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

Daniel Janke[1], Steffen Staab[1,2], and Matthias Thimm[1]

[1] Institute for Web Science and Technologies
Universität Koblenz-Landau, Germany
{danijank,staab,thimm}@uni-koblenz.de
[2] Web and Internet Science Group
University of Southampton, UK
s.r.staab@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.

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.

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 performance analyses. Especially, the second ability is important since comparing performances of alternative implementations helps to identify weaknesses and thus, indicate directions for future improvements.

Our contribution is the open source glass box profiling system Koral [1]. It is a distributed RDF store that is such modularized that the inter-dependencies between its components are reduced to an extent that each component can be exchanged with alternative implementations. This allows for comparing the performance of alternative implementations of the same component. Together with the wide variety of provided metrics, Koral allows for in-depth performance analyses of approaches tackling the challenges of distributed RDF stores. A detailed and extensive evaluation of different graph cover strategies with the help of Koral can be found in [4].

2 Glass Box Profiling System Koral

Koral consists of one master node and several slaves as shown in Fig. 1. The network managers maintain peer-to-peer network connections and manage the 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 (i.e. strategies to assign triples to compute nodes) 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. 2.1). The encoded graph is then used by the graph cover creator to create the requested graph chunks (i.e. the sets of all triples assigned to the individual compute nodes). If unencoded 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, statistical information about the content of each graph chunk is collected 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 [7]). 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.

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.

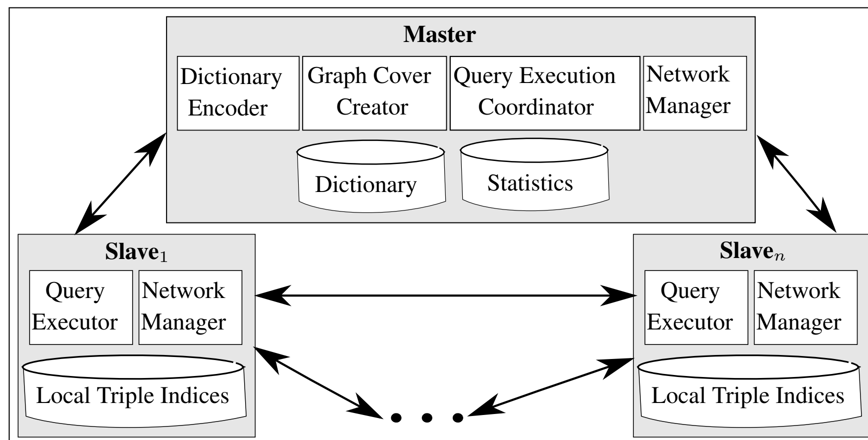

Fig. 1. Architecture of Koral.

2.1 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 graph cover creator component. First of all, each graph cover strategy can define, which elements of a triple can be encoded during the initial dictionary encoding phase and which ones are required in their textual representation. The main functionality of each graph cover strategy is the creation of the graph cover out of this initially encoded input RDF graph. The created graph chunks are used during the succeeding 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. This graph cover-independent query execution strategy is the default implementation in Koral (see Sec. 2.2).

The existing implementations of Koral comprises three graph cover strategies:

1. The *hash cover* [3] assigns triples to chunks according to the hash value computed on their subjects.
2. The *hierarchical hash cover* [6] creates a hash cover only on common IRI prefixes.
3. The *minimal edge-cut cover* [5] 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.

2.2 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. For a newly received query, the query execution coordinator parses the query and creates the query execution trees that are sent to the slaves. These trees can be adjusted for each individual slave based on the query execution strategy.

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 query operations are initialized. Thereby, the operations get access to the network manager, to send messages to other operations, and to the local triple indices. After initialization, the worker thread circularly executes all query operations assigned to him. During the execution of a query operation it receives messages from child operations and send messages to its parent operation. When an operation is finished, the worker thread unregisters it and instructs the query operation to shut down.

The existing implementation of Koral realises an extension of the state-of-the-art asynchronous execution mechanism realised in TriAD [2]. This extension makes it independent of the used graph cover strategy. When the master receives a query, the query execution coordinator parses is and creates the query execution tree. The complete tree is submitted to all slaves and thereafter executed. To avoid (some) duplicate joins and corresponding data transfer, each join is uniquely assigned to the slave responsible for the join of the resource. The responsibility was determined during the graph loading.

3 Evaluation Measures

Loading Time. Koral measures the overall load time, but also the run times of the individual loading steps are of interest. For instance, the graph cover computation time can be used to compare different graph cover strategies.

Storage imbalance. Scaling the cloud for handling growing memory needs may be jeopardized by graph cover strategies reducing 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.

Querying Time. For the overall query performance, Koral measures the time until the complete result is delivered. This measurement is crucial, e.g., for statistical reports. In a fact-finding mission the fast return of a few top-*k* results may be more important. Hence, Koral provides the time interval between issuing a query and the times when the individual results are returned. To support performance comparisons of different query operator variants, Koral measures the idling and working time of each query operation.

Network Usage. Time-based measurements depend on the exact configuration of the system such as network bandwidth and latency. In order to measure the volume of transferred data, we measure the number of transferred variable bindings as well as the number of bound variables. Beside the volume Koral also measures the number of sent packages, since in a high-latency network it might have a strong effect on the run time.

Query Workload. An interesting question to answer would be, how many join comparisons might be executed by different compute nodes in parallel. To answer this question Koral measures the number of comparisons performed by each join operation on each slave. With these measures it can be identified how balanced the join comparisons are distributed over all compute nodes and how large the total computational effort is.

4 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 existing approaches.

References

1. Koral. https://github.com/Institute-Web-Science-and-Technologies/koral, accessed: 2017-07-12
2. 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)
3. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: Proc. of LA-WEB '05. pp. 71—-. IEEE (2005)
4. 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
5. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. 20(1), 359–392 (1998)
6. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB 6(14), 1894–1905 (Sep 2013)
7. Wood, D., Gearon, P., Adams, T.: Kowari: A platform for semantic web storage and analysis. In: In XTech 2005 Conference. pp. 05–0402 (2005)