 259–268 (Sep 2012). doi: 10.1109/ICPP.2012.47

[151] Wang, X., Tiropanis, T., Davis, H.C.: LHD: Optimising Linked Data Query Processing Using Parallelisation. In: Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013 (2013), http://ceur-ws.org/Vol-996/papers/ldow2013-paper-06.pdf

[152] White, T.: Hadoop: The Definitive Guide. O'Reilly, Beijing, 4 edn. (2015), https://www.safaribooksonline.com/library/view/hadoop-the-definitive/9781491901687/

[153] Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases **1**(1), 103–128 (jan 1993). doi: 10.1007/BF01277522

[154] Wood, D., Gearon, P., Adams, T.: Kowari: A platform for semantic web storage and analysis. In: In XTech 2005 Conference. pp. 05–0402 (2005)

[155] Wu, B., Zhou, Y., Yuan, P., Liu, L., Jin, H.: Scalable SPARQL querying using path partitioning. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 795–806 (apr 2015). doi: 10.1109/ICDE.2015.7113334

[156] Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SemStore: A Semantic-Preserving Distributed RDF Triple Store. In: CIKM-2014 (2014)

[157] Wylot, M., Cudré-Mauroux, P.: Diplocloud: Efficient and scalable management of rdf data in the cloud. IEEE Transactions on Knowledge and Data Engineering **28**(3), 659–674 (2016). doi: 10.1109/TKDE.2015.2499202

[158] Xu, Z., Chen, W., Gai, L., Wang, T.: SparkRDF: In-Memory Distributed RDF Management Framework for Large-Scale Social Data. In: Dong, X.L., Yu, X., Li, J., Sun, Y. (eds.) Web-Age Information Management. pp. 337–349. Springer International Publishing, Cham (2015)

[159] Yang, S., Yan, X., Zong, B., Khan, A.: Towards Effective Partition Management for Large Graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 517–528. SIGMOD '12, ACM, New York, NY, USA (2012). doi: 10.1145/2213836.2213895

[160] Yang, T., Chen, J., Wang, X., Chen, Y., Du, X.: Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In: Meng, W., Feng, L., Bressan, S., Winiwarter, W., Song, W. (eds.) Database Systems for Advanced Applications. pp. 244–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[161] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. p. 10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), `http://dl.acm.org/citation.cfm?id=1863103.1863113`

[162] Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. PVLDB **6**(4), 265–276 (Feb 2013). doi: 10.14778/2535570.2488333

[163] Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In: ICDE-2013. pp. 565–576 (Apr 2013). doi: 10.1109/ICDE.2013.6544856

[164] Zhang, X., Chen, L., Wang, M.: Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In: Ailamaki, A., Bowers, S. (eds.) Scientific and Statistical Database Management, Lecture Notes in Computer Science, vol. 7338, pp. 250–259. Springer Berlin Heidelberg (2012). doi: 10.1007/978-3-642-31235-9_16

APPENDIX A

Examples of Distributed Query Execution

A.1. Example of Distributed Query Execution without Triple Replication

In the example setting we assume that there exists three compute nodes c_1, c_2 and c_3. Therefore the set of all compute nodes C is $C = \{c_1, c_2, c_3\}$. The compute nodes have the order $c_1 < c_2 < c_3$.

A graph cover called cov distributes a graph G over all three computes nodes. The resulting chunks are:

$$\begin{aligned}
\text{chunk}_{\text{cov}}(c_1) &= \{(1,a,2),(1,b,2),(2,c,5),(2,c,6),(2,c,7)\} \\
\text{chunk}_{\text{cov}}(c_2) &= \{(2,a,3),(2,b,3),(3,b,2)\} \\
\text{chunk}_{\text{cov}}(c_3) &= \{(2,a,1)\}
\end{aligned}$$

Thereby, $1,2,3,5,6,7$ represent resources at the subject or object position whereas a,b,c represent resources at the property position.

While loading the graph, the following join responsibilities will be defined based on the number of occurrences at the subject position in the different chunks. If a resource does not occur as a subject in the complete graph G then the number of its occurrences at the object and thereafter at the property position is chosen. One special case is `a`. `a` occurs in each chunk only once as a property. In this case the smallest compute node c_1 on which it occurs is chosen.

$$\begin{aligned}
\text{jResp}(1) &= c_1 \\
\text{jResp}(2) &= c_1 \\
\text{jResp}(3) &= c_2 \\
\text{jResp}(5) &= c_1 \\
\text{jResp}(6) &= c_1 \\
\text{jResp}(7) &= c_1 \\
\text{jResp}(a) &= c_1 \\
\text{jResp}(b) &= c_2 \\
\text{jResp}(c) &= c_1
\end{aligned}$$

The following query Q should be executed with a bushy query execution tree.

```
SELECT ?X, ?Y, ?Z, ?W WHERE {
   ?X a ?Y.
   ?Y b ?Z.
   ?Z b ?W.
   ?W a ?X
}
```

The bushy execution tree for query Q will result in the query execution tree shown in Figure A.1. Each compute node will execute the complete query execution tree. Additionally, the variables have the order `?X` $<$ `?Y` $<$ `?Z` $<$ `?W`.

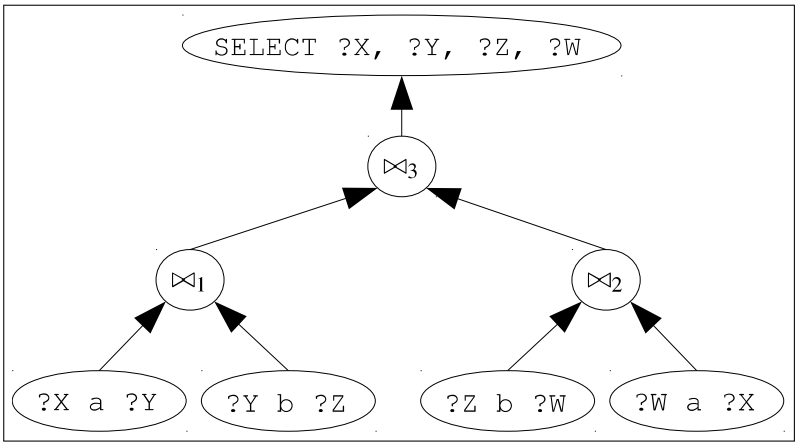


Figure A.1.: Query execution tree of example query Q.

Processing of ⟪`?X a ?Y`⟫

Each compute node matches the triple pattern `?X a ?Y` on its locally stored chunk. The results are visualized in Figure A.2.
Compute node c_1 computes $[\![\texttt{?X a ?Y}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?X},1),(\texttt{?Y},2)\}\}$
Compute node c_2 computes $[\![\texttt{?X a ?Y}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?X},2),(\texttt{?Y},3)\}\}$
Compute node c_3 computes $[\![\texttt{?X a ?Y}]\!]^{c_3}_{\text{cov}} = \{\{(\texttt{?X},2),(\texttt{?Y},1)\}\}$

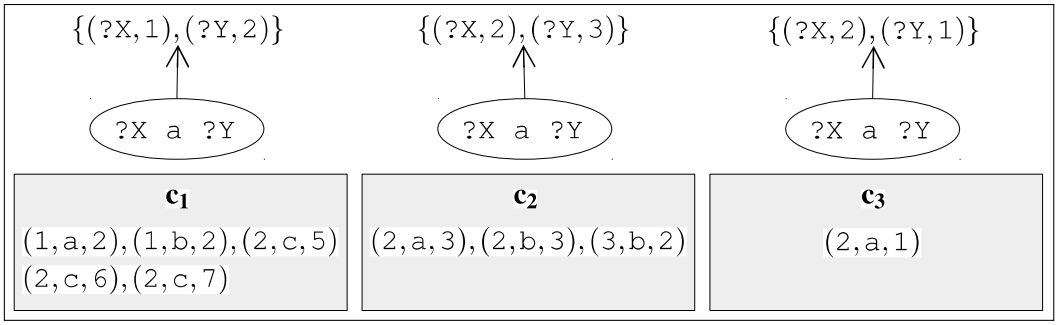


Figure A.2.: Results of evaluating `?X a ?Y` on the different compute nodes.

Processing of ⟪`?Y b ?Z`⟫

Each compute node matches the triple pattern `?Y b ?Z` on its locally stored chunk. The results are visualized in Figure A.3.
Compute node c_1 computes $[\![\texttt{?Y b ?Z}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?Y},1),(\texttt{?Z},2)\}\}$
Compute node c_2 computes $[\![\texttt{?Y b ?Z}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{(\texttt{?Y},3),(\texttt{?Z},2)\}\}$
Compute node c_3 computes $[\![\texttt{?Y b ?Z}]\!]^{c_3}_{\text{cov}} = \{\}$

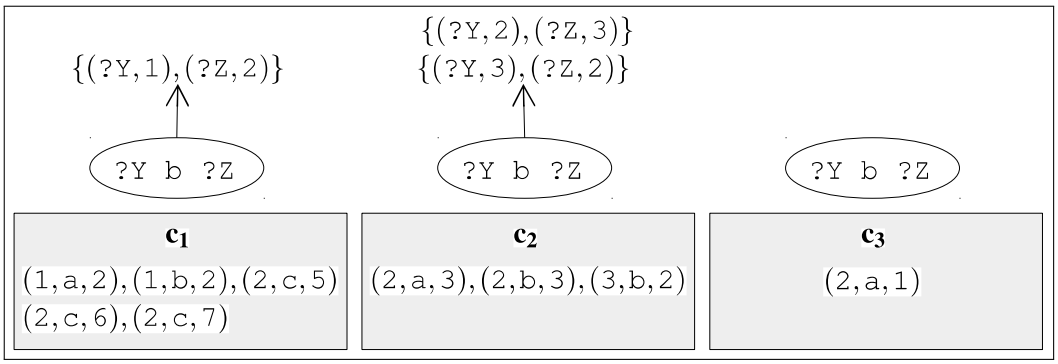


Figure A.3.: Results of evaluating `?Y b ?Z` on the different compute nodes.

Processing of ⟪⟪?X a ?Y⟫.⟪?Y b ?Z⟫⟫

In order to join the results of the triple patterns `?X a ?Y` and `?Y b ?Z` the join variables need to be identified first. In this case the join variables are $\text{cVars}(\langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle) = \{\texttt{?Y}\}$.

One difficulty of distributed joins is to compute the complete set of results. To ensure this, all variable bindings that assign the same resource to the join variable needs to be transferred to the same compute node where they will be joined. The compute node on which a specific resource will be joined is defined by its join responsibility jResp. For instance, the variable binding $\mu = \{(\texttt{?X},1),(\texttt{?Y},2)\}$ will be joined on compute node $\text{jResp}(\mu(\texttt{?Y})) = \text{jResp}(2) = c_1$.

Reminder: The join responsibilities of `1`, `2` and `3` are $\text{jResp}(1) = c_1$, $\text{jResp}(2) = c_1$ and $\text{jResp}(3) = c_1$.

Transferring the Results of the Evaluation of `?X a ?Y`

The results of $[\![\texttt{?X a ?Y}]\!]^{c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_1}_{\text{cov}}) &= \{\{(\texttt{?X},1),(\texttt{?Y},2)\}\} \\
\text{route}(c_2, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_1}_{\text{cov}}) &= \{\} \\
\text{route}(c_3, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?X a ?Y}]\!]^{c_2}_{\text{cov}}$ produced on compute node c_2 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_2}_{\text{cov}}) &= \{\} \\
\text{route}(c_2, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_2}_{\text{cov}}) &= \{\{(\texttt{?X},2),(\texttt{?Y},3)\}\} \\
\text{route}(c_3, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_2}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?X a ?Y}]\!]^{c_3}_{\text{cov}}$ produced on compute node c_3 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_3}_{\text{cov}}) &= \{\{(\texttt{?X},2),(\texttt{?Y},1)\}\} \\
\text{route}(c_2, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_3}_{\text{cov}}) &= \{\} \\
\text{route}(c_3, \langle\langle\langle\langle \texttt{?X a ?Y}\rangle\rangle.\langle\langle \texttt{?Y b ?Z}\rangle\rangle\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c_3}_{\text{cov}}) &= \{\}
\end{aligned}$$

Transferring the Results of the Evaluation of `?Y b ?Z`

The results of $[\![\texttt{?Y b ?Z}]\!]^{c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_1}_{\text{cov}}) &= \{\{(\texttt{?Y},1),(\texttt{?Z},2)\}\} \\
\text{route}(c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_1}_{\text{cov}}) &= \{\} \\
\text{route}(c_3, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Y b ?Z}]\!]^{c_2}_{\text{cov}}$ produced on compute node c_2 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_2}_{\text{cov}}) &= \{\{(\texttt{?Y},2),(\texttt{?Z},3)\}\} \\
\text{route}(c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_2}_{\text{cov}}) &= \{\{(\texttt{?Y},3),(\texttt{?Z},2)\}\} \\
\text{route}(c_3, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_2}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Y b ?Z}]\!]^{c_3}_{\text{cov}}$ produced on compute node c_3 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}(c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_3}_{\text{cov}}) &= \{\} \\
\text{route}(c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_3}_{\text{cov}}) &= \{\} \\
\text{route}(c_3, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^{c_3}_{\text{cov}}) &= \{\}
\end{aligned}$$

Figure A.4 shows the result after transferring all variable bindings.

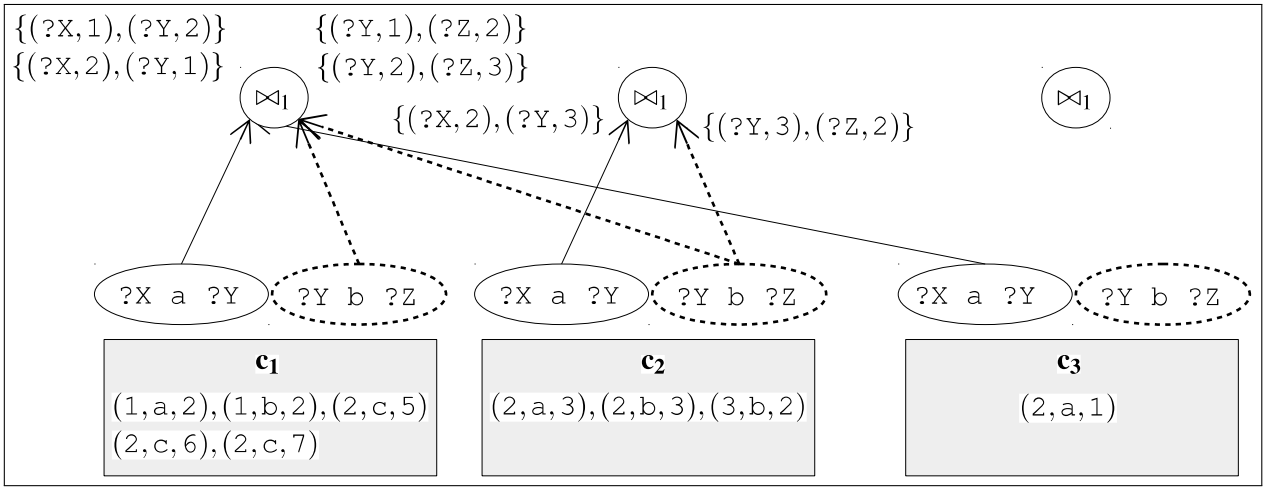


Figure A.4.: Results of transferring the variable bindings according to their join responsibility.

Joining the Results of the Evaluations of `?X a ?Y` *and* `?Y b ?Z`

Compute node c_1 has received the following results of the evaluation of `?X a ?Y` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{c}_{\text{cov}}) = \{\{(\texttt{?X},1),(\texttt{?Y},2)\},\{(\texttt{?X},2),(\texttt{?Y},1)\}\}$$

Furthermore, compute node c_1 received the following results of the evaluation of `?Y b ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{(\texttt{?Y},2),(\texttt{?Z},3)\}\}$$

The join of both variable binding sets results in:
$[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_1}_{\text{cov}}$
$= \{\{(\texttt{?X},1),(\texttt{?Y},2)\},\{(\texttt{?X},2),(\texttt{?Y},1)\}\} \bowtie \{\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{(\texttt{?Y},2),(\texttt{?Z},3)\}\}$
$= \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{(\texttt{?X},2),(\texttt{?Y},1),(\texttt{?Z},2)\}\}$

Compute node c_2 has received the following results of the evaluation of `?X a ?Y` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?X a ?Y}]\!]^c_{\text{cov}}) = \{\{(\texttt{?X},2),(\texttt{?Y},3)\}\}$$

Furthermore, compute node c_3 received the following results of the evaluation of `?Y b ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y b ?Z} \rangle\rangle, [\![\texttt{?Y b ?Z}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Y},3),(\texttt{?Z},2)\}\}$$

The join of both variable binding sets results in:
$[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?X},2),(\texttt{?Y},3)\}\} \bowtie \{\{(\texttt{?Y},3),(\texttt{?Z},2)\}\} = \{\{(\texttt{?X},2),(\texttt{?Y},3),(\texttt{?Z},2)\}\}$

Compute node c_3 has not received any variable bindings. Therefore, $[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_3}_{\text{cov}} = \{\}$.
Figure A.5 shows the variable bindings resulting of the joins.

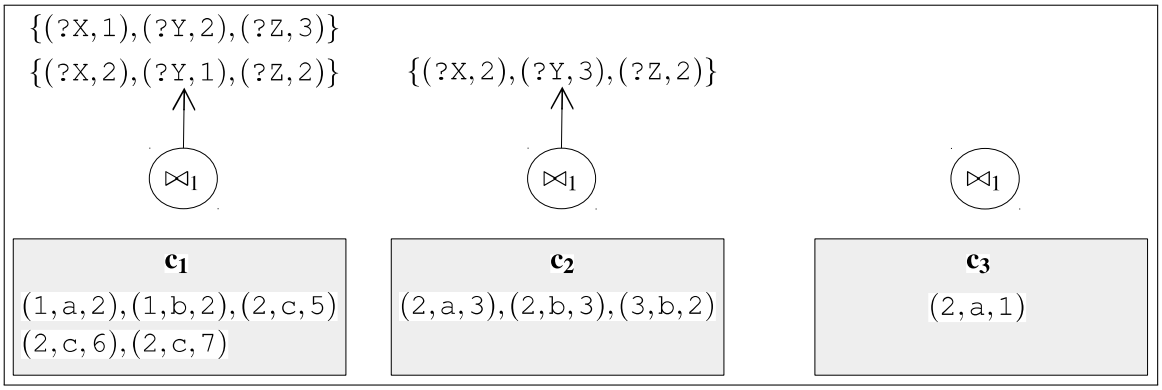


Figure A.5.: Results of joining the results of `?X a ?Y` and `?Y b ?Z`.

Processing of ⟪`?Z b ?W`⟫

Each compute node matches the triple pattern `?Z b ?W` on its locally stored chunk. The results are visualized in Figure A.6.
Compute node c_1 computes $[\![\texttt{?Z b ?W}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?Z},1),(\texttt{?W},2)\}\}$
Compute node c_2 computes $[\![\texttt{?Z b ?W}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?Z},2),(\texttt{?W},3)\},\{(\texttt{?Z},3),(\texttt{?W},2)\}\}$
Compute node c_3 computes $[\![\texttt{?Z b ?W}]\!]^{c_3}_{\text{cov}} = \{\}$

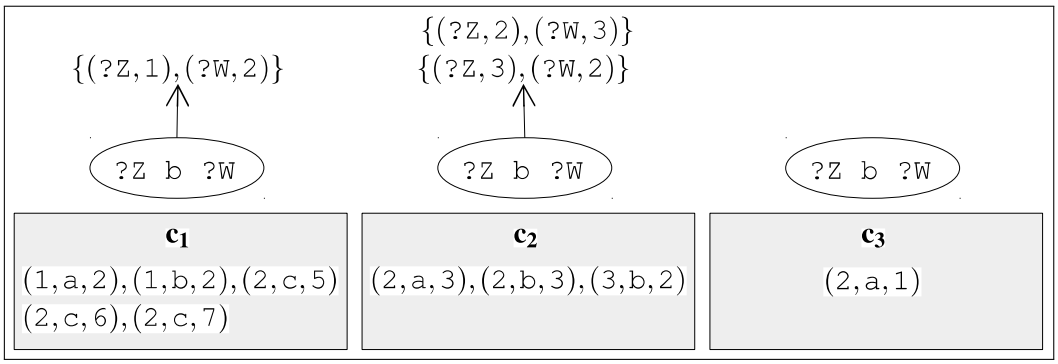


Figure A.6.: Results of evaluating `?Z b ?W` on the different compute nodes.

Processing of ⟪`?W a ?X`⟫

Each compute node matches the triple pattern `?W a ?X` on its locally stored chunk. The results are visualized in Figure A.7.
Compute node c_1 computes $[\![\texttt{?W a ?X}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?W},1),(\texttt{?X},2)\}\}$
Compute node c_2 computes $[\![\texttt{?W a ?X}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?W},2),(\texttt{?X},3)\}\}$
Compute node c_3 computes $[\![\texttt{?W a ?X}]\!]^{c_3}_{\text{cov}} = \{\{(\texttt{?W},2),(\texttt{?X},1)\}\}$

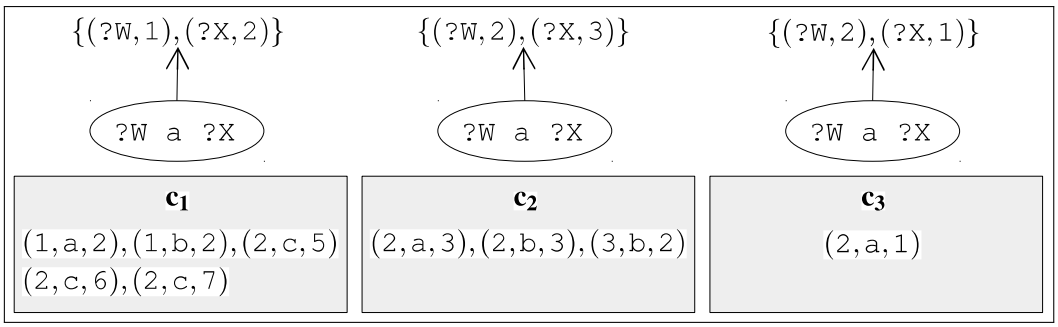


Figure A.7.: Results of evaluating `?W a ?X` on the different compute nodes.

Processing of ⟪⟪`?Z b ?W`⟫.⟪`?W a ?X`⟫⟫

In order to join the results of the triple patterns `?Z b ?W` and `?W a ?X` the join variables need to be identified first. In this case the join variables are $\text{cVars}(\langle\langle\texttt{?Z b ?W}\rangle\rangle.\langle\langle\texttt{?W a ?X}\rangle\rangle) = \{\texttt{?W}\}$. The variable bindings produced by the triple patterns `?Z b ?W` and `?W a ?X` are transferred to the compute node which is responsible for joining the resource bound to `?W`.

Reminder: The join responsibilities of `1`, `2` and `3` are $\text{jResp}(\texttt{1}) = c_1$, $\text{jResp}(\texttt{2}) = c_1$ and $\text{jResp}(\texttt{3}) = c_1$.

Figure A.8 shows the result after transferring all variable bindings.

Compute node c_1 has received the following results of the evaluation of `?Z b ?W` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, \langle\langle\texttt{?Z b ?W}\rangle\rangle.\langle\langle\texttt{?W a ?X}\rangle\rangle, [\![\texttt{?Z b ?W}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Z},1),(\texttt{?W},2)\},\{(\texttt{?Z},3),(\texttt{?W},2)\}\}$$

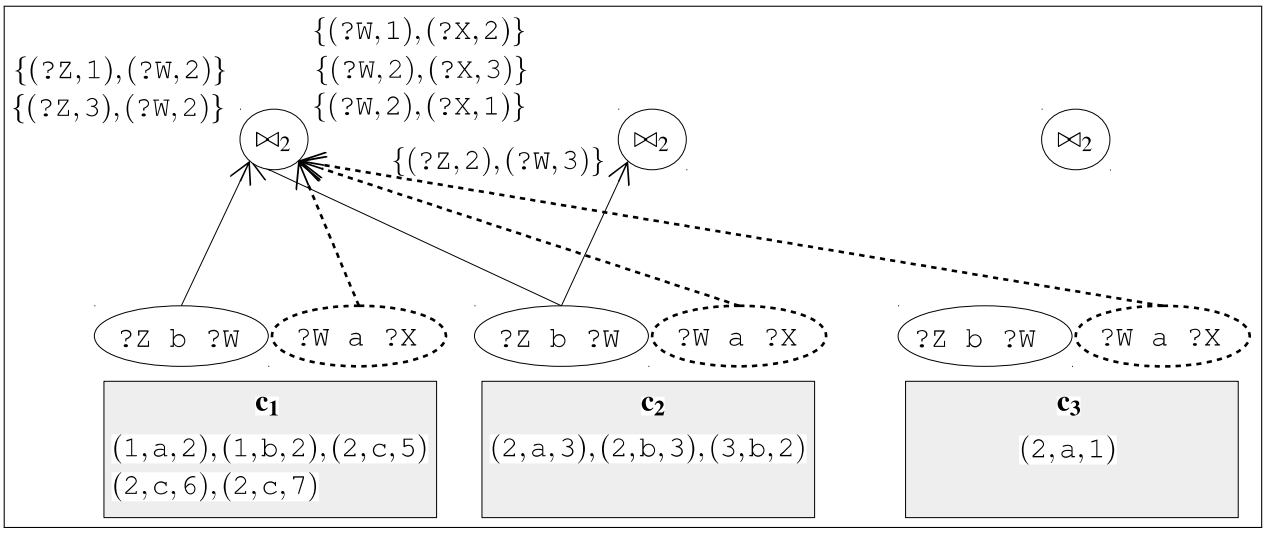


Figure A.8.: Results of transferring the variable bindings according to their join responsibility.

Furthermore, compute node c_1 received the following results of the evaluation of `?W a ?X` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, \langle\langle \texttt{?Z b ?W} \rangle\rangle . \langle\langle \texttt{?W a ?X} \rangle\rangle, [\![\texttt{?W a ?X}]\!]^c_{\text{cov}})$$
$$= \{\{(\texttt{?W},1),(\texttt{?X},2)\},\{(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?W},2),(\texttt{?X},1)\}\}$$

The join of both variable binding sets results in:
$[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_1}_{\text{cov}}$
$= \{\{(\texttt{?Z},1),(\texttt{?W},2)\},\{(\texttt{?Z},3),(\texttt{?W},2)\}\} \bowtie \{\{(\texttt{?W},1),(\texttt{?X},2)\},\{(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?W},2),(\texttt{?X},1)\}\}$
$= \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},1)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},1)\}\}$

Compute node c_2 has received the following results of the evaluation of `?Z b ?W` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, \langle\langle \texttt{?Z b ?W} \rangle\rangle . \langle\langle \texttt{?W a ?X} \rangle\rangle, [\![\texttt{?Z b ?W}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Z},2),(\texttt{?W},3)\}\}$$

Furthermore, compute node c_3 received the following results of the evaluation of `?W a ?X` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, \langle\langle \texttt{?Z b ?W} \rangle\rangle . \langle\langle \texttt{?W a ?X} \rangle\rangle, [\![\texttt{?W a ?X}]\!]^c_{\text{cov}}) = \{\}$$

The join of both variable binding sets results in:
$[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?Z},2),(\texttt{?W},3)\}\} \bowtie \{\} = \{\}$

Compute node c_3 has not received any variable bindings. Therefore, $[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_3}_{\text{cov}} = \{\}$.
Figure A.9 shows the variable bindings resulting of the joins.

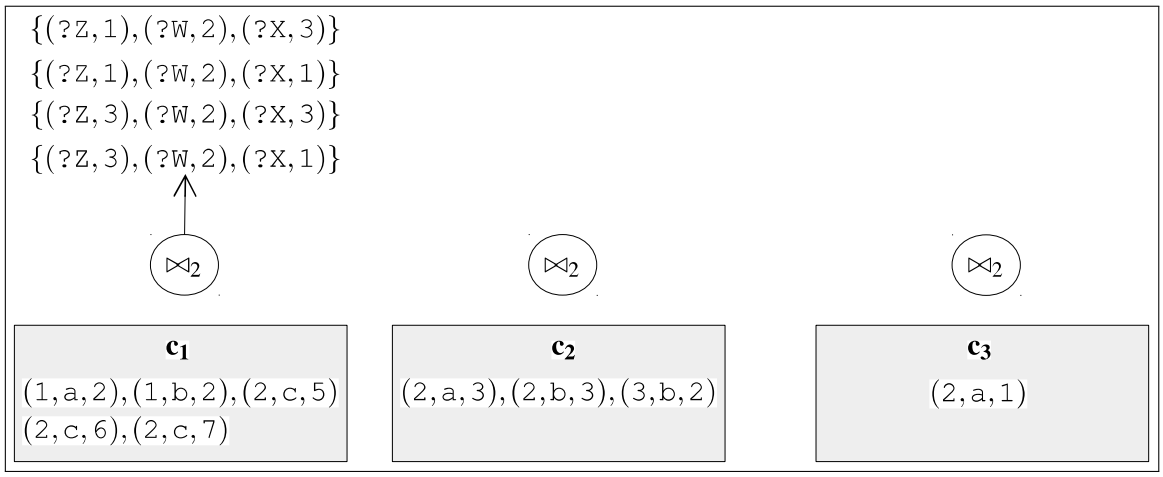


Figure A.9.: Results of joining the results of `?Z b ?W` and `?W a ?X`.

Processing of ⟪⟪⟪?X a ?Y⟫.⟪?Y b ?Z⟫⟫.⟪⟪?Z b ?W⟫.⟪?W a ?X⟫⟫⟫

In order to join the results of the two previous joins `?X a ?Y. ?Y b ?Z` and `?Z b ?W. ?W a ?X` the join variables need to be identified first. In this case the join variables are

$$\text{cVars}(⟪⟪⟪\texttt{?X a ?Y}⟫.⟪\texttt{?Y b ?Z}⟫⟫.⟪⟪\texttt{?Z b ?W}⟫.⟪\texttt{?W a ?X}⟫⟫⟫) = \{\texttt{?X}, \texttt{?Z}\}$$

Since there are two join variables now, the smallest variable is chosen based on the ordering of the variables. In this case $\min_{<_V}(\{\texttt{?X}, \texttt{?Z}\}) = \texttt{?X}$. Therefore, variable bindings produced by both previous joins are transferred to the compute node which is responsible for joining the resource bound to `?X`. In order to abbreviate the following formulas, the term ⟪⟪⟪?X a ?Y⟫.⟪?Y b ?Z⟫⟫.⟪⟪?Z b ?W⟫.⟪?W a ?X⟫⟫⟫ will be named as Q_j.

Reminder: The join responsibilities of `1`, `2` and `3` are $\text{jResp}(\texttt{1}) = c_1$, $\text{jResp}(\texttt{2}) = c_1$ and $\text{jResp}(\texttt{3}) = c_1$.

As described before, the join `?X a ?Y. ?Y b ?Z` produces the following results on the different compute nodes:

$[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{(\texttt{?X},2),(\texttt{?Y},1),(\texttt{?Z},2)\}\}$
$[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_2}_{\text{cov}} = \{\{(\texttt{?X},2),(\texttt{?Y},3),(\texttt{?Z},2)\}\}$
$[\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^{c_3}_{\text{cov}} = \{\}$

As described before, the join `?Z b ?W. ?W a ?X` produces the following results on the different compute nodes:

$[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_1}_{\text{cov}} = \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},1)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},3)\},$
$\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},1)\}\}\}$
$[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_2}_{\text{cov}} = \{\}$
$[\![\texttt{?Z b ?W. ?W a ?X}]\!]^{c_3}_{\text{cov}} = \{\}$

Figure A.10 shows the result after transferring all variable bindings. Since no operation on compute node c_3 is involved its operations are not shown.

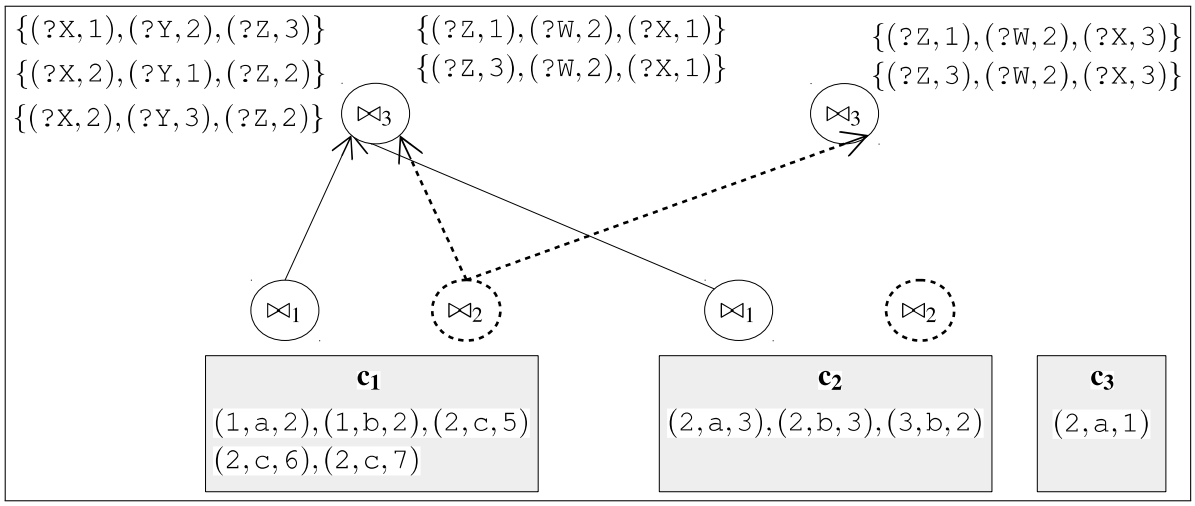


Figure A.10.: Results of transferring the variable bindings according to their join responsibility.

Compute node c_1 has received the following results of the evaluation of `?X a ?Y. ?Y b ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, Q_j, [\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^c_{\text{cov}})$$
$$= \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{(\texttt{?X},2),(\texttt{?Y},1),(\texttt{?Z},2)\},\{(\texttt{?X},2),(\texttt{?Y},3),(\texttt{?Z},2)\}\}$$

Furthermore, compute node c_1 received the following results of the evaluation of `?Z b ?W. ?W a ?X` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_1, Q_j, [\![\texttt{?Z b ?W. ?W a ?X}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},1)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},1)\}\}$$

The join of both variable binding sets results in:
$[\![\texttt{?X a ?Y. ?Y b ?Z. ?Z b ?W. ?W a ?X}]\!]^{c_1}_{\text{cov}}$
$= \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{(\texttt{?X},2),(\texttt{?Y},1),(\texttt{?Z},2)\},\{(\texttt{?X},2),(\texttt{?Y},3),(\texttt{?Z},2)\}\}$
$\quad \bowtie \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},1)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},1)\}\}$
$= \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3),(\texttt{?W},2)\}\}$

Compute node c_2 has received the following results of the evaluation of `?X a ?Y. ?Y b ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, Q_j, [\![\texttt{?X a ?Y. ?Y b ?Z}]\!]^c_{\text{cov}}) = \{\}$$

Furthermore, compute node c_3 received the following results of the evaluation of `?Z b ?W. ?W a ?X` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}(c_2, Q_j, [\![\texttt{?Z b ?W. ?W a ?X}]\!]^c_{\text{cov}}) = \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},3)\}\}$$

The join of both variable binding sets results in:
$[\![$`?X a ?Y. ?Y b ?Z. ?Z b ?W. ?W a ?X`$]\!]^{c_2}_{\text{cov}}$
$= \{\} \bowtie \{\{(\texttt{?Z},1),(\texttt{?W},2),(\texttt{?X},3)\},\{(\texttt{?Z},3),(\texttt{?W},2),(\texttt{?X},3)\}\}$
$= \{\}$

Compute node c_3 has not received any variable bindings.
$[\![$`?X a ?Y. ?Y b ?Z. ?Z b ?W. ?W a ?X`$]\!]^{c_3}_{\text{cov}} = \{\}$.

Figure A.11 shows the variable bindings resulting of the joins.

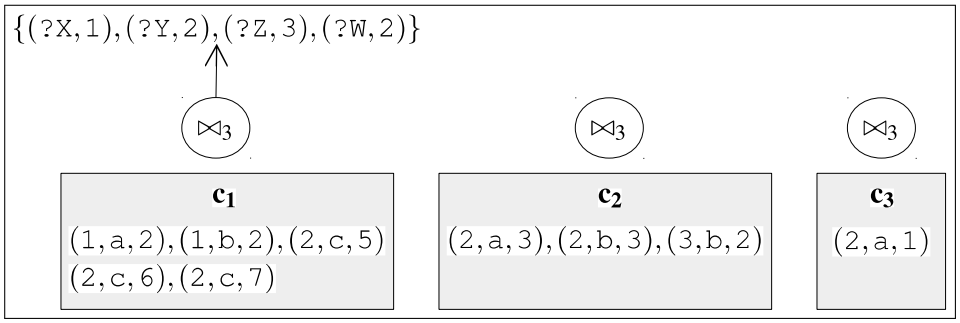


Figure A.11.: Results of joining the results of both previous joins.

Processing of
⟪SELECT ?X, ?Y, ?Z, ?W⟪⟪⟪?X a ?Y⟫.⟪?Y b ?Z⟫⟫.⟪⟪?Z b ?W⟫.⟪?W a ?X⟫⟫⟫⟫

The projection of the variables can be performed locally on each compute node without the need to exchange variable bindings before. Since in this example no variable is filtered out the results on the different compute nodes remain unchanged.

$$\begin{aligned}
[\![Q]\!]^{c_1}_{\text{cov}} &= \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3),(\texttt{?W},2)\}\} \\
[\![Q]\!]^{c_2}_{\text{cov}} &= \{\} \\
[\![Q]\!]^{c_3}_{\text{cov}} &= \{\}
\end{aligned}$$

Union of the Result Sets of all Compute Nodes

The last step performed is to union all result sets before sending the results to the requesting client.

$$[\![Q]\!]_{\text{cov}} = [\![Q]\!]^{c_1}_{\text{cov}} \cup [\![Q]\!]^{c_2}_{\text{cov}} \cup [\![Q]\!]^{c_3}_{\text{cov}} = \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3),(\texttt{?W},2)\}\}$$

A.2. Example of Distributed Query Execution with Triple Replication

In the example setting we assume that there exists three compute nodes c_1, c_2 and c_3. Therefore the set of all compute nodes C is $C = \{c_1, c_2, c_3\}$. The compute nodes have the order $c_1 < c_2 < c_3$.

A graph cover called cov distributes a graph G over all three computes nodes. The resulting chunks are:

$$\begin{aligned} \text{chunk}_{\text{cov}}(c_1) &= \{(\texttt{1},\texttt{a},\texttt{2}),(\texttt{2},\texttt{a},\texttt{3})\} \\ \text{chunk}_{\text{cov}}(c_2) &= \{(\texttt{2},\texttt{a},\texttt{3}),(\texttt{3},\texttt{b},\texttt{1})\} \\ \text{chunk}_{\text{cov}}(c_3) &= \{(\texttt{3},\texttt{b},\texttt{1}),(\texttt{1},\texttt{a},\texttt{2})\} \end{aligned}$$

Thereby, `1,2,3` represent resources at the subject or object position whereas `a,b` represent resources at the property position. Even though it is not defined formally, we add the set of compute nodes on which the triples occur in the following list of chunks in order to increase the comprehensibility of this example. In the formal definition the sets of compute nodes are determined by the graph cover function cov when evaluating the triple patterns.

$$\begin{aligned} \text{chunk}_{\text{cov}}(c_1) &= \{((\texttt{1},\texttt{a},\texttt{2}),\{c_1,c_3\}),((\texttt{2},\texttt{a},\texttt{3}),\{c_1,c_2\})\} \\ \text{chunk}_{\text{cov}}(c_2) &= \{((\texttt{2},\texttt{a},\texttt{3}),\{c_1,c_2\}),((\texttt{3},\texttt{b},\texttt{1}),\{c_2,c_3\})\} \\ \text{chunk}_{\text{cov}}(c_3) &= \{((\texttt{3},\texttt{b},\texttt{1}),\{c_2,c_3\}),((\texttt{1},\texttt{a},\texttt{2}),\{c_1,c_3\})\} \end{aligned}$$

Furthermore, awe assume that the following join responsibility is defined.

$$\begin{aligned} \text{jResp}(\texttt{1}) &= c_1 \\ \text{jResp}(\texttt{2}) &= c_1 \\ \text{jResp}(\texttt{3}) &= c_2 \\ \text{jResp}(\texttt{a}) &= c_1 \\ \text{jResp}(\texttt{b}) &= c_2 \end{aligned}$$

The following query Q should be executed with a bushy query execution tree.

```
SELECT ?X, ?Y, ?Z, ?W WHERE {
   ?X a ?Y.
   ?Y a ?Z.
   ?Z b ?X
}
```

The bushy execution tree for query Q will result in the query execution tree shown in Figure A.12. Each compute node will execute the complete query execution tree. Additionally, the variables have the order `?X` $<$ `?Y` $<$ `?Z`.

Processing of ⟪`?X a ?Y`⟫

Each compute node matches the triple pattern `?X a ?Y` on its locally stored chunk. The results are visualized in Figure A.13.
Compute node c_1 computes $[\![\texttt{?X a ?Y}]\!]^{loc,c_1}_{\text{cov}} = \{(\{(\texttt{?X},\texttt{1}),(\texttt{?Y},\texttt{2})\},\{c_1,c_3\}),(\{(\texttt{?X},\texttt{2}),(\texttt{?Y},\texttt{3})\},\{c_1,c_2\})\}$
Compute node c_2 computes $[\![\texttt{?X a ?Y}]\!]^{loc,c_2}_{\text{cov}} = \{(\{(\texttt{?X},\texttt{2}),(\texttt{?Y},\texttt{3})\},\{c_1,c_2\})\}$
Compute node c_3 computes $[\![\texttt{?X a ?Y}]\!]^{loc,c_3}_{\text{cov}} = \{(\{(\texttt{?X},\texttt{1}),(\texttt{?Y},\texttt{2})\},\{c_1,c_3\})\}$

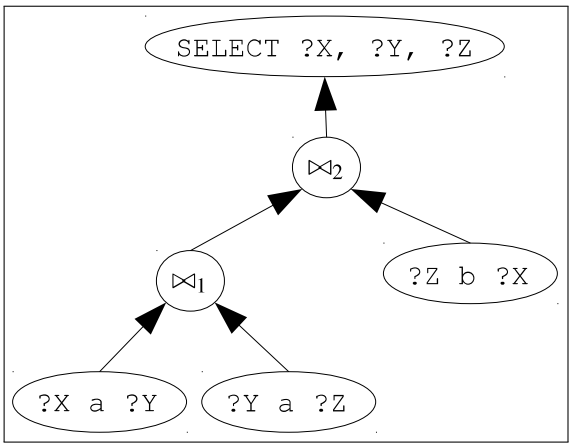


Figure A.12.: Query execution tree of example query Q.

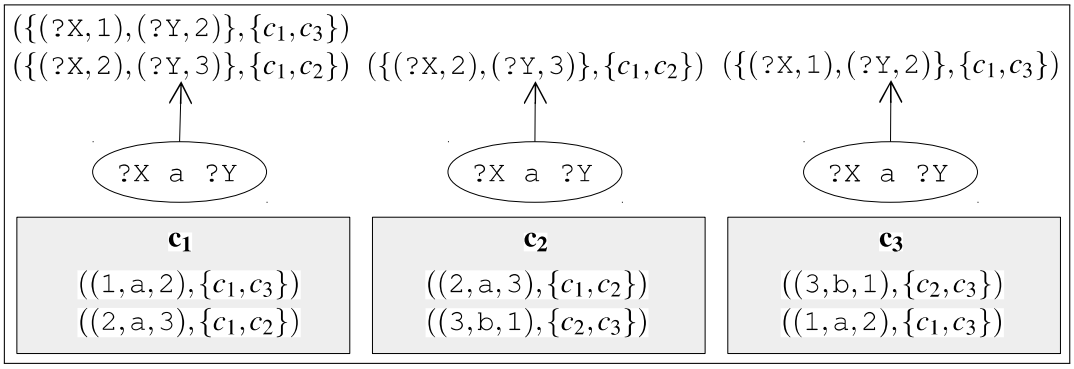


Figure A.13.: Results of evaluating `?X a ?Y` on the different compute nodes.

Processing of ⟪`?Y a ?Z`⟫

Each compute node matches the triple pattern `?Y a ?Z` on its locally stored chunk. The results are visualized in Figure A.14.
Compute node c_1 computes $[\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c_1} = \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\}),(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}$
Compute node c_2 computes $[\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c_2} = \{(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}$
Compute node c_3 computes $[\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c_3} = \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\})\}$

Processing of ⟪⟪`?X a ?Y`⟫.⟪`?Y a ?Z`⟫⟫

In order to join the results of the triple patterns `?X a ?Y` and `?Y a ?Z` the join variables need to be identified first. In this case the join variables are $\text{cVars}(\langle\!\langle\langle\!\langle\texttt{?X a ?Y}\rangle\!\rangle.\langle\!\langle\texttt{?Y a ?Z}\rangle\!\rangle\rangle\!\rangle) = \{\texttt{?Y}\}$.

One difficulty of distributed joins is to compute the complete set of results. To ensure this, all localized variable bindings that assign the same resource to the join variable needs to be transferred to the same compute node where they will be joined. The compute node on which a specific resource will be joined is defined by its join responsibility jResp. For instance, the localized variable binding $(\mu, C') = (\{(\texttt{?X},1),(\texttt{?Y},2)\},\{c_1,c_3\})$ produced on compute node c_3 would be joined on compute node $\text{jResp}(\mu(\texttt{?Y})) = \text{jResp}(2) = c_1$. Since this localized variable binding is already known on compute node c_1, i.e., $c_1 \in C'$ with $C' = \{c_1,c_3\}$, it is not

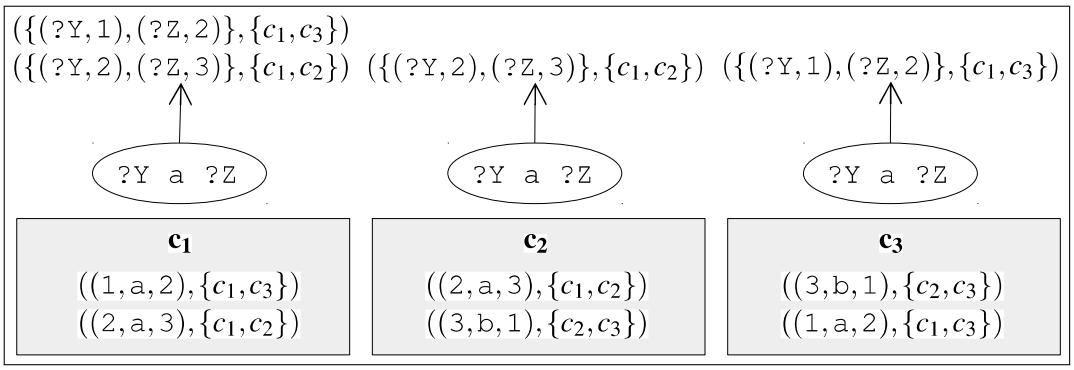


Figure A.14.: Results of evaluating `?Y a ?Z` on the different compute nodes.

transferred to compute node c_1. Instead, it is transferred to the parent join on the same compute node c_3 to allow for future joins with localized variable bindings on this compute node.

Reminder: The join responsibilities of `1`, `2` and `3` are $\text{jResp}(1) = c_1$, $\text{jResp}(2) = c_1$ and $\text{jResp}(3) = c_1$.

Transferring the Results of the Evaluation of `?X a ?Y`

The results of $[\![\texttt{?X a ?Y}]\!]^{loc,c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_1, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_1}_{\text{cov}}) &= \{(\{(\texttt{?X},1),(\texttt{?Y},2)\},\{c_1,c_3\}),\\
&\quad\ (\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\}\\
\text{route}^{\text{loc}}(c_1, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_1, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?X a ?Y}]\!]^{loc,c_2}_{\text{cov}}$ produced on compute node c_2 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_2, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_2}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_2, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_2}_{\text{cov}}) &= \{(\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\}\\
\text{route}^{\text{loc}}(c_2, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_2}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?X a ?Y}]\!]^{loc,c_3}_{\text{cov}}$ produced on compute node c_3 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_3, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_3}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_3, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_3}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_3, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c_3}_{\text{cov}}) &= \{(\{(\texttt{X},1),(\texttt{?Y},2)\},\{c_1,c_3\})\}
\end{aligned}$$

Transferring the Results of the Evaluation of `?Y a ?Z`

The results of $[\![\texttt{?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_1, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\}),\\
&\quad\ (\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}\\
\text{route}^{\text{loc}}(c_1, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_1, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Y a ?Z}]\!]^{loc,c_2}_{\text{cov}}$ produced on compute node c_2 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_2, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_2}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_2, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_2}_{\text{cov}}) &= \{(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}\\
\text{route}^{\text{loc}}(c_2, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_2}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Y a ?Z}]\!]^{loc,c_3}_{\text{cov}}$ produced on compute node c_3 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_3, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_3}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_3, c_2, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_3}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_3, c_3, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?Y a ?Z}]\!]^{loc,c_3}_{\text{cov}}) &= \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\})\}
\end{aligned}$$

Figure A.15 shows the result after transferring all localized variable bindings.

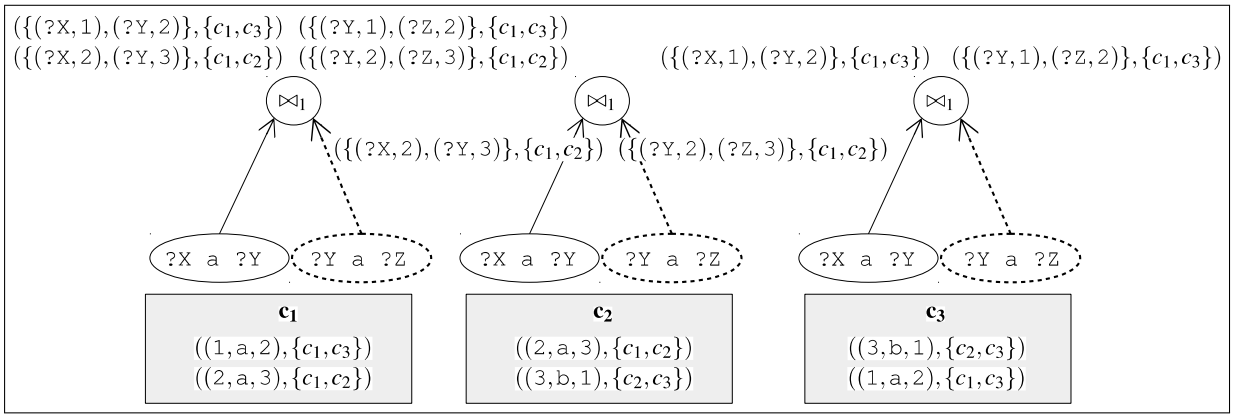


Figure A.15.: Results of transferring the localized variable bindings.

Joining the Results of the Evaluations of `?X a ?Y` *and* `?Y a ?Z`

Compute node c_1 has received the following results of the evaluation of `?X a ?Y` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_1, \langle\!\langle \texttt{?X a ?Y} \rangle\!\rangle . \langle\!\langle \texttt{?Y a ?Z} \rangle\!\rangle\!\rangle, [\![\texttt{?X a ?Y}]\!]^{loc,c}_{\text{cov}}) =$$
$$\{(\{(\texttt{?X},1),(\texttt{?Y},2)\},\{c_1,c_3\}),(\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\}$$

Furthermore, compute node c_1 received the following results of the evaluation of `?Y a ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_1, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y a ?Z} \rangle\rangle, [\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c}) =$$
$$\{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\}),(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}$$

The join of both localized variable binding sets results in:

$$[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]_{\text{cov}}^{loc,c_1}$$
$$= \{(\{(\texttt{?X},1),(\texttt{?Y},2)\},\{c_1,c_3\}),(\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\}$$
$$\bowtie \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\}),(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}$$
$$= \{(\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1\})\}$$

Note that the joined result is only known by compute node c_1 since it is the only compute node that knows both localized variable bindings from which it is created, i.e., $\{c_1\} = \{c_1,c_3\} \cap \{c_1,c_2\}$.

Compute node c_2 has received the following results of the evaluation of `?X a ?Y` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y a ?Z} \rangle\rangle, [\![\texttt{?X a ?Y}]\!]_{\text{cov}}^{loc,c}) = \{(\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\}$$

Furthermore, compute node c_3 received the following results of the evaluation of `?Y a ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_2, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y a ?Z} \rangle\rangle, [\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c}) = \{(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\}$$

The join of both localized variable binding sets results in:

$$[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]_{\text{cov}}^{loc,c_2} = \{(\{(\texttt{?X},2),(\texttt{?Y},3)\},\{c_1,c_2\})\} \bowtie \{(\{(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1,c_2\})\} = \{\}$$

Compute node c_3 has received the following results of the evaluation of `?X a ?Y` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_3, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y a ?Z} \rangle\rangle, [\![\texttt{?X a ?Y}]\!]_{\text{cov}}^{loc,c}) = \{(\{(\texttt{X},1),(\texttt{?Y},2)\},\{c_1,c_3\})\}$$

Furthermore, compute node c_3 received the following results of the evaluation of `?Y a ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_3, \langle\langle \texttt{?X a ?Y} \rangle\rangle . \langle\langle \texttt{?Y a ?Z} \rangle\rangle, [\![\texttt{?Y a ?Z}]\!]_{\text{cov}}^{loc,c}) = \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\})\}$$

The join of both localized variable binding sets results in:

$$[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]_{\text{cov}}^{loc,c_3} = \{(\{(\texttt{?X},1),(\texttt{?Y},2)\},\{c_1,c_3\})\} \bowtie \{(\{(\texttt{?Y},1),(\texttt{?Z},2)\},\{c_1,c_3\})\} = \{\}$$

Figure A.16 shows the localized variable bindings resulting of the joins.

Processing of $\langle\langle$ `?Z b ?X` $\rangle\rangle$

Each compute node matches the triple pattern `?Z b ?X` on its locally stored chunk. The results are visualized in Figure A.17.

Compute node c_1 computes $[\![\texttt{?Z b ?X}]\!]_{\text{cov}}^{loc,c_1} = \{\}$

Compute node c_2 computes $[\![\texttt{?Z b ?X}]\!]_{\text{cov}}^{loc,c_2} = \{(\{(\texttt{?Z},3),(\texttt{?X},1)\},\{c_2,c_3\})\}$

Compute node c_3 computes $[\![\texttt{?Z b ?X}]\!]_{\text{cov}}^{loc,c_3} = \{(\{(\texttt{?Z},3),(\texttt{?X},1)\},\{c_2,c_3\})\}$

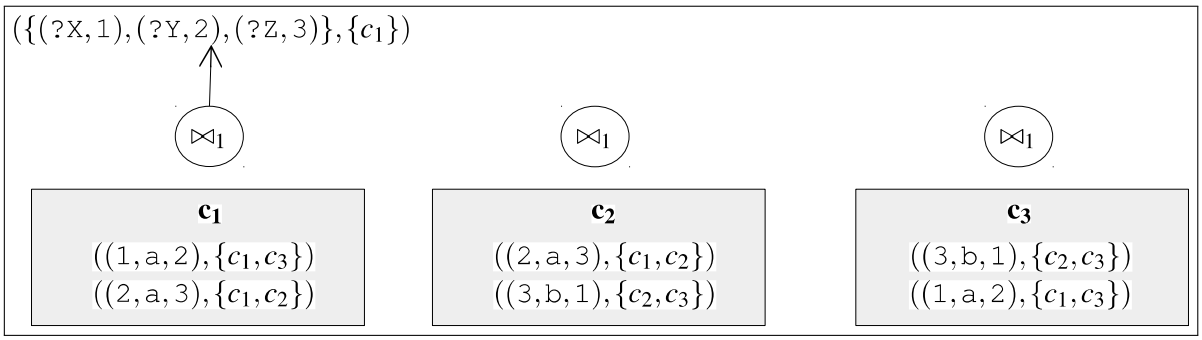


Figure A.16.: Results of joining the results of `?X a ?Y` and `?Y a ?Z`.

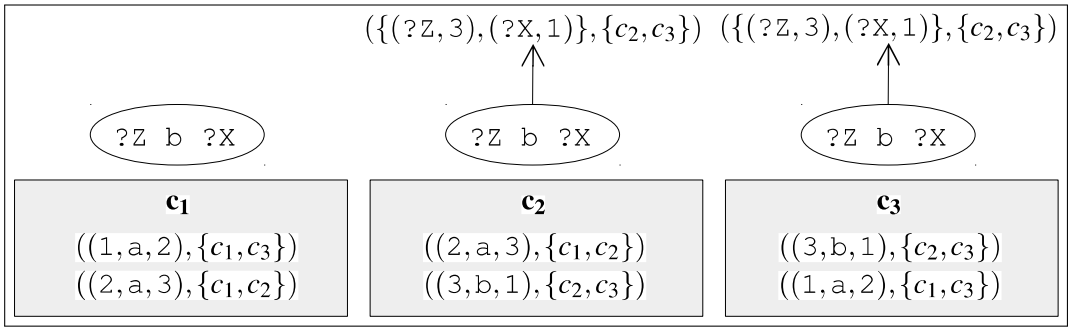


Figure A.17.: Results of evaluating `?Z b ?X` on the different compute nodes.

Processing of $\langle\langle\langle\langle$`?X a ?Y`$\rangle\rangle.\langle\langle$`?Y a ?Z`$\rangle\rangle\rangle,\langle\langle$`?Z b ?X`$\rangle\rangle\rangle$

In order to join the results of the join $\langle\langle$`?X a ?Y`$\rangle\rangle.\langle\langle$`?Y a ?Z`$\rangle\rangle\rangle$ and the triple pattern `?Z b ?X` the join variables need to be identified first. In this case the join variables are

$$\text{cVars}(\langle\langle\langle\texttt{?X a ?Y}\rangle\rangle.\langle\langle\texttt{?Y a ?Z}\rangle\rangle\rangle,\langle\langle\texttt{?Z b ?X}\rangle\rangle) = \{\texttt{?X, ?Z}\}$$

Since there are two join variables now, the smallest variable is chosen based on the ordering of the variables. In this case $\min_{<_V}(\{\texttt{?X},\texttt{?Z}\}) = \texttt{?X}$. Therefore, localized variable bindings produced by both previous joins are transferred to the compute node which is responsible for joining the resource bound to `?X`. In order to abbreviate the following formulas, the term $\langle\langle\langle\langle$`?X a ?Y`$\rangle\rangle.\langle\langle$`?Y a ?Z`$\rangle\rangle\rangle,\langle\langle$`?Z b ?X`$\rangle\rangle\rangle$ will be named as Q_j.

Reminder: The join responsibilities of `1`, `2` and `3` are $\text{jResp}(\texttt{1}) = c_1$, $\text{jResp}(\texttt{2}) = c_1$ and $\text{jResp}(\texttt{3}) = c_1$. As described before, the join `?X a ?Y. ?Y a ?Z` produces no join results on compute nodes c_2 and c_3. On compute node c_1 the localized variable binding $\{(\{(\texttt{?X},\texttt{1}),(\texttt{?Y},\texttt{2}),(\texttt{?Z},\texttt{3})\},\{c_1\})\}$ is produced.

Transferring the Results of the Evaluation of $\langle\langle$`?X a ?Y`$\rangle\rangle.\langle\langle$`?Y a ?Z`$\rangle\rangle\rangle$

The results of $[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_1,c_1,Q_j,[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{(\{(\texttt{?X},\texttt{1}),(\texttt{?Y},\texttt{2}),(\texttt{?Z},\texttt{3})\},\{c_1\})\}\\
\text{route}^{\text{loc}}(c_1,c_2,Q_j,[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}\\
\text{route}^{\text{loc}}(c_1,c_3,Q_j,[\![\texttt{?X a ?Y. ?Y a ?Z}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

Since there were no join results on compute nodes c_2 and c_3, they are omitted here.

Transferring the Results of the Evaluation of `?Z b ?X`

The triple pattern `?Z b ?X` has produced the localized variable binding $(\mu, C') = (\{(\texttt{?Z}, 3), (\texttt{?X}, 1)\}, \{c_2, c_3\})$ on compute nodes c_2 and c_3. Based on the join responsibility of the resource to which `?X` is bound this localized variable binding has to be joined on compute node $\text{jResp}(\mu(\texttt{?X})) = \text{jResp}(1) = c_1$. Since c_1 does not know the localized variable binding – i.e., $c_1 \notin \{c_2, c_3\}$ – it has to be transferred there. In order to produce the complete result set c_1 only needs to receive (μ, C') once. Therefore, only the smallest compute node $min_{<_C}(\{c_2, c_3\}) = c_2$ knowing (μ, C') will send it to c_1. Compute node c_3 will discard the localized variable binding. Since after the transfer only compute node c_1 will know the localized variable binding, it will be transferred as $(\{(\texttt{?Z}, 3), (\texttt{?X}, 1)\}, \{c_1\})$.

The results of $[\![\texttt{?Z b ?X}]\!]^{loc,c_1}_{\text{cov}}$ produced on compute node c_1 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_1, c_1, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_1}_{\text{cov}}) &= \{\} \\
\text{route}^{\text{loc}}(c_1, c_2, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_1}_{\text{cov}}) &= \{\} \\
\text{route}^{\text{loc}}(c_1, c_3, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_1}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Z b ?X}]\!]^{loc,c_2}_{\text{cov}}$ produced on compute node c_2 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_2, c_1, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_2}_{\text{cov}}) &= \{(\{(\texttt{?Z}, 3), (\texttt{?X}, 1)\}, \{c_1\})\} \\
\text{route}^{\text{loc}}(c_2, c_2, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_2}_{\text{cov}}) &= \{\} \\
\text{route}^{\text{loc}}(c_2, c_3, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_2}_{\text{cov}}) &= \{\}
\end{aligned}$$

The results of $[\![\texttt{?Z b ?X}]\!]^{loc,c_3}_{\text{cov}}$ produced on compute node c_3 will be transferred to the compute nodes c_1, c_2 and c_3 as follows:

$$\begin{aligned}
\text{route}^{\text{loc}}(c_3, c_1, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_3}_{\text{cov}}) &= \{\} \\
\text{route}^{\text{loc}}(c_3, c_2, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_3}_{\text{cov}}) &= \{\} \\
\text{route}^{\text{loc}}(c_3, c_3, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c_3}_{\text{cov}}) &= \{\}
\end{aligned}$$

Figure A.18 shows the result after transferring all localized variable bindings.

Joining the Results of the Evaluations of `?X a ?Y. ?Y a ?Z` *and* `?Z b ?X`

Compute node c_1 has received the following results of the evaluation of `?X a ?Y. ?Y a ?Z` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_1, Q_j, [\![\texttt{?X a ?Y. ?Y a ?Z}]\!]^{loc,c}_{\text{cov}}) = \{(\{(\texttt{?X}, 1), (\texttt{?Y}, 2), (\texttt{?Z}, 3)\}, \{c_1\})\}$$

Furthermore, compute node c_1 received the following results of the evaluation of `?Z b ?X` from the different compute nodes:

$$\bigcup_{c \in C} \text{route}^{\text{loc}}(c, c_1, Q_j, [\![\texttt{?Z b ?X}]\!]^{loc,c}_{\text{cov}}) = \{(\{(\texttt{?Z}, 3), (\texttt{?X}, 1)\}, \{c_1\})\}$$

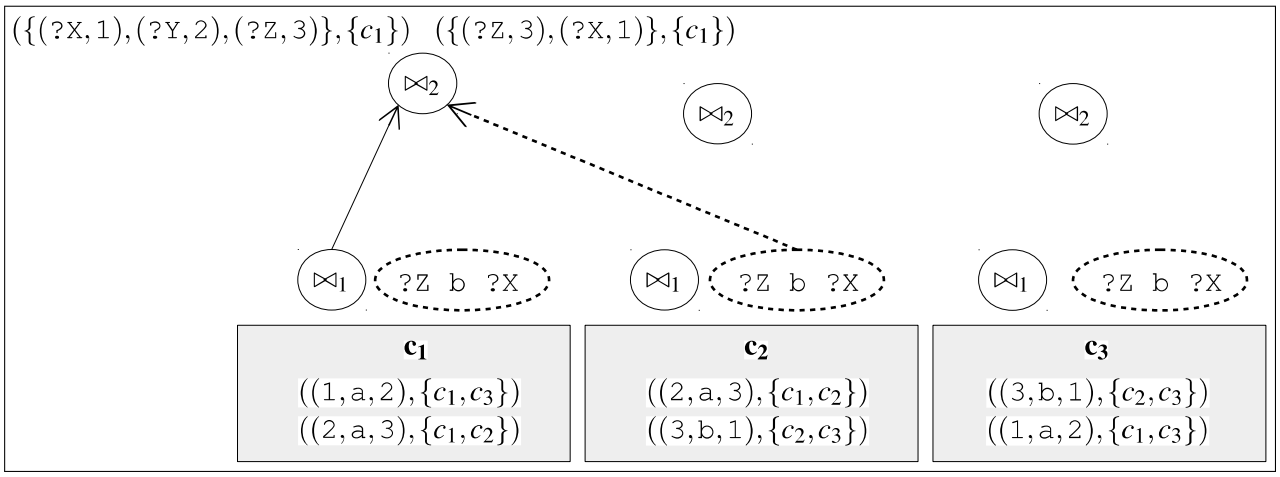


Figure A.18.: Results of transferring the localized variable bindings.

The join of both localized variable binding sets results in:
$[\![$`?X a ?Y. ?Y a ?Z. ?Z b ?X`$]\!]^{loc,c_1}_{\text{cov}}$
$= \{(\{(?X,1),(?Y,2),(?Z,3)\},\{c_1\})\} \bowtie \{(\{(?Z,3),(?X,1)\},\{c_1\})\}$
$= \{(\{(?X,1),(?Y,2),(?Z,3)\},\{c_1\})\}$

Compute node c_2 has not received any localized variable bindings. Therefore, no results are created.
$[\![$`?X a ?Y. ?Y a ?Z. ?Z b ?X`$]\!]^{loc,c_2}_{\text{cov}} = \{\}$.

Compute node c_3 has not received any localized variable bindings. Therefore, no results are created.
$[\![$`?X a ?Y. ?Y a ?Z. ?Z b ?X`$]\!]^{loc,c_3}_{\text{cov}} = \{\}$.

Figure A.19 shows the localized variable bindings resulting of the joins.

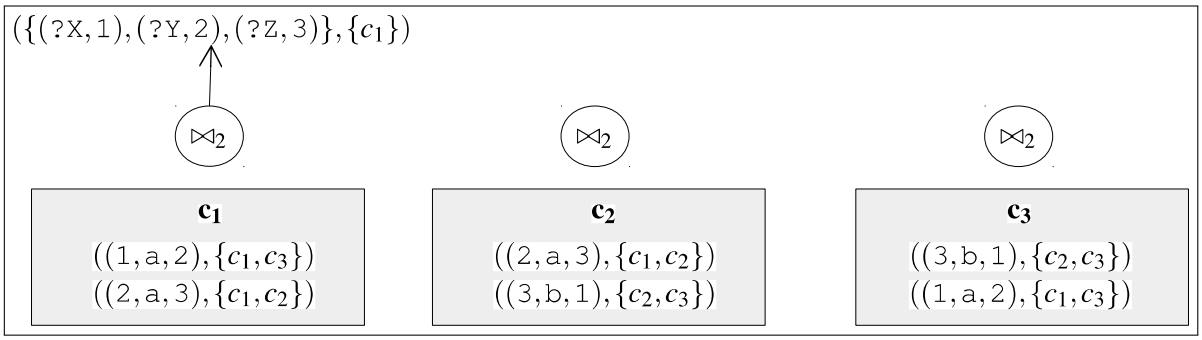


Figure A.19.: Results of joining the results of `?X a ?Y. ?Y a ?Z` and `?Z b ?X`.

Processing of ⟪SELECT ?X, ?Y, ?Z⟪⟪?X a ?Y⟫.⟪?Y a ?Z⟫⟫.⟪?Z b ?X⟫⟫⟫

The projection of the variables can be performed locally on each compute node without the need to exchange localized variable bindings before. Since in this example no variable is filtered out the results on the different compute nodes remain unchanged.

$$\begin{aligned}
[\![Q]\!]_{\text{cov}}^{loc,c_1} &= \{(\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\},\{c_1\})\} \\
[\![Q]\!]_{\text{cov}}^{loc,c_2} &= \{\} \\
[\![Q]\!]_{\text{cov}}^{loc,c_3} &= \{\}
\end{aligned}$$

Union of the Result Sets of all Compute Nodes

The last step performed is to union all result sets and removing the localization before sending the results to the requesting client.

$$[\![Q]\!]_{\text{cov}}^{loc} = \{\{(\texttt{?X},1),(\texttt{?Y},2),(\texttt{?Z},3)\}\}$$

APPENDIX B

Proof of Semantic Correctness and Completeness of Koral's Distributed Query Execution Strategy

In this section the semantic correctness and completeness of the distributed query execution strategy that ignores the triple replication is proven in Section B.1. The semantic correctness and completeness of the distributed query execution strategy with triple replication is proven in Section B.3. In order to proof it, an additional theorem is required that is presented and proven in Section B.2. The equality of both query execution strategies for graph covers that assign each triple to exactly one graph chunk is proven in Section B.5. In order to proof it, Section B.4 defines and proves an additional theorem. This chapter was taken from [69].

B.1. Proof of Semantic Correctness and Completeness for the Distributed Query Execution Strategy Ignoring Triple Replication

The semantic correctness and completeness of the distributed execution mechanism described in Sec. 4.4.2 is shown by proving Theorem 1.

Theorem 1. The centralized evaluation of query Q produces exact the same results as its distributed evaluation, i.e.

$$[\![Q]\!]_{\text{cover}} = [\![Q]\!]_G \ .$$

Proof. The proof of Theorem 1 is an induction over the structure of query Q. As induction bases it will be shown for all queries that do not contain subqueries. In the context of this thesis this means $Q = tp$. Based on this proof the induction hypothesis is raised that the theorem holds for queries B, B_1 and B_2. Afterwards the induction step is shown for the remaining query constructs that are the join $Q = B_1.B_2$ and the projection $Q =$ `SELECT` W `WHERE` $\{B\}$.

Definition 2 ensures that an arbitrary graph cover called "cover" always assigns each triple of the graph G to at least one compute node. This means that (B.1) holds.

$$G = \bigcup_{c \in C} \text{chunk}_{\text{cover}}(c) \ . \tag{B.1}$$

Induction basis: Q = tp.

$$\begin{aligned}
\mu \in [\![tp]\!]_{\text{cover}} &\overset{\text{Def. 27}}{\iff} \mu \in \bigcup_{c\in C} [\![tp]\!]^c_{\text{cover}} \\
&\overset{\text{Def. 26}}{\iff} \mu \in \bigcup_{c\in C} \left\{\mu' \middle| \text{dom}(\mu') = \text{var}(tp) \wedge \mu'(tp) \in \text{chunk}_{\text{cover}}(c)\right\} \\
&\iff \bigvee_{c\in C} (\text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in \text{chunk}_{\text{cover}}(c)) \\
&\iff \text{dom}(\mu) = \text{var}(tp) \wedge \bigvee_{c\in C} \mu(tp) \in \text{chunk}_{\text{cover}}(c) \\
&\iff \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in \bigcup_{c\in C} \text{chunk}_{\text{cover}}(c) \\
&\overset{\text{(B.1)}}{\iff} \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in G \\
&\overset{\text{Def. 14}}{\iff} \mu \in [\![tp]\!]_G
\end{aligned}$$

Induction hypothesis: Theorem 1 is valid for queries B_1, B_2 and B.

Induction step: $\mathbf{Q = B_1.B_2}$.

This proof is structured in a way that first $[\![B_1.B_2]\!]_{\text{cover}}$ is transformed into a set S_{left} and then $[\![B_1.B_2]\!]_G$ is transformed into a set S_{right}. Finally, the equality of S_{left} and S_{right} is shown.

$$\begin{aligned}
\mu \in [\![B_1.B_2]\!]_{\text{cover}} &\overset{\text{Def. 27}}{\iff} \mu \in \bigcup_{c\in C} [\![B_1.B_2]\!]^c_{\text{cover}} \\
&\overset{\text{Def. 26}}{\iff} \bigvee_{c\in C} \mu \in \left(\left(\bigcup_{c_1\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_1}_{\text{cover}})\right) \bowtie \left(\bigcup_{c_2\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_2}_{\text{cover}})\right)\right) \\
&\overset{\text{Def. 13}}{\iff} \bigvee_{c\in C} \mu) \in \left\{\mu_1 \cup \mu_2 \middle| \mu_1 \sim \mu_2 \wedge \mu_1 \in \left(\bigcup_{c_1\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_1}_{\text{cover}})\right)\right. \\
&\qquad \left. \wedge \mu_2 \in \left(\bigcup_{c_2\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_2}_{\text{cover}})\right)\right\} \\
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \\
&\quad \bigvee_{c\in C} \left(\mu_1 \in \left(\bigcup_{c_1\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_1}_{\text{cover}})\right) \wedge \mu_2 \in \left(\bigcup_{c_2\in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_2}_{\text{cover}})\right)\right)
\end{aligned}$$

$$
\begin{aligned}
&\overset{\text{Def. 25}}{\Longleftrightarrow} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} \mu_1 \in \left\{ \mu_1' \in [\![B_1]\!]^{c_1}_{\text{cover}} \middle| \mu_1' = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \right. \\
&\quad \left. \vee \left(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1'(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c\right) \right\} \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} \mu_2 \in \left\{ \mu_2' \in [\![B_2]\!]^{c_2}_{\text{cover}} \middle| \mu_2' = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \right. \\
&\quad \left. \vee \left(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2'(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c\right) \right\} \Bigg) \Bigg) \\
&\Longleftrightarrow \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c)) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c)) \Bigg) \Bigg) \Bigg) \\
&\Longrightarrow \mu \in S_{left}
\end{aligned}
$$

Now, $[\![B_1.B_2]\!]_G$ is transformed into a set S_{right}.

$$\begin{aligned}
\mu \in [\![B_1.B_2]\!]_G &\overset{\text{Def. 14}}{\iff} \mu \in [\![B_1]\!]_G \bowtie [\![B_2]\!]_G \\
&\overset{\text{Def. 13}}{\iff} \mu \in \{\mu_1 \cup \mu_2 | \mu_1 \sim \mu_2 \wedge \mu_1 \in [\![B_1]\!]_G \wedge \mu_2 \in [\![B_2]\!]_G\} \\
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \mu_1 \in [\![B_1]\!]_G \wedge \mu_2 \in [\![B_2]\!]_G \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \mu_1 \in [\![B_1]\!]_{\text{cover}} \wedge \mu_2 \in [\![B_2]\!]_{\text{cover}} \\
&\overset{\text{Def. 27}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \mu_1 \in \bigcup_{c_1 \in C} [\![B_1]\!]^{c_1}_{\text{cover}} \\
&\qquad \wedge \mu_2 \in \bigcup_{c_2 \in C} [\![B_2]\!]^{c_2}_{\text{cover}} \\
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \right) \wedge \left(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \right) \\
&\implies \mu \in S_{right}
\end{aligned}$$

The sets S_{left} and S_{right} contain only the elements determined by the equations above. The equality of S_{left} and S_{right} is proven by distinguishing the following cases:

1. $\varnothing \in S_{left} \cap S_{right}$
2. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
3. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
4. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
5. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
6. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
7. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$

The proofs of the third case is analogue to the second case and the fifth case is analogue to the fourth case. Therefore, the proofs of the third and fifth case are not shown.

Case 1: $\varnothing \in \mathbf{S_{left}} \cap \mathbf{S_{right}}$

$$
\begin{aligned}
\mu \in S_{left} \iff & \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
& \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \right. \\
& \left. \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \right) \Bigg) \\
& \wedge \left(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \right. \\
& \left. \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \right) \Bigg) \Bigg) \\
\overset{\text{case 1}}{\iff} & \exists \mu_1, \mu_2 \wedge \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
& \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \mu_1 = \varnothing) \right) \wedge \left(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \mu_2 = \varnothing) \right) \Bigg) \\
\iff & \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
& \wedge \left(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \right) \wedge \left(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \right) \\
\iff & \mu \in S_{right}
\end{aligned}
$$

Case 2: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_\mathbf{2} \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1.B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S_{left} \iff & \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
& \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \Bigg) \\
& \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \Bigg) \Bigg) \\
\overset{\text{case 2}}{\iff} & \exists \mu_1, \mu_2 \wedge \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \mu_1 = \varnothing) \Bigg) \\
& \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \Bigg) \Bigg) \\
\iff & \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
& \wedge \Bigg(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \Bigg) \wedge \Bigg(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \Bigg) \\
\iff & \mu \in S_{right}
\end{aligned}
$$

Case 4: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_\mathbf{2} \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\Longleftrightarrow \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \Bigg) \Bigg) \\
&\overset{\text{case 4}}{\Longleftrightarrow} \exists \mu_1, \mu_2 \wedge \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \mu_1 = \varnothing) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c) \Bigg) \Bigg) \\
&\Longleftrightarrow \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \Bigg) \wedge \Bigg(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \Bigg) \\
&\Longleftrightarrow \mu \in S_{right}
\end{aligned}
$$

Case 6: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_\mathbf{1} \sim \mu_\mathbf{2} \wedge \mu_\mathbf{1} \neq \varnothing \wedge \mu_\mathbf{2} \neq \varnothing \wedge \mathrm{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \mathrm{jResp}(\mu_1(\min_{<_V}(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \Bigg) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \mathrm{jResp}(\mu_2(\min_{<_V}(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \Bigg) \Bigg) \Bigg) \\
&\overset{\text{case 6}}{\iff} \exists \mu_1, \mu_2 \wedge \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \Bigg) \Bigg) \\
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \Bigg) \wedge \Bigg(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \Bigg) \\
&\iff \mu \in S_{right}
\end{aligned}
$$

Case 7: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_\mathbf{1} \sim \mu_\mathbf{2} \wedge \mu_\mathbf{1} \neq \varnothing \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\mu_1 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \Bigg) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\mu_2 = \varnothing \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = \min_{<_C}(C)) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \Bigg) \Bigg) \Bigg) \\
&\overset{\text{case 7}}{\iff} \exists \mu_1, \mu_2 \wedge \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c) \Bigg) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(\min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c) \Bigg) \Bigg) \\
&\overset{\text{Def. 13}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\bigvee_{c_1 \in C} \mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \Bigg) \wedge \Bigg(\bigvee_{c_2 \in C} \mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \Bigg) \\
&\iff \mu \in S_{right}
\end{aligned}
$$

Induction step: Q = `SELECT` W `WHERE` {B}.

$$
\begin{aligned}
\mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]_{\text{cover}} &\overset{\text{Def. 27}}{\iff} \mu \in \bigcup_{c \in C} [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^c_{\text{cover}} \\
&\iff \bigvee_{c \in C} \mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^c_{\text{cover}} \\
&\overset{\text{Def. 26}}{\iff} \bigvee_{c \in C} \mu \in \left\{ \mu'_{|_W} \,\middle|\, \mu' \in [\![B]\!]^c_{\text{cover}} \right\} \\
&\iff \exists \mu' : \mu = \mu'_{|_W} \wedge \bigvee_{c \in C} \mu' \in [\![B]\!]^c_{\text{cover}} \\
&\iff \exists \mu' : \mu = \mu'_{|_W} \wedge \mu' \in \bigcup_{c \in C} [\![B]\!]^c_{\text{cover}} \\
&\overset{\text{Def. 27}}{\iff} \exists \mu' : \mu = \mu'_{|_W} \wedge \mu' \in [\![B]\!]_{\text{cover}} \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu' : \mu = \mu'_{|_W} \wedge \mu' \in [\![B]\!]_G \\
&\overset{\text{Def. 14}}{\iff} \mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]_G
\end{aligned}
$$

□

B.2. Proof of Knows Lemma

The knows Lemma 1 is required to proof the semantic correctness and completeness of the replication-aware distributed query execution strategy in the following section. It says that whenever a localized variable binding is produced on a compute node c, then this variable binding will be known by c.

Lemma 1. If the replication-aware distributed query execution strategy produces a localized variable binding μ' while evaluating the BGP B on any compute node c, then μ' is known on c:

$$\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : (\mu, C') \in [\![B]\!]^{loc,c}_{\text{cover}} \Rightarrow c \in C' \quad .$$

Proof. The proof of Lemma 1 is an induction over the structure of basic graph pattern B. As induction bases it will be shown for all queries that do not contain other basic graph patterns. In the context of this thesis this means $B = tp$. Based on this proof the induction hypothesis is raised that the theorem holds for basic graph patterns B_1 and B_2. Afterwards the induction step is shown for the remaining basic graph pattern that is the join $B = B_1.B_2$.

Induction basis: B = *tp*.

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : (\mu, C') \in [\![tp]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{Def. 31}}{\iff} \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in \text{chunk}_{\text{cover}}(c) \wedge C' = \text{cover}(\mu(tp)) \\
&\implies \mu(tp) \in \text{chunk}_{\text{cover}}(c) \wedge C' = \text{cover}(\mu(tp)) \\
&\overset{\text{Def. 3}}{\iff} c \in \text{cover}(\mu(tp)) \wedge C' = \text{cover}(\mu(tp)) \\
&\implies c \in C'
\end{aligned}
$$

Induction hypothesis: Lemma 1 is valid for queries B_1, B_2 and B.

Induction step: $\mathbf{B = B_1.B_2}$.

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : (\mu, C') \in [\![B_1.B_2]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{Def. 31}}{\Longleftrightarrow} (\mu, C') \in \left(\bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \bowtie \left(\bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 29}}{\Longleftrightarrow} \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\qquad \left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\qquad \left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right)
\end{aligned}
$$

In the following we distinguishing the following cases:

1. $\mu_1 = \varnothing \wedge \mu_2 = \varnothing$
2. $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
3. $\mu_1 \neq \varnothing \wedge \mu_2 = \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
4. $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
5. $\mu_1 \neq \varnothing \wedge \mu_2 = \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
6. $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
7. $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$

The proofs of the third case is analogue to the second case and the fifth case is analogue to the fourth case. Therefore, the proofs of the third and fifth case are not shown.

Case 1: $\mu_1 = \varnothing \wedge \mu_2 = \varnothing$

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \\
&\mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 30}}{\Longleftrightarrow} \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \\
&\qquad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c = min_{<_c}(C_1') \wedge C_1 = C \right) \right) \wedge \\
&\qquad \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c = min_{<_c}(C_2') \wedge C_2 = C \right) \right) \\
&\Longrightarrow C' = C_1 \cap C_2 \wedge C_1 = C \wedge C_2 = C \\
&\Longleftrightarrow C' = C \cap C = C \\
&\overset{c \in C}{\Longrightarrow} c \in C'
\end{aligned}
$$

Case 2: $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \\
&\mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 30}}{\iff} \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \\
&\qquad \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c = min_{<_C}(C_1') \wedge C_1 = C \Big) \right) \wedge \\
&\qquad \left(\exists C_2' : \bigvee_{c_2 \in C} \Big(\Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \right. \right. \\
&\qquad \left. \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \Big) \vee \right. \\
&\qquad \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \\
&\qquad \left. \left. \wedge c_2 = c \wedge C_2 = \{c\}) \right) \right) \\
&\implies C' = C_1 \cap C_2 \wedge C_1 = C \wedge C_2 = \{c\} \\
&\iff C' = C \cap \{c\} = \{c\} \\
&\implies c \in C'
\end{aligned}
$$

Case 4: $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \\
&\mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 30}}{\iff} \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \\
&\qquad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c = min_{<_C}(C_1') \wedge C_1 = C \right) \right) \wedge \\
&\qquad \left(\exists C_2' : \bigvee_{c_2 \in C} \left(\left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \right. \right. \right. \\
&\qquad \left. \left. \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_2' \wedge C_2 = \{c\} \right) \vee \right. \\
&\qquad \left. \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \right. \right. \\
&\qquad \left. \left. \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge c_u \in C_2' \wedge C_2 = C_2' \right) \right) \\
&\overset{\text{ind. hyp.}}{\implies} C' = C_1 \cap C_2 \wedge C_1 = C \wedge (C_2 = \{c\} \vee (c_2 \in C_2 \wedge c_2 = c)) \\
&\iff C' = C \cap \{c\} = \{c\} \vee \left(C' = C \cap C_2 = C_2 \wedge c \in C_2 \right) \\
&\implies C' = \{c\} \vee c \in C' \\
&\implies c \in C'
\end{aligned}
$$

Case 6: $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \\
&\mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 30}}{\iff} \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \\
&\qquad \left(\exists C_1' : \bigvee_{c_1 \in C} \Big(\Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \notin C_1' \right. \right. \\
&\qquad \left. \wedge c_1 = min_{<_c}(C_1') \wedge C_1 = \{c\} \Big) \vee \\
&\qquad \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \in C_1' \\
&\qquad \left. \wedge c_1 = c \wedge C_1 = \{c\}) \Big) \right) \wedge \\
&\qquad \left(\exists C_2' : \bigvee_{c_2 \in C} \Big(\Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \notin C_2' \right. \right. \\
&\qquad \left. \wedge c_2 = min_{<_c}(C_2') \wedge C_2 = \{c\} \Big) \vee \\
&\qquad \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \in C_2' \\
&\qquad \left. \wedge c_2 = c \wedge C_2 = \{c\}) \Big) \right) \\
&\implies C' = C_1 \cap C_2 \wedge C_1 = \{c\} \wedge C_2 = \{c\} \\
&\iff C' = \{c\} \cap \{c\} = \{c\} \\
&\implies c \in C'
\end{aligned}
$$

Case 7: $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
&\exists \mu \in \hat{\Omega} : \exists C' \subseteq C : \forall c \in C : \exists \mu_1, \mu_2 \in \hat{\Omega}, \exists C_1, C_2 \subseteq C : \\
&\mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \wedge \\
&\left((\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 30}}{\iff} \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \\
&\qquad \left(\exists C_1' : \bigvee_{c_1 \in C} \Big(\Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_c}(C_1') \right. \right. \\
&\qquad \left. \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_1' \wedge C_1 = \{c\} \Big) \vee \\
&\qquad \Big((\mu_2, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \\
&\qquad \left. \left. \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge c_u \in C_1' \wedge C_1 = C_1' \Big) \Big) \right) \wedge \\
&\qquad \left(\exists C_2' : \bigvee_{c_2 \in C} \Big(\Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_c}(C_2') \right. \right. \\
&\qquad \left. \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_2' \wedge C_2 = \{c\} \Big) \vee \\
&\qquad \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \\
&\qquad \left. \left. \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge c_u \in C_2' \wedge C_2 = C_2' \Big) \Big) \right) \\
&\overset{\text{ind. hyp.}}{\Longrightarrow} C' = C_1 \cap C_2 \wedge (C_1 = \{c\} \vee (c_1 \in C_1 \wedge c_1 = c)) \wedge (C_2 = \{c\} \vee (c_2 \in C_2 \wedge c_2 = c)) \\
&\iff C' = \{c\} \cap \{c\} = \{c\} \vee \big(C' = \{c\} \cap C_2 \wedge c \in C_2\big) \vee \big(C' = C_1 \cap \{c\} \wedge c \in C_1\big) \vee \\
&\qquad \big(C' = C_1 \cap C_2 \wedge c \in C_1 \wedge c \in C_2\big) \\
&\Longrightarrow C' = \{c\} \vee c \in C' \\
&\Longrightarrow c \in C'
\end{aligned}
$$

□

B.3. Proof of Semantic Correctness and Completeness for the Distributed Query Execution Strategy With Triple Replication

The semantic correctness and completeness of the replication-aware distributed execution mechanism described in Sec. 4.4.2 is shown by proving Theorem 2.

Theorem 2. The centralized evaluation of query Q produces exact the same results as its replication-aware distributed evaluation, i.e.

$$[\![Q]\!]^{loc}_{\text{cover}} = [\![Q]\!]_G \ .$$

Proof. The proof of Theorem 2 is an induction over the structure of query Q. As induction bases it will be shown for all queries that do not contain subqueries. In the context of this thesis this means $Q = tp$. Based on this proof the induction hypothesis is raised that the theorem holds for queries B, B_1 and B_2. Afterwards the induction step is shown for the remaining query constructs that are the join $Q = B_1.B_2$ and the projection $Q = \texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}$.

Induction basis: $\mathbf{Q = tp}$.

$$\begin{aligned}
\mu \in [\![tp]\!]^{loc}_{\text{cover}} &\overset{\text{Def. 32}}{\Longleftrightarrow} \exists C' : (\mu, C') \in \bigcup_{c \in C} [\![tp]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{Def. 31}}{\Longleftrightarrow} \exists C' : \bigvee_{c \in C} \left(\text{dom}(\mu) = var(tp) \wedge \mu(tp) \in chunk_{\text{cover}}(c) \wedge C' = \text{cover}(\mu(tp))\right) \\
&\Longleftrightarrow \text{dom}(\mu) = var(tp) \wedge \exists C' : C' = \text{cover}(\mu(tp)) \wedge \bigvee_{c \in C} \mu(tp) \in chunk_{\text{cover}}(c) \\
&\overset{\text{Def. 2}}{\Longleftrightarrow} \text{dom}(\mu) = var(tp) \wedge \bigvee_{c \in C} \mu(tp) \in chunk_{\text{cover}}(c) \\
&\overset{\text{(B.1)}}{\Longleftrightarrow} \text{dom}(\mu) = var(tp) \wedge \mu(tp) \in G \\
&\overset{\text{Def. 14}}{\Longleftrightarrow} \mu \in [\![tp]\!]_G
\end{aligned}$$

Induction hypothesis: Theorem 2 is valid for queries B_1, B_2 and B.

For every basic graph pattern B the induction hypothesis is:

$$[\![B]\!]_G = [\![B]\!]^{loc}_{\text{cover}} \overset{\text{Def. 32}}{=} \left\{ \mu \,\middle|\, \exists C' : (\mu, C') \in \bigcup_{c \in C} [\![B]\!]^{loc,c}_{\text{cover}} \right\}$$

This can be rephrased as

$$\mu \in [\![B]\!]_G \Leftrightarrow \exists C' : \bigvee_{c \in C} (\mu, C') \in [\![B]\!]^{loc,c}_{\text{cover}} \ .$$

Induction step: $\mathbf{Q = B_1.B_2}$.

This proof is structured in a way that first $[\![B_1.B_2]\!]^{loc}_{\text{cover}}$ is transformed into a set S_{left} and then $[\![B_1.B_2]\!]_G$ is transformed into a set S_{right}. Finally, the equality of S_{left} and S_{right} is shown.

$$\begin{aligned}
\mu \in [\![B_1.B_2]\!]^{loc}_{\text{cover}} &\overset{\text{Def. 32}}{\iff} \exists C' : \bigvee_{c \in C} (\mu, C') \in [\![B_1.B_2]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{Def. 31}}{\iff} \exists C' : \bigvee_{c \in C} (\mu, C') \in \left(\bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \\
&\qquad \bowtie \left(\bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 29}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\qquad \wedge \bigvee_{c \in C} \left((\mu_1, C_1) \in \bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right. \\
&\qquad\qquad \left. \wedge (\mu_2, C_2) \in \bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\implies \mu \in S_{left}
\end{aligned}$$

Now, $[\![B_1.B_2]\!]_G$ is transformed into a set S_{right}.

$$\begin{aligned}
\mu \in [\![B_1.B_2]\!]_G &\overset{\text{Def. 14}}{\iff} \mu \in [\![B_1]\!]_G \bowtie [\![B_2]\!]_G \\
&\overset{\text{Def. 13}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \mu_1 \in [\![B_1]\!]_G \wedge \mu_2 \in [\![B_2]\!]_G \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} (\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \right) \\
&\qquad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} (\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \right) \\
&\implies \mu \in S_{right}
\end{aligned}$$

The sets S_{left} and S_{right} contain only the elements determined by the equations above. The equality of S_{left} and S_{right} is proven by distinguishing the following cases:

1. $\varnothing \in S_{left} \cap S_{right}$
2. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
3. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
4. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
5. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
6. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
7. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$

The proofs of the third case is analogue to the second case and the fifth case is analogue to the fourth case. Therefore, the proofs of the third and fifth case are not shown.

Case 1: $\varnothing \in \mathbf{S_{left}} \cap \mathbf{S_{right}}$

$$
\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = C \right) \right) \Bigg) \\
&\overset{C'=C \cap C = C \subseteq C}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c_2 = min_{<_C}(C_2') \right) \right) \Bigg) \\
&\overset{a \vee a = a}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c_2 = min_{<_C}(C_2') \right) \right) \\
&\overset{\text{case 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge c_1 = min_{<_C}(C_1') \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge c_2 = min_{<_C}(C_2') \right) \right) \\
&\overset{\text{Lemma 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} (\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \right) \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} (\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \right) \\
&\iff \mu \in S_{right}
\end{aligned}
$$

Case 2: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \mathrm{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_c}(C_1') \wedge C_1 = C \Big) \right) \\
&\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_c}(C_2') \wedge C_2 = \{c\} \big) \vee \\
&\left. \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big))) \right) \Bigg) \\
&\overset{\text{case 2}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge c_1 = min_{<_c}(C_1') \wedge C_1 = C \Big) \right) \\
&\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\big(c = min_{<_c}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_c}(C_2') \wedge C_2 = \{c\} \big) \vee \\
&\left. \big(c = min_{<_c}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big))) \right) \Bigg) \\
&\overset{\text{Lemma 1}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge C_1 = C \Big) \right) \\
&\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\big(c = min_{<_c}(C) \wedge c \notin C_2' \wedge C_2 = \{c\} \big) \vee \\
&\left. \big(c = min_{<_c}(C) \wedge c \in C_2' \wedge C_2 = \{c\} \big))) \right) \Bigg) \\
&\overset{\forall c \in C : \exists C' : C' = C \cap \{c\} = \{c\} \subseteq C}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \Big) \right) \\
&\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\left. \big(c = min_{<_c}(C) \wedge c \notin C_2' \big) \vee \big(c = min_{<_c}(C) \wedge c \in C_2' \big))) \right) \Bigg)
\end{aligned}
$$

$$\begin{aligned}
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \right) \right) \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \right) \right) \Bigg) \\
&\overset{a \vee a = a}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \right) \right) \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \right) \right) \\
&\iff \mu \in S_{right}
\end{aligned}$$

Case 4: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \right) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \big) \vee \\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge \\
&\quad c_u \in C_2' \wedge C_2 = C_2'))) \Bigg) \Bigg) \\
&\overset{\text{case 4}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
&\quad \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \right) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
&\quad \big(c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_2' \wedge C_2 = \{c\} \big) \vee \\
&\quad \big(c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge c_u \in C_2' \wedge C_2 = C_2'))) \Bigg) \Bigg)
\end{aligned}$$

$$\overset{\exists C' : \forall C_2 \in 2^C : C' = C \cap C_2 = C_2 \subseteq C}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge c_1 = min_{<_C}(C_1') \Big) \Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\big(c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \notin C_2' \big) \vee \big(c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c_u \wedge c_u \in C_2' \big) \big) \Big) \Bigg) \Bigg)$$

$$\overset{\text{Lemma 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \Big) \Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\big(\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \notin C_2' \big) \vee \big(c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c_u \wedge c_u \in C_2' \big) \big) \Big) \Bigg) \Bigg)$$

$$\overset{c_2 = c \Rightarrow c_u = c, \text{Lemma 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \Big) \Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\big(\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \notin C_2' \big) \vee \big(\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \in C_2' \big) \big) \Big) \Bigg) \Bigg)$$

$$\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \bigvee_{c \in C} \Bigg(\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \Big) \Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \Big) \Bigg) \Bigg)$$

$$\overset{a \vee a = a}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge \Bigg(\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \Big) \Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \Big) \Bigg) \Bigg)$$

$$\iff \mu \in S_{right}$$

□

Case 6: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_\mathbf{1} \sim \mu_\mathbf{2} \wedge \mu_\mathbf{1} \neq \varnothing \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S_{left} \overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_1' \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = \{c\} \big) \vee \\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_1' \wedge c_1 = c \wedge C_1 = \{c\} \big))) \Bigg) \\
\wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \big) \vee \\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big))) \Bigg) \Bigg) \\
\overset{\text{case 6}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
\big(c = min_{<_C}(C) \wedge c \notin C_1' \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = \{c\} \big) \vee \\
\big(c = min_{<_C}(C) \wedge c \in C_1' \wedge c_1 = c \wedge C_1 = \{c\} \big))) \Bigg) \\
\wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
\big(c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \big) \vee \\
\big(c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big))) \Bigg) \Bigg)
\end{aligned}
$$

$$\overset{\exists C':\forall c\in C:C'=\{c\}\cap\{c\}=\{c\}\subseteq C}{\Longleftrightarrow} \exists\mu_1,\mu_2:\mu=\mu_1\cup\mu_2\wedge\mu_1\sim\mu_2\wedge\bigvee_{c\in C}\Bigg($$
$$\Bigg(\exists C_1':\bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\wedge($$
$$\big(c=min_{<_C}(C)\wedge c\notin C_1'\wedge c_1=min_{<_C}(C_1')\big)\vee$$
$$\big(c=min_{<_C}(C)\wedge c\in C_1'\wedge c_1=c\big)\big)\Big)\Bigg)$$
$$\wedge\Bigg(\exists C_2':\bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\wedge($$
$$\big(c=min_{<_C}(C)\wedge c\notin C_2'\wedge c_2=min_{<_C}(C_2')\big)\vee$$
$$\big(c=min_{<_C}(C)\wedge c\in C_2'\wedge c_2=c\big)\big)\Big)\Bigg)\Bigg)$$

$$\overset{\text{Lemma 1}}{\Longleftrightarrow} \exists\mu_1,\mu_2:\mu=\mu_1\cup\mu_2\wedge\mu_1\sim\mu_2\wedge\bigvee_{c\in C}\Bigg($$
$$\Bigg(\exists C_1':\bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\wedge($$
$$\big(c=min_{<_C}(C)\wedge c\notin C_1'\big)\vee\big(c=min_{<_C}(C)\wedge c\in C_1'\big)\big)\Big)\Bigg)$$
$$\wedge\Bigg(\exists C_2':\bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\wedge($$
$$\big(c=min_{<_C}(C)\wedge c\notin C_2'\big)\vee\big(c=min_{<_C}(C)\wedge c\in C_2'\big)\big)\Big)\Bigg)\Bigg)$$

$$\Longleftrightarrow \exists\mu_1,\mu_2:\mu=\mu_1\cup\mu_2\wedge\mu_1\sim\mu_2\wedge\bigvee_{c\in C}\Bigg($$
$$\Bigg(\exists C_1':\bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\Big)\Bigg)\wedge\Bigg(\exists C_2':\bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\Big)\Bigg)\Bigg)$$

$$\overset{a\vee a=a}{\Longleftrightarrow} \exists\mu_1,\mu_2:\mu=\mu_1\cup\mu_2\wedge\mu_1\sim\mu_2\wedge$$
$$\Bigg(\exists C_1':\bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\Big)\Bigg)\wedge\Bigg(\exists C_2':\bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\Big)\Bigg)$$

$$\Longleftrightarrow \mu\in S_{right}$$

Case 7: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1.B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S_{left} \overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
c \notin C_1' \wedge C_1 = \{c\}\big) \vee \\
(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_1} \wedge \\
c_{u_1} \in C_1' \wedge C_1 = C_1')))\Bigg) \\
\wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
c \notin C_2' \wedge C_2 = \{c\}\big) \vee \\
(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_2} \wedge \\
c_{u_2} \in C_2' \wedge C_2 = C_2')))\Bigg)\Bigg) \\
\overset{\text{case 7}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \wedge \bigvee_{c \in C} \Bigg(\\
\Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
\big(c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_1' \wedge C_1 = \{c\}\big) \vee \\
\big(c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_1} \wedge c_{u_1} \in C_1' \wedge C_1 = C_1')\big)\Big)\Bigg) \\
\wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
\big(c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge c \notin C_2' \wedge C_2 = \{c\}\big) \vee \\
\big(c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_2} \wedge c_{u_2} \in C_2' \wedge C_2 = C_2')\big)\Big)\Bigg)\Bigg)
\end{aligned}
$$

$$\overset{\forall C_1,C_2\subseteq C: C_1\cap C_2\subseteq C}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu=\mu_1\cup\mu_2 \wedge \mu_1\sim\mu_2 \wedge \bigvee_{c\in C}\Bigg($$

$$\Bigg(\exists C_1' : \bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\Big.$$

$$\Big(c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c \wedge c\notin C_1'\Big)\vee$$

$$\Big.\Big(c_1=c\wedge\text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c_{u_1}\wedge c_{u_1}\in C_1'\Big)\Big)\Bigg)$$

$$\wedge\Bigg(\exists C_2' : \bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\Big.$$

$$\Big(c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c \wedge c\notin C_2'\Big)\vee$$

$$\Big.\Big(c_2=c\wedge\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c_{u_2}\wedge c_{u_2}\in C_2'\Big)\Big)\Bigg)\Bigg)$$

$$\overset{c_i=c\Rightarrow c_{u_i}=c,\text{Lemma 1}}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu=\mu_1\cup\mu_2 \wedge \mu_1\sim\mu_2 \wedge \bigvee_{c\in C}\Bigg($$

$$\Bigg(\exists C_1' : \bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\Big.$$

$$\Big(\text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c \wedge c\notin C_1'\Big)\vee$$

$$\Big.\Big(\text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c\wedge c\in C_1'\Big)\Big)\Bigg)$$

$$\wedge\Bigg(\exists C_2' : \bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\Big.$$

$$\Big(\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c \wedge c\notin C_2'\Big)\vee$$

$$\Big.\Big(\text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2\rangle\!\rangle))))=c\wedge c\in C_2'\Big)\Big)\Bigg)\Bigg)$$

$$\Longleftrightarrow \exists \mu_1,\mu_2 : \mu=\mu_1\cup\mu_2 \wedge \mu_1\sim\mu_2 \wedge \bigvee_{c\in C}\Bigg($$

$$\Bigg(\exists C_1' : \bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\Big)\Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\Big)\Bigg)\Bigg)$$

$$\overset{a\vee a=a}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu=\mu_1\cup\mu_2 \wedge \mu_1\sim\mu_2 \wedge$$

$$\Bigg(\exists C_1' : \bigvee_{c_1\in C}\Big((\mu_1,C_1')\in[\![B_1]\!]^{loc,c_1}_{\text{cover}}\Big)\Bigg) \wedge \Bigg(\exists C_2' : \bigvee_{c_2\in C}\Big((\mu_2,C_2')\in[\![B_2]\!]^{loc,c_2}_{\text{cover}}\Big)\Bigg)$$

$$\Longleftrightarrow \mu\in S_{right}$$

Induction step: Q = `SELECT` W `WHERE` {B}.

$$\begin{aligned}
\mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^{loc}_{\text{cover}} &\overset{\text{Def. 32}}{\iff} \exists C' : (\mu, C') \in \bigcup_{c \in C} [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{Def. 31}}{\iff} \exists C' : \exists \mu' : \mu = \mu'_{|_W} \wedge \bigvee_{c \in C} (\mu', C') \in [\![B]\!]^{loc,c}_{\text{cover}} \\
&\iff \exists \mu' : \mu = \mu'_{|_W} \wedge \exists C' : \bigvee_{c \in C} (\mu', C') \in [\![B]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu' : \mu = \mu'_{|_W} \wedge \mu' \in [\![B]\!]_G \\
&\overset{\text{Def. 14}}{\iff} \mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]_G
\end{aligned}$$

□

B.4. Proof of Unique Knows Theorem

Lemma 2. In case of a graph cover without triple replication, i.e., $\forall t \in G : |\text{cover}(t)| = 1$, the distributed query execution strategy and the replication-aware distributed query execution strategy will only produce located variable bindings (μ, C') with $|C'| = 1$ if $\mu \neq \varnothing$ for the query Q:

$$\forall c \in C : \left(\mu, C'\right) \in [\![Q]\!]^{loc,c}_{\text{cover}} \wedge \mu \neq \varnothing \Rightarrow |C'| = 1 \ .$$

Proof. The proof of Lemma 2 is an induction over the structure of query Q. As induction bases it will be shown for all queries that do not contain subqueries. In the context of this thesis this means $Q = tp$. Based on this proof the induction hypothesis is raised that the theorem holds for queries B, B_1 and B_2. Afterwards the induction step is shown for the remaining query constructs that are the join $Q = B_1.B_2$ and the projection $Q = \texttt{SELECT } W \texttt{ WHERE } \{B\}$.

Induction basis: Q = *tp*.

$$\begin{aligned}
\forall c \in C : \exists C' : (\mu, C') \in [\![tp]\!]^{loc,c}_{\text{cover}} &\overset{\text{Def. 31}}{\iff} \exists C' : \text{dom}(\mu) = var(tp) \wedge \mu(tp) \in chunk_{\text{cover}}(c) \wedge C' = \text{cover}(\mu(tp)) \\
&\overset{\forall t \in G : |\text{cover}(t)| = 1}{\Longrightarrow} |C'| = 1
\end{aligned}$$

Induction hypothesis: Lemma 2 is valid for queries B_1, B_2 and B.

Induction step: $\mathbf{Q = B_1.B_2}$.

$$\begin{aligned}
\forall c \in C : \exists C' : (\mu, C') \in [\![B_1.B_2]\!]^{loc,c}_{\text{cover}} \wedge \mu \neq \varnothing &\overset{\text{Def. 31}}{\iff} \exists C' : (\mu, C') \in \left(\bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \\
&\quad \bowtie \left(\bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 29}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \bigvee_{c_1 \in C} (\mu_1, C_1) \in \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \\
&\quad \wedge \bigvee_{c_2 \in C} (\mu_2, C_2) \in \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \wedge \mu \neq \varnothing \\
&\implies \mu \in S \wedge \mu \neq \varnothing
\end{aligned}$$

$(\mu, C') \in S \wedge \mu \neq \varnothing \Rightarrow |C'| = 1$ is proven by distinguishing the following cases:

1. $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
2. $\mu_1 \neq \varnothing \wedge \mu_2 = \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
3. $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
4. $\mu_1 \neq \varnothing \wedge \mu_2 = \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
5. $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
6. $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$

The proofs of the second case is analogue to the first case and the fourth case is analogue to the third case. Therefore, the proofs of the second and fourth case are not shown.

Case 1: $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1.B_2} \rangle\!\rangle) = \varnothing$

$$\begin{aligned}
\mu \in S &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_c}(C_1') \wedge C_1 = C \Big) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \\
&\qquad \wedge \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_c}(C_2') \wedge C_2 = \{c\} \big) \\
&\qquad \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_c}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big) \Big) \Bigg) \\
&\implies C' = C \cap \{c\} = \{c\} \\
&\implies |C'| = 1
\end{aligned}$$

Case 3: $\mu_1 = \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc, c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \Big) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc, c_2}_{\text{cover}} \wedge (\\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge \\
&\quad c_u \in C_2' \wedge C_2 = C_2'))) \Bigg) \\
&\overset{\text{ind. hyp.}}{\implies} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc, c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \Big) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc, c_2}_{\text{cover}} \wedge |C_2'| = 1 (\\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge \\
&\quad c_u \in C_2' \wedge C_2 = C_2'))) \Bigg) \\
&\overset{\text{Lemma 1}}{\implies} C' = C \cap \{c\} = \{c\} \\
&\implies |C'| = 1
\end{aligned}
$$

Case 5: $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S \overset{\text{Def. 30}}{\iff} & \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
& \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \\
& \wedge \big(\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_1' \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = \{c\} \big) \\
& \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_1' \wedge c_1 = c \wedge C_1 = \{c\} \big) \big) \Big) \Bigg) \\
& \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \\
& \wedge \big(\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \big) \\
& \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big) \big) \Big) \Bigg) \\
\Longrightarrow & C' = \{c\} \cap \{c\} = \{c\} \\
\Longrightarrow & |C'| = 1
\end{aligned}
$$

Case 6: $\mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S \overset{\text{Def. 30}}{\iff} & \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
& \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
& c \notin C_1' \wedge C_1 = \{c\} \big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_1} \wedge \\
& c_{u_1} \in C_1' \wedge C_1 = C_1'))) \Bigg) \\
& \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
& c \notin C_2' \wedge C_2 = \{c\} \big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_2} \wedge \\
& c_{u_2} \in C_2' \wedge C_2 = C_2'))) \Bigg)
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\text{ind. hyp.}}{\Longrightarrow} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge |C_1'| = 1 \, (\\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_c}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_1' \wedge C_1 = \{c\} \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_1} \wedge \\
&\quad c_{u_1} \in C_1' \wedge C_1 = C_1'))) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge |C_2'| = 1 \, (\\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_c}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_{u_2} \wedge \\
&\quad c_{u_2} \in C_2' \wedge C_2 = C_2'))) \Bigg) \\
&\overset{\text{Lemma 1}}{\Longrightarrow} C' = \{c\} \cap \{c\} = \{c\} \\
&\Longrightarrow |C'| = 1
\end{aligned}
$$

Induction step: Q = `SELECT` W `WHERE` {B}.

$$
\begin{aligned}
&\forall c \in C : \exists C' : (\mu, C') \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^{loc,c}_{\text{cover}} \wedge \mu \neq \varnothing \\
&\qquad \overset{\text{Def. 31}}{\Longleftrightarrow} \exists \mu' : \mu = \mu'_{|_W} \wedge (\mu', C') \in [\![B]\!]^{loc,c}_{\text{cover}} \wedge \mu \neq \varnothing \\
&\qquad \Longleftrightarrow \exists \mu' : \mu = \mu'_{|_W} \wedge (\mu', C') \in [\![B]\!]^{loc,c}_{\text{cover}} \wedge \mu \neq \varnothing \\
&\qquad\quad \wedge \mu' \neq \varnothing \\
&\qquad \overset{\text{ind. hyp.}}{\Longrightarrow} |C'| = 1
\end{aligned}
$$

□

B.5. Proof of Similarity of the Replication-Aware Extension

Theorem 3 says that in the case of graph cover strategies that do not replicate triples the distributed query execution strategy and the replication-aware distributed query execution strategy work identically.

Theorem 3. In case of a graph cover without triple replication, i.e., $\forall t \in G : |\text{cover}(t)| = 1$, the distributed query execution strategy and the replication-aware distributed query execution strategy evaluate the query Q identically:

$$\forall c \in C : \left\{ \mu \middle| (\mu, C') \in [\![Q]\!]^{loc,c}_{\text{cover}} \right\} = [\![Q]\!]^{c}_{\text{cover}} \ .$$

Proof. The proof of Theorem 3 is an induction over the structure of query Q. As induction bases it will be shown for all queries that do not contain subqueries. In the context of this thesis this means $Q = tp$. Based on this proof the induction hypothesis is raised that the theorem holds for queries B, B_1 and B_2. Afterwards the induction step is shown for the remaining query constructs that are the join $Q = B_1.B_2$ and the projection $Q = \texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}$.

Induction basis: $\mathbf{Q = tp}$.

$$\begin{aligned}
\forall c \in C : \exists C' : (\mu, C') \in [\![tp]\!]^{loc,c}_{\text{cover}} &\overset{\text{Def. 31}}{\iff} \exists C' : \text{dom}(\mu) = var(tp) \wedge \mu(tp) \in chunk_{\text{cover}}(c) \wedge C' = \text{cover}(\mu(tp)) \\
&\iff \text{dom}(\mu) = var(tp) \wedge \mu(tp) \in chunk_{\text{cover}}(c) \\
&\overset{\text{Def. 26}}{\iff} \mu \in [\![tp]\!]^{c}_{\text{cover}}
\end{aligned}$$

Induction hypothesis: Theorem 3 is valid for queries B_1, B_2 and B.

For every basic graph pattern B the induction hypothesis is:

$$[\![B]\!]^{c}_{\text{cover}} = \left\{ \mu \middle| \exists C' : (\mu, C') \in \bigcup_{c \in C} [\![B]\!]^{loc,c}_{\text{cover}} \right\}$$

This can be rephrased as

$$\mu \in [\![B]\!]^{c}_{\text{cover}} \Leftrightarrow \exists C' : (\mu, C') \in [\![B]\!]^{loc,c}_{\text{cover}} \ .$$

Induction step: $\mathbf{Q = B_1.B_2}$.

This proof is structured in a way that first $\{\mu | \exists C' : (\mu, C') \in [\![B_1.B_2]\!]^{loc,c}_{\text{cover}}\}$ is transformed into a set S_{left} and then $[\![B_1.B_2]\!]^{c}_{\text{cover}}$ is transformed into a set S_{right}. Finally, the equality of S_{left} and S_{right} is shown.

$$\begin{aligned}
\forall c \in C : \exists C' : (\mu, C') \in [\![B_1.B_2]\!]^{loc,c}_{\text{cover}} &\overset{\text{Def. 31}}{\iff} \exists C' : (\mu, C') \in \left(\bigcup_{c_1 \in C} \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \right) \\
&\qquad \bowtie \left(\bigcup_{c_2 \in C} \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 29}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\qquad \wedge \bigvee_{c_1 \in C} (\mu_1, C_1) \in \text{route}^{\text{loc}}(c_1, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{loc,c_1}_{\text{cover}}) \\
&\qquad \wedge \bigvee_{c_2 \in C} (\mu_2, C_2) \in \text{route}^{\text{loc}}(c_2, c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{loc,c_2}_{\text{cover}}) \\
&\Longrightarrow \mu \in S_{left}
\end{aligned}$$

Now, $[\![B_1.B_2]\!]^c_{\text{cover}}$ is transformed into a set S_{right}.

$$\begin{aligned}
\forall c \in C : \mu \in [\![B_1.B_2]\!]^c_{\text{cover}} &\overset{\text{Def. 26}}{\iff} \mu \in \left(\bigcup_{c_1 \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_1}_{\text{cover}}) \right) \\
&\quad \bowtie \left(\bigcup_{c_2 \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_2}_{\text{cover}}) \right) \\
&\overset{\text{Def. 13}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \bigvee_{c_1 \in C} \mu_1 \in \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c_1}_{\text{cover}}) \\
&\quad \wedge \bigvee_{c_2 \in C} \mu_2 \in \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c_2}_{\text{cover}}) \\
&\implies \mu \in S_{right}
\end{aligned}$$

The sets S_{left} and S_{right} contain only the elements determined by the equations above. The equality of S_{left} and S_{right} is proven by distinguishing the following cases:

1. $\varnothing \in S_{left} \cap S_{right}$
2. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
3. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
4. $\mu \in S_{left} \cap S_{right} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
5. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \varnothing \wedge \mu_1 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$
6. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing$
7. $\mu \in S_{left} \cap S_{right} \wedge \mu = \mu_1 \sim \mu_2 \wedge \mu_1 \neq \varnothing \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing$

The proofs of the third case is analogue to the second case and the fifth case is analogue to the fourth case. Therefore, the proofs of the third and fifth case are not shown.

Case 1: $\varnothing \in \mathbf{S_{left} \cap S_{right}}$

$$\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = C \right) \right)
\end{aligned}$$

$$\overset{C'=C\cap C=C\subseteq C}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2$$
$$\wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \Big) \right)$$
$$\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \wedge c_2 = min_{<_C}(C_2') \Big) \right)$$
$$\overset{\text{Lemma 1}}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2$$
$$\wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \Big) \right)$$
$$\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \mu_2 = \varnothing \Big) \right)$$
$$\overset{\text{ind. hyp.}}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2$$
$$\wedge \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \mu_1 = \varnothing) \right)$$
$$\wedge \left(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \mu_2 = \varnothing) \right)$$
$$\overset{\text{Def. 25}}{\Longleftrightarrow} \mu \in S_{right}$$

Case 2: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_2 \wedge \mu_2 \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1.B_2} \rangle\!\rangle) = \varnothing$

$$\mu \in S_{left} \overset{\text{Def. 30}}{\Longleftrightarrow} \exists C', C_1, C_2 : \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2$$
$$\wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \Big) \right)$$
$$\wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \right.$$
$$\wedge \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \big)$$
$$\left. \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big) \big) \right)$$

$$
\begin{aligned}
&\overset{C'=C\cap\{c\}=\{c\}\subseteq C}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]_{\text{cover}}^{loc,c_1} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \right) \right) \\
&\qquad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]_{\text{cover}}^{loc,c_2} \\
&\qquad\quad \wedge \big(\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \big) \\
&\qquad\quad \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \big) \big) \Big) \Bigg) \\
&\overset{\text{Lemma 1}}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]_{\text{cover}}^{loc,c_1} \wedge \mu_1 = \varnothing \right) \right) \\
&\qquad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]_{\text{cover}}^{loc,c_2} \\
&\qquad\quad \wedge \big(\big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \big) \\
&\qquad\quad \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \big) \big) \Big) \Bigg) \\
&\Longleftrightarrow \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]_{\text{cover}}^{loc,c_1} \wedge \mu_1 = \varnothing \right) \right) \\
&\qquad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]_{\text{cover}}^{loc,c_2} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \right) \right) \\
&\overset{\text{ind. hyp.}}{\Longleftrightarrow} \exists \mu_1,\mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]_{\text{cover}}^{c_1} \wedge \mu_1 = \varnothing) \right) \\
&\qquad \wedge \left(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]_{\text{cover}}^{c_2} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C)) \right) \\
&\overset{\text{Def. 25}}{\Longleftrightarrow} \mu \in S_{right}
\end{aligned}
$$

Case 4: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \varnothing \sim \mu_\mathbf{2} \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1.B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = C \Big) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\quad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge \\
&\quad \left. c_u \in C_2' \wedge C_2 = C_2'))) \right) \\
&\overset{C'=C \cap C_2 = C_2 \subseteq C}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \Big) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c_2 = min_{<_C}(C_2') \wedge c \notin C_2' \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c_u \wedge \\
&\quad \left. c_2 = c \wedge c_u \in C_2'))) \right) \\
&\overset{\text{Lemma 2}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \mu_1 = \varnothing \wedge c_1 = min_{<_C}(C_1') \Big) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\right. \\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad c_2 = min_{<_C}(C_2') \wedge c \notin C_2' \big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle)))) = c \wedge \\
&\quad \left. c_2 = c \wedge c \in C_2'))) \right)
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\text{Lemma 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]_{\text{cover}}^{loc,c_1} \wedge \mu_1 = \varnothing \right) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]_{\text{cover}}^{loc,c_2} \wedge (\\
&\qquad \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \notin C_2' \big) \\
&\qquad \vee \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge c \in C_2' \big) \big) \Big) \Bigg) \\
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]_{\text{cover}}^{loc,c_1} \wedge \mu_1 = \varnothing \right) \right) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]_{\text{cover}}^{loc,c_2} \wedge (\\
&\qquad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c) \Big) \Bigg) \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]_{\text{cover}}^{c_1} \wedge \mu_1 = \varnothing) \right) \\
&\quad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]_{\text{cover}}^{c_2} \wedge (\\
&\qquad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c)) \Bigg) \\
&\overset{\text{Def. 25}}{\iff} \mu \in S_{right}
\end{aligned}
$$

Case 6: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_{\mathbf{1}} \sim \mu_{\mathbf{2}} \wedge \mu_{\mathbf{1}} \neq \varnothing \wedge \mu_{\mathbf{2}} \neq \varnothing \wedge \mathrm{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) = \varnothing$

$$
\begin{aligned}
\mu \in S_{left} &\overset{\text{Def. 30}}{\iff} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_1' \wedge c_1 = min_{<_C}(C_1') \wedge C_1 = \{c\} \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_1' \wedge c_1 = c \wedge C_1 = \{c\} \big) \big) \big) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \wedge C_2 = \{c\} \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \wedge C_2 = \{c\} \big) \big) \big) \Bigg) \\
&\overset{C'=\{c\}\cap\{c\}=\{c\}\subseteq C}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_1' \wedge c_1 = min_{<_C}(C_1') \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_1' \wedge c_1 = c \big) \big) \big) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \wedge c_2 = min_{<_C}(C_2') \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \wedge c_2 = c \big) \big) \big) \Bigg) \\
&\overset{\text{Lemma 1}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_1' \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_1' \big) \big) \big) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \\
&\quad \wedge \big(\big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \notin C_2' \big) \\
&\quad \vee \big(\mathrm{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \wedge c \in C_2' \big) \big) \big) \Bigg)
\end{aligned}
$$

$$
\begin{aligned}
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \left(\exists C_1' : \bigvee_{c_1 \in C} \left((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \right) \right) \\
&\quad \wedge \left(\exists C_2' : \bigvee_{c_2 \in C} \left((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C) \right) \right) \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \left(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C)) \right) \\
&\qquad \wedge \left(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) = \varnothing \wedge c = min_{<_C}(C)) \right) \\
&\overset{\text{Def. 25}}{\iff} \mu \in S_{right}
\end{aligned}
$$

Case 7: $\mu \in \mathbf{S_{left}} \cap \mathbf{S_{right}} \wedge \mu = \mu_\mathbf{1} \sim \mu_\mathbf{2} \wedge \mu_\mathbf{1} \neq \varnothing \wedge \mu_\mathbf{2} \neq \varnothing \wedge \text{cVars}(\langle\!\langle \mathbf{B_1}.\mathbf{B_2} \rangle\!\rangle) \neq \varnothing$

$$
\begin{aligned}
\mu \in S_{left} \overset{\text{Def. 30}}{\iff} & \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
& \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \notin C_1' \wedge C_1 = \{c\}\big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c_{u_1} \wedge \\
& c_{u_1} \in C_1' \wedge C_1 = C_1'))) \Bigg) \\
& \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \notin C_2' \wedge C_2 = \{c\}\big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c_{u_2} \wedge \\
& c_{u_2} \in C_2' \wedge C_2 = C_2'))) \Bigg) \\
\overset{\text{Thm. 2}}{\iff} & \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
& \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = min_{<_C}(C_1') \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \notin C_1' \wedge C_1 = \{c\}\big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_1 = c \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \in C_1' \wedge C_1 = C_1'))) \Bigg) \\
& \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
& \big(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = min_{<_C}(C_2') \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \notin C_2' \wedge C_2 = \{c\}\big) \\
& \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge c_2 = c \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
& c \in C_2' \wedge C_2 = C_2'))) \Bigg)
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\text{Lemma 1}}{\Longleftrightarrow} \exists C', C_1, C_2 : \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \wedge C' = C_1 \cap C_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \notin C_1' \wedge C_1 = \{c\} \Big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \in C_1' \wedge C_1 = C_1'))) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \notin C_2' \wedge C_2 = \{c\} \Big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \in C_2' \wedge C_2 = C_2'))) \Bigg) \\
&\overset{\forall C_1, C_2 \subseteq C : C_1 \cap C_2 \subseteq C}{\Longleftrightarrow} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \notin C_1' \Big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \in C_1'))) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
&\quad (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \notin C_2' \Big) \\
&\quad \vee (\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))))) = c \wedge \\
&\quad c \in C_2'))) \Bigg)
\end{aligned}
$$

$$
\begin{aligned}
&\iff \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\quad \wedge \Bigg(\exists C_1' : \bigvee_{c_1 \in C} \Big((\mu_1, C_1') \in [\![B_1]\!]^{loc,c_1}_{\text{cover}} \wedge (\\
&\qquad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \\
&\quad \wedge \Bigg(\exists C_2' : \bigvee_{c_2 \in C} \Big((\mu_2, C_2') \in [\![B_2]\!]^{loc,c_2}_{\text{cover}} \wedge (\\
&\qquad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu_1, \mu_2 : \mu = \mu_1 \cup \mu_2 \wedge \mu_1 \sim \mu_2 \\
&\qquad \wedge \Bigg(\bigvee_{c_1 \in C} (\mu_1 \in [\![B_1]\!]^{c_1}_{\text{cover}} \wedge (\\
&\qquad\quad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_1(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \\
&\qquad \wedge \Bigg(\bigvee_{c_2 \in C} (\mu_2 \in [\![B_2]\!]^{c_2}_{\text{cover}} \wedge (\\
&\qquad\quad \text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle) \neq \varnothing \wedge \text{jResp}(\mu_2(min_{<_V}(\text{cVars}(\langle\!\langle B_1.B_2 \rangle\!\rangle))) = c)) \Bigg) \\
&\overset{\text{Def. 25}}{\iff} \mu \in S_{right}
\end{aligned}
$$

Induction step: Q = `SELECT` W `WHERE` {B}.

$$
\begin{aligned}
\forall c \in C : \exists C' : (\mu, C') \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^{loc,c}_{\text{cover}} &\overset{\text{Def. 31}}{\iff} \exists C' : \exists \mu' : \mu = \mu'_{|_W} \wedge (\mu', C') \in [\![B]\!]^{loc,c}_{\text{cover}} \\
&\overset{\text{ind. hyp.}}{\iff} \exists \mu' : \mu = \mu'_{|_W} \wedge \mu' \in [\![B]\!]^{c}_{\text{cover}} \\
&\overset{\text{Def. 26}}{\iff} \mu \in [\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]^{c}_{\text{cover}}
\end{aligned}
$$

□

APPENDIX C

Additional Evaluation Details

C.1. Characteristics of the Used Data Sets

Table C.1 shows the characteristics of the 500M, 1G and 2G triples subsets of the real-world billion triple challenge dataset from 2014 [76] as well as both data sets generated with the Waterloo SPARQL Diversity Test Suite v0.6 [15] used in the evaluations.

dataset	BTC500M	BTC1000M	BTC2000M	WatDiv100M	WatDiv1000M
# triples	500M	1,000M	2,000M	109,786,094	1,099,208,068
# unique top level domains	9k	13k	21k	1	1
# unique subjects	50,602k	90,713k	170,246k	5,212k	52,120k
# unique properties	179k	412k	812k	86	86
# unique objects	76,642k	147,625k	259,945k	9,743k	92,220k

Table C.1.: Data set characteristics.

C.2. Generated Queries

Queries for the BTC2014 Data Sets

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://purl.org/ontology/bibo/Webpage> ?v1.
    ?v1 <http://purl.org/dc/terms/created> ?v2.
} LIMIT 1000000
```

Listing C.1: Query 1: `so #tp=2 #ds=1 sel=0.001`

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://vivo.ufl.edu/ontology/vivo-ufl/dateTimeIntervalFor> ?v1.
    ?v1 <http://www.w3.org/2000/01/rdf-schema#label> ?v2.
} LIMIT 1000000
```

Listing C.2: Query 2: `so #tp=2 #ds=1 sel=0.01`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v1.
    ?v1 <http://purl.org/dc/terms/hasPart> ?v2.
    ?v2 <http://www.metalex.eu/metalex/2008-05-02#variant> ?v3.
    ?v3 <http://www.metalex.eu/metalex/2008-05-02#fragment> ?v4.
    ?v4 <http://www.metalex.eu/metalex/2008-05-02#fragment> ?v5.
    ?v5 <http://xmlns.com/foaf/0.1/primaryTopic> ?v6.
    ?v6 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v7.
    ?v7 <http://purl.org/dc/terms/hasPart> ?v8.
} LIMIT 1000000
```

Listing C.3: Query 3: `so #tp=8 #ds=1 sel=0.001`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://vivo.ufl.edu/ontology/vivo-ufl/dateTimeIntervalFor> ?v1.
    ?v1 <http://vivoweb.org/ontology/core#contributingRole> ?v2.
    ?v2 <http://vivoweb.org/ontology/core#teacherRoleOf> ?v3.
    ?v3 <http://vivo.ufl.edu/ontology/vivo-ufl/homeDept> ?v4.
    ?v4 <http://vivo.ufl.edu/ontology/vivo-ufl/homeDeptFor> ?v5.
    ?v5 <http://vivo.ufl.edu/ontology/vivo-ufl/homeDept> ?v6.
    ?v6 <http://vivoweb.org/ontology/core#subOrganizationWithin> ?v7.
    ?v7 <http://www.w3.org/2000/01/rdf-schema#label> ?v8.
} LIMIT 1000000
```

Listing C.4: Query 4: `so #tp=8 #ds=1 sel=0.01`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v1.
    ?v1 <http://www.w3.org/2007/05/powder-s#describedby> ?v2.
    ?v2 <http://www.openlinksw.com/schema/attribution#isDescribedUsing> ?v3.
    ?v3 <http://www.w3.org/2007/05/powder-s#describedby> ?v4.
    ?v4 <http://www.openlinksw.com/schema/attribution#isDescribedUsing> ?v5.
    ?v5 <http://www.w3.org/2007/05/powder-s#describedby> ?v6.
    ?v6 <http://www.openlinksw.com/schema/attribution#isDescribedUsing> ?v7.
    ?v7 <http://purl.org/dc/terms/title> ?v8.
} LIMIT 1000000
```

Listing C.5: Query 5: `so #tp=8 #ds=3 sel=0.001`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v1.
    ?v1 <http://purl.org/dc/terms/hasFormat> ?v2.
    ?v2 <http://purl.org/dc/terms/hasVersion> ?v3.
    ?v3 <http://purl.org/dc/terms/isFormatOf> ?v4.
    ?v4 <http://purl.org/dc/terms/hasFormat> ?v5.
    ?v5 <http://purl.org/dc/terms/isVersionOf> ?v6.
    ?v6 <http://purl.org/dc/terms/type> ?v7.
    ?v7 <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?v8.
} LIMIT 1000000
```

Listing C.6: Query 6: `so #tp=8 #ds=3 sel=0.01`

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#realizedBy> ?v1.
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#fragment> ?v2.
} LIMIT 1000000
```

Listing C.7: Query 7: `ss #tp=2 #ds=1 sel=0.001`

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://vivo.ufl.edu/ontology/vivo-ufl/dateTimeIntervalFor> ?v1.
    ?v0 <http://vivoweb.org/ontology/core#start> ?v2.
} LIMIT 1000000
```

Listing C.8: Query 8: `ss #tp=2 #ds=1 sel=0.01`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#realizedBy> ?v1.
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#fragment> ?v2.
    ?v0 <http://xmlns.com/foaf/0.1/isPrimaryTopicOf> ?v3.
    ?v0 <http://purl.org/vocab/frbr/core#realization> ?v4.
    ?v0 <http://purl.org/dc/terms/valid> ?v5.
    ?v0 <http://purl.org/dc/terms/description> ?v6.
    ?v0 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v7.
    ?v0 <http://purl.org/dc/terms/created> ?v8.
} LIMIT 1000000
```

Listing C.9: Query 9: `ss #tp=8 #ds=1 sel=0.001`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#realizedBy> ?v1.
    ?v0 <http://www.metalex.eu/metalex/2008-05-02#fragment> ?v2.
    ?v0 <http://purl.org/vocab/frbr/core#realization> ?v3.
    ?v0 <http://purl.org/dc/terms/description> ?v4.
    ?v0 <http://xmlns.com/foaf/0.1/isPrimaryTopicOf> ?v5.
    ?v0 <http://purl.org/dc/terms/type> ?v6.
    ?v0 <http://purl.org/dc/terms/created> ?v7.
    ?v0 <http://purl.org/dc/terms/title> ?v8.
} LIMIT 1000000
```

Listing C.10: Query 10: `ss #tp=8 #ds=1 sel=0.01`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.w3.org/1999/xhtml/vocab#stylesheet> ?v1.
    ?v0 <http://www.w3.org/1999/xhtml/vocab#icon> ?v2.
    ?v0 <http://ogp.me/ns#description> ?v3.
    ?v0 <http://www.w3.org/1999/xhtml/vocab#next> ?v4.
    ?v0 <http://ogp.me/ns#title> ?v5.
    ?v0 <http://ogp.me/ns#url> ?v6.
    ?v0 <http://ogp.me/ns#type> ?v7.
    ?v0 <http://www.w3.org/1999/xhtml/vocab#prev> ?v8.
} LIMIT 1000000
```

Listing C.11: Query 11: `ss #tp=8 #ds=3 sel=0.001`

```
SELECT ?v0 ?v8 WHERE {
    ?v0 <http://www.w3.org/2002/07/owl#equivalentClass> ?v1.
    ?v0 <http://www.w3.org/2004/02/skos/core#prefLabel> ?v2.
    ?v0 <http://www.w3.org/2003/06/sw-vocab-status/ns#term_status> ?v3.
    ?v0 <http://purl.obolibrary.org/obo/IAO_0000111> ?v4.
    ?v0 <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> ?v5.
    ?v0 <http://www.w3.org/2000/01/rdf-schema#label> ?v6.
    ?v0 <http://www.w3.org/2000/01/rdf-schema#comment> ?v7.
    ?v0 <http://eagle-i.org/ont/app/1.0/preferredLabel> ?v8.
} LIMIT 1000000
```

Listing C.12: Query 12: `ss #tp=8 #ds=3 sel=0.01`

Queries for the WatDiv Data Sets

Since the queries generated for the WatDiv100M and the WatDiv1000M data sets were identically, they are only listed once. In order to improve the layout some line breaks were inserted into some triple patterns without changing their semantics.

```
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 C.13: Query `C1`

```
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 C.14: Query `C2`

```
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 C.15: Query C3

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
        ?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic57> .
        ?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 C.16: Query F1

```
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/SubGenre49> .
}
```

Listing C.17: Query F2

```
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/SubGenre23> .
        ?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 C.18: Query F3

```
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/Topic39> .
        ?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 C.19: Query F4

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
        ?v0 <http://purl.org/goodrelations/includes> ?v1 .
        <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer6238>
                        <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 C.20: Query F5

```
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/Website15805> .
        ?v2 <http://schema.org/caption> ?v3 .
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v2 .
}
```

Listing C.21: Query L1

```
SELECT ?v1 ?v2 WHERE {
        <http://db.uwaterloo.ca/~galuc/wsdbm/City142>
                        <http://www.geonames.org/ontology#parentCountry> ?v1 .
        ?v2 <http://db.uwaterloo.ca/~galuc/wsdbm/likes>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/Product0> .
        ?v2 <http://schema.org/nationality> ?v1 .
}
```

Listing C.22: Query L2

```
SELECT ?v0 ?v1 WHERE {
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/Website47601> .
}
```

Listing C.23: Query L3

```
SELECT ?v0 ?v2 WHERE {
        ?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic167> .
        ?v0 <http://schema.org/caption> ?v2 .
}
```

Listing C.24: Query L4

```
SELECT ?v0 ?v1 ?v3 WHERE {
        ?v0 <http://schema.org/jobTitle> ?v1 .
        <http://db.uwaterloo.ca/~galuc/wsdbm/City212>
                        <http://www.geonames.org/ontology#parentCountry> ?v3 .
        ?v0 <http://schema.org/nationality> ?v3 .
}
```

Listing C.25: Query L5

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
        ?v0 <http://purl.org/goodrelations/includes> ?v1 .
        <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer3584>
                        <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 C.26: Query S1

```
SELECT ?v0 ?v1 ?v3 WHERE {
        ?v0 <http://purl.org/dc/terms/Location> ?v1 .
        ?v0 <http://schema.org/nationality>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/Country18> .
        ?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 C.27: Query S2

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
        ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory8> .
        ?v0 <http://schema.org/caption> ?v2 .
        ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v3 .
        ?v0 <http://schema.org/publisher> ?v4 .
}
```

Listing C.28: Query S3

```
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 <http://xmlns.com/foaf/age> <http://db.uwaterloo.ca/~galuc/wsdbm/AgeGroup7> .
        ?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 C.29: Query S4

```
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory3> .
        ?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 C.30: Query S5

```
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/SubGenre57> .
}
```

Listing C.31: Query S6

```
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/User97304>
                        <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}
```

Listing C.32: Query S7

C.3. Query Execution Times

Table C.2 shows the query execution times of all graph covers for the 10 slaves and the 1 billion triples data set BTC1000M in seconds. Table C.3 shows the query execution times of the molecule hash covers with the varying molecule diameters for 10 slaves and the BTC1000M data set in seconds. Table C.4 shows the query execution times for the different graph cover strategies for the 1 billion triples data set BTC1000M when using 10, 20 and 40 slaves in seconds. Additionally, this table includes the execution times for the centralized query execution. Table C.5 shows the query execution times for the different graph cover strategies for 20 slaves when using the 500 million, 1 billion and 2 billion triples data sets (i.e., BTC500M, BTC1000M and

BTC2000M) in seconds. When interpreting the query execution times, it should be noted that the number of returned results differ for the different data set sizes and the 2-hop hash cover. Query execution times of aborted queries are highlighted. For the WatDiv100M and WatDiv1000M data sets the query execution times are shown in Table C.6 and Table C.7, respectively.

query	hash	hierarchical	minimal edge-cut	vertical	2-hop hash	molecule hash	MEC over
q_1	2.2	2.4	2.5	45.1	1.1	2.2	0.7
q_2	16.5	15.1	23.4	3,502.1	354.8	14.6	5.4
q_3	245.4	244.2	261.4	1,133.2	143.7	108.2	15.9
q_4	45.6	42.0	49.9	3,459.8	289.8	32.2	27.4
q_5	**121.5**	**76.1**	**55.4**	**2,516.0**	**24.5**	**43.4**	**29.2**
q_6	2,505.6	2,460.0	2,311.4	2,455.7	**91.0**	1,926.5	69.2
q_7	**36.0**	**38.3**	**54.6**	**57.4**	**39.9**	**33.8**	**38.8**
q_8	4.4	4.4	7.3	44.7	14.2	2.5	0.1
q_9	**22.5**	**23.6**	**25.6**	**201.0**	**34.4**	**22.8**	**173.7**
q_{10}	**21.9**	**24.0**	**23.8**	**1,176.5**	**30.9**	**22.1**	**83.8**
q_{11}	**28.0**	**22.0**	**38.7**	**1,099.1**	**28.2**	**37.3**	**10.3**
q_{12}	8.0	25.7	20.8	3,908.2	667.8	10.6	2.4

Table C.2.: Query execution times in sec for the comparison of the different graph cover strategies using the BTC1000M data set and 10 slaves.

query	diameter 2	diameter 3	diameter 4	diameter 5	diameter 6	diameter 7
q_1	2.2	2.2	0.6	0.7	2.2	0.3
q_2	12.7	14.6	14.2	16.7	15.5	15.6
q_3	166.1	108.2	105.4	104.8	100.8	100.8
q_4	37.1	32.2	30.7	29.1	30.7	33.2
q_5	**53.3**	**43.4**	**52.9**	**55.1**	**50.6**	**62.6**
q_6	2,197.1	1,926.5	1,865.3	1,897.6	1,875.9	1,847.6
q_7	**43.2**	**33.8**	**39.5**	**41.9**	**35.6**	**46.2**
q_8	3.0	2.5	2.9	2.9	2.5	2.9
q_9	**26.9**	**22.8**	**24.1**	**25.1**	**22.9**	**24.0**
q_{10}	**24.8**	**22.1**	**23.6**	**24.0**	**22.2**	**24.2**
q_{11}	**36.7**	**37.3**	**35.6**	**39.0**	**30.7**	**33.1**
q_{12}	7.8	10.6	8.8	8.8	10.0	8.2

Table C.3.: Query execution times in sec for the different diameters of the molecule hash cover using the BTC1000M data set and 10 slaves.

	hash			hierarchical			minimal edge-cut		
query	10	20	40	10	20	40	10	20	40
q_1	2.2	5.4	11.5	2.4	5.6	12.8	2.5	5.4	8.0
q_2	16.5	18.0	15.2	15.1	12.0	19.3	23.4	13.6	16.6
q_3	245.4	174.1	188.9	244.2	162.9	172.6	261.4	168.4	174.2
q_4	45.6	53.6	81.0	42.0	57.6	103.3	49.9	56.4	88.0
q_5	121.5	62.0	45.0	76.1	39.2	56.4	55.4	49.0	91.8
q_6	2,505.6	3,756.1	2,799.9	2,460.0	3,500.4	2,186.9	2,311.4	2,206.7	2,639.7
q_7	36.0	20.3	36.9	38.3	50.6	57.8	54.6	47.4	60.2
q_8	4.4	4.4	8.0	4.4	5.5	8.0	7.3	4.3	8.0
q_9	22.5	25.2	24.4	23.6	25.1	35.6	25.6	24.6	40.1
q_{10}	21.9	24.9	27.5	24.0	34.7	44.3	23.8	24.8	39.8
q_{11}	28.0	22.5	25.0	22.0	24.0	41.4	38.7	24.3	36.5
q_{12}	8.0	6.0	3.3	25.7	7.5	3.4	20.8	11.1	4.1

(a) Query execution times for frequently used graph cover strategies.

	molecule hash			overpartitioned minimal edge-cut			centralized
query	10	20	40	10	20	40	
q_1	2.2	4.3	8.0	0.7	0.4	0.4	7.8
q_2	14.6	9.6	13.6	5.4	16.6	25.7	238.2
q_3	108.2	87.9	107.8	15.9	30.2	59.8	69.6
q_4	32.2	44.0	70.2	27.4	29.6	59.7	276.2
q_5	43.4	41.2	79.3	29.2	15.0	27.0	55.0
q_6	1,926.5	2,945.9	3,643.2	69.2	24.0	61.9	247.3
q_7	33.8	40.0	55.7	38.8	21.2	33.6	23.6
q_8	2.5	4.0	7.9	0.1	8.9	19.8	3.3
q_9	22.8	26.1	45.3	173.7	45.2	59.6	49.4
q_{10}	22.1	25.6	57.6	83.8	38.6	59.4	57.2
q_{11}	37.3	23.3	36.8	10.3	3.0	59.9	124.9
q_{12}	10.6	5.8	1.8	2.4	30.6	59.9	313.6

(b) Query execution times for molecule hash cover, overpartitioned minimal edge-cut cover and the centralized setting.

Table C.4.: Query execution times in sec when scaling the number of slaves for the BTC1000M data set.

	hash			hierarchical			minimal edge-cut		
query	500M	1000M	2000M	500M	1000M	2000M	500M	1000M	2000M
q_1	5.7	5.4	5.5	4.0	5.6	5.4	5.1	5.4	5.4
q_2	10.3	18.0	19.6	9.4	12.0	89.5	15.1	13.6	80.1
q_3	97.8	174.1	1,516.7	94.6	162.9	1,956.8	99.8	168.4	2,290.7
q_4	46.9	53.6	64.5	34.4	57.6	65.0	49.3	56.4	70.0
q_5	**27.6**	**62.0**	**46.1**	**28.0**	**39.2**	**51.4**	**42.9**	**49.0**	**52.9**
q_6	2,883.0	3,756.1	**2,561.1**	2,570.5	3,500.4	**2,613.4**	1,368.8	2,206.7	**1,929.2**
q_7	**27.9**	**20.3**	**43.1**	**29.0**	**50.6**	**39.7**	**28.9**	**47.4**	**42.0**
q_8	5.0	4.4	8.7	3.9	5.5	8.4	4.0	4.3	5.0
q_9	**25.7**	**25.2**	**29.1**	**25.2**	**25.1**	**25.0**	**25.8**	**24.6**	**24.6**
q_{10}	**26.7**	**24.9**	**26.1**	**30.5**	**34.7**	**23.4**	**29.2**	**24.8**	**23.7**
q_{11}	**22.3**	**22.5**	**32.9**	**21.2**	**24.0**	**33.2**	**23.7**	**24.3**	**23.8**
q_{12}	1.3	6.0	12.2	3.4	7.5	14.2	4.4	11.1	19.6

(a) Query execution times for frequently used graph cover strategies.

	molecule hash			overpartitioned minimal edge-cut		
query	500M	1000M	2000M	500M	1000M	2000M
q_1	4.9	4.3	4.2	0.4	0.4	0.7
q_2	8.2	9.6	17.6	11.6	16.6	24.6
q_3	67.0	87.9	2,258.9	29.1	30.2	8.4
q_4	39.6	44.0	53.6	29.2	29.6	37.3
q_5	**50.3**	**41.2**	**40.1**	**19.4**	**15.0**	**14.0**
q_6	2,859.5	2,945.9	**1,583.8**	4.9	24.0	**44.3**
q_7	**34.7**	**40.0**	**40.1**	**21.5**	**21.2**	**23.3**
q_8	3.9	4.0	4.1	8.7	8.9	11.1
q_9	**32.0**	**26.1**	**25.5**	**30.0**	**45.2**	**427.0**
q_{10}	**32.7**	**25.6**	**24.9**	**29.3**	**38.6**	**257.5**
q_{11}	**26.4**	**23.3**	**29.5**	**1.3**	**3.0**	**19.1**
q_{12}	4.0	5.8	13.1	29.3	30.6	67.8

(b) Query execution times for molecule hash cover and the overpartitioned minimal edge-cut cover.

Table C.5.: Query execution times in sec when scaling the dataset size for 20 slaves.

query	hash	hierarchical	minimal edge-cut	molecule hash	MEC over
C1	68.3	67.9	182.4	64.4	3.3
C2	444.7	443.7	816.6	34.5	19.1
C3	2,581.2	3,580.9	3,584.0	1,242.4	1,519.6
F1	22.3	24.4	32.8	22.2	12.1
F2	3,420.3	3,427.0	3,414.7	3,379.0	3,389.5
F3	2,884.8	2,886.5	2,886.5	2.884.0	2.881.3
F4	3,226.0	3,210.5	3,199.0	247.1	3,166.8
F5	6.9	7.7	9.8	6.0	3.4
L1	28.9	28.6	29.6	29.1	29.1
L2	0.1	0.1	0.1	0.1	0.04
L3	0.5	0.3	1.8	0.5	0.3
L4	1.9	1.9	1.9	1.9	1.9
L5	50.6	50.5	49.9	50.7	47.9
S1	17.2	16.1	66.6	17.4	16.6
S2	0.8	0.7	2.8	0.9	0.8
S3	6.6	6.6	6.8	6.8	6.6
S4	1.4	1.6	2.4	0.6	0.5
S5	2.9	2.8	3.0	0.2	0.2
S6	0.9	0.5	1.4	0.8	0.9
S7	4.9	4.9	2.9	5.2	4.9

Table C.6.: Query execution times in sec for the comparison of the different graph cover strategies using the WatDiv100M data set and 10 slaves.

query	hash	hierarchical	molecule hash
C1	570.6	573.4	553.4
C2	3,465.0	3,288.1	609.6
C3	3,581.5	3,593.9	3.590.7
F1	2,873.8	3,479.5	2.660.9
F2	4,661.7	4,662.4	3,080.3
F3	3,968.0	3,966.3	839.8
F4	4,279.2	4,257.0	4,263.1
F5	192.6	195.5	126.5
L1	230.8	231.9	230.9
L2	532.0	541.1	521.8
L3	11.5	13.3	11.4
L4	2.0	1.9	1.9
L5	1.8	1.9	1.9
S1	490.0	441.7	368.6
S2	8.3	10.6	11.0
S3	7.8	10.5	7.6
S4	436.4	677.4	8.8
S5	7.2	7.0	6.9
S6	8.9	10.9	11.7
S7	13.9	16.8	15.1

Table C.7.: Query execution times in sec for the comparison of the different graph cover strategies using the WatDiv1000M data set and 10 slaves.

List of Figures

List of Tables

List of Listings

Curriculum Vitae

Daniel Dominik Janke

Institute for Web Science and Technologies,
Universität Koblenz-Landau,
Universitätsstr. 1,
56070 Koblenz,
Germany.

+49 (0) 261 287-2747

danijank@uni-koblenz.de

February 5, 2020

Education and Qualification

2012	M.Sc. in Computer Science	Universität Koblenz-Landau, Koblenz.
2010	B.Sc. in Computer Science	Universität Koblenz-Landau, Koblenz.

Positions

2013–	**Research assistant**, Institute for Web Science and Technologies, Universität Koblenz-Landau.
2012-2013	**Research assistant**, Institute for Software Engineering, Universität Koblenz-Landau.

Research

My current research is on distributed graph databases. My focus is on the effects of the data placement strategies on the query performance. Additionally, I also investigate distributed query execution strategies as well as finding new persistent data structures that allow for efficient updates.

Invited Talks

- Summer school lecturer, *14th Reasoning Web Summer School*, Esch-sur-Alzette, September 2018.

Teaching

2019	Big Data Tutorial.
2018	Foundations in Databases Tutorial.
2014	Data Science Tutorial.
2013	Proseminar Linked Data.

Supervised Theses

- Optimizing a Statistics Database for Big Data, bachelor thesis, August 2018.
- Quintuple Extension of RDF and SPARQL, bachelor thesis, December 2016.
- Benchmarks for SPARQL Property Paths, bachelor thesis, July 2016.

Reviewing Activities

2017-2018	Extended Semantic Web Conference.
2017	International Semantic Web Conference.

Publications

1. Janke, D., Staab, S., Thimm, M.: Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores. Tech. rep., WeST (2016), https://owncloud.uni-koblenz-landau.de/owncloud/s/5c25skgCkkDgxyQ

2. Janke, D., Staab, S., Thimm, M.: On Data Placement Strategies in Distributed RDF Stores. In: Proceedings of The International Workshop on Semantic Big Data. pp. 1:1–1:6. SBD '17, ACM, New York, NY, USA (2017). doi: 10.1145/3066911.3066915

3. Janke, D., Skubella, A., Staab, S.: Evaluating sparql 1.1 property path support. In: Usbeck, R., Ngonga, A., Kim, J.D., Choi, K.S., Cimiano, P., Fundulaki, I., Krithara, A. (eds.) Joint Proceedings of BLINK2017: Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data (BLINK2017-NLIWoD3). No. 1932 in CEUR Workshop Proceedings, Aachen (2017), http://ceur-ws.org/Vol-1932/paper-04.pdf

4. Janke, D., Staab, S., Thimm, M.: Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In: Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017. (2017), http://ceur-ws.org/Vol-1932/paper-05.pdf

5. Janke, D., Staab, S., Thimm, M.: Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In: Nikitina, N., Song, D., Fokoue, A., Haase, P. (eds.) ISWC 2017 Posters & Demonstrations and Industry Tracks. No. 2963 in CEUR Workshop Proceedings, Aachen (2017), http://ceur-ws.org/Vol-1963/paper489.pdf

6. Janke, D., Staab, S., Thimm, M.: Impact analysis of data placement strategies on query efforts in distributed RDF stores. Journal of Web Semantics **50**, 21 – 48 (2018). doi: 10.1016/j.websem.2018.02.002, http://www.websemanticsjournal.org/index.php/ps/article/view/516

7. Janke, D., Staab, S.: Storing and Querying Semantic Data in the Cloud. In: D'Amato, C., Theobald, M. (eds.) Reasoning Web. Learning, Uncertainty, Streaming, and Scalability: 14th International Summer School 2018, Esch-sur-Alzette, Luxembourg, September 22–26, 2018, Tutorial Lectures. pp. 173–222. Springer International Publishing, Cham (2018). doi: 10.1007/978-3-030-00338-8_7

8. Skubella, A., Janke, D., Staab, S.: Beseppi: Semantic-based benchmarking of property path implementations. In: The Semantic Web - 15th International Conference (06/06/19) (June 2019), https://eprints.soton.ac.uk/429356/