# On Data Placement Strategies in Distributed RDF Stores

Daniel Janke
Universität Koblenz-Landau, Germany
Institute for Web Science and Technologies
danijank@uni-koblenz.de

Steffen Staab
Universität Koblenz-Landau, Germany
Institute for Web Science and Technologies
University of Southampton, UK
Web and Internet Science Group
staab@uni-koblenz.de

Matthias Thimm
Universität Koblenz-Landau, Germany
Institute for Web Science and Technologies
thimm@uni-koblenz.de

## 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) different query results may be computed on several compute nodes in parallel (vertical parallelization) and (b) 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.

## CCS CONCEPTS

•**Information systems** → **Graph-based database models; Database performance evaluation;** *Parallel and distributed DBMSs;*

## KEYWORDS

Distributed RDF stores, graph partitioning, benchmark

## 1 INTRODUCTION

In the last years, the requirement for RDF stores that can cope with 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 [24]. Another example is the European Bioinformatics Institute (EMBL-EBI) that would like to convert its datasets into RDF resulting in a graph consisting of trillions of triples. To date no such scalable RDF store exists and the current EBI RDF Platform can handle only 10 billion triples [22].

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.

*SBD 2017, Chicago, USA*
© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

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 [7], YARS2 [12, 13], Clustered TDB [25] and Trinity.RDF [30]). 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 [21]. 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, 15, 31]).

It is a commonly held belief that query completion is optimized by approaches that emphasize local computation such as minimal edge-cut (see [15, 20, 31]). The first major contribution of this paper is to challenge this assumption by new experiments. Our results indicate that contrary to commonly held beliefs, hash covers may outperform, e.g., minimal edge-cut covers.

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:

- *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.
- *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.

Using these measurements, we derive the second important contribution of this paper. We understand 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 [15], [20] and [31] suffered from a setting with highly inefficient methods for data transfer (i.e. based on the Hadoop/HDFS infrastructure) (see [17]).

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. To avoid the bias of, e. g. inefficient methods for data transfer, the third important contribution is the flexible open source platform Koral. It executes queries on arbitrary graph covers and transfers intermediate results within the network.

The remainder of this work is structured as follows. In Sec. 2 the frequently used graph cover strategies are described. Thereafter, our novel benchmark methodology is described (see Sec. 3) and in Sec. 4 our analysis indicating that (i) hash covers may outperform minimal edge-cut covers and (ii) vertical parallelization may be more important than horizontal containment is described. Finally, we describe why previous evaluations concluded that the amount of data transfer is crucial for the query execution effort (see Sec. 5) before we conclude in Sec. 6.

## 2 GRAPH COVER STRATEGIES

To formalize the problem, we define RDF graphs like in [11]. 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 2.1*: 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 vertices contained in $G$ 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* $o$. 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 notes is denoted as $C$ in the rest of this paper.

*Definition 2.2*: Let $G$ denote an RDF graph. Then a *graph cover* is a function cover: $G \rightarrow 2^C$, that assigns each triple of a graph $G$ to at least one compute node.

*Definition 2.3*: The function chunk returns the triples assigned to a specific compute node by a graph cover (*graph chunks*). It is defined as

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

Beside the graph cover strategies that are described in the following, there exist other approaches such as the ones used by Partout [8], COSI [5], [4], WARP [14], VB-Partitioner [20] and [29]. 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. 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 [7], YARS2 [12, 13], Clustered TDB [25] and Trinity.RDF [30].

**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 [21] 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.

**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 [19]. 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 [19]. RDF stores like EAGRE [31], [26] and [15] 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.

## 3 METHODOLOGY FOR BENCHMARKING GRAPH COVER STRATEGIES

When defining a methodology for investigating the effects of graph cover strategies on distributed RDF stores, several complexities arise. Beyond overall performance for the SPARQL query processing [27], we want to observe indications that contribute to understanding how graph cover strategies may relate to scalability. Sec. 1 has explained several high level indicators, which are formally defined in Sec. 3.1.

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 (see Sec. 3.2). Furthermore, actual query execution constitutes a highly influential factor, too, for which we need to specify execution strategies (see Sec. 3.3) as well as execution operation (see Sec. 3.4). 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.

### 3.1 Evaluation Measures

In this subsection, we define the measures we have found most useful to characterize different graph cover strategies.

**Overall performance.** The query execution time is measured as the overall performance characteristics of an RDF store. More precisely it is the time interval between issuing the query $q$ (more precisely the query execution tree as elaborated on in Sec. 3.3) at time $t_0^q$ and the time when the last result is returned at $t_{K^q}^q$ with $K^q$ representing the overall number of query results for query $q$. We drop $^q$ when it is unambiguous from context as in the following definitions.

*Definition 3.1*: Overall query performance is evaluated by the query time to completion: $exTime := t_K - t_0$.

**Vertical parallelization.** In order to measure workload independently of time needed, 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 to estimate their load. 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 entropy.

*Definition 3.2*: For a $cover$ and a query execution tree $q$, workload entropy $W$ is:

$$W := -\sum_{c \in C} \frac{w(c)}{w(C)} \log_2 \frac{w(c)}{w(C)},\ 0 \leq W \leq -\log_2\left(\frac{1}{|C|}\right),$$

where the workload of a compute node $w(c)$ is defined by the number of join comparisons on $c$ and $w(C) := \sum_{c \in C} w(c)$ denotes the number of join comparisons overall.

In the strict sense, workload entropy 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 entropy and measures for horizontal containment (see Table 1).

| | Data transfer low | Data transfer high |
|---|---|---|
| W low | low vertical parallelization | low vertical parallelization (unlikely situation) |
| W high | high vertical parallelization | low to medium vertical parallelization |

**Table 1: Measurement of vertical parallelization.**

**Horizontal containment.** Time-based measurements such as the query execution time $exTime$ depend on the exact configuration of the system such as network bandwidth and latency. Workload and workload entropy 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:[1]

*Definition 3.3*: 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 $T_c := \sum_{op} m^{op} \cdot |\mathrm{dom}(\mu_1^{op})|$. Each join operation $op$ that leads to the sending of variable bindings to another compute node $c' \neq c$ contributes with data size $m^{op} \cdot |\mathrm{dom}(\mu_1^{op})|$ where $|\mathrm{dom}(\mu_1^{op})|$ are the number of variables of a binding and $m$ are the number of bindings $m = |\{\mu_i^{op}\}|$.

The data transfer is sometimes also used as the preferred measurement for overall query efforts in the cloud (see [26]), 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 balance.

[1]We follow common notation and use $\mu$ to represent a variable binding, i.e. a function that maps from variables to values, and $\mathrm{dom}(\mu)$ to refer to the set of variables of this binding.

## 3.2 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 [9], which given arbitrary real-world datasets varies the query characteristics:

**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.

## 3.3 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.

## 3.4 Distributed RDF Store for Arbitrary Graph Covers (Koral)

The distributed RDF store for arbitrary graph covers (Koral) [3] 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. Due to the space limitations, Koral will only be sketched here. Its formal definition and proofs of soundness and completeness are given in [16].

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. Koral consists of one master node and $|C|$ slave nodes. When the master receives a query, a query execution coordinator is instantiated. The coordinator creates the query execution tree that specifies the query execution strategy (bushy, left-linear, right-linear) for the given query and sends the query execution tree to all slaves. To *vertically parallelize* workload execution, each slave executes each query operator. Aiming at *horizontal containment*, Koral avoids (some) duplicate joins and corresponding data transfer by assigning each join uniquely to the slave responsible for the join of the resource, i.e. $\mu(v)$ (where $v$ is the join variable).

## 4 EVALUATION

The experimental setup we have used for the impact analysis of different graph cover strategies on the query execution effort is explained in Sec. 4.1. Our results are described in Sec. 4.2.

### 4.1 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 and a minimal edge-cut cover are compared. Both hash covers are reimplemented following the descriptions in [21]. Thereby, the hash is computed only on the subject of each triple. For the creation of the minimal edge-cut cover we use METIS [19] as done by other distributed RDF stores like [26], [15], D-SPARQ [23] and WARP [14].

**Dataset and Queries.** In order to avoid effects that occur due to the generation process of a synthetic dataset, we use a subset of the real-world billion triple challenge dataset from 2014 (BTC2014) [18]. This dataset has been generated by crawling data from several data sources of the linked open data cloud. The used subset contains the first one billion syntactically correct triples.

Following the strategy explained in Sec. 3.2, we generate basic graph patterns with

**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 the extensible evaluation platform for graph cover strategies (CEP) [1], we set up the evaluation as follows. CEP downloads the BTC2014 dataset, removes all syntactically incorrect triples and creates the one billion triple dataset. The resulting dataset is used by SPLODGE [9], configured as described above, to generate the query set for the benchmark. For each graph cover strategy Koral is cleared, 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 query execution and creates tables and corresponding diagrams.

**Computer and Software Environment.** The graph cover evaluation platform CEP is executed on a VM with 4 cores and 8 GB RAM. Koral is executed on 21 VMs. The master has 4 cores and 64 GB RAM and the 20 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 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 5.1.0.dfsg-2 is used.

### 4.2 Results

As the possible configurations of independent variables (configuration settings) and dependent variables (evaluation measures) is staggering, we focus analysis results by (i) depicting overall query performance under a larger variation of independent variables, and (ii) showing indicators for vertical parallelization and horizontal containment based on few selected independent variables. An extensive evaluation and guide through our large set of experiments is to be found in the long version of this paper [16].

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 matching triples from 3 data sources and the sum of the selectivities of all triple patterns is 0.01.

#### 4.2.1 *Measuring Overall Query Performance under Varying Independent Variables.*

When measuring the overall query performance we could observe similarly to [28] that the bushy query execution strategy requires less execution time for all graph covers for almost all queries. Due to the space limitations, we present only results of the bushy query execution in the following. The more comprehensive analysis of the different query execution strategies can be found in [16].

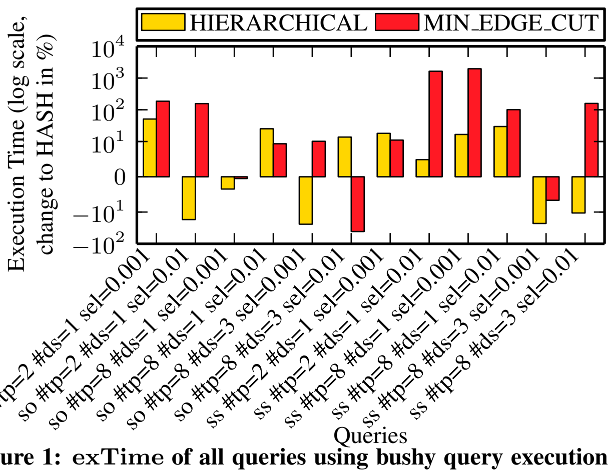

**Figure 1: exTime of all queries using bushy query execution relative to the hash cover.**

Figure 1 shows the execution times $exTime$ of the queries for all graph covers. In most cases, the minimal edge-cut cover causes the longest query execution times. When comparing the hash cover with the hierarchical hash cover, none of them is faster in general.

**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. When focussing on the query shape, star-shaped queries tend to be faster than path-shaped queries. This can be explained by the unnecessity of data transfer as described in the following section. In two cases the minimal edge-cut cover is significantly slower for star-shaped queries. In both cases the low workload entropy $W$ (see Fig. 3) indicates an unbalanced distribution of the workload as possible cause. When we examined the execution behaviour of this query in more detail, we observed that a few graph chunks of the minimal edge-cut cover contains most triples required to produce query results whereas the other graph chunks produce only very few results. This leads to the longer execution time.

**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. See [16] for more details.

### 4.2.2 Measuring Dependent Variables.

**Horizontal Containment.** All examined graph cover strategies assign triples with the same subject to the same chunk. Therefore, all triples required to produce one result of a 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 could be observed. Thus, all graph cover strategies result in a perfect horizontal containment for star-shaped queries.

In our evaluation, the data transfer $T$ increases for all graph cover strategies, if the number of triple patterns included in the path-shaped query increases. Thus, the likelihood to leave a graph chunk during query processing increases for all graph cover strategies, when the length of the queried path increases. Nevertheless, the minimal edge-cut cover produces the least data transfer for all path-shaped queries. Except for query `so #tp=2 #ds=1 sel=0.001`, the data transfer could be reduced by 18%-38% (see Fig. 2). Thus, it has the best horizontal containment. The data transfer of the hash and the hierarchical hash cover is nearly the same for all queries.

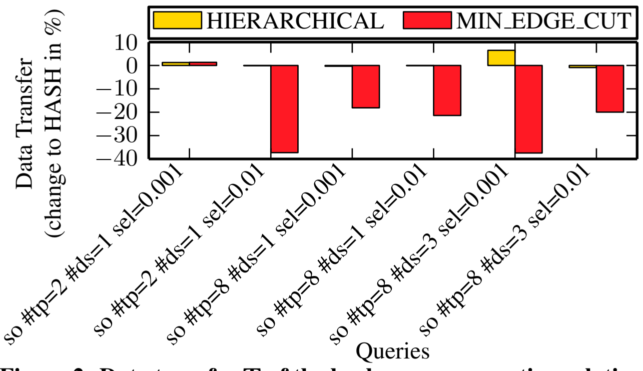

**Figure 2: Data transfer T of the bushy query execution relative to the hash cover.**

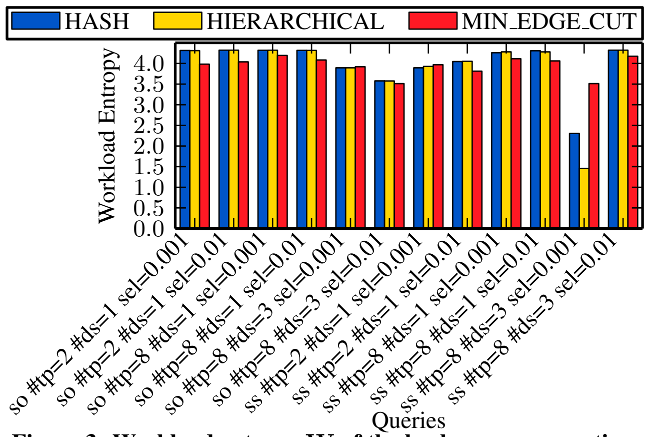

**Figure 3: Workload entropy W of the bushy query execution.**

**Vertical Parallelization.** When analysing the workload entropy $W$ in Fig. 3, the minimal edge-cut cover has the most unbalanced workload of all graph covers. This is caused by four small graph chunks that have a much smaller workload than the other graph chunks. Whereas, one huge graph chunk with 130k triples does not produce a higher workload than the other graph chunks.

$W$ of query `ss #tp=8 #ds=3 sel=0.001` is for all graph covers low, since the query uses some triple patterns for which only a few matching triples exist in the dataset. Thus only the slaves which store these triples produce join comparisons. Especially in the case of the hierarchical hash cover, joins were only computed on three slaves whereas the minimal edge-cut cover spreads these instances across 18 Koral slaves.

Combining the low workload entropy $W$ with the low data transfer $T$, 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 high vertical parallelization.

## 4.3 Discussion

In our evaluation 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, and the minimal edge-cut cover, which assigns triples to chunks based on structural information of the graph. The latter 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 [21, 26, 29] 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 data transfer by 18% to 38% in comparison to hash-based strategies (see Fig. 2), due to a more unbalanced workload (see Fig. 3), the query execution time of minimal edge-cut is effectively slower (see Fig. 1).

Our investigation suggests that in our setting the minimal edge-cut cover does not perform better over all (see Fig. 1). Nevertheless, the minimal edge-cut cover might still be a good choice in setting in which locality is important, e. g. in heterogeneous networks with unreliable compute nodes. 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.

## 5 RELATED WORK

In the literature, papers like [6], [15], [20] and [31] 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 [2] 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 [17]). 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 [26] concluded that the minimal edge-cut cover outperforms hash-based graph covers. Since their 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 [30] are the closest to ours. They compare the system presented in [15], 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: [15] 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.

## 6 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. These results let us expect that data placement strategies like Partout [8], WARP [14], COSI [5] or [4] that aim to balance future workload by balancing the workload of historic queries will produce better query performances than data placement strategies distributing data based only on the graph structure or hashes. The evaluation of this conjecture will be done in the future. 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] 2016. CEP. (2016). Retrieved 2016-10-24 from https://github.com/Institute-Web-Science-and-Technologies/cep
[2] 2016. Hadoop. (2016). Retrieved 2016-10-21 from https://hadoop.apache.org/
[3] 2016. Koral. (2016). Retrieved 2016-10-24 from https://github.com/Institute-Web-Science-and-Technologies/koral
[4] C. Basca and A. Bernstein. 2013. Distributed SPARQL Throughput Increase: On the effectiveness of Workload-driven RDF partitioning. In *ISWC2013*.
[5] M Bröcheler, A Pugliese, and V S Subrahmanian. 2010. COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In *Advances in Social Networks Analysis and Mining (ASONAM)*. 248–255.
[6] O. Curé, H. Naacke, M. A. Baazizi, and B. Amann. 2015. 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)*. 16–31.
[7] Orri Erling and Ivan Mikhailov. 2008. Towards Web Scale RDF. In *4th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*.
[8] Luis Galarraga, Katja Hose, and Ralf Schenkel. 2012. Partout: A Distributed Engine for Efficient RDF Processing. *CoRR* abs/1212.5 (2012).
[9] O. Görlitz, M. Thimm, and S. Staab. 2012. Splodge: Systematic generation of sparql benchmark queries for linked open data. *The Semantic Web–ISWC 2012* (2012), 116–132.
[10] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD*. 289–300.
[11] Claudio Gutierrez, Carlos Hurtado, and Alberto O Mendelzon. 2004. Foundations of Semantic Web Databases. In *PODS*. ACM, 95–106.
[12] Andreas Harth and Stefan Decker. 2005. Optimized Index Structures for Querying RDF from the Web. In *Proc. of LA-WEB '05*. IEEE, 71—-.
[13] A. Harth, J. Umbrich, A. Hogan, and S. Decker. 2007. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC-2007*. Vol. 4825. 211–224.
[14] K Hose and R Schenkel. 2013. WARP: Workload-aware replication and partitioning for RDF. In *Data Engineering Workshops (ICDEW)*. 1–6.
[15] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4, 11 (2011), 1123–1134.
[16] Daniel Janke, Steffen Staab, and Mathias Thimm. 2016. *Impact Analysis of Data Placement Strategies on Query Efforts in Distributed RDF Stores*. Technical Report. Institute for WeST. http://west.uni-koblenz.de/sites/default/files/research/publications/janke2016iao_technicalreport.pdf
[17] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. The Performance of MapReduce: An In-depth Study. *PVLDB* 3, 1 (2010), 472–483.
[18] Tobias Käfer and Andreas Harth. 2014. Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2014/. (2014).
[19] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.
[20] Kisung Lee and Ling Liu. 2013. 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, 46:1—-46:12.
[21] Kisung Lee and Ling Liu. 2013. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB* 6, 14 (Sept. 2013), 1894–1905.
[22] J. McMurry, S. Jupp, J. Malone, T. Burdett, A.y Jenkinson, H. Parkinson, M. Davies, M. Brandizi, and et al. 2015. Report on the scalability of semantic web integration in BioMedBridges. http://dx.doi.org/10.5281/zenodo.14071. (2015).
[23] R. Mutharaju, S. Sakr, A. Sala, and P. Hitzler. 2013. D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine. In *ISWC (Posters & Demos)'13*. 261–264.
[24] Peter Norvig. 2016. 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, 1–1.
[25] A. Owens, A. Seaborne, N. Gibbins, and Mc schraefel. 2008. Clustered TDB: A Clustered Triple Store for Jena. (Nov. 2008). http://eprints.soton.ac.uk/266974/
[26] Anthony Potter, Boris Motik, and Ian Horrocks. 2014. Querying Distributed RDF Graphs: The Effects of Partitioning. In *Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014)*. 29–44.
[27] Eric Prud'hommeaux, Steve Harris, and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. W3C Recommendation. W3C. http://www.w3.org/TR/sparql11-query/
[28] María-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. 2010. *Efficiently Joining Group Patterns in SPARQL Queries*. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–242.
[29] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. 2014. SemStore: A Semantic-Preserving Distributed RDF Triple Store. In *CIKM-2014*.
[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (Feb. 2013), 265–276.
[31] X. Zhang, L. Chen, Y. Tong, and M. Wang. 2013. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE-2013*. 565–576.