

Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores

Daniel Janke¹, Steffen Staab^{1,2}, and Matthias Thimm¹

¹ Institute for Web Science and Technologies
Universität Koblenz-Landau, Germany
{dani.jank, staab, thimm}@uni-koblenz.de
<http://west.uni-koblenz.de/>

² Web and Internet Science Group
University of Southampton, UK
s.r.staab@soton.ac.uk
<http://wais.ecs.soton.ac.uk/>

Abstract. In the last years, scalable RDF stores in the cloud have been developed increasing the complexity of RDF stores running on a single computer. In order to gain a deeper understanding how, e.g., the data placement or the distributed query execution strategies affect the performance, we have developed the modular glass box profiling system Koral. With its help, it is possible to test the behaviour of already existing or newly created strategies tackling the challenges caused by the distribution in a realistic distributed RDF store. Thereby, the design goal of Koral is that only the evaluated component needs to be exchanged and the adaptation of other components is aimed to be minimal. The wide variety of measurements allow for an in-depth investigation of the performance. With Koral we analyse the impact of the three most commonly used data placement strategies and found out that balancing query workload reduces the query execution time more than reducing the data transfer.

Keywords: distributed RDF store, glass box, profiling system

1 Introduction

In the last years, several scalable RDF stores in the cloud were developed, in which graph data is distributed over compute and storage nodes for scaling efforts of query processing and memory needs. This distribution over several compute and storage nodes introduces a higher degree of complexity. In contrast to centralized RDF stores, distributed RDF stores need strategies for data placement over compute and storage nodes, for distributed query processing and for handling failures of compute or storage nodes. Several approaches aiming to improve these aspects of distributed RDF stores were developed in the recent years. This includes new graph cover strategies like [19, 10] and new distributed query processing approaches like [23, 25].

In order to improve the current state-of-the-art, the strength and weaknesses of the already existing techniques as well as their impacts on the individual components of a distributed RDF store need to be identified. Therefore, glass box profiling systems are required that (i) profile the performance of a component in a distributed RDF store,

(ii) allow for a fair comparison of alternative implementations of a single component and (iii) provide measurements for in-depth analyses of the performance. Especially, the second ability is important since comparing the performance of alternative implementations helps to identify their weaknesses and thus, indicate directions for future improvements.

Evaluation platforms like Granula [20] allow for analysing the performance of large-scale graph processing systems. Thereby, they provide insights on the performance of the individual components used by the tested systems. Their drawback is that, e.g. alternative data placement strategies can only be compared by comparing systems that use different data placement strategies. Since only the differences between complete systems can be profiled, we call platforms like Granula black box evaluation platforms. These black box evaluation platforms can hardly answer the question of whether an observed difference in the network traffic is caused by, e.g. the new data placement strategy, a better query execution or optimization technique. The identification of the actual cause requires systems that are identical in all but the examined component.

Due to the absence of glass box profiling systems in which alternative implementations of the same component can be profiled within one system, [5, 11] suggest the usage of distributed batch processing systems like Apache Hadoop [1] and Apache Spark [2] to evaluate individual components of a distributed RDF store. These systems use distributed file systems for the data exchanges between individual compute nodes leading to a slower data exchange than systems using direct peer-to-peer communication [14]. Thus, approaches reducing the network traffic showed a better performance.

To enable fairer in-depth performance analyses of alternative implementations of individual components, our first contribution is the open source glass box profiling system Koral [3]. It is a modularized distributed RDF store in which the inter-dependencies between its components are reduced to an extent that each component can be exchanged with alternative implementations. Together with the wide variety of provided metrics, Koral allows for in-depth performance analyses of approaches tackling the challenges of distributed RDF stores.

Our second contribution is a case study in which we use Koral to examine the effect of frequently used data placement strategies on the query performance. We could observe that a data placement strategy that reduces the data transfer during query processing performed worse than data placement strategies that balanced the query workload equally among all compute nodes. A more detailed and extensive evaluation of graph cover strategies and their effect on the query execution when scaling the number of compute nodes can be found in [12].

In short, the contributions of this paper are:

1. The open source glass box profiling system Koral that (a) profiles the performance of different variants of the same component and (b) provides metrics for an in-depth investigation of the observed behaviour (Sec. 3).
2. A profiling of frequently used data placement strategies using Koral indicating that a more balanced query workload might have a higher impact on the query performance than a reduction of the data transfer (Sec. 4).

2 Formalisation of Challenges

Distributed RDF stores have several challenges. In the context of this paper, we will focus on the challenges of the data placement and the distributed query processing. Their formalization is given in the following two sections.

2.1 Formalisation of Data Placement

To formalize the Data Placement challenge, we define RDF graphs like in [8]. Assume a signature $\sigma = (I, B, L)$, where I , B and L are pairwise disjoint infinite sets of IRIs, blank nodes and literals, respectively. The union of these sets is abbreviated as IBL .

Definition 1. The set of all possible RDF triples T for signature σ is defined by $T = (I \cup B) \times I \times IBL$. An RDF graph G or simply graph is defined as $G \subseteq T$.

$(s, p, o) \in T$ is also called a triple with *subject* s , *property* p and *object* o . To simplify later definitions, the functions $\text{subj}(t)$, $\text{obj}(t)$ and $\text{prop}(t)$ return the subject, object or property of triple t , respectively. Likewise, we use $\text{subj}(T)$, $\text{obj}(T)$ and $\text{prop}(T)$ 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 .

Definition 2. Let G denote an RDF graph. Then a graph cover is a function $\text{cover} : G \rightarrow 2^C$, 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

$$\begin{aligned} \text{chunk}_{\text{cover}} : C &\rightarrow 2^G \\ \text{chunk}_{\text{cover}}(c) &:= \{t \mid c \in \text{cover}(t)\} \end{aligned}$$

2.2 Formalisation of Distributed Query Execution Strategy

The challenge of the distributed query execution is to find a strategy to execute a query on several compute nodes each of them storing a different graph chunk. The result of the distributed query execution should be the same as when executed on an RDF store running on a single compute node and storing the complete graph. To formalize the distributed query execution, we define a SPARQL core as done in [24], [21] and [4]. 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 4. A basic graph pattern (BGP) is a

1. triple pattern, i.e. an element of the set $TP = (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$
2. a conjunction $B_1.B_2$ of two BGPs B_1 and B_2 .

Definition 5. A SELECT query is defined as $\text{SELECT } W \text{ WHERE } \{B\}$ with $W \subseteq V$ and B a BGP.

Before the semantics of a SPARQL query can be defined, some additional definitions are required. In the following \mathcal{Q} represents the set of all SPARQL queries and the partial function $\mu : V \rightarrow IBL$ represents a variable binding. The abbreviated notation $\mu(t)$ with $t \in TP$ means that the variables in t are substituted according to μ .

Definition 6. Two variable bindings μ_i and μ_j are compatible, denoted by $\mu_i \sim \mu_j$, if $\forall ?x \in \text{dom}(\mu_i) \cap \text{dom}(\mu_j) : \mu_i(?x) = \mu_j(?x)$.³

Definition 7. The join of two sets of variable bindings Ω_1 and Ω_2 is defined as $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$. The variables contained in $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ are called join variables.

[21] and [4] define the semantics of a SPARQL query as follows:

Definition 8. The evaluation of a SPARQL query Q over an RDF Graph G , denoted by $\llbracket Q \rrbracket_G$, is defined recursively as follows, with $\text{var}(tp)$ returning all variables occurring in triple pattern tp :

1. If $tp \in TP$ then $\llbracket tp \rrbracket_G = \{\mu \mid \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in G\}$.
2. If B_1 and B_2 are BGPs, then $\llbracket B_1.B_2 \rrbracket_G = \llbracket B_1 \rrbracket_G \bowtie \llbracket B_2 \rrbracket_G$.
3. If $W \subseteq V$ and B is a BGP, then $\llbracket \text{SELECT } W \text{ WHERE } \{B\} \rrbracket_G = \text{project}(W, \llbracket B \rrbracket_G) = \{\mu|_W \mid \mu \in \llbracket B \rrbracket_G\}$.⁴

Definition 9. 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 $\llbracket Q \rrbracket_{\text{cover}}$, is defined as $\llbracket Q \rrbracket_{\text{cover}} := \bigcup_{c \in C} \llbracket Q \rrbracket_{\text{cover}}^c$. Thereby, $\llbracket Q \rrbracket_{\text{cover}}^c$ is the set of all results produced on compute node c during the distributed query evaluation⁵.

In order to be equivalent to the centralized query evaluation, every distributed execution mechanism has to be semantically correct.

Theorem 1. The centralized evaluation of query Q over graph G $\llbracket Q \rrbracket_G$ produces exactly the same results as its distributed evaluation, i.e.

$$\llbracket Q \rrbracket_{\text{cover}} = \llbracket Q \rrbracket_G \ .$$

3 Glass Box Profiling System Korall

In order to gain deeper insights in the strength and weaknesses of individual approaches tackling the challenges of distributed RDF stores, glass box profiling systems are required. These systems should be:

realistic so that the profiled performance is similar to a realistic distributed RDF store.

modular to test varying approaches tackling the same distributed RDF store challenge.

investigative by performing measurements that allow for an in-depth analysis of the performance of the examined components.

³ $\text{dom}(\mu)$ refers to the set of variables of this binding.

⁴ $\mu|_W$ means that the domain of μ is restricted to the variables in W .

⁵ The formal definition of $\llbracket Q \rrbracket_{\text{cover}}^c$ can be found in [12].

The glass box profiling system Korall [3] is *realistic* since it is designed as a distributed RDF store. Its architecture is presented in Sec. 3.1.

We achieved *modularity* by separating the core functionalities into individual components whose functionality used by other components are declared by interfaces. Furthermore, we reduced the inter-dependencies between components to an extent that each component can be exchanged with alternative implementations. With its current state of modularity, components tackling the following challenges of distributed RDF stores can be profiled: the data placement strategy, the centralized indexing of all graph chunks, the distributed query execution strategy including query optimization, the handling of compute node failures as well as the efficient data transfer between compute nodes. Due to space limitations we will only present the exchangeability of the graph cover strategy (Sec. 3.2) and of the distributed query execution mechanism (Sec. 3.3).

In order to be *investigative*, Korall provides a wide variety of measures as described in Sec. 3.4. Beside time-based measures, Korall also provides several time-independent measures to investigate the performance without influences caused by the experimental setting. The limitations of Korall are discussed in Sec. 3.5.

3.1 Architecture Overview

Korall consists of one master node and several slaves as shown in Fig. 1. In general, the master creates the graph cover, assigns chunks to slaves and coordinates the query execution. The slaves are responsible for the query processing. The network managers maintain peer-to-peer network connections and manage the network communication.

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. Since some graph cover strategies might require, e.g. subjects, as plain text, the dictionary encoder encodes only those parts of the triples that are not required in their textual representation (see Sec. 3.2). The encoded graph is then used by the graph cover creator to create the requested graph chunks. If unencoded

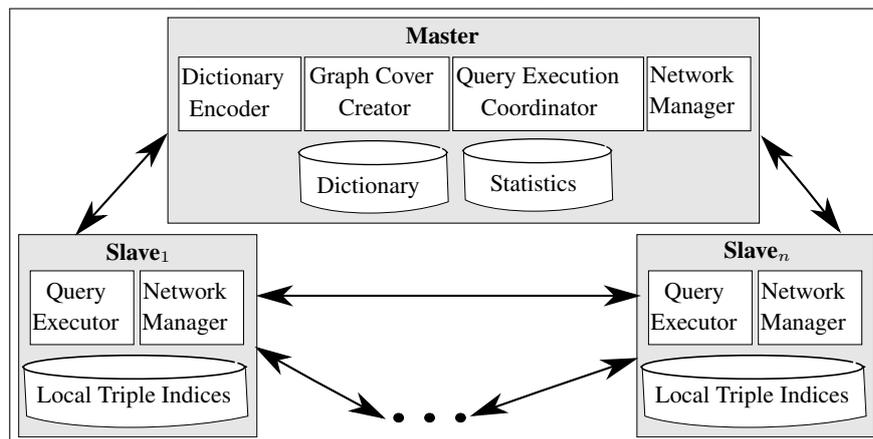


Fig. 1: Architecture of Korall.

triple elements exist, they are encoded after the graph cover creation in order to reduce the size of the graph chunks.

In order to perform, e.g. cost estimations required for query optimization or a load balancing during query execution, statistical information about the content of each graph chunk may be required. Therefore, the frequency of the different resources in the different chunks is counted and stored in a statistics database.

Some distributed query execution strategies might require some preprocessing steps of the input data like appending additional information to the encoded resource ids. Therefore, the master iterates all graph chunks a last time before they are sent to the slaves. The slaves create local index structures (SPO, OSP, and POS indices as described in [27]). 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.

At run-time, a query execution coordinator is instantiated for each received query. After the initial parsing step including the encoding of constants, the query execution trees for the slaves are created and sent to the corresponding slave.

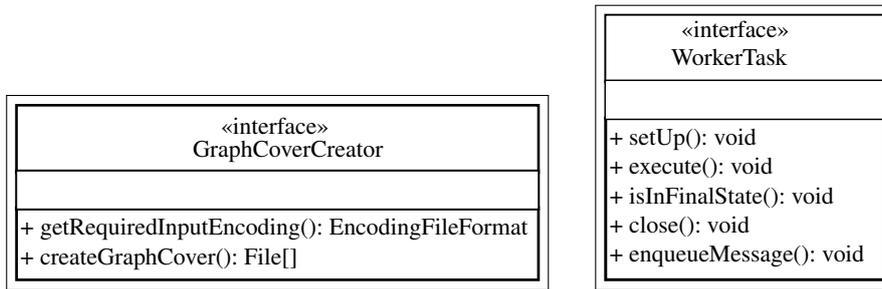
Each slave executes the query execution tree assigned to him. The match operations use the local triple indices to find matches for the corresponding triple pattern. The resulting variable bindings are transferred to the succeeding operation on the same or any other slave. In order to make better use of the network bandwidth, several intermediate results are bundled together and sent to the receiving slave within one package. The final query results are sent to the query coordinator. The coordinator decodes the ids using the dictionary and sends the decoded variable bindings to the sender of the query.

3.2 Exchangeability of Graph Cover Strategies

To allow for testing new graph cover strategies, all methods used by Koral are declared in the interface of the `GraphCoverCreator` component (Fig. 2a). During the initial dictionary encoding phase all elements of the RDF tuple that are not required in their textual representation for the graph cover creation are encoded. With the method `getRequiredInputEncoding()` each graph cover strategy can define, whether it requires the subject, property or object in its textual representation. These elements will be automatically encoded after the graph cover is created.

The actual graph cover creation is performed by `createGraphCover()`. It receives the initially encoded input RDF file, a working directory where the graph chunks should be created and the number of graph chunks to be created as input. For the sake of brevity, these input parameters are omitted in Fig. 2a. The created graph chunks are finally returned by this method and processed in the succeeding graph loading steps.

In order to avoid restrictions on the graph cover strategies that can be used in Koral, a distributed query execution strategy is required that works with arbitrary graph covers including triples assigned to more than one compute node. This graph cover-independent query execution strategy is the default implementation in Koral. This strategy is explained in the following section.



(a) The interface of the graph cover creator component. (b) The interface for query operations.

Fig. 2: The interfaces for graph cover strategies and query operations.

The existing implementations of Koral comprises three graph cover strategies:

1. The *hash cover* [9] 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.
2. The *hierarchical hash cover* [18] is inspired by the observations that IRIs have a path hierarchy and IRIs with a common hierarchy prefix are often queried together. Therefore, this cover creates a hash cover only on IRI prefixes.
3. The *minimal edge-cut cover* is a vertex-centred partitioning which tries to solve the k-way graph partitioning problem as described in [16]. It aims at minimizing the number of edges between vertices of different partitions under the condition that each partition contains approximately the same number of vertices.

3.3 Exchangeability of Distributed Query Execution Strategies

Distributed query execution strategies may vary in (i) additional information added to the graph chunks, (ii) the way query execution trees are created for the individual slaves, and (iii) the actual implementation of the query operations executed on the slaves.

In order to encode additional information into the graph chunks, the loading procedure of the graph includes a step for final adjustments of the created graph chunks. In order to implement such final adjustments, the method `performFinalAdjustments()` in class `GraphLoaderTask` needs to be implemented. It receives the graph chunks as input and returns the adjusted graph chunks.

After the graph is loaded, the master accepts queries. For each query, a new query execution coordinator is started. This coordinator parses the query and creates the query execution trees that are sent to the slaves. The query execution trees sent to the individual slaves can be adjusted by implementing `executePreStartStep()` in `QueryExecutionCoordinator`. Since statistical information about the occurrences of resources in the individual graph chunks might be required, the method provides access to the statistics database.

New query operations can be created by implementing the `WorkerTask` interface (Fig. 2b). Each slave has a query executor component that runs for each available CPU core one worker thread. The query executor registers query operations at the worker threads based on their current workload. After the registration the method `setUp()` is

called. Thereby, the operation gets access to the network manager, to send messages to other operations, and to the local triple indices. After initialization, the worker thread circularly calls `execute()` of all query operations assigned to him. During these method calls each operation performs its work and processes received messages. Incoming messages are announced via `enqueueMessage()`. When an operation is finished, the worker thread unregisters it and calls `close()`.

The existing implementation of Koral extends the state-of-the-art asynchronous execution mechanism realised in TriAD [7]. This extension makes it independent of the used graph cover strategy. Its formal definition and the proofs of soundness and completeness are given in [12].

In order to reduce the number of transferred intermediate results, each resource is uniquely assigned to a slave that is responsible for joining it during the query processing. This assignment of a resource is based on the frequency with which it occurs in the different graph chunks. Therefore, when the statistical data have been completely collected and the loading process iterates over all graph chunks a last time, 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.

When the master receives a query, it creates a query execution coordinator. This coordinator parses the query and creates the query execution tree. This tree is submitted to all slaves. During the query execution on the slaves, each operation transfers its resulting variable bindings to the succeeding join operation on the slave responsible for the join of the resource. Whenever the join operation receives a variable binding, it is joined with the cached variable bindings. The join results are directly sent to the succeeding operation. When all child operations of a query operation o in the query execution tree are finished and no further input needs to be processed, it sends a finish notification 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. The root operation in the query execution tree sends its results to the query coordinator. The coordinator forwards them to the sender of the query after they have been decoded.

3.4 Evaluation Measures

In order to gain deeper insights about the effect of a graph cover strategy or a distributed query execution strategy, Koral provides a wide variety of measurements. Beside different run time measurements, Koral measures also many time-independent measurements like the storage imbalance, the workload and the network usage. With the help of these measurements, meaningful metrics can be defined that allow for comparing different strategies.

Loading Time. Loading a dataset 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 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. Perform query strategy-dependent adjustments of graph chunks.
6. Transfer of data chunks to compute nodes.
7. Indexing of data chunks at local compute nodes.

Given a dataset and a graph cover strategy, the overall load time comprises these 7 steps, but also the run times of the individual steps are of interest. For instance, the graph cover computation time can be used to compare different graph cover strategies, whereas the time required for the query strategy-dependent adjustments of the graph chunks might be one important factor for the comparison of distributed query execution strategies.

Storage imbalance. Scaling the cloud for handling growing memory needs may be jeopardized by graph cover strategies avoiding data transfer. They might generate a skewed distribution delegating expensive tasks on few compute nodes. Therefore, Koral counts the number of triples stored in each graph chunk. These measurements are used to evaluate the quality of the storage distribution resulting from a graph cover strategy.

Querying Time. For the overall query performance, different performance characteristics of an RDF store may be desirable. While the time to deliver 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 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 definition. With the t_i^q values, the frequency in which the individual query results are returned can be analysed. With the help of K^q the query time to completion can be computed.

Definition 10. *Overall query performance is evaluated by the query time to completion $exTime := t_K - t_0$.*

To compare the performance of different implementations of the same query operator or to find bottlenecks in a query execution strategy, a more detailed measurement of the individual query operation run times may be required. Therefore, Koral measures for each operation on each slave how long this operation idles and how long it works.

Network Usage. Time-based measurements such as $exTime$ depend on the exact configuration of the system such as network bandwidth and latency. In a distributed system, the transfer of data via the network is a time-consuming operation.

In order to measure the volume of data transferred between compute nodes, we measure for each query operation the number of variable bindings transferred to each slave as well as the number of bound variables.

Beside the volume of transferred data also the number of sent packages are part of the network usage, since in a network with high latency this number might have a strong effect on the query run time. The network manager of Koral collects a bunch of variable bindings before it sends them as a single package to another slave. Thereby it counts the number of packages transferred to each other slave.

Definition 11. *For a given cover and a given query execution tree q , we define the number of transferred packages $P := \sum_{c \in C} P_c$, where P_c is the number of packages 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 network usage and the workload imbalance.

Query Workload. An interesting question to answer would be, 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 the number of comparisons performed by each join operation on each slave and computing the Gini coefficient on the resulting distribution of join comparisons over different compute nodes.

Definition 12. For a cover and a query execution tree q , workload imbalance W is the Gini coefficient:

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

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

3.5 Limitations

During the design of Koral we tried to design the components independent of other components. Since a complete separation of concerns for all invented or not-yet-invented methods is not possible, some of the components provided by Koral might punish some methods with a poorer performance. Nevertheless, in this case the modularity of Koral allows for exchanging the punishing component by an improved implementation and using this one for a fairer evaluation.

Furthermore, the current design of Koral does not foresee the evaluation of transactions. In order to support transactions, the architecture might be extended by additional components and already existing components need to be adapted.

4 Case Study Evaluation

In a graph cover benchmark 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. By using Koral for our benchmark⁶, we can compare the performance of the different graph cover strategies without varying any other part of the distributed RDF store. Therefore, most factors influencing the performance comparison negatively could be avoided. Other influencing factors might be the dataset and

⁶ A more detailed description of this benchmark and its results can be found in [13].

the queries used for the benchmark. To reduce the impact of these factors, we use a real-world dataset and a diverse set of queries as described in Sec. 4.1. The results of our evaluation are shown 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 and (ii) the set of different query-dataset combinations.

Compared Graph Cover Strategies. During the evaluation, a hash cover, a hierarchical hash cover and a minimal edge-cut cover are compared. We use the implementations provided by Koral.

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) [15]. 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.

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 use SPLODGE [6] to generate queries with the following characteristics for the one billion triple subset:

Number of triple patterns: 2 and 8 triple patterns.

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

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

Number of data sources: 1 and 3 source data sets.

Execution Strategy. We downloaded the BTC2014 dataset, removed all syntactically incorrect triples and created the one billion triple dataset. The resulting dataset is used by SPLODGE [6], 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. The effect of outliers caused by, e.g. garbage collection is prevented by ignoring the best and the worst execution time computing the arithmetic mean of the remaining values as *exTime*.

Computer and Software Environment. Koral is executed on 11 VMs. The master has 4 cores and 64 GB RAM and the 10 slaves have 1 core and 2 GB RAM each. Since the Koral master VM needs to store the complete dataset, it has a 1 TB hard disk. The slaves have 300 GB hard disks. The physical computers on which the VMs run are connected via a 1 Gigabit Ethernet network.

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

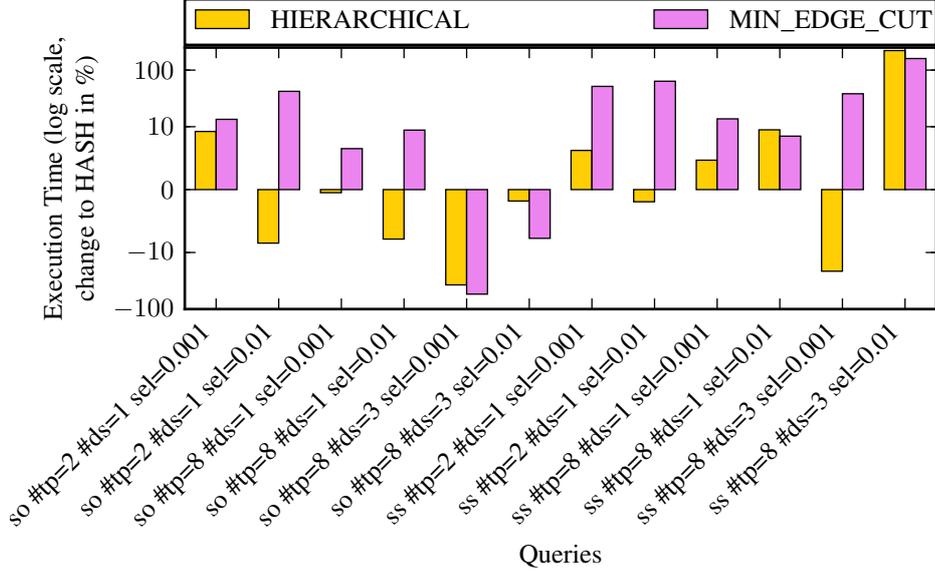


Fig. 3: *exTime* of all queries relative to the hash cover.

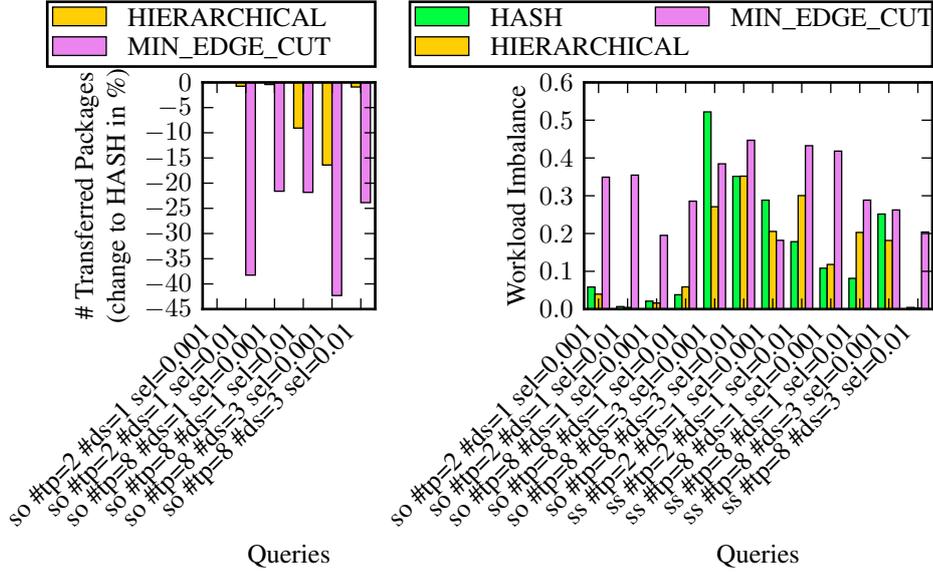
4.2 Results

As the possible configurations of independent variables (configuration settings) and dependent variables (evaluation measures) is staggering, we focus on analysis results by depicting (i) overall query performance, (ii) network usage and (iii) query workload. 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.

Overall Query Performance. We measure the overall query performance in terms of the execution times *exTime*. Fig. 3 shows the *exTime* of all queries. Due to the huge differences in the execution times of the varying queries, the execution times are shown relative to the *exTime* required for the hash cover. The figure shows, that the minimal edge-cut cover causes the longest query execution times in most cases. When comparing the hash cover with the hierarchical hash cover, none of them is faster in general.

Network Usage. 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. For query `so #tp=2 #ds=1 sel=0.001` every graph cover strategy produces almost the same number of transferred packages. For all other path-shaped queries the minimal edge-cut cover reduces the data transfer by 20%-42% (see Fig. 4a). The number of transferred packages of the hash and the hierarchical hash cover is almost the same for all but two queries.

Query Workload. We investigate the query workload by comparing the workload imbalance W of all queries. Fig. 4b shows that the minimal edge-cut cover has the highest



(a) The number of transferred packages P relative to the hash cover.

(b) The workload imbalance W .

Fig. 4: The number of transferred packages and the workload imbalance.

workload imbalance of all graph covers. This is caused by two small graph chunks that have a much lower workload than the other graph chunks. Whereas, one huge graph chunk with 160M triples does not produce a higher workload than the other chunks.

W of queries so #tp=8 #ds=1 sel=0.001 and ss #tp=8 #ds=3 sel=0.01 are for all graph covers low, since these queries use 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 6 Koral slaves.

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 [18, 22, 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 data transfer by 20% to 42% in comparison to hash-based strategies (see Fig. 4a), due to a more imbalanced workload (see Fig. 4b), the query execution time of minimal edge-cut is effectively slower (see Fig. 3).

Our investigation suggests that in our setting the minimal edge-cut cover does not perform better over all (see Fig. 3). 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

Without profiling platforms, distributed RDF stores are usually profiled and compared as black boxes as done in [23, 26, 29, 30]. Black box evaluations do not allow to improve the current state-of-the-art since they cannot identify bottlenecks within a system. To identify bottlenecks, glass box profiling is required in which the performance of individual components is profiled.

Black box valuation platforms like Granula [20] help to perform in-depth analyses of large-scale graph processing systems and their components. These evaluation platforms are useful to identify components that are the bottlenecks of these systems. The usually strong dependencies between the individual components of distributed RDF store limit the search space for more performant components to similar approaches with relatively small modifications. The evaluation of fundamentally different approaches would be feasible by evaluating different systems usually varying in several aspects.

In order to profile fundamentally different approaches tackling the same challenge of distributed RDF store, [5, 11, 17, 31] propose the usage of distributed batch processing platforms like Apache Hadoop [1] or Apache Spark [2]. The drawback of using these systems is that they punish data transfer by the potentially huge overhead of possibly several Hadoop jobs and the usage of distributed file systems for the data transfer between compute nodes (see [14]).

To the best of our knowledge, Koral is the only glass box profiling system that (i) profiles the performance of a component in a distributed RDF store, (ii) allows for a fair comparison of alternative implementations of a single component due to its modularization and (iii) provides measurements for in-depth analyses of the performance.

6 Conclusion

We have presented our versatile open source glass box profiling system Koral. It is a modularized distributed RDF store in which the inter-dependencies between its components are reduced to an extent so that each component can be exchanged with alternative implementations. Thus, it allows for profiling novel approaches tackling the challenges introduced by the distribution and compare them with already existing strategies. We demonstrated the advantages of such a profiling platform with a case study evaluation of different graph cover strategies revealing 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 huge variety of measurements provided by Koral, 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. The common believe was raised by evaluations like [5, 11] in which the authors reduced the

implementation effort by simulating distributed RDF stores with batch processing systems that delay network traffic in contrast to direct peer-to-peer communication. With our novel open source glass box profiling system Koral, freely available on the Web [3], further investigation of distributed RDF data management challenges with only little implementation efforts are possible.

The alternative implementations of Koral's components used in future investigations can be provided to the public. This growing set of implementations will support the community by simplifying the comparison with state-of-the-art approaches or by identifying novel combinations of components producing better overall performances.

References

1. Apache hadoop. <https://hadoop.apache.org/>, accessed: 2017-07-12
2. Apache spark. <https://spark.apache.org/>, accessed: 2017-07-12
3. Koral. <https://github.com/Institute-Web-Science-and-Technologies/koral>, accessed: 2017-07-12
4. Arenas, M., Pérez, J.: Federation and Navigation in SPARQL 1.1. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web. Semantic Technologies for Advanced Query Answering, Lecture Notes in Computer Science, vol. 7487, pp. 78–111. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33158-9_3
5. Curé, O., Naacke, H., Baazizi, M.A., Amann, B.: On the evaluation of RDF distribution algorithms implemented over apache spark. In: Proc. of the 11th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (at ISWC-2015). pp. 16–31 (2015)
6. Görlitz, O., Thimm, M., Staab, S.: Splodge: Systematic generation of sparql benchmark queries for linked open data. The Semantic Web–ISWC 2012 pp. 116–132 (2012)
7. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: SIGMOD. pp. 289–300 (2014)
8. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of Semantic Web Databases. In: PODS. pp. 95–106. ACM (2004)
9. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: Proc. of LA-WEB '05. pp. 71—. IEEE (2005)
10. Hose, K., Schenkel, R.: WARP: Workload-aware replication and partitioning for RDF. In: Data Engineering Workshops (ICDEW). pp. 1–6 (Apr 2013)
11. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
12. Janke, D., Staab, S., Thimm, M.: Impact analysis of data placement strategies on query efforts in distributed rdf stores. Tech. rep., Institute for WeST (2016), http://west.uni-koblenz.de/sites/default/files/research/publications/janke2016iao_technicalreport.pdf
13. Janke, D., Staab, S., Thimm, M.: On data placement strategies in distributed rdf stores. In: Proceedings of The International Workshop on Semantic Big Data. pp. 1:1–1:6. SBD '17, ACM, New York, NY, USA (2017), <http://doi.acm.org/10.1145/3066911.3066915>
14. Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The performance of mapreduce: An in-depth study. PVLDB 3(1), 472–483 (2010), <http://www.comp.nus.edu.sg/~vladb2010/proceedings/files/papers/E03.pdf>
15. Käfer, T., Harth, A.: Billion Triples Challenge data set. Downloaded from <http://km.aifb.kit.edu/projects/btc-2014/> (2014)

16. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20(1), 359–392 (1998)
17. Lee, K., Liu, L.: Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud. In: *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*. pp. 46:1—46:12. ACM (2013)
18. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB* 6(14), 1894–1905 (Sep 2013)
19. Lee, K., Liu, L., Tang, Y., Zhang, Q., Zhou, Y.: Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud. In: *IEEE CLOUD '13*. pp. 327–334 (2013)
20. Ngai, W.L., Hegeman, T., Heldens, S., Iosup, A.: Granula: Toward Fine-grained Performance Analysis of Large-scale Graph Processing Platforms. In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. pp. 8:1—8:6. *GRADES'17*, ACM, New York, NY, USA (2017), <http://doi.acm.org/10.1145/3078447.3078455>
22. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* 34(3), 16:1—16:45 (Sep 2009), <http://doi.acm.org/10.1145/1567274.1567278>
22. Potter, A., Motik, B., Horrocks, I.: Querying Distributed RDF Graphs: The Effects of Partitioning. In: *Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014)*. pp. 29–44 (2014)
23. Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Distributed RDF Query Answering with Dynamic Data Exchange, pp. 480–497. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-46523-4_{_}29
24. Prud'hommeaux, E., Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3c recommendation, W3C (2013), <http://www.w3.org/TR/sparql11-query/>
25. Saleem, M., Ngonga Ngomo, A.C.: HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation, pp. 176–191. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-07443-6_{_}13
26. Saleem, M., Ngonga Ngomo, A.C., Xavier Parreira, J., Deus, H.F., Hauswirth, M.: DAW: Duplicate-Aware Federated Query Processing over the Web of Data, pp. 574–590. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-41335-3_{_}36
27. Wood, D., Gearon, P., Adams, T.: Kowari: A platform for semantic web storage and analysis. In: *In XTech 2005 Conference*. pp. 05–0402 (2005)
28. Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SemStore: A Semantic-Preserving Distributed RDF Triple Store. In: *CIKM-2014* (2014)
29. Wylot, M., Cudré-Mauroux, P.: Diplocloud: Efficient and scalable management of rdf data in the cloud. *IEEE Transactions on Knowledge and Data Engineering* 28(3), 659–674 (2016)
30. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6(4), 265–276 (Feb 2013)
31. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In: *ICDE-2013*. pp. 565–576 (Apr 2013)