# Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores[☆]

Daniel Janke[a,*], Steffen Staab[a,b,*], Matthias Thimm[a,*]

[a] *Universität Koblenz-Landau, Institute for Web Science and Technologies, Universitätsstr. 1, 56070 Koblenz, Germany*
[b] *University of Southampton, Web and Internet Science Group, Building 32, Highfield Campus, SO17 1BJ Southampton, United Kingdom*

**Abstract**

In the last years, scalable RDF stores in the cloud have been developed, where graph data is distributed over compute and storage nodes for scaling efforts of query processing and memory needs. One main challenge in these RDF stores is the data placement strategy that can be formalized in terms of graph covers. These graph covers determine whether (a) the triples distribution is well-balanced over all storage nodes (storage balance) (b) different query results may be computed on several compute nodes in parallel (vertical parallelization) and (c) individual query results can be produced only from triples assigned to few — ideally one — storage node (horizontal containment). We analyse the impact of three most commonly used graph cover strategies in these terms and found out that balancing query workload reduces the query execution time more than reducing data transfer over network. To this end, we present our novel benchmark and open source evaluation platform Koral.

*Keywords:* Distributed RDF stores, graph partitioning, benchmark

## 1. Introduction

In the last years, the requirement for RDF stores that can cope with several trillions of triples has emerged. For instance, the number of Schema.org-based facts that are extracted out of the Web have reached the size of three trillions [2]. Another example is the European Bioinformatics Institute (EMBL-EBI) that would like to convert its datasets into RDF resulting in a graph consisting of several trillions of triples. To date no such scalable RDF store exists and the current EBI RDF Platform can handle only 10 billion triples [3].

We pursue the development of a scalable RDF store 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 paper, we focus on comparing the performance of data placement strategies.

Strategies for data placement may be formalized in terms of graph covers. Each compute and storage node hosts a graph chunk. Each triple is assigned to (at least) one graph chunk and the union of all graph chunks define a (possibly redundant) graph cover. When a query is requested to an RDF store in the cloud, the query is distributed over the different compute and storage nodes. Each node applies the query operators assigned to it on its local data. If the query requires the combination of data from different chunks, the required information has to be transferred between compute nodes.

One graph cover strategy commonly used is the hash cover that assigns triples to compute and storage nodes according to the hash value of, e. g., their subject (e. g., used by Virtuoso Clustered Edition [4], YARS2 [5, 6], Clustered TDB [7] and Trinity.RDF [8]). In order to reduce the number of transferred intermediate results, hierarchical hash has been proposed as an extension of the hash cover strategy that computes the hash only on IRI prefixes [9]. Another commonly used graph cover strategy is the minimal edge-cut cover that assigns vertices to similarly-sized partitions in a way that the number of edges connecting vertices assigned to different partitions is minimised (e. g., used by [10–12]). Fur-

[☆]This paper extends the 6 page workshop paper [1].
[*]Corresponding author
*Email addresses:* `danijank@uni-koblenz.de` (Daniel Janke), `staab@uni-koblenz.de` (Steffen Staab), `thimm@uni-koblenz.de` (Matthias Thimm)

thermore, the vertical cover strategy is inspired by relational databases. It partitions the dataset by storing all triples with the same property in one table. Finally, these tables are then distributed among all compute and storage nodes according to the hash on the property. It is used by, e. g. HadoopRDF [13], Jena-HBase [14] and [15]. In order to reduce the number of transferred intermediate results, [11] proposed to replicate triples at the border of the graph chunks. This idea is also used by systems like VB-Partitioner [16] and D-SPARQ [17].

It is a commonly held belief that query completion is optimized by approaches that emphasize local computation such as minimal edge-cut (cf. [11, 12, 16]). *The first major contribution* of this paper is to challenge this assumption by new experiments. Our results indicate that contrary to commonly held beliefs, *query answering with hash covers may outperform query answering with, e.g., minimal edge-cut covers* since the load on the different machines is more balanced. Furthermore, when *replicating triples* on several computers, the high number of duplicate computations may overcome the benefits of the reduced data transfer via network and *lead to a worse query performance*.

We have performed our experiments with the aim to understand interdependencies of the involved query processing. Thus, we have devised new measures and do not only compare graph cover strategies in terms of query processing time, but in addition we investigate the following dimensions:

- *Load time* describes the time it takes to create a graph cover. This is an indicator how well a graph cover strategy can scale by horizontal scaling of the cloud.
- *Storage balance* describes to which extent graph chunks are of similar size. This is an indicator that memory needs can be met with increasing data size by horizontal scaling of the cloud.
- *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.
- *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.

Using these measurements, we derive *the second important contribution of this paper*. We discovered from the analysis of query processing using different graph cover strategies 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). Furthermore, our analysis revealed that previous experiments like [11], [16] and [12] suffered from a setting with highly inefficient methods for data transfer (i.e. based on the Hadoop/HDFS infrastructure) (see [18]).

In order to determine to which extent graph cover strategies lead to efficient query answering, they have to be implemented and evaluated in distributed RDF stores. For instance, [8] and [12] evaluate various RDF stores that use different graph cover strategies, but these evaluations compared the RDF stores as wholes. Thus, their results also reflect the effects of, e. g., the different indexing strategies and persistence strategies (i. e., main memory vs. hard disk) used by the different stores. In order to focus on the effects of the different graph cover strategies, other evaluations have used the same system to measure the execution time ([16] and [19]). These systems use technologies like Hadoop or HDFS that cause an overhead for data transfer. To avoid the bias of this overhead, *the third important contribution of this paper is the flexible open source platform* Koral. It executes queries on arbitrary graph covers and transfers the intermediate results within the network.

In short, the contributions of this paper are:

1. An explanation why previous evaluations concluded that the amount of data transfer caused by a graph cover strategy is crucial for the query execution effort (Section 6).
2. An analysis indicating that (i) hash covers outperform minimal edge-cut covers and vertical covers, (ii) vertical parallelization is more important than horizontal containment and (iii) triple replication reduces query performance due to a high number of duplicate computations (Section 5).
3. A benchmark methodology and its implementation that allows for a detailed understanding of the interdependencies of the graph cover strategy and the query processing (Section 4).

## 2. Formalisation of Graph Cover Strategies

In order to illustrate different graph cover strategies, we use Figure 1 as our 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.

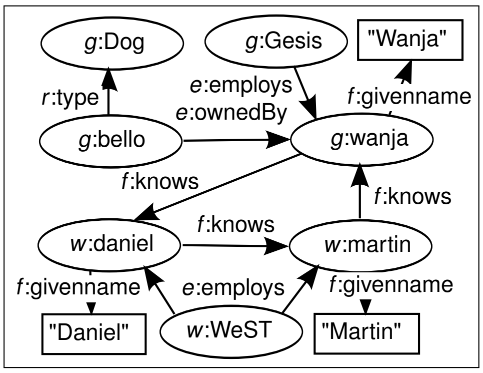


Figure 1: The example graph describing the knows relationships between some employees of WeST and Gesis.

To formalize the problem, we define RDF graphs like in [20]. 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* $\sigma$ *is defined by* $T = (I \cup B) \times I \times IBL$. *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, we use subj($T$), obj($T$) and prop($T$) 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 and storage nodes (in the following, we refer to them more briefly as *compute nodes*). The finite set of compute nodes is denoted as $C$ in the rest of this paper.

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

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

$$\text{chunk}_{\text{cover}}: C \to 2^G$$

$$\text{chunk}_{\text{cover}}(c) := \{t | c \in \text{cover}(t)\} \quad .$$

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

*Example 1*: The graph cover shown in Figure 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 4. 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* path $P$ *is a sequence* $t_0, t_1, ..., t_n$, *if* $\forall i \in [0, n] : t_i \in G \wedge \forall j \in [0, n], j \neq i : t_j \neq t_i$ *and* $\forall i \in [1, n] : t_{i-1} = (s_{i-1}, p_{i-1}, s_i) \wedge t_i = (s_i, p_i, o_i)$. *The* length of path $P$ *is* $n + 1$.

*Example 2*: In the example Graph shown in Figure 1, (w:daniel, f:knows, w:martin), (w:martin, f:knows, g:wanja) is a path of length 2.

Beside the graph cover strategies that are described in the following, there exist other approaches such as the ones used by Partout [21], COSI [22], [23], WARP [24], VB-Partitioner [16] and [25] (see Section 6 for more details). Since these graph cover strategies are only used by a single system, we focussed our evaluation on the graph cover strategies that are used most frequently.

## Hash Cover

A hash cover assigns triples to chunks according to the hash value computed on their subjects modulo the number of compute nodes[1]. Thus, all triples with the same subject are located in the same graph chunk. This graph cover strategy is used, for instance, by Virtuoso Clustered Edition [4], YARS2 [5, 6], ZipG [26], Clustered TDB [7] and Trinity.RDF [8].

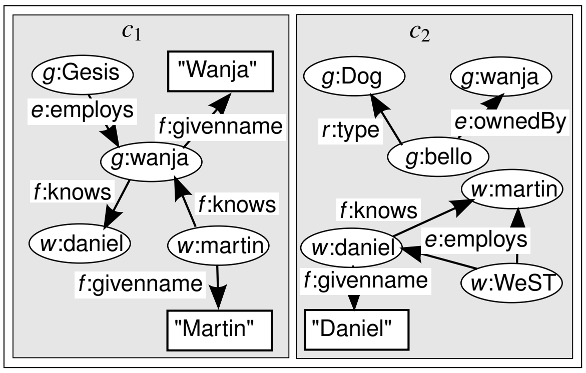


Figure 2: An example hash cover of the example graph.

[1]Beside the subject, the property or the object are used to compute the hash value, for instance, see [16], but the subject-based hash is chosen most frequently.

*Example 3*: The following hash function produces the graph cover shown in Figure 2.

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

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).

**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 [9] 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 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 4*: Assume the hash is computed on the prefixes gesis and west of the subject IRIs in the example graph. If the hash function returns 1 for gesis and 2 for west the resulting hierarchical hash cover is shown in Figure 3.

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.

**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 [27]. 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 [27]. RDF stores like EAGRE [12], [28] and [11] convert the outcome of the minimal edge-cut algorithm, i.e., a partitioning of $V_G$, into a graph cover of $G$ by assigning each triple to the compute node to which its subject has been assigned.

*Example 5*: A minimal edge-cut algorithm might 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.

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 is assigned to $c_1$, this triple is assigned to $c_1$. The subject of the second triple w:martin is assigned to $c_2$. Therefore, this triple is 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 might 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.

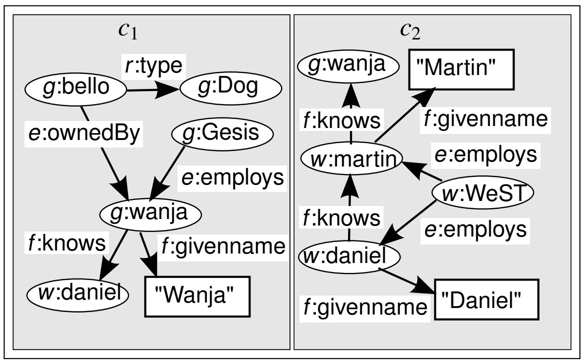


Figure 3: An example hierarchical hash cover which is also a minimal edge-cut cover of the example graph.

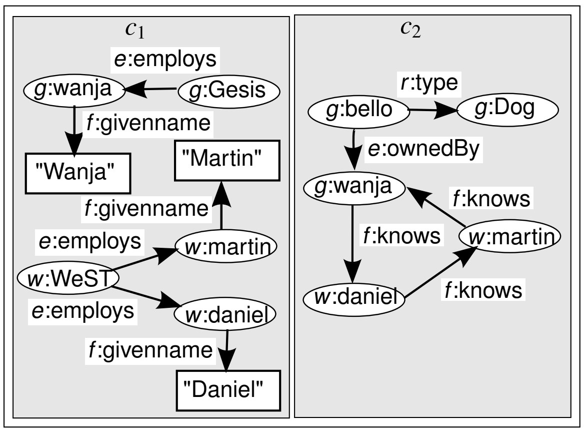


Figure 4: An example vertical cover of the example graph.

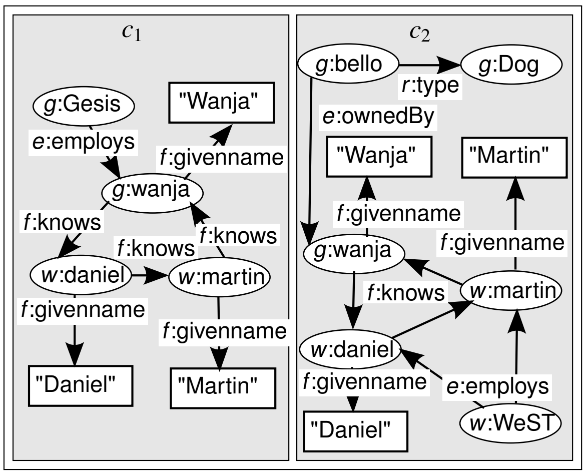


Figure 5: The 2-hop extension of the hash cover in Figure 2.

### Vertical Cover

The basic idea of the vertical cover originated in [29] to store RDF data in a relational database so 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 HadoopRDF [13], Jena-HBase [14] and [15] distribute the individual tables among the compute nodes by the hash of their property modulo the number of compute nodes.

*Example 6*: The following hash function produces the graph cover shown in Figure 4.

$$\forall r \in \{\text{f:givenname}, \text{x:employs}\}: \text{hash}(r) := 1$$

$$\forall r \in \{\text{f:knows}, \text{r:type}, \text{e:ownedBy}\}: \text{hash}(r) := 2 \ .$$

The vertical cover assigns triples to chunks based on the hash of their properties. 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.

### *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 [11], VB-Partitioner [16] and D-SPARQ [17].

*Example 7*: Applying the 2-hop replication extension on the hash cover in Figure 2 results in the 2-hop hash cover shown in Figure 5. In this cover a query could match with the path (g:bello, e:ownedBy, g:wanja) , (g:wanja, f:knows, w:daniel) 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 increase 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 intermediate results is slow and the number of replicated triples is low.

## 3. Formalisation of Query Execution Strategy

For the impact analysis done in this paper we have extended a state-of-the-art asynchronous execution mechanisms such as realised in TriAD [10]. The extensions render the query execution mechanism independent from the underlying graph cover. In order to formalise our query execution mechanism in Section 3.2 we first introduce the required formal definitions of a small subset of the standard query language SPARQL in the next section. This section contains only common definitions to provide the notions for Section 3.2. The formal proof of the completeness and correctness of our execution strategy can be found in [30].

### *3.1. SPARQL*

We define the used SPARQL core as done in [31], [32] and [33]. 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 6.** *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 7.** *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 8.** *A* SELECT query *is defined as* `SELECT` $W$ `WHERE` $\{B\}$ *with* $W \subseteq V$ *and* $B$ *a BGP.*

*Example 8*: 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 $Q$ represents the set of all SPARQL queries.

**Definition 9.** *The function* var : $Q \rightarrow V$ *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 10.** *A* variable binding *is a partial function* $\mu : V \nrightarrow IBL$. *The* set of all variable bindings *is* $O$.

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

*Example 9*: The following three partial functions are variable bindings, that assign values to some variables. $\mu_1$ would be an intermediate result produced by the first triple pattern of the example query in example 8 whereas $\mu_2$ and $\mu_3$ 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 11.** *Two variable bindings* $\mu_i$ *and* $\mu_j$ *are* compatible, *denoted by* $\mu_i \sim \mu_j$, *if* $\forall ?x \in \text{dom}(\mu_i) \cap \text{dom}(\mu_j) : \mu_i(?x) = \mu_j(?x)$.[2]

*Example 10*: The variable bindings $\mu_1$ and $\mu_2$ from example 9 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. $\mu_1$ and $\mu_3$ as well as $\mu_2$ and $\mu_3$ are not compatible because they assign different values to common variables.

**Definition 12.** *The* join *of two sets of variable bindings* $\Omega_1$ *and* $\Omega_2$ *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 11*: The join of the two variable bindings sets $\{\mu_1\}$ and $\{\mu_2, \mu_3\}$ from example 9 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 $\mu_1$ and $\mu_2$ are compatible.

[32] and [33] define the semantics of a SPARQL query as follows:

**Definition 13.** *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* $[\![\texttt{SELECT } W \texttt{ WHERE } \{B\}]\!]_G = \text{project}(W, [\![B]\!]_G) = \{\mu_{|W} | \mu \in [\![B]\!]_G\}$.[3]

### *3.2. Query Execution Strategy*

In order to compare arbitrary graph cover strategies, a distributed query execution strategy is required that can also deal with replicated triples. For its formal definition we first define query execution trees. In order to increase the comprehensibility, we first define our graph-cover-independent distributed query execution strategy

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

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

without taking care of replicated triples. Thereafter, we extend this query execution strategy to benefit of triple replication. Finally, we discuss the limitations of our query execution strategy.

### Query Execution Trees

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 14.** *Let $L_{node}$ be the set of node labels and $\Upsilon = L_{node} \times 2^V \times 2^\Upsilon$ the set of all query execution trees, 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 $\langle\!\langle B_1.B_2 \rangle\!\rangle = (join, var(B_1) \cup var(B_2), \{\langle\!\langle B_1 \rangle\!\rangle, \langle\!\langle B_2 \rangle\!\rangle\})$.*
3. *If $W \subseteq V$ and $B$ is a BGP, then $\langle\!\langle \texttt{SELECT } W \texttt{ WHERE } \{B\} \rangle\!\rangle = (project, W, \langle\!\langle B \rangle\!\rangle)$.*

**Definition 15.** *The* variables common for all child trees *of a query execution tree are defined as follows.*

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

*Example 12*: Figure 10b shows a graphical representation of one query execution tree for the example query from example 8. The following query execution tree represents the first join in its mathematical representation. It has the two child trees (tp, $\{?v_1, ?v_2\}$, $\varnothing$) and (tp, $\{?v_2, ?v_3\}$, $\varnothing$). Their common variables are $\{?v_2\}$.

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

### Distributed Query Execution Strategy Ignoring Triple Replication

Our 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 16.** *The* join responsibility of a resource *is a function* jResp : *$IBL \to C$ 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 13*: Assume the example hash cover from Figure 2. The join responsibilities for the different resources are:

$\forall r \in$ {g:Gesis, g:wanja, w:martin, "Wanja", "Martin", f:knows, f:givenname}: jResp($r$) := $c_1$
$\forall r \in$ {g:bello, w:WeST, w:daniel, r:type, e:employs, g:Dog, e:ownedBy, "Daniel"}: 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, we assume an arbitrary but fixed strict total order $<_V$ is defined on $V$ (e.g., a lexicographic order on the variable names). With its help we can extract the least variable out of the set of all join variables.

In order to simplify the definition of the distributed query execution strategy, we define a function 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, we assume 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 17.** *Let $<_S$ be a strict total order defined on a set $S$, then the* minimum *is defined as follows.*

$$\begin{aligned}\min_{<_S} &: 2^S \to S\\ \min_{<_S}(\hat{S}) &:= s, \text{ iff } \hat{S} \neq \emptyset \land s \in \hat{S}\\ &\land \forall s' \in \hat{S} : s' \neq s \Rightarrow s < s' \ .\end{aligned}$$

**Definition 18.** *The* route function *extracts all variable bindings from a set of variable bindings $\hat{\Omega}$ that will be processed on a compute node $c$. For an arbitrary but fixed join responsibility function* jResp*, it is defined as follows.*

$$\begin{aligned}\text{route} &: C \times \Upsilon \times O \to O\\ \text{route}(c, qet, \hat{\Omega}) &:= \Big\{\mu \in \hat{\Omega} \Big| \mu = \emptyset\\ &\lor (\text{cVars}(qet) = \emptyset \land c = \min_{<_C}(C))\\ &\lor (\text{cVars}(qet) \neq \emptyset\\ &\land \text{jResp}(\mu(\min_{<_V}(\text{cVars}(qet)))) = c)\Big\} \ .\end{aligned}$$

*Example 14*: If we execute the triple pattern `?v1 <f:knows> ?v2` from the query in example 8 on compute node $c_1$ of the example hash cover in Figure 2, the variable binding $\mu_1 = \{(\texttt{?v1}, \text{g:wanja}), (\texttt{?v2}, \text{w:daniel})\}$ is produced as shown in Figure 6. The succeeding join operation will join on variable `?v2`. $\mu_1$ maps `?v2` on the IRI w:daniel. The join responsibility for this IRI was assigned to jResp(w:daniel) $= c_2$. Therefore, $\text{route}(c_2, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_1\}) = \{\mu_1\}$ will identify $\mu_1$ to be transferred to compute node $c_2$.

The triple pattern `?v2 <f:givenname> ?v3` will create the variable binding $\mu_2 = \{(\texttt{?v2}, \text{w:daniel}), (\texttt{?v3}, \text{"Daniel"})\}$ on compute node $c_2$. The succeeding join operation will join on variable `?v2`. $\mu_2$ maps this variable on the same IRI as $\mu_1$. Thus, $\text{route}(c_2, \langle\!\langle tp_1.tp_2 \rangle\!\rangle, \{\mu_2\}) = \{\mu_2\}$ will identify $\mu_2$ to be joined on the same compute node $c_2$ on which it was produced. Since now, both compatible mapping $\mu_1$ and $\mu_2$ are assigned to the same join operation on the same compute node, they can be joined.

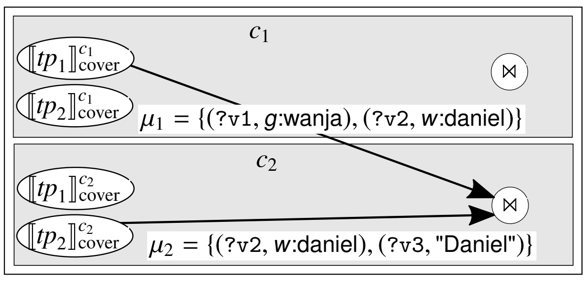

$c_1$
$[\![tp_1]\!]^{c_1}_{\text{cover}}$
$[\![tp_2]\!]^{c_1}_{\text{cover}}$
$\bowtie$
$\mu_1 = \{(\texttt{?v1}, g\text{:wanja}), (\texttt{?v2}, w\text{:daniel})\}$
$c_2$
$[\![tp_1]\!]^{c_2}_{\text{cover}}$
$[\![tp_2]\!]^{c_2}_{\text{cover}}$
$\bowtie$
$\mu_2 = \{(\texttt{?v2}, w\text{:daniel}), (\texttt{?v3}, \text{"Daniel"})\}$

Figure 6: An illustration of the distributed join.

**Definition 19.** *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 $[\![tp]\!]^c_{cover} = \{\mu | \operatorname{dom}(\mu) = \operatorname{var}(tp) \land \mu(tp) \in \text{chunk}_{\text{cover}}(c)\}$.*
2. *If $B_1$ and $B_2$ are BGPs, then $[\![B_1.B_2]\!]^c_{\text{cover}} = \left(\bigcup_{c' \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_1]\!]^{c'}_{\text{cover}})\right) \bowtie \left(\bigcup_{c' \in C} \text{route}(c, \langle\!\langle B_1.B_2 \rangle\!\rangle, [\![B_2]\!]^{c'}_{\text{cover}})\right)$.*
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 20.** *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 [30].

## 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 [9] 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.

In order to benefit of triple replication in the same way as systems like [9] do, we extend our distributed

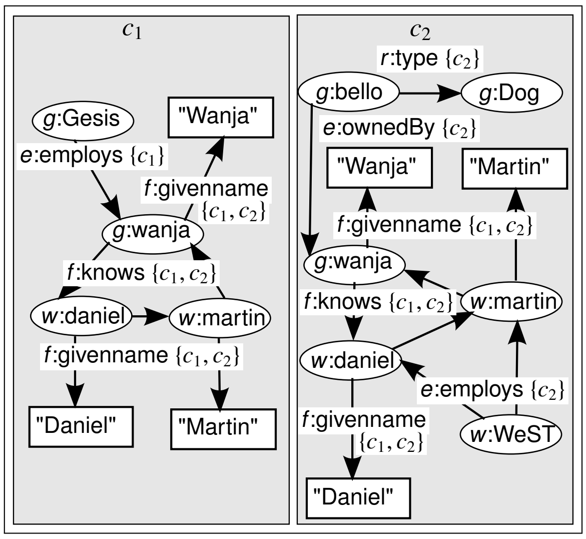


Figure 7: The 2-hop replication extension of the hash cover in Figure 5 with annotated compute nodes.

query execution strategy. 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, we annotate each triple with the compute nodes to which it was assigned during the creation of the graph cover.

*Example 15*: Figure 7 shows the 2-hop replication extension of a hash cover from Figure 5. The information to which compute nodes each triple was assigned to is annotated behind the edge labels. In the following we will use this cover 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:

$\forall r \in$ {g:Gesis, g:wanja, "Wanja", "Martin", "Daniel", f:knows, f:givenname}: $\text{jResp}(r) := c_1$
$\forall r \in$ {g:bello, w:WeST, w:daniel, w:martin, r:type, e:employs, g:Dog, e:ownedBy}: $\text{jResp}(r) := c_2$ .

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 $\mu^l$ – know on which compute nodes they could be created.

**Definition 21.** *A* localized variable binding *is a tuple $(\mu, C')$ with the set of compute nodes $C' \subseteq C$ on which the variable binding $\mu$ is known. The set of localized variable bindings denoted as $\mathcal{O}'$.*

*Example 16*: When finding matches for the triple pattern $tp_1$ = `?v1 <f:kknows> ?v2` from the example query in example 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 $\mu_1^l$ was created on both compute nodes.

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

In order to join two localized variable binding, we extend definition 12. A variable binding that is created by joining two localized variable bindings $\mu_1^l$ and $\mu_2^l$ will be created and known on all compute nodes that know $\mu_1^l$ and $\mu_2^l$.

**Definition 22.** *The* join *of two sets of localized variable bindings $\Omega_1'$ and $\Omega_2'$ is defined as*

$$\begin{aligned}\Omega_1' \bowtie \Omega_2' = \quad & \{(\mu_1 \cup \mu_2, C_1' \cap C_2') \,|\, (\mu_1, C_1') \in \Omega_1' \\ & \wedge (\mu_2, C_2') \in \Omega_2' \wedge \mu_1 \sim \mu_2\} \enspace .\end{aligned}$$

*Example 17*: The join of $\mu_1^l$ and $\mu_2^l$ 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 $\mu_2^l$ 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′ function in definition 23. 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 node. 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.

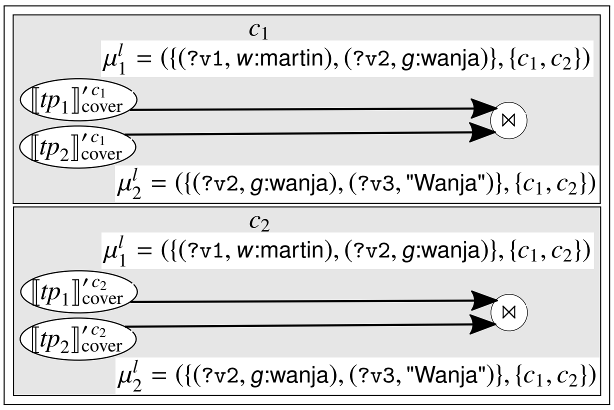
In case of a join operation with at least one join variable, it is checked first, whether the compute node responsible for performing the join already knows the variable binding or not. If the compute node does not know it, it is transferred by the first compute node knowing it. As a consequence only 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 23.** *The* replication-aware route function *extracts all variable bindings from a set of localized variable bindings $\hat{\Omega}'$ produced on compute node $c_s$ that will be processed on a compute node $c_t$. For an arbitrary but fixed join responsibility function* jResp*, it is defined as follows.*

$$\text{route}' : C \times C \times \Upsilon \times O' \rightarrow O'$$

$$\begin{aligned}\text{route}'(c_s, c_t, qet, \hat{\Omega}') := \Big\{(\mu, C'_r) \Big| \exists\, (\mu, C') \in \hat{\Omega}' : \\ &(\mu = \varnothing \land c_s = \min_{<_C}(C') \land C'_r = C) \\ &\lor (\text{cVars}(qet) = \varnothing \land c_t = \min_{<_C}(C) \\ &\quad \land c_t \notin C' \land c_s = \min_{<_C}(C') \\ &\quad \land C'_r = \{c_t\}) \\ &\lor (\text{cVars}(qet) = \varnothing \land c_t = \min_{<_C}(C) \\ &\quad \land c_t \in C' \land c_s = c_t \land C'_r = \{c_t\}) \\ &\lor (\text{cVars}(qet) \neq \varnothing \land c_s = \min_{<_C}(C') \\ &\quad \land \text{jResp}(\mu(\min_{<_V}(\text{cVars}(qet)))) = c_t \\ &\quad \land c_t \notin C' \land C'_r = \{c_t\}) \\ &\lor (\text{cVars}(qet) \neq \varnothing \land c_s = c_t \\ &\quad \land \text{jResp}(\mu(\min_{<_V}(\text{cVars}(qet)))) = c_u \\ &\quad \land c_u \in C' \land C'_r = C')\Big\} \quad .\end{aligned}$$

*Example 18*: When executing the the example query from example 8 on the 2-hop hash cover from Figure 7, the first triple pattern will create the localized variable binding $\mu^l_1$ on both compute nodes as shown in Figure 8. The following join on ?v2. Based on the join responsibility of $\mu^l_1$(?v2) = g:wanja compute node $c_1$ would be responsible for joining it. Since $c_1$ knows the variable binding already, route′ determines both $\mu^l_1$ to be transferred to the local join operations, i.e., $\text{route}'(c_1, c_1, \langle\!\langle tp_1.tp_2\rangle\!\rangle, \{\mu^l_1\}) = \text{route}'(c_2, c_2, \langle\!\langle tp_1.tp_2\rangle\!\rangle, \{\mu^l_1\}) = \{\mu^l_1\}$. The second triple



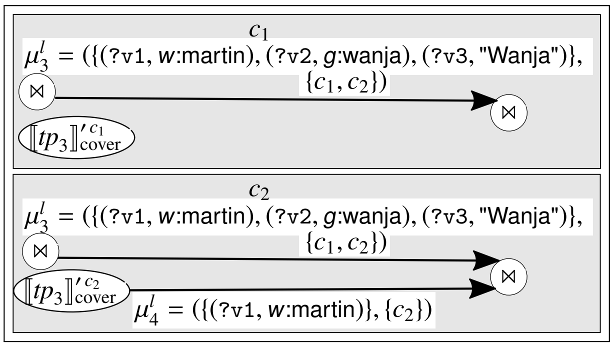
$c_1$
$\mu^l_1$ = ({(?v1, w:martin), (?v2, g:wanja)}, {$c_1$, $c_2$})
$[\![tp_1]\!]'^{c_1}_{\text{cover}}$
$[\![tp_2]\!]'^{c_1}_{\text{cover}}$
⋈
$\mu^l_2$ = ({(?v2, g:wanja), (?v3, "Wanja")}, {$c_1$, $c_2$})
$c_2$
$\mu^l_1$ = ({(?v1, w:martin), (?v2, g:wanja)}, {$c_1$, $c_2$})
$[\![tp_1]\!]'^{c_2}_{\text{cover}}$
$[\![tp_2]\!]'^{c_2}_{\text{cover}}$
⋈
$\mu^l_2$ = ({(?v2, g:wanja), (?v3, "Wanja")}, {$c_1$, $c_2$})

Figure 8: Forwarding of duplicate localized variable bindings at the presence of replicated triples.

pattern produces the localized variable binding $\mu^l_2$ 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 $\mu^l_3$ that is also known by both compute nodes. As shown in Figure 9 this variable binding is forwarded to the succeeding local join operations. The third triple pattern will produce the localized variable binding $\mu^l_4$ only on compute node $c_2$. Since the join responsibility of w:martin is $c_2$ and $\mu^l_4$ is already known on $c_2$ it is forwarded to the succeeding join operation on $c_2$.

When joining $\mu^l_3$ and $\mu^l_4$ on $c_2$ the resulting localized variable binding $\mu^l_5$ = ({(?v1, w:martin), (?v2, g:wanja), (?v3, "Wanja")}, {$c_2$}) is only known on compute node $c_2$, since $\mu^l_4$ is only known on $c_2$. The succeeding join operation would have the join variable ?v2. $\mu^l_5$ assigns g:wanja to this variable. Since compute node $c_1$ is responsible for joining it and $c_1$ does not know the localized



$c_1$
$\mu^l_3$ = ({(?v1, w:martin), (?v2, g:wanja), (?v3, "Wanja")}, {$c_1$, $c_2$})
⋈
⋈
$[\![tp_3]\!]'^{c_1}_{\text{cover}}$
$c_2$
$\mu^l_3$ = ({(?v1, w:martin), (?v2, g:wanja), (?v3, "Wanja")}, {$c_1$, $c_2$})
⋈
⋈
$[\![tp_3]\!]'^{c_2}_{\text{cover}}$
$\mu^l_4$ = ({(?v1, w:martin)}, {$c_2$})

Figure 9: Forwarding of unique localized variable bindings at the presence of replicated triples.

variable binding yet, it is transferred to $c_1$, i.e. $\text{route}'(c_2, c_1, \langle\!\langle\langle\!\langle\langle\!\langle tp_1.tp_2\rangle\!\rangle.tp_3\rangle\!\rangle.tp_4\rangle\!\rangle, \{\mu_5^l\}) = \{\mu_5^l\}$.

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

**Definition 24.** *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* $[\![Q]\!]'^{c}_{\text{cover}}$*, is defined recursively as follows:*

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

**Definition 25.** *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* $[\![Q]\!]'_{\text{cover}}$*, is defined as* $[\![Q]\!]'_{\text{cover}} := \left\{\mu \middle| (\mu, C') \in \bigcup_{c \in C} [\![Q]\!]'^{c}_{\text{cover}}\right\}$.

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.*

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

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:*

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

The proof of Theorems 2 and 3 are shown in [30].

**Limitations**

A single graph cover strategies 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 $\text{hash}(s) mod |C|$ 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 [9] and [11] 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.

We developed a simple distributed query execution strategy that similar to [9] and [11] executes the complete query execution tree on all compute nodes. But the decomposition of queries into subqueries that match with paths that have a length of at most $n$ was not done, since it would be a property that is specific for the $n$-hop replication. To become more graph cover-independent we keep track on which compute nodes the individual triples and intermediate results are known. With this information, we can avoid transferring intermediate results to compute nodes that know these results already. Similar to [9], our 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 our approach and the systems [9] and [11] is how the join of intermediate results from different compute nodes are handled. [9] and [11] use MapReduce to join them with the consequence that these joins via MapReduce punish network traffic with an overhead. To avoid this punishment we adapted the approach of TriAD [10] 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.

## 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 overall performance for the processing of SPARQL queries [31], we want to observe indications that contribute to understanding how graph cover strategies may relate to scalability. Section 1 has already explained several high-level indicators, which 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 factor are 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 we need to specify execution strategies (cf. Section 4.3) as well as execution operation (cf. Section 4.4). For these two factors, our 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.

### 4.1. Evaluation Measures

In this subsection, we define the measures we have found most useful to characterize different graph cover strategies. We have experimented with further measurement and statistics functions, e. g. standard deviation instead of Gini coefficient, but found them to be correlated and then decided in favour of the ones we found most intuitive to interpret.

*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, we define the load time $L$ as the time required for the computation of the graph cover. We perform this measurement as a weak indicator for setting up a data set in a cloud RDF store. However, we are aware that all other steps by themselves are complex enough to warrant deeper investigation.

*Storage imbalance*

Scaling the cloud for handling growing memory needs may be jeopardized by graph cover strategies aiming at horizontal containment. They might generate a skewed distribution delegating expensive tasks on few compute nodes. Therefore, we evaluate the quality of the storage distribution resulting from a graph cover strategy with the storage imbalance $b$.

**Definition 26.** *For a given* cover, storage imbalance $b$ *is defined by the Gini coefficient*

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

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

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, we also experimented with the standard deviation and entropy but we decided to use the Gini coefficient, since the produced values are within a fixed range between 0 and 1 independent of the actual chunk sizes and thus better comparable. Furthermore, our 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, we define the storage redundancy 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 27.** *For a given* cover *of a graph G, the* storage redundancy *is defined as*

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

*whereby* $\mathrm{vol}(c) := |\mathrm{chunk}_{\mathrm{cover}}(c)|$ *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, we provide different kinds of performance characteristics. 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 $q$. We drop the superscript $^q$ when it is clear from context as in the following definitions.

**Definition 28.** *Overall query performance is evaluated by the following functions, with K being the overall number of query results:*

$$\begin{aligned} \textit{Query time to completion:} \quad & exTime := t_K - t_0 \\ \textit{Result curve function:} \quad & \chi(t) := \frac{|\{t_i | t_i - t_0 \leq t\}|}{K} \end{aligned}$$

$\chi(t)$ 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. Workload and workload imbalance are means to capture the computing efforts at an abstract level of operation. In a distributed system, the second — and 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, we measure an abstract level of data transfer:

**Definition 29.** *For a given cover and a given query execution tree q, we define overall data transfer* $T := \sum_{c \in C} T_c$. *Data transfer is measured at each compute node c as* $T_c := \sum_{op} m^{op} \cdot \left|\mathrm{dom}(\mu_1^{op})\right|$. *Each join operation op that leads to the sending of variable bindings to another compute node* $c' \neq c$ *contributes with data size* $m^{op} \cdot \left|\mathrm{dom}(\mu_1^{op})\right|$ *where* $\left|\mathrm{dom}(\mu_1^{op})\right|$ *are the number of variables of a variable binding and m are the number of variable bindings* $m = \left|\{\mu_i^{op}\}\right|$.

Additionally to the data transfer which measures the amount of transferred data, we also measure the amount of transferred packets:

**Definition 30.** *For a given cover and a given query execution tree q, we define the number of transferred packets* $P := \sum_{c \in C} P_c$, *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, we prefer to measure the data transfer and the workload imbalance.

*Vertical parallelization*

In order to measure workload independently of time, we observe the number of join comparisons to be performed. Given a query execution tree the overall workload will be identical for all graph cover strategies. With vertical parallelization we are interested in 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’. We pursue a simple, but effective strategy here, by simply measuring how the workload is distributed over different compute nodes using the Gini coefficient.

**Definition 31.** *For a cover and a query execution tree q,* workload imbalance *W is the Gini coefficient:*

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

*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 we arrive at the following interpretation table that lets us derive at a comprehensive picture when jointly considering workload imbalance and measures for horizontal containment (see Table 1). Based on our evaluation we would suggest that the workload imbalance may be seen as low for a $W < 0.1$ and the horizontal containment may be seen as low if less than 0.01 packets would be transferred to produce a single query result.

| | Horizontal containment low | Horizontal containment high |
|---|---|---|
| W low | high vertical parallelization | low to medium vertical parallelization |
| W high | low vertical parallelization | low vertical parallelization (unlikely situation) |

Table 1: Measurement of vertical parallelization.

### *4.2. Strategy for Generating Queries*

Since the core functionality of SPARQL is provided by matching basic graph patterns, we follow the strategy of most other benchmarks, performing evaluations with varied basic graph pattern structures. In particular, we adopt the strategy of SPLODGE [34], which varies the query characteristics given arbitrary real-world datasets:

**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).

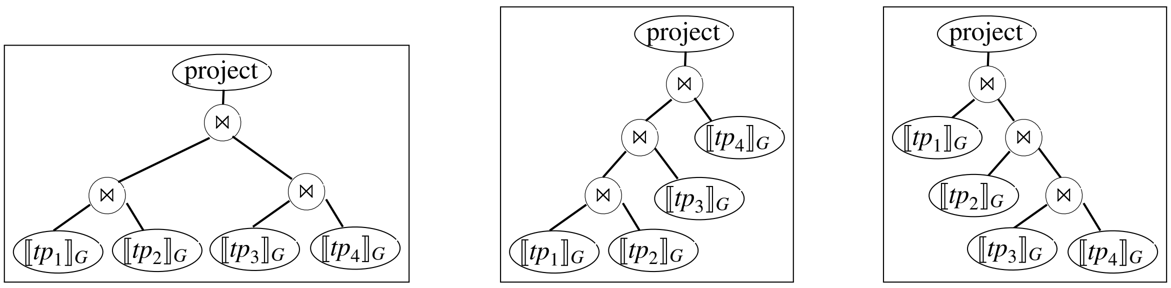
While the first three are common to most benchmarks, the last one has been specifically added to SPLODGE for benchmarking federated stores. Varying this parameter between 1 and several units is important in this context, as several graph cover strategies may easily collocate data from a single data source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

### *4.3. Query Execution Strategies*

In order to find out about weaknesses and strengths of graph cover strategies, we need to determine 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. We remedy this issue in a similar way as we do for dataset and queries: We systematically explore the suitability of the different graph cover strategies under variations of query executions. Thus, we do not measure the performance of "the best run", which would be hard to achieve anyway, but we characterize the robustness and susceptibility of graph cover strategies vs. execution strategies.

Specifically, we use (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. Thus, we have trees of different heights and topological sorting. To evaluate the performance of graph cover strategies under variations of query execution trees, we have devised an operative environment that can handle different graph cover strategies and such variations of query execution trees. This environment is described next.

*Example 19*: Figure 10 shows the three different query execution trees generated for the query from example 8. The bushy query execution tree joins all consecutive triple patterns pairwise as shown in Figure 10a. The resulting intermediate results are joined pairwise again until all joins are performed. The left-linear query execution tree joins the triple patterns in the sequence they were defined in the query (see Figure 10b) whereas the right-linear query execution tree joins them in the reverse sequence as shown in Figure 10c.



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

Figure 10: The three different query execution strategies for the query from example 8.

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

The distributed RDF store for arbitrary graph covers (Koral)[4] [35] 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 3 and the proofs of soundness and completeness are given in [30].

Koral is an extension of state-of-the-art asynchronous execution mechanisms such as realised in TriAD [10]. The extensions render the query execution mechanism independent from the underlying graph cover. The architecture of Koral is depicted in Figure 11. 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] `https://github.com/Institute-Web-Science-and-Technologies/koral`

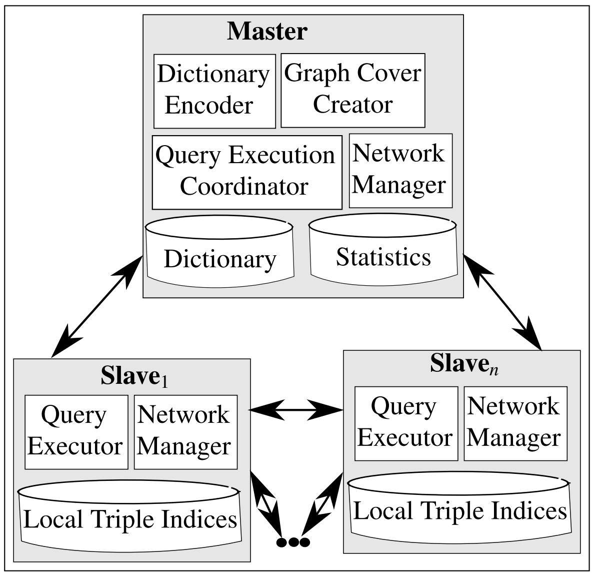


Figure 11: Architecture of Koral.

### 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.

As described in Section 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 our 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 [36]) 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.

**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 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 slave to abort the query execution.

## 5. Evaluation

The experimental setup we have used for the impact analysis of different graph cover strategies on the query execution effort is explained in Section 5.1. Our results are described in Section 5.2.[5]

### *5.1. Experimental Setup*

The set of configurations in our benchmark results from the multiplicative combination of (i) the set of different graph cover strategies, (ii) the set of different query-dataset 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 [9]. For both hash covers, the hash is computed only on the subject of each triple. For the creation of the minimal edge-cut cover we use METIS [27] as done by other distributed RDF stores like [28], [11], D-SPARQ [17] and WARP [24].

Additionally, we examined the effect of the $n$-hop replication. [11] and [16] identified a value of $n = 2$

[5]A script to generate the used datasets, 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`.

to be a good balance between storage redundancy and gained query performance. Since it is expectational that the random distribution of a hash cover will lead to a high number of exchanged intermediate results, we evaluated a 2-hop hash cover to have the greatest benefits from the reduced network traffic.

### Dataset and Queries

In order to avoid effects that occur due to the generation process of a synthetic dataset, we use 500M, 1000M and 2000M triples subsets of the real-world billion triple challenge dataset from 2014 (BTC2014) [37]. The dataset has been generated by crawling data from several data sources of the linked open data cloud. The used subsets contain the first 500 million, one billion and two billion syntactically correct triples. Some characteristics of these subsets are shown in Appendix A.

In comparison to evaluations as described in [8] that store 1 billion triples per compute node, the dataset we use are relatively small. Due to our limited computational resources, we could not perform experiments with 40 compute nodes that are capable to process 1 billion triples each. In order to compensate this, we used smaller datasets but also much smaller compute nodes. Whereas [8] used compute nodes with 6 CPU cores and 96 GB RAM, our compute nodes had only 1 CPU core and 2 GB RAM as described below.

Following the strategy explained in Section 4.2, we generate basic graph patterns with SPLODGE varying the query characteristics. In order to measure the effect of an increasing number of joins, we generate queries with one join involving two triple patterns and 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 be dependent on the join patterns, we generate queries for both join pattern types.

Some graph cover strategies might locate triples from the same dataset 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, we generate queries that require triples only from one data source and queries that combine triples from three different data sources.

In order to generate queries that produce different amounts of intermediate results, we generate queries with a selectivity between 0.001% and 0.01% for the 1000M triples subset. Thus, the match operations alone guarantee that there will be between 1 million and 10 million intermediate results. We experimented with a selectivity rate of 0.1% but, since we do not apply any query optimization strategies, the join operations of these queries produced such an amount of intermediate results that it exceeded the available resources. The generated queries fulfilling these characteristics are given in Appendix C. 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.

### Evaluation Setup using the Graph Cover Evaluation Platform (CEP)

Using our extensible evaluation platform for graph cover strategies (CEP)[6] [38], we set up the evaluation as follows. CEP downloads the BTC2014 dataset, removes all syntactically incorrect triples and creates the 500M, 1000M and 2000M triples dataset. The resulting 1000M dataset is used by SPLODGE [34] configured as described above to generate the query set for the benchmark. For each graph cover strategy Koral is initialized, the dataset 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, we use the 1000M triples dataset and 11, 21, and 41 virtual machines (VMs) to evaluate graph covers with 10, 20 and 40 graph chunks, respectively. Thereafter, we use 21 virtual machines to evaluate the graph cover strategies with the 500M, 1000M and 2000M triples datasets.

### Computer and Software Environment

The graph cover evaluation platform CEP is executed on

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

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 dataset, 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.

**Summary of Evaluation Setup**

Table 2 summarizes the setup of the performed evaluations. We compared the hash cover with the 2-hop hash cover, the hierarchical hash cover, the minimal edge-cut cover and the vertical cover with the 1 billion triples dataset distributed among 10 slaves. Additionally, we executed the master and one slave 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, we used the 1 billion triples dataset that were distributed among 10, 20 and 40 slaves. Finally, he effect of scaling the dataset 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.

| Dataset | Number of slaves | | |
|---|---|---|---|
| size | 10 | 20 | 40 |
| 500M | - | hash<br>hierarchical<br>edge-cut | - |
| 1000M | hash<br>hierarchical<br>edge-cut<br>vertical<br>2-hop hash | hash<br>hierarchical<br>edge-cut | hash<br>hierarchical<br>edge-cut |
| 2000M | - | hash<br>hierarchical<br>edge-cut | - |

Table 2: Summary of the evaluation setup.

## 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 we first compare 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 in Section 5.2.1. For graph cover strategies that can process the queries faster than in the centralized setting, we will first present our analysis of measurements that do not depend on queries in Section 5.2.2. These measurements comprises 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 we analyse indicators for horizontal containment and vertical parallelization based on few selected independent variables. Additionally, we investigate how the observed results change, when scaling the number of virtual machines with a fixed dataset size as well as scaling the dataset 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 we name the queries 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 3 shows the number of results returned by the queries for the different dataset sizes. For all queries the number of results increased while scaling up the dataset 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.

### *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.

| Query | #Results for 500M triples | #Results for 1000M triples | #Results for 2000M triples |
|---|---|---|---|
| $q_1$: `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 3: Number of query results.

- The 2-hop hash cover has a slower query execution time since the high amount of replicated triples cause many intermediate results being computed several times.

To check whether the examined graph cover strategies could reduce the query execution time, we executed one Koral master and one slave on the master VM with 64 GB main memory, loaded the 1 billion triples dataset and executed the queries on it. Then, we compare the measured execution time 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 12. Since the execution times vary strongly, we show the change in comparison to the hash cover.

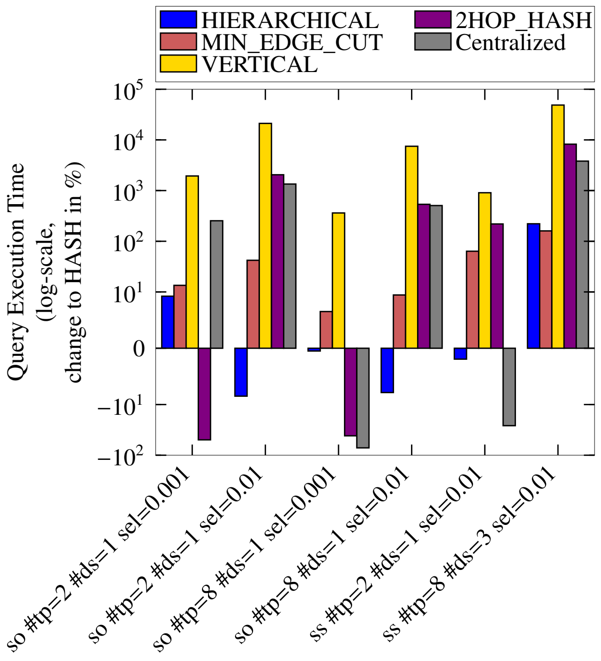


Figure 12: Change of the *exTimes* of all finished queries relative to the hash cover using bushy query execution with 10 slaves.

*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 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 need 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 13 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

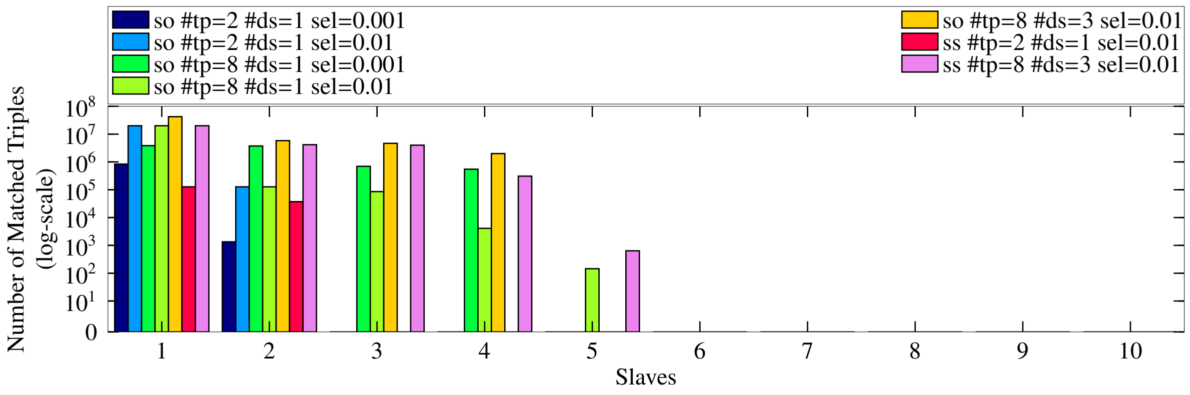


Figure 13: Number of triples that were matched on the individual graph chunks of the vertical cover.

than the chunk with the most matches for almost all queries. Since the vertical cover has matches only on a few slaves its workload 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 we will focus on the hash cover, the hierarchical hash cover and the minimal edge-cut cover in the following.

### 5.2.2. *Query Independent Measurements*

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

- The minimal edge-cut cover takes the longest time to be created and produces the most imbalanced graph chunks.
- Each graph cover strategy cuts almost every second edge but 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 datasets. The loading $L$ time 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 we will focus here.

As shown in Figure 14, the hash cover is created the fasted. It takes around one hour to iterate the dataset 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 dataset twice and the additional computation to find the optimal IRI hierarchy level for

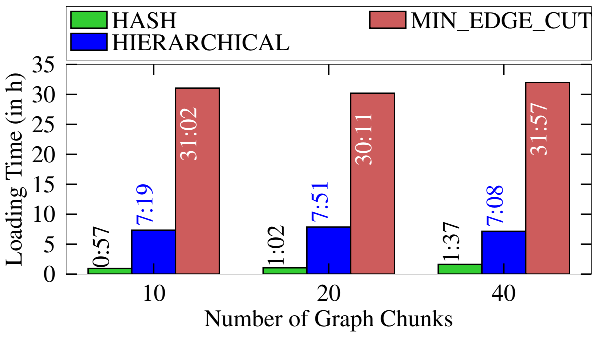


Figure 14: Creation time of the different graph covers for different number of slaves.

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 might be caused by a much higher number of switches between the 40 chunk files when assigning the triples to the chunks. When we scaled the size of the dataset, the creation time increased to the same extent as the dataset size grew for both hash-based covers. Only the creation time of the minimal edge-cut cover became three times higher when doubling the dataset size.

In our current implementation we mainly focussed on the graph cover creation step. We did not optimize the loading steps that are the same for all graph cover strategies 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 our 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 might have a higher query workload than the compute nodes storing the smaller chunks. In order to visualize the storage imbalance, Figure 15 shows, how many triples are contained in the different graph chunks sorted in descending order for 20 graph chunks[7]. As shown in Table 4 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.

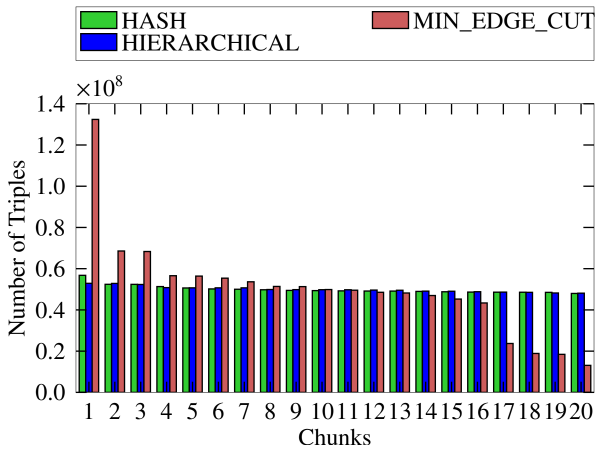


Figure 15: Number of triples contained in each of 20 graph chunks.

| # chunks | 10 | 20 | 40 |
|---|---|---|---|
| Hash cover | 0.0167 | 0.0196 | 0.0275 |
| Hierarchical hash cover | 0.0119 | 0.0160 | 0.0240 |
| Minimal edge-cut cover | 0.1787 | 0.2418 | 0.2441 |

Table 4: The storage imbalance $b$ of the different graph covers at different number of graph chunks for the 1 billion triple dataset.

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 15 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.

*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 our evaluation we determine the size of a graph chunk 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, we define a cut edge 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.

[7]The figures for 10 and 40 graph chunks and for the 500M triples and 2000M triples datasets do not provide additional insights and are therefore omitted.

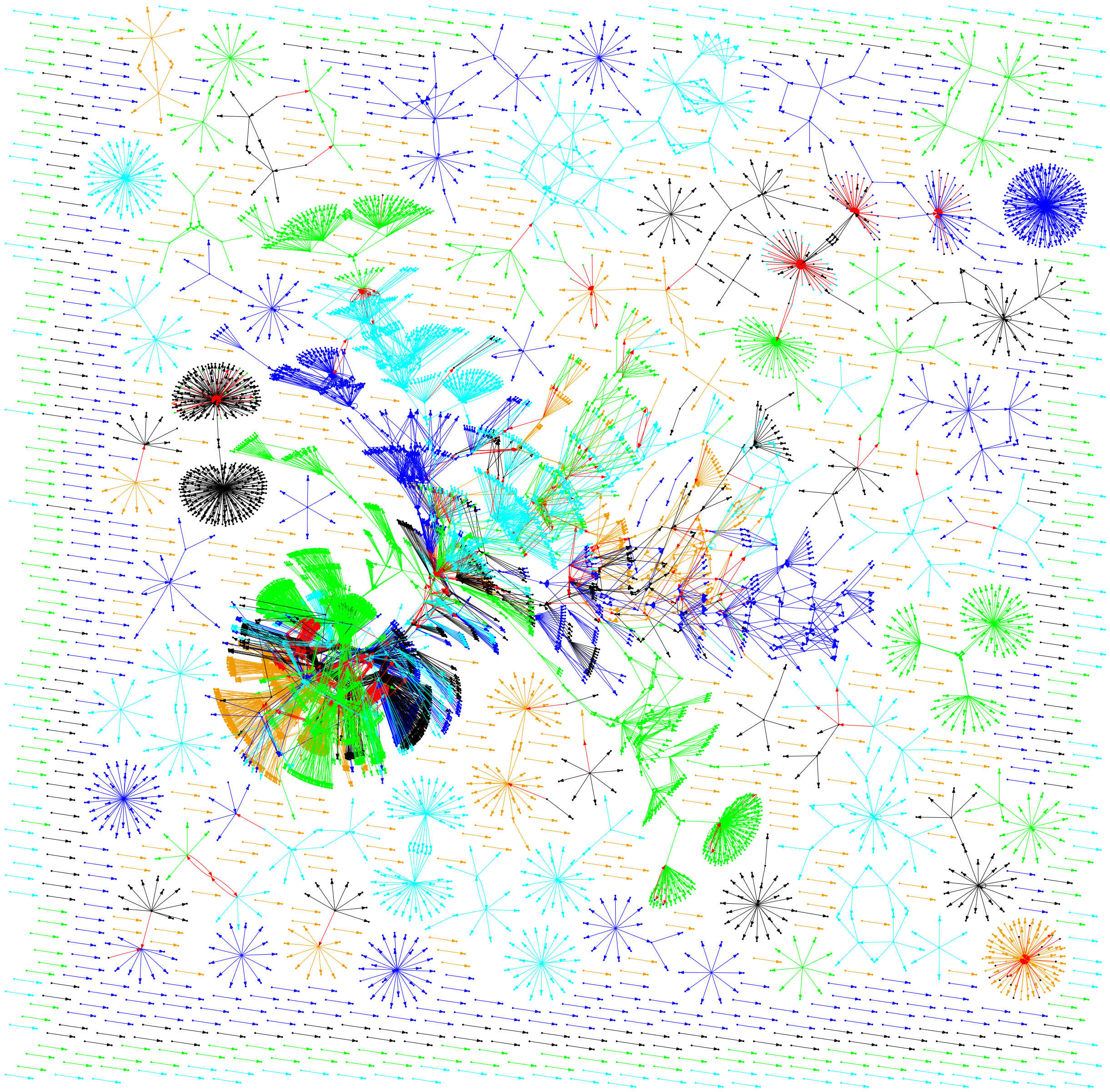


Figure 16: A plot of a minimal edge-cut cover of a 10k triples subset consisting of five chunks. Each chunk is drawn by a different colour. Cut edges are indicated by red arrows.

When measuring the number of cut edges for the different graph cover strategies, we observe 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.[8]

In order to understand, why nearly every second triple is a cut edge, we plotted 1k to 30k triples subsets of our dataset. Similar to [39], we found out that the graph consists of one huge, densely-connected core and several small sets of vertices which are densely connected with each other, but are only loosely or not at all con-

[8]During the creation of the minimal edge-cut cover, we removed the rdfs:type triples and added them 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 our procedure.

nected to the huge core. Additionally, around 20% of all triples used a subject and object that were not used by any other triple. When colouring the triples by the chunks to which they were assigned to, we could see that each of the examined graph cover strategies cuts the huge densely connected core, leading to a high number of cut edges.

Colouring 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 16. 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.** Our scalability experiments showed that the effects of scaling the number of graph chunks and the dataset 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 datasets, the hierarchical hash cover has 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 our study of the effect of the graph cover strategy on the query performance, we simulate a graph cover-independent query optimizer by using three different query execution strategies for each query (see Section 4.3). When investigating the effect of the different strategies, we could observe that 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 [40], the bushy query execution strategy is faster than the other strategies in our evaluation, we focus on this strategy in the following.

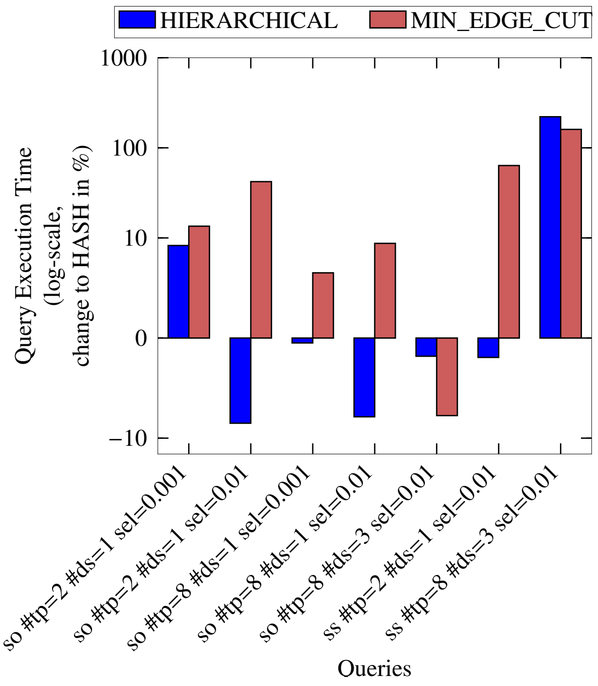


Figure 17: Change of the *exTimes* of all finished queries relative to the hash cover using bushy query execution with 10 slaves.

*Query Performance for 10 Slaves.* In order to examine the overall query performance, we investigate the queries that were finished completely. The corresponding query execution times are given in Appendix B. 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 17 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 Figure24). 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 22). 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.

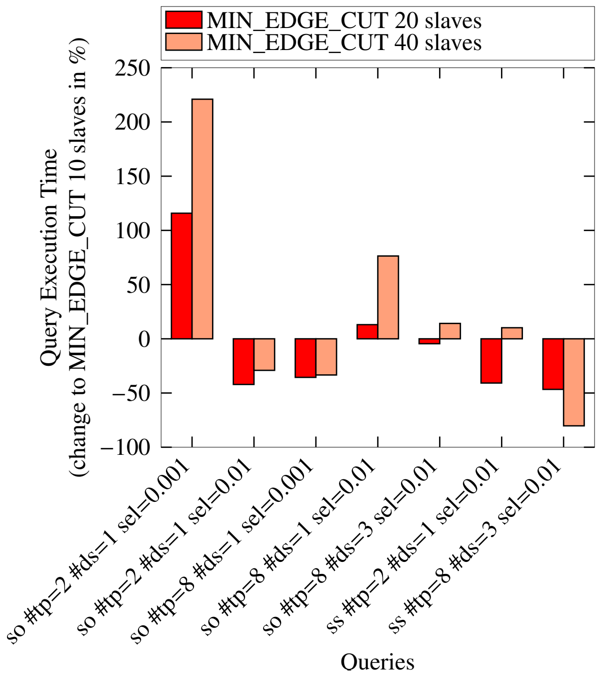


Figure 18: Change of the *exTimes* of all finished queries relative to 10 slaves using bushy query execution for the minimal edge-cut cover.

## Scaling Number of Slaves

When scaling up the number of slaves, the effect on the query execution times for the different graph covers is similar. Therefore, we discuss the effect of scaling up the number of slaves only for the minimal edge-cut cover here. Figure 18 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 slave. 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 our experimental setting, as already identified in [41], chapter 24.3 for gigabit networks in general. Our experiments show that the speed-up when scaling with the number of slaves is limited, as described by rules like the Universal Scalability Law [42].

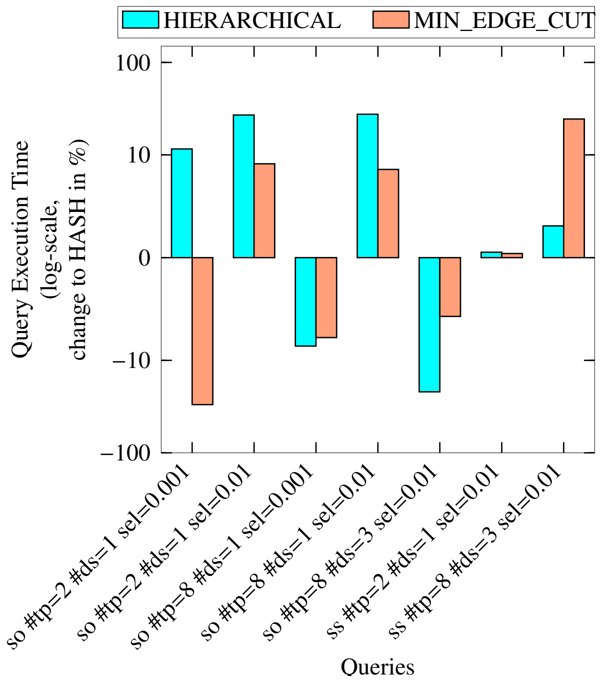


Figure 19: Change of the *exTimes* of all finished queries relative to the hash cover using bushy query execution with 40 slaves.

*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 19. 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.

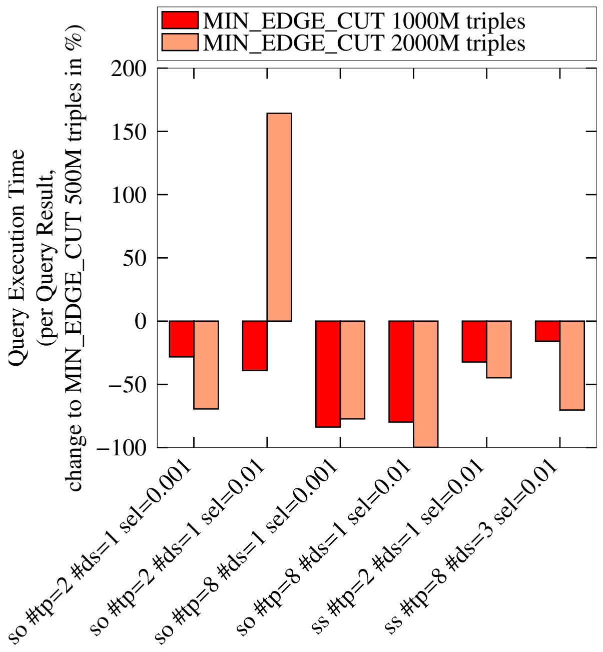


Figure 20: Change of the *exTimes* per query result of all finished queries relative to the 500M triples dataset using bushy query execution for the minimal edge-cut cover.

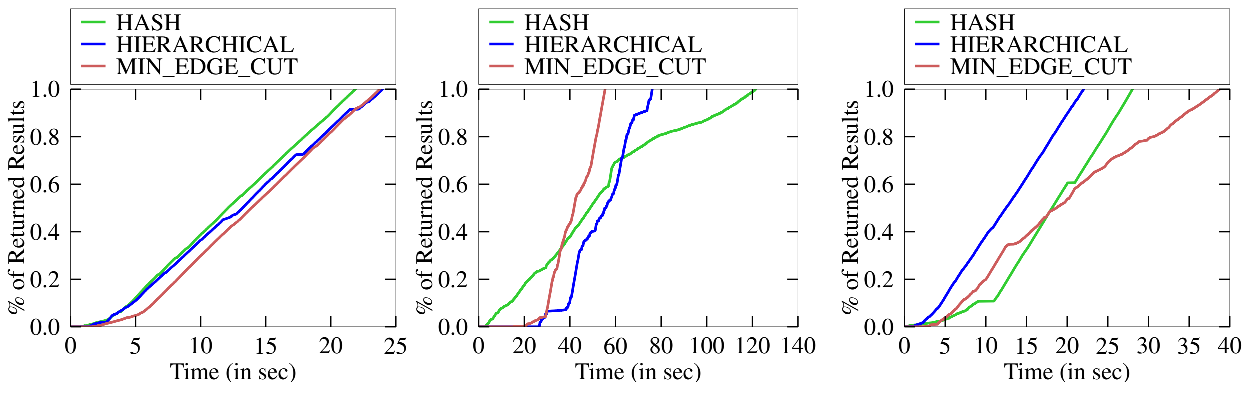
## Scaling Dataset Size

When scaling up the dataset 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, we divide *exTime* by the number of query results leading to the execution time per query result shown in Figure 20 for the minimal edge-cut cover. For almost all queries the execution time per results decreases when the dataset 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 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 query result 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 query result 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 Dataset Sizes.* Since the query execution time speed ups affect the different graph cover strategies to a different extent, we compare the different strategies for every dataset size. For the 500M triples dataset 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 2000M dataset, 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 500M dataset whereas the opposite observation can be found for the 2000M dataset.

## Results Over Time

When investigating how fast the different graph cover strategies produce their results over time, most queries have result curve functions $\chi$ like the ones shown in Figure 21a 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, 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 21b shows the $\chi$ 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



(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 21: $\chi$ for some queries at scale 10 for the 1000M triples dataset.

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 21c 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 might be caused by a more balanced workload in the beginning of the query execution.

### Susceptibility to Query Size and Shape

Our 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 might be caused by the larger chunk diameters for the minimal edge-cut cover (see Section 5.2.2). When focussing on the query shape, star-shaped queries tend to be faster than path-shaped queries. The explanation is that in our 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 our 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, we analyse the indicators for horizontal containment and vertical parallelization, 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 dataset 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 dataset 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 our 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

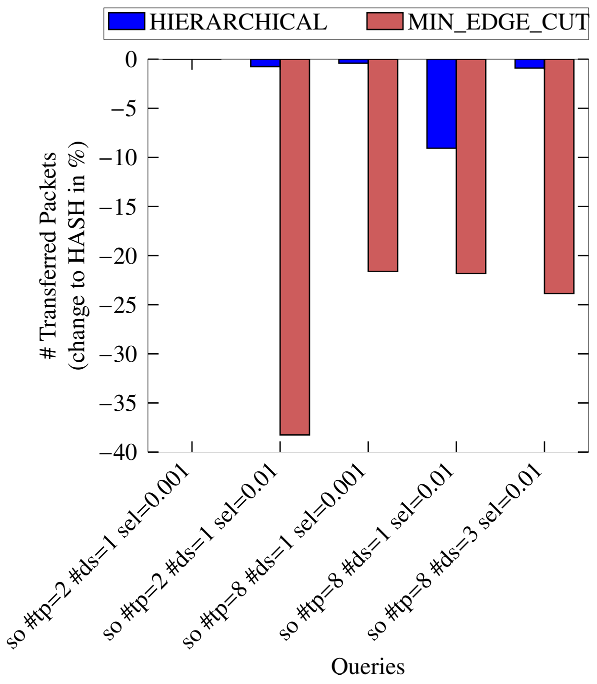


Figure 22: The relative change in the number of transferred packets *P* of the bushy query execution. Comparison of the different graph covers at 10 slaves.

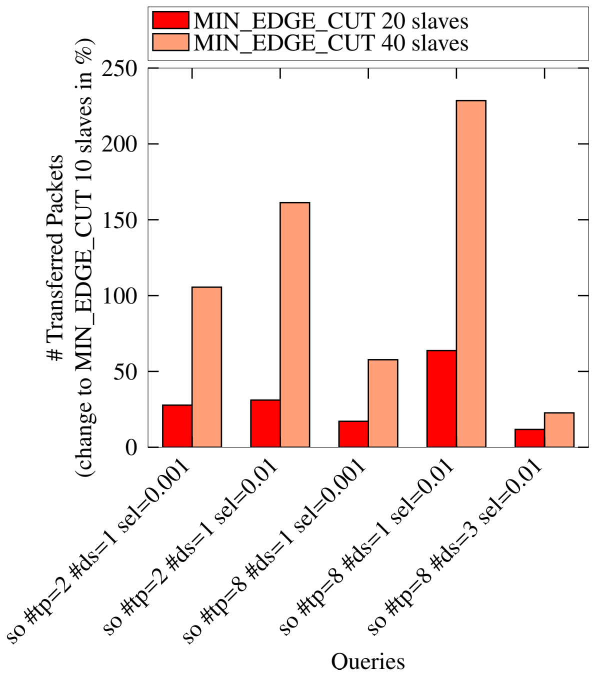


Figure 23: The relative change in the number of transferred packets *P* of the bushy query execution. Comparison of the different number of slaves for the minimal edge-cut cover.

length of the queried path increases. In order to examine the horizontal containment of the different graph cover strategies, Figure 22 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 we used queries with different characteristics, we investigated which of theses characteristics lead to an increased data transfer and number of transferred packets. We found out that long path-shaped queries have the highest data transfer and star-shaped queries the smallest. The overall query selectivity cannot be used to estimate the data transfer. In our evaluation, the impact of the number of used data sources cannot be separated from the influence of the number of results, since both queries with `#ds=3` produce also much more results than the other queries.

**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 23 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 our 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 Dataset Size.** Scaling up the dataset 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 query result decreases since more query results might be computed with the data of a single chunk. In contrast to this assumption, we found out that the number of transferred packets per query result increases by up to 100% for all queries when scaling up to the 1000M triples dataset. When scaling up to the 2000M triples dataset the number of transferred packets per query result 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 dataset 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, we examined

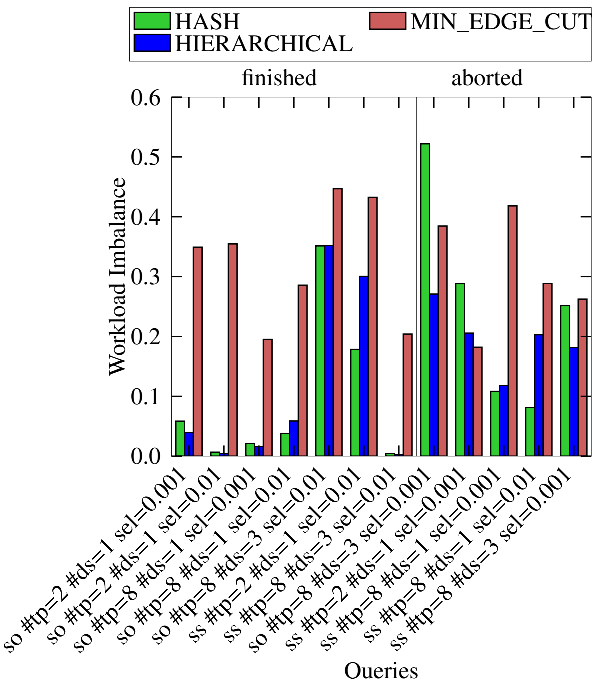


Figure 24: Workload imbalance $W$ of the bushy query execution. Comparison of the different graph covers at 10 slaves.

how the total computational effort $w(C)$ changes. As expected, we observed that in our 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 dataset size, the computational effort increases for all graph cover strategies equally, since the queries produce more results.

*Workload Imbalance.* When analysing the workload imbalance $W$ as shown in Figure 24, 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 dataset (see Section 5.2.2). Since these chunks contain less 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 are 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, we can observe that 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.

**Scaling Number of Slaves.** In order to visualize the effect of the number of slaves on the workload imbalance better, Figure 25 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 had the most imbalanced workloads for every examined number of slaves.

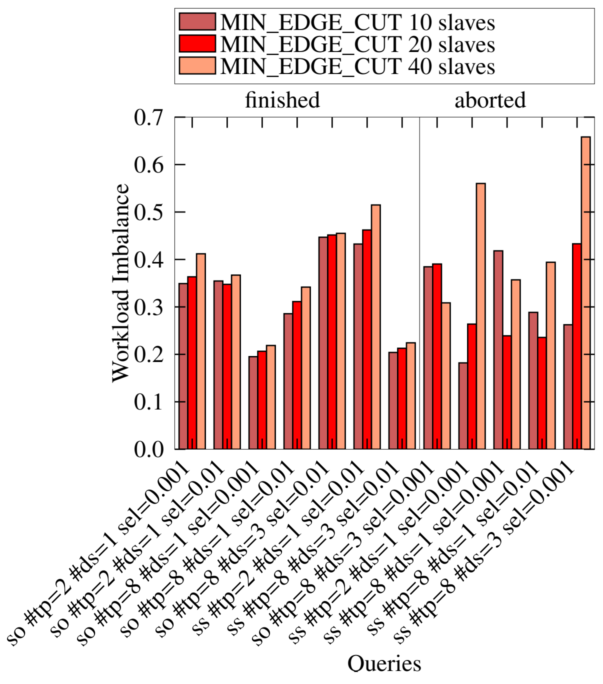


Figure 25: Workload imbalance $W$ of the bushy query execution. Comparison of the different number of slaves for the minimal edge-cut cover.

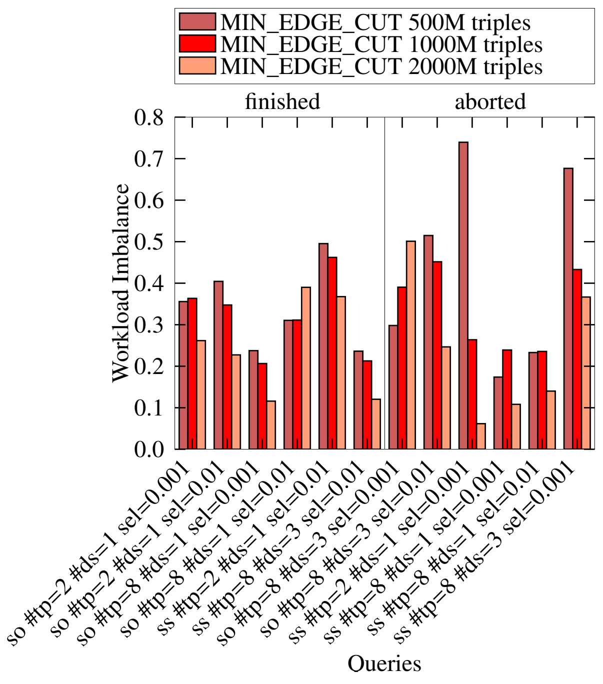


Figure 26: Workload imbalance $W$ of the bushy query execution. Comparison of the different dataset sizes for the minimal edge-cut cover.

**Scaling Dataset Size.** As shown in Figure 26, the workload imbalance decreases for the minimal edge-cut cover for most queries, when the dataset 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 datasets.

## 5.3. *Lessons Learned*

When we analysed the results of our experiments, our observations confirmed several of our expectations:

- The distributed query execution speeds up the query execution for the hash-based and the minimal edge-cut 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.
- Scaling up the number of slaves reduces the horizontal containment.
- The vertical cover needs to transfer the most packets and has the highest workload imbalance since the triple pattern matches 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, our evaluation showed 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 dataset 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 dataset 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 might be more important than the data transfer.
- Both hash-based covers perform nearly the same.

Only for small datasets, the hierarchical hash cover has a slightly better overall performance than the hash cover.

- The *n*-hop replication reduces the number of transferred packets drastically but the overall query performance is decreases since duplicate intermediate results increases the total computational effort strongly.

A few aspects that we wanted to investigate could not be analysed by our experiments:

- Since most of our generated queries combining triples from three data sources were aborted after one million results, the impact of 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 our study we have examined the impact of two hash-based graph covers, which assign triples to graph chunks based on the hash of the complete IRI or only an IRI prefix, the minimal edge-cut cover, which assigns triples to chunks based on structural information of the graph, the vertical cover, which assigns triples to chunks based on their properties, and the 2-hop extension of the hash cover. The minimal edge-cut cover strategy takes more effort to be prepared 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 [9, 25, 28] 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 22), due to a more unbalanced workload (see Figure 24), the query execution time of minimal edge-cut is effectively slower (see Figure 17).

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 expectational 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 till 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 avoid the computation of duplicate results in order to benefit from the triple replication.

Our investigation suggests that in our setting the minimal edge-cut cover takes the most effort to be prepared (see Figure 14) but does not perform better over all (see Figure 17). Since both hash-based covers perform similarly, the simpler hash cover implementation might be preferred, if other functionality such as prefix matching does not benefit from the hierarchical hash cover.

## 6. Related Work

There are two categories of work related to our study. The first type consists of other graph cover evaluations and is described in Section 6.1. The second type consists of graph cover strategies that we have not evaluated, yet. They are described in Section 6.2.

### *6.1. Other Studies of Graph Cover Strategies*

In the literature, papers like [19], [11], [16] and [12] have compared the effect of the minimal edge-cut cover strategy with hash-based cover strategies. They reported that the minimal edge-cut cover produces the least query execution time since it reduces the amount of transferred intermediate results. But they have neglected that using Apache Hadoop [43] or its distributed file system to join partial results from different compute nodes punishes data transfer by the potentially huge overhead of possibly several Hadoop jobs (see [18]). The results of our experiments indicate that in a system without this overhead, the workload balance may have a higher impact on the overall query performance than the data transfer.

Also [28] came to the result that the minimal edge-cut cover outperforms hash-based graph covers. Since their proposed distributed query execution mechanism was not implemented at that time, their study was limited to investigating to which extent certain covers produce complete or intermediate query results. These numbers can be seen as an estimation of the expected data transfer but they do not reflect whether a minimal edge-cut cover will lead to a better overall query performance.

To the best of our knowledge, the findings in [8] are the closest to ours. They compare the system presented in [11], which uses a minimal edge-cut cover strategy, with Microsoft's Trinity.RDF, which uses a hash cover strategy. It indicates that local queries can be executed faster using the minimal edge-cut cover but if intermediate results need to be transferred between chunks the hash cover executes the queries faster. The two compared systems work fundamentally differently: [11] uses centralized RDF stores for the local query processing and Apache Hadoop for the join of partial results from different graph chunks, whereas Trinity.RDF is realized with a single distributed in-memory column store. Thus, it is not clear whether their observations are caused by the different graph cover strategies. We could confirm that the hash cover leads to a shorter query execution time, if intermediate results have to be transferred. But the queries are also executed faster, if only local data is used.

**Further approaches**

In [44] two distributed RDF stores are compared that use different extensions of the minimal edge-cut cover. Since no additional hash-based system is evaluated, it is not comparable with our study.

[45] measures the execution time of queries on different random distributions of a synthetic RDF graph, but they used the system AVALANCHE which is not designed to produce complete query results. Instead, it is optimised to produce initial results quickly without guaranteeing that a complete result set will ever be returned. In our evaluation we also examined the speed in which query results are produced.

Large datasets can also be managed by a federated RDF store, which consists of several centralized RDF stores and a query federator. The latter decomposes a query into subqueries that are processed by the individual RDF stores and merges the returned partial results. The impact of the data distribution strategy in such a system has been evaluated by [46, 47] using different query federation strategies. Although they have control of the data placement on the different RDF stores, no centralized index is used to simplify the identification of graph chunks involved in the processing of query. Therefore, the query federator needs to send additional queries to the RDF stores to identify the required chunks. This additional effort makes their evaluation not comparable with ours.

The importance of graph cover strategies has also been investigated in distributed reasoning platforms that infer new statements from an existing RDF graph. In [48] the performance of WebPIE is compared with other distributed reasoning platforms each of them using a different graph cover strategy. Since the platforms which are compared differ in more components than just the used graph cover strategy, it is difficult to estimate to which extent the measured differences are caused by the different graph cover strategies.

## *6.2. Other Graph Cover Strategies*

Our study has focussed on the most commonly used graph cover strategies, but several other graph cover strategies can be found in the literature. For the sake of completeness the other graph cover strategies, we have found so far, are mentioned below.

RDF stores that run on a single compute node use a local database as a storage back-end for the RDF graph. In order to create a distributed RDF store, this storage back-end is exchanged with a distributed general purpose database as done by, e. g., Rya [49], $H_2$RDF+ [50, 51], CumulusRDF [52] and Jena-HBase [14, 53]. In these cases, the assignment of triples to compute nodes is left to the distributed database. Thus, the assignment decision is driven by metrics not related to graphs like the number of rows in table.

Other distributed RDF stores are realised with Apache Hadoop[9] as done by, e. g., [15], PigSPARQL [54], SHARD [55], RAPID+ [56, 57] and HadoopRDF [13, 58]. They store the RDF graph in several files in the Hadoop Distributed File System (HDFS). This file system splits the files into blocks of a fixed size and distributes the resulting blocks equally among all compute nodes. Thus, the resulting graph cover is a random assignment of triples to compute nodes.

The VB-Partitioner [16] assumes that resources that frequently occur as a subject – i.e., vertices with a high degree – are frequently involved in the processing of queries. Furthermore, it assumes that such frequent subjects, which are connected via a small number of triples, are frequently queried together. Founded on these assumptions, the used graph cover identifies sets of triples with closely connected resources as subject. These sets of triples are equally distributed among the compute nodes. Thereby, triples of one set are assigned to the same compute node.

[59] splits the RDF graph into set of triples with identical subject. Sets whose contained triples have similar properties are then combined. The resulting triple sets are assigned to the compute node in a way that the graph chunks have a similar size.

The graph cover strategy proposed by [25] basically identifies all resources that only occur as subjects first. Then, all triples occurring in any path starting at one

[9]`https://hadoop.apache.org/`

of these resources are grouped into one triple set called rooted subgraphs. The resulting rooted subgraphs are assigned to compute nodes such that the number of triples that are assigned to several compute nodes is minimal.

Another type of graph cover strategies assume that the query workload does not change much over the 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 a optimal graph cover for future queries. These approaches are, for instance:

- The novel idea applied in WARP [24] 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 [22] 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 an improved horizontal containment.
- In [23] 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 [21] the queries contained in the historic query workload are first generalized by replacing every 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.

## 7. Conclusion

We have presented a comprehensive methodology and its implementation for analysing the impact of graph cover strategies on the performance of distributed RDF stores in the cloud. Our systematically varied, broad set of experiments has revealed that contrary to common assumption the minimal edge-cut cover may have a worse overall query execution performance than hash-based data placement strategies. With the provided set of varying metrics, we found out that balancing the query workload across all compute nodes may be more important for a fast query execution than the amount of network traffic. Even without knowing the future query workload, our study gives hints, how triples should be distributed to improve the query performance: (i) to reduce the data transfer, the triples stored on a single compute node should be connected and allow the traversal of longer paths (ii) triples of densely connected subgraphs should be equally distributed among all compute nodes to achieve a balanced workload. The evaluation of further graph cover strategies as well as reducing the amount of redundant computations in case of triple replication will be done in the future. Further future work will be applying compression techniques as described in [26] or using query optimization techniques like [60] to improve the performance of our distributed RDF store for arbitrary graph covers Koral. Part of our contribution are the tools CEP and Koral which are open source available on the Web for further investigation of distributed RDF data management challenges.

## References

[1] D. Janke, S. Staab, M. Thimm, On data placement strategies in distributed rdf stores, in: Proceedings of The International Workshop on Semantic Big Data, SBD '17, ACM, New York, NY, USA, 2017, pp. 1:1–1:6. doi:10.1145/3066911.3066915. URL `http://doi.acm.org/10.1145/3066911.3066915`

[2] P. Norvig, The semantic web and the semantics of the web: Where does meaning come from?, in: Proceedings of the 25th International Conference on World Wide Web, WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 2016, pp. 1–1.

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

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

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

[6] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: ISWC-2007, Vol. 4825, 2007, pp. 211–224. doi:10.1007/978-3-540-76298-0_16.

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

[8] K. Zeng, J. Yang, H. Wang, B. Shao, Z. Wang, A Distributed Graph Engine for Web Scale RDF Data, PVLDB 6 (4) (2013) 265–276. doi:10.14778/2535570.2488333.

[9] K. Lee, L. Liu, Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning, PVLDB 6 (14) (2013) 1894–1905.

[10] S. Gurajada, S. Seufert, I. Miliaraki, M. Theobald, TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing, in: SIGMOD, 2014, pp. 289–300.

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

[12] X. Zhang, L. Chen, Y. Tong, M. Wang, EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud, in: ICDE-2013, 2013, pp. 565–576. doi:10.1109/ICDE.2013.6544856.

[13] M. Farhan Husain, L. Khan, M. Kantarcioglu, B. Thuraisingham, Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools, in: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, 2010, pp. 1–10. doi:10.1109/CLOUD.2010.36.

[14] V. Khadilkar, M. Kantarcioglu, B. M. Thuraisingham, P. Castagna, 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.
URL `http://ceur-ws.org/Vol-914/paper_14.pdf`

[15] X. Zhang, L. Chen, M. Wang, Towards Efficient Join Processing over Large RDF Graph Using MapReduce, in: A. Ailamaki, S. Bowers (Eds.), Scientific and Statistical Database Management, Vol. 7338 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 250–259. doi:10.1007/978-3-642-31235-9_16.
URL `http://dx.doi.org/10.1007/978-3-642-31235-9_16`

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

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

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

[19] O. Curé, H. Naacke, M. A. Baazizi, B. Amann, 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)., 2015, pp. 16–31.

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

[21] L. Galarraga, K. Hose, R. Schenkel, Partout: A Distributed Engine for Efficient RDF Processing, CoRR abs/1212.5.
URL `http://arxiv.org/abs/1212.5636`

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

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

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

[25] B. Wu, Y. Zhou, P. Yuan, H. Jin, L. Liu, SemStore: A Semantic-Preserving Distributed RDF Triple Store, in: CIKM-2014, 2014.

[26] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, I. Stoica, ZipG: A Memory-efficient Graph Store for Interactive Queries, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, ACM, New York, NY, USA, 2017, pp. 1149–1164. doi:10.1145/3035918.3064012.
URL `http://doi.acm.org/10.1145/3035918.3064012`

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

[28] A. Potter, B. Motik, I. Horrocks, Querying Distributed RDF Graphs: The Effects of Partitioning, in: Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014), 2014, pp. 29–44.

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

[30] D. Janke, S. Staab, M. Thimm, Impact analysis of data placement strategies on query efforts in distributed rdf stores, Tech. rep., Institute for WeST, `http://west.uni-koblenz.de/sites/default/files/research/publications/janke2016iao_technicalreport.pdf` (2016).
URL `http://west.uni-koblenz.de/sites/default/files/research/publications/janke2016iao_technicalreport.pdf`

[31] E. Prud'hommeaux, S. Harris, A. Seaborne, SPARQL 1.1 Query Language, W3c recommendation, W3C (2013).
URL `http://www.w3.org/TR/sparql11-query/`

[32] J. Pérez, M. Arenas, C. Gutierrez, Semantics and Complexity of SPARQL, ACM Trans. Database Syst. 34 (3) (2009) 16:1—-16:45. doi:10.1145/1567274.1567278.
URL `http://doi.acm.org/10.1145/1567274.1567278`

[33] M. Arenas, J. Pérez, Federation and Navigation in SPARQL 1.1, in: T. Eiter, T. Krennwallner (Eds.), Reasoning Web. Semantic Technologies for Advanced Query Answering, Vol. 7487 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 78–111. doi:10.1007/978-3-642-33158-9_3.
URL `http://dx.doi.org/10.1007/978-3-642-33158-9\_3`

[34] O. Görlitz, M. Thimm, S. Staab, Splodge: Systematic generation of sparql benchmark queries for linked open data, The Semantic Web–ISWC 2012 (2012) 116–132.

[35] Koral, https://github.com/Institute-Web-Science-and-Technologies/koral, accessed: 2016-10-24.

[36] D. Wood, P. Gearon, T. Adams, Kowari: A platform for semantic web storage and analysis, in: In XTech 2005 Conference, 2005, pp. 05–0402.

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

[38] Cep, https://github.com/Institute-Web-Science-and-Technologies/cep, accessed: 2016-10-24.

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

[40] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, A. Polleres, Efficiently Joining Group Patterns in SPARQL Queries, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 228–242. doi:10.1007/978-3-642-13486-9_16.
URL `http://dx.doi.org/10.1007/978-3-642-13486-9_16`

[41] W. R. Stevens, TCP/IP Illustrated (Vol. 1): The Protocols, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.
URL `http://www.pcvr.nl/tcpip/`

[42] N. J. Gunther, 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.

[43] Apache hadoop, `https://hadoop.apache.org/`, accessed: 2016-10-21.

[44] A. Potter, B. Motik, Y. Nenov, I. Horrocks, Distributed RDF Query Answering with Dynamic Data Exchange, Springer International Publishing, Cham, 2016, pp. 480–497. doi:10.1007/978-3-319-46523-4_29.
URL `http://dx.doi.org/10.1007/978-3-319-46523-4{\_}29`

[45] C. Basca, A. Bernstein, Querying a Messy Web of data with AVALANCHE, Web Semantics: Science, Services and Agents on the World Wide Web 26.

[46] N. A. Rakhmawati, M. Hausenblas, On the Impact of Data Distribution in Federated SPARQL Queries, in: Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on, 2012, pp. 255–260. doi:10.1109/ICSC.2012.72.

[47] N. A. Rakhmawati, M. Karnstedt, M. Hausenblas, S. Decker, On Metrics for Measuring Fragmentation of Federation over SPARQL Endpoints, in: Proceedings of the 10th International Conference on Web Information Systems and Technologies, 2014, pp. 119–126. doi:10.5220/0004760101190126.

[48] J. Urbani, S. Kotoulas, J. Massen, F. van Harmelen, H. Bal, Webpie: A web-scale parallel inference engine using mapreduce, Web Semantics 10.

[49] R. Punnoose, A. Crainiceanu, D. Rapp, Rya: A scalable rdf triple store for the clouds, in: 1st Int. Workshop on Cloud Intelligence, ACM, 2012, pp. 4:1–4:8.

[50] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, N. Koziris, H2RDF+: High-performance distributed joins over large-scale RDF graphs, in: Big Data, 2013 IEEE International Conference on, 2013, pp. 255–263. doi:10.1109/BigData.2013.6691582.

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

[52] G. Ladwig, A. Harth, 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), 2011.

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

[54] A. Schätzle, M. Przyjaciel-Zablocki, G. Lausen, PigSPARQL: Mapping SPARQL to Pig Latin, in: Proceedings of the International Workshop on Semantic Web Information Management, SWIM '11, ACM, New York, NY, USA, 2011, pp. 4:1—-4:8. doi:10.1145/1999299.1999303.
URL `http://doi.acm.org/10.1145/1999299.1999303`

[55] K. Rohloff, R. E. Schantz, High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store, in: Programming Support Innovations for Emerging Distributed Applications, PSI EtA '10, ACM, New York, NY, USA, 2010, pp. 4:1—-4:5. doi:10.1145/1940747.1940751.
URL `http://doi.acm.org/10.1145/1940747.1940751`

[56] P. Ravindra, H. Kim, K. Anyanwu, An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce, in: G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, J. Pan (Eds.), The Semanic Web: Research and Applications, Vol. 6644 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 46–61. doi:10.1007/978-3-642-21064-8_4.
URL `http://dx.doi.org/10.1007/978-3-642-21064-8_4`

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

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

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

[60] I. Trummer, C. Koch, Solving the Join Ordering Problem via Mixed Integer Linear Programming, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, ACM, New York, NY, USA, 2017, pp. 1025–1040. doi:10.1145/3035918.3064039.
URL `http://doi.acm.org/10.1145/3035918.3064039`

## Appendix A. Characteristics of the Used Datasets

Table A.5 shows the characteristics of the 500M, 1G and 2G triples subsets of the real-world billion triple challenge dataset from 2014 [37] used in our evaluation.

| dataset | 500M | 1G | 2G |
|---|---|---|---|
| # triples | 500M | 1,000M | 2,000M |
| # unique graphs | 9k | 13k | 21k |
| # unique subjects | 50,602k | 90,713k | 170,246k |
| # unique properties | 179k | 412k | 812k |
| # unique objects | 76,642k | 147,625k | 259,945k |

Table A.5: Dataset characteristics.

## Appendix B. Query Execution Times

Table B.6 shows the query execution times of all graph covers for the 10 slaves and the 1 billion triples dataset in seconds. Table B.7 shows the query execution times for the hash cover, hierarchical cover and minimal edge-cut cover for the 1 billion triples dataset when using 10, 20 and 40 slaves in seconds. Additionally, this table includes the execution times for the centralized query execution. Table B.8 shows the query execution times for the hash cover, hierarchical cover and minimal edge-cut cover for 20 slaves when using the 500 million, 1 billion and 2 billion triples dataset in seconds. When interpreting the query execution times, it should be noted that the number of returned results differ for the different dataset sizes and the 2-hop hash cover. Query execution times of aborted queries are highlighted.

| query | hash | hierarchical | minimal edge-cut | vertical | 2-hop hash |
|---|---|---|---|---|---|
| $q_1$ | 2.2 | 2.4 | 2.5 | 45.1 | 1.1 |
| $q_2$ | 16.5 | 15.1 | 23.4 | 3,502.1 | 354.8 |
| $q_3$ | 245.4 | 244.2 | 261.4 | 1,133.2 | 143.7 |
| $q_4$ | 45.6 | 42.0 | 49.9 | 3,459.8 | 289.8 |
| $q_5$ | 121.5 | 76.1 | 55.4 | 2,516.0 | 24.5 |
| $q_6$ | 2,505.6 | 2,460.0 | 2,311.4 | 2,455.7 | 91.0 |
| $q_7$ | 36.0 | 38.3 | 54.6 | 57.4 | 39.9 |
| $q_8$ | 4.4 | 4.4 | 7.3 | 44.7 | 14.2 |
| $q_9$ | 22.5 | 23.6 | 25.6 | 201.0 | 34.4 |
| $q_{10}$ | 21.9 | 24.0 | 23.8 | 1,176.5 | 30.9 |
| $q_{11}$ | 28.0 | 22.0 | 38.7 | 1,099.1 | 28.2 |
| $q_{12}$ | 8.0 | 25.7 | 20.8 | 3,908.2 | 667.8 |

Table B.6: Query execution times in sec for the comparison of the different graph cover strategies.

|  | hash |  |  | hierarchical |  |  | minimal edge-cut |  |  | centralized |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 7.8 |
| $q_2$ | 16.5 | 18.0 | 15.2 | 15.1 | 12.0 | 19.3 | 23.4 | 13.6 | 16.6 | 238.2 |
| $q_3$ | 245.4 | 174.1 | 188.9 | 244.2 | 162.9 | 172.6 | 261.4 | 168.4 | 174.2 | 69.6 |
| $q_4$ | 45.6 | 53.6 | 81.0 | 42.0 | 57.6 | 103.3 | 49.9 | 56.4 | 88.0 | 276.2 |
| $q_5$ | 121.5 | 62.0 | 45.0 | 76.1 | 39.2 | 56.4 | 55.4 | 49.0 | 91.8 | 55.0 |
| $q_6$ | 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 | 247.3 |
| $q_7$ | 36.0 | 20.3 | 36.9 | 38.3 | 50.6 | 57.8 | 54.6 | 47.4 | 60.2 | 23.6 |
| $q_8$ | 4.4 | 4.4 | 8.0 | 4.4 | 5.5 | 8.0 | 7.3 | 4.3 | 8.0 | 3.3 |
| $q_9$ | 22.5 | 25.2 | 24.4 | 23.6 | 25.1 | 35.6 | 25.6 | 24.6 | 40.1 | 49.4 |
| $q_{10}$ | 21.9 | 24.9 | 27.5 | 24.0 | 34.7 | 44.3 | 23.8 | 24.8 | 39.8 | 57.2 |
| $q_{11}$ | 28.0 | 22.5 | 25.0 | 22.0 | 24.0 | 41.4 | 38.7 | 24.3 | 36.5 | 124.9 |
| $q_{12}$ | 8.0 | 6.0 | 3.3 | 25.7 | 7.5 | 3.4 | 20.8 | 11.1 | 4.1 | 313.6 |

Table B.7: Query execution times in sec when scaling the number of slaves.

|  | 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 |

Table B.8: Query execution times in sec when scaling the dataset size.

## Appendix C. Generated Queries

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://purl.org/ontology/bibo/Webpage> ?v1.
    ?v1 <http://purl.org/dc/terms/created> ?v2.
} LIMIT 1000000
```

Listing 1: Query `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 2: Query `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 3: Query `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 4: Query `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 5: Query `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 6: Query `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 7: Query `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 8: Query `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 9: Query `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 10: Query `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 11: Query `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 12: Query `ss #tp=8 #ds=3 sel=0.01`