The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)

At the 9th International Semantic Web Conference (ISWC2010), Shanghai, China, November, 2010

SSWS 2010 PC Co-chairs' Message

SSWS 2010 was the sixth instance in the sequence of successful Scalable Semantic Web Knowledge Base Systems workshops. This workshop focused on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages like OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the applications semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 12 submissions. Each paper was carefully evaluated by two or three workshop Program Committee members. Based on these reviews, we accepted eight papers for presentation. The topics of the selected papers span the areas of large scale data stores, optimized representation mechanisms, and query processing. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

October 2010

Achile Fokoue
Yuanbo Guo
Thorsten Liebig

Program Commitee

Achile Fokoue
IBM Watson Research Center, USA

Yuanbo Guo
Microsoft, USA

Jeff Heflin
Lehigh University, USA

Thorsten Liebig
Ulm University, Germany

Ian Horrocks
University of Oxford, UK

Pascal Hitzler
Wright State University, Ohio, USA

Kavitha Srinivas
IBM Watson Research Center, USA

Raúl García-Castro
Univ. Politecnica de Madrid, Spain

Aditya Kalyanpur
IBM Watson Research Center, USA

Oscar Corcho
University of Manchester, UK

Marko Luther
DoCoMo Eurolabs Munich, Germany

Andy Seaborne
Hewlett-Packard, UK

Volker Haarslev
Condordia University, Canada

Mariano Rodriguez
Free University of Bolzano, Italy

Mike Dean
BBN Technologies, USA

Additional Reviewers

Kejia Wu
Condordia University, Canada

Jinan El Hachem
Condordia University, Canada

Ming Zuo
Condordia University, Canada

Yingjie Li
Lehigh University, USA

Dezhao Song
Lehigh University, USA

Table of Contents

Configuring a Self-Organized Semantic Storage Service

H. Mühleisen, T. Walther, A. Augustin, M. Harasic & R. Tolksdorf

Freie Universität Berlin
Department of Computer Science – Networked Information Systems Group
Königin-Luise-Str. 24/26, D-14195 Berlin, Germany
{muehleis,twalther,aaugusti,harasic}@inf.fu-berlin.de, tolk@ag-nbi.de
http://digipolis.ag-nbi.de

Abstract. Scalability requirements for semantic stores lead to distributed hardware-independent solutions to handle and analyze massive amounts of semantic data. We use a different approach by imitating the behaviour of swarm individuals to achieve this scalability. We have implemented our concept of a Self-organized Semantic Storage Service (S4) and present preliminary evaluation results in order to investigate to what extent the performance of a distributed and swarm-based storage system is dependent on its configuration.

1 Introduction and Motivation

Most Semantic Web applications require a semantic store, a specialized database for semantic data storage and analysis. Data for such applications is becoming more and more available, which leads to increased performance and scalability requirements for semantic stores. While considerable amounts of semantic data have been successfully processed on a single computer, distributed hardware-independent solutions are necessary to handle and analyze the massive amount of semantic data expected to become available in the near future.

Distributed storage poses two main questions: Where should an arbitrary data item be stored, and how should a specific stored item be located and retrieved efficiently. Many systems use a central catalog server, others maintain overlay network structures to answer these questions. We propose a different approach, where no node in the storage network has any distinct functionality, and where the tasks of data distribution and retrieval are performed by autonomous processes imitating the behaviour of swarm individuals, which have been shown to accomplish astonishing tasks using strictly local knowledge, limited memory, a limited set of rules, and a simple yet scalable way of passing information to other individuals.

This concept has been researched and simulated before as we will show in the following section, in this paper we focus on the presentation of central configuration parameters controlling the behaviour of our swarm-based system. We present preliminary evaluation results in order to answer our research question:

To what extent is the performance of a swarm-based system dependent on single configuration parameters?

To answer the research question in a setting as realistic as possible, we have implemented central parts from our concept of a Self-organized Semantic Storage Service (S4) and deployed this implementation on our lab network consisting of 150 virtual machines. For every relevant configuration parameter identified, a number of test runs was performed using different settings for this parameter. Test runs were comprised of storing a fixed dataset the system, and then testing whether the stored data could be retrieved efficiently from an arbitrary node.

The remainder of this paper is structured as follows: First, in Section 2, we introduce our relevant previous work and then continue with describing various approaches in distributed semantic storage and analysis. Section 3 then describes the basic concepts for retrieval and storage of data within our swarm-based S4 system with special focus on the relevant configuration parameters. Using our implementation, Section 4 presents our preliminary evaluation results both for storage and retrieval of a LUBM test data set. Finally, Section 5 concludes this paper by discussing evaluation results and describing our next steps.

2 Previous and Related Work

The need for distributed storage solutions emerges from the inherent limitations present on every stand-alone computer system. For those distributed solutions, two general approaches can be followed: Centralized network structures rely on systems orchestrating storage operations between storage nodes, while decentralized structures make no conceptual distinction between nodes, thus eliminating the single point of failure.

For the distributed decentralized storage of semantic information, various concepts such as Edutella [13], RDFPeers [2], GridVine [3] or YARS [9] have been proposed. They make use of Peer-to-Peer (P2P) technology to create an overlay network to store and retrieve semantic information in a distributed way.

Apart from mere storage, reasoning is also a crucial requirement for a semantic storage system. However, the support for reasoning is limited at best in the proposed concepts. So far, distributed reasoning has been attempted using different distribution techniques: Urbani et al. employ MapReduce to achieve reasoning over a very large amount of semantic data [18], Oren et al. use distributed hash tables (DHTs) in their MaRVIN reasoning system [15], and Dentler et al. rely on swarm intelligence for scalable reasoning [4].

Swarm intelligence has been identified to be a powerful family of methods by Bonabeau et al. [1]. Different applications of this family were described by Mamei et al. [10]. Menezes and Tolksdorf applied swarm intelligence to a distributed tuple space built to implement the Linda coordination model [6]. They introduced basic concepts of ant colony algorithms that are suitable for tuple storage and retrieval [11]. Tolksdorf and Augustin then applied this idea to distributed RDF storage and used a syntax-based similarity metric to cluster syntactically similar resources on neighboring storage nodes. Their concept was evaluated using simulation runs [16]. A similarity metric based on semantic similarity measures and aimed at achieving the clustering of related concepts was introduced in [17]. Harasic et al. proposed a different similarity metric using a hash function and also contributed an implementation architecture and evaluation results from a first prototype [8].

3 Self-Organized Semantic Storage Service - Concepts

In this section, we introduce basic concepts and operations of our Self-Organized Semantic Storage System (S4). The S4 system is a distributed semantic storage system. Triples are stored on a number of nodes that have been added to a storage network by configuration. This network does not contain a single central component, all a node has to know in order to join the network is the address of an arbitrary member. Each node maintains a list of neighbor nodes it is connected to. This list is built by a simple bootstrap algorithm. The algorithm simply asks all known nodes for other nodes, and tries to connect to them. The connection is successful, if the involved nodes have not yet reached their configurable neighbor limit. The process is repeated until a node has reached a minimum amount of neighbors, which is typically set to half the neighbor limit. It is expected that a higher neighbor limit leads to a better system performance, as a higher connectivity in the network reduces the average hop count to find a single node, and thus improves overall response time.

S4 offers two basic storage and retrieval operations, which are exported over an API available on every node. For both operations, custom adaptions of ant colony algorithms are employed. These algorithms have been described in [12], but we will outline the basic ideas and introduce the relevant configuration parameters as well as their expected effects below:

3.1 Retrieval

The system is able to locate information by a single key using a method of foraging found in the behaviour of several species of ants. Searching is performed by following virtual pheromone trails left behind by previous operations.

These trails are located to each connection to another node, thus by checking all present pheromones on a single node, an operation is able to determine which node to visit next, where it can access the triples stored there. This process is repeated until either a match was found or the configured maximum hop count has been reached. If a match was found, the path taken is tracked back in order to spread pheromones for the current key to be used by subsequent operations.

Pheromones are volatile due to the dynamics of the stored triples. For example, if a triple is deleted the pheromones leading to it should also disappear. Since manual removal would require broadcast messages through the entire network, their intensity decays by a configurable percentage per time unit. The following behaviour is expected for different decay rate settings: A smaller decay rate allows pheromones outlive the duration of few operations, and also leads to more triples being found.

For every retrieval operation, only a subset of the potential matches is to be returned. This is due to the intended use of our system by applications and also the expected size of the storage network. If the amount of data stored is exceeding a certain amount, it may be not feasible to return *all* matching results. Instead the user is asked to specify time an result set limit for the operation. Retrieval operations are terminated if one of these limits is reached.

3.2 Clustered Storage

A basic concept is the clustering of the stored triples by their similarity, which is intended to lead to triples with similar keys being stored on the same or neighboring storage nodes. This is achieved by a similarity metric and the "Brood Sorting" ant-based clustering algorithm. Storage operations are designed to store new triples on a node or in a neighborhood where similar triples are stored, according to the similarity metric. The new triples are taken on a path through the storage network similar to the retrieval operations and every node is checked for similar triples. If a sufficient amount of similar triples are found, the new triples are stored on the current node. Additional effort is invested to move misplaced triples to a node or a neighborhood where similar triples are stored, a special move operation performs this task by picking up triples not fitting on a node and moving them to another location. For information on the concrete similarity measures, we refer to the related work introduced in section 2.

3.3 Reasoning Support

The system performs forward-chaining assertional reasoning based on the knowledge that is held in the store and writes the inferred triples back into it. This section provides an overview of the basic concepts for reasoning used in the

semantic store. However, we do not not focus on this aspect yet, as the underlying storage layer has to reach a stable state first. A detailed discussion of the theoretical underpinnings has been given in [14]. The introduced basic storage operations cannot always guarantee correct results, reasoning on top of our system also trades away completeness for the degree of scalability we aim to achieve. Inferred statements are retrieved using the described read operations, there is no systemic differentiation between explicit and inferred triples.

In S4, terminological and assertional triples are stored together, i.e. TBox and ABox of the description logic are separated only conceptionally, not physically. For every terminological axiom that is inserted in the TBox a reasoning process is started. This process applies the axiom to matching assertions within the ABox, following the virtual pheromone trails through the network for the localization of the fitting triple clusters. Once a match is found, the resulting assertion is derived and then written back in the store in order to make it available for retrieval operations.

As an example we consider an example axiom from [14]:

$$ta := professor \sqcap researcher \sqsubseteq seniorresearcher$$

a part of the TBox. This means that every resource that is an instance of $professor$ and $researcher$ is also an instance of $seniorresearcher$.

In order to generate the inferred axioms, a retrieval operation for the triples matching the different predicates of the expression is executed. In this case the first step is to look for instances of the type *professor*. In a second step the corresponding axiom, that defines the matching instance to be also of the type *researcher* is looked for, again following the pheromone trails in the store. If the process identifies a matching instance, the resulting axiom to define the instance to be also of the type *seniorresearcher* is inferred and written back using the standard storage process as described.

Since all axioms for the reasoning process are retrieved from the store, the reasoning process is subject to the probabilistic influences of the swarm algorithms. Because of the constantly expanding data base in the store and its decentralized nature the reasoning itself is a continuous process. Thus, the inferral of new axioms can take some time as well as there is no guarantee for a certain inferred axiom to be retrievable at a certain time.

RDFS Inferencing To give an example of our approach on reasoning, we will present our implementations for a selection of RDF Schema (RDFS) inferencing rules in the following. For each rule, we present the task given to the reasoning operations, which then move through the storage network to fulfill their respective tasks. A similar approach has also been followed in [4], where swarm in-

dividuals are used to locate seldom-visited parts of an RDF graph. In our case, however, the storage network possesses the ability to find the path to nodes where triples matching the inferencing rules are stored, thus making a far larger amount of nodes possible.

For any property, a domain and a range can be defined. If a property has a concept as its domain, every resource annotated with this property is an instance of this concept. Range is very similar: If a property has a concept as its range, every resource referred to by this property is an instance of this concept. The formal definition for `rdfs:domain` and `rdfs:range` is given as follows:

```
(?x ?p ?y), (?p rdfs:domain ?c) -> (?x rdf:type ?c)
(?x ?p ?y), (?p rdfs:range ?c) -> (?y rdf:type ?c)
```

For our reasoning operation, domain definitions are evaluated using several steps. The following list shows the required steps to evaluate this rule within our swarm-based system.

1. Read a `rdfs:domain` statement on the local node in the form `(?p rdfs:domain ?c)`, bind `p` and `c` to values from the matching statement.
2. Using `p` as lookup key for routing, move to the next node
3. Locate all statements in the form `(?r1 p ?r2)`, bind `r1` to values from the matching statement.
4. Create new statements of the form `(?r1 rdf:type c)` for all matches.
5. Write new statements to the storage network.
6. Continue with step 2 until no more matches have been found for a configurable number of steps.

Range definitions are evaluated using the same method, but with `(?r2 rdf:type c)` as new statement in step 4.

3.4 Optimizations and System Behaviour

In order to efficiently compare the new triples with a potentially large amount of locally stored triples, the statements are again organized into local clusters [12]. The same local clustering algorithm which is based on the agglomerative hierarchical clustering method is used to limit the amount of pheromones present on each neighbor connection. Our clustering algorithm is configured by a limit for the maximum number of local clusters. A higher number of local clusters is expected to increase the accuracy of the global clustering, and hence the performance of the retrieval and storage processes.

Read operations are restarted, until a user-defined timeout or maximum result count is reached. If a read operation has been successful in locating triples matching its pattern on a particular node, results are sent back to the host the

query originated from. Since the notion of our similarity measures and global clustering supports the assumption of fitting triples being available on nearby nodes, the read operation is continued on the neighboring nodes.

The decisions on which node to go to next or whether new triples should be stored on the current node are influenced by various random factors in accordance to the basic principles of swarm-based algorithms. This leads to non-deterministic behaviour of the entire system. Thus, this approach can only be verified by simulations or test runs. In the conceptual phase, one can only make educated guesses about the influence single configuration parameters would have on the behaviour of the entire system.

3.5 Advantages of the Swarm-Based Approach

In contrast to other distributed storage systems, S4 does not require a central catalog server nor an overlay network structure that is costly to maintain in the event of network topology changes. Network organization is decentralized and robust to changes, as node failures only affect the data stored on that very node and perhaps the nodes the failed one was connected to. The remainder of the potentially huge storage network is completely unaffected. Every node has sufficient local information to take all decisions required from them by the straightforward swarm algorithms, hence eliminating error-prone synchronization. The main advantage over deterministic solutions is the ability of the swarm algorithms to adapt to an ever-changing environment very well, whether a single node may be overloaded with data or requests, or the mentioned node failure issue: Swarm algorithms have the potential to handle these issues without significant overhead, while still being able to efficiently respond to the various requests. For example, triples with the RDF:type property describing the type of an resource occurs in approx. 20% of the triples in our test data set. If data distribution is determined by an hash function, all those triples will be stored on and retrieved from the same node, which will then be soon overloaded, if a lot of queries contain this property (which is the case). In our system, a node will slowly start to reject triples if it detects its load approaching a certain limit. This will lead to these triples being stored on other nodes, all using only local knowledge and status and no observer whatsoever.

4 Performance Evaluation

In this section, preliminary evaluation results of our current S4 development version are presented. We have implemented the S4 system as a distributed Java application [12] and deployed it onto a cluster of 150 virtual Linux nodes running on a server equipped with eight 2.6 GHz processors and a total of 64 GB

memory. For each test run, a predefined data set generated by the LUBM data generator [7] containing 1.3 million triples was written to the storage system. After a short cool-down period, a single query was sent to all cluster nodes. The amount of results returned as well as the time required to return those results was measured. In order to determine the influence of the relevant configuration parameters, a set of configuration files covering various settings for each of the relevant parameters was created and a full test run was performed with all configuration files. In particular, the influence of the following parameters conceptually influencing the swarm algorithms was evaluated:

- `CLUSTER_LIMIT` - The maximum number of local clusters allowed for triple storage and pheromone management
- `MAX_STEPS` - The maximum amount of hops between nodes a single operation is allowed to perform
- `NEIGHBOR_LIMIT` - The amount of neighbor nodes to connect to
- `DECAY_RATE` - The decay rate of the virtual pheromone trails per time unit

Due to the probabilistic behaviour of the algorithms employed in the S4 system, no two test runs yield entirely equivalent results. The results presented below were taken from one single test cycle with identical software versions containing 30 test runs, each with a different configuration. However, these results are regarded to be exemplary, as other test cycles showed comparable results, and the test runs presented here have been selected to show the influence of the single parameters as clearly as possible. For this preliminary evaluation, the parameters were considered to be independent, in an attempt to reduce the search space.

4.1 Storage Performance

The LUBM-10 dataset we used to evaluate the system performance is written into 189 files by the data generator, each containing around 7000 triples. We used the HTTP API on an arbitrary node to store each file sequentially into the S4 system. Obviously, not all triples were stored on the node the requests were issued to. Fig. 1 shows the distribution of the amount of triples stored per storage node using a value of 25 for the `MAX_STEPS` parameter.

The time required to write a single file of the test data set into the S4 system was measured. Box plots describing the statistical trends of the write time for all files in the different configuration tested are given in Fig. 2:

As expected, a higher `CLUSTER_LIMIT` leads to a larger amount of comparably expensive cluster maintenance operations, therefore the write times for

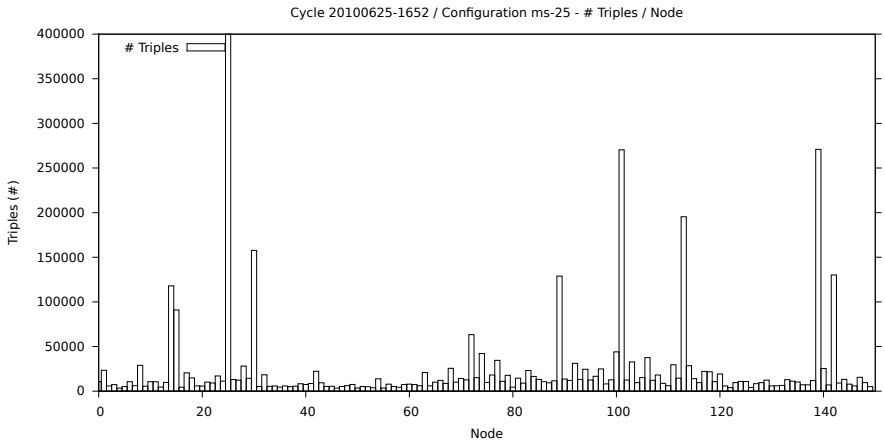


Fig. 1. Storage load distribution

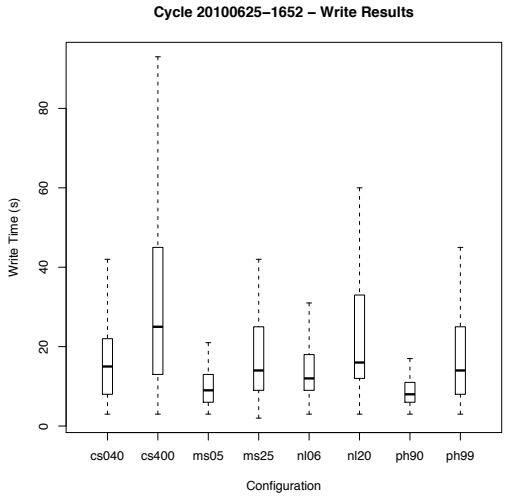


Fig. 2. Storage performance for LUBM-10 dataset files

configuration `cs400` with a cluster limit of 400 greatly exceed those of configuration `cs040` with a limit of 40.

The influence of the amount of steps to be taken, as configured by the `MAX_STEPS` parameter was tested in configuration sets `ms05` and `ms25` with 5 and 25 for maximum step number, respectively. As the transition of an operation from one storage node to another one is an expensive operation as well, a smaller value also leads to an improved write performance.

System performance was also expected to be influenced by the amount of neighbor nodes a single node connects itself to (`NEIGHBOR_LIMIT`). In contrast to our first expectation of a larger amount of neighbor nodes also distributing the load over more shoulders, configuration sets `nl06` and `nl20` with 6 and 20 as neighbor limits show a smaller neighborhood having a beneficial impact on the write performance. This may be due to the larger amount of possible paths that can be marked by the pheromones and thus missing pheromone data on some of the nodes.

A central configuration parameter for all systems based on ant foraging is the pheromone decay rate [5, p. 212 ff], thus the influence of this value over the parameter `DECAY_RATE` was also tested: Configuration `ph90` used a decay rate of 90, in which pheromone intensities are reduced by 10% every second, `ph99` employed only a 1% decay rate. Again, the results did not meet our expectations: Normally, longer pheromone endurance would result in better paths through the storage network, and thus reduced step counts, but the test result showed the opposite to be true for write operations. An possible explanation for this phenomenon might be the following: As pheromones on more frequently used paths are also updated more often, sub-optimal paths are then discarded sooner.

4.2 Retrieval Performance

To measure the performance for operations retrieving data from the S4 system, a SPARQL query was evaluated several times on each of the 150 storage nodes sequentially. The query contained a single triple pattern matching approx. 20% of the stored triples. Each node started the corresponding read operations within the S4 system, and collected the results from inside the storage network. The time limit for each query was set to five seconds, and the requested result set size was 1000 results. For each node, the amount of results returned as well as the time required to deliver the results was measured by the test driver. After three warm-up runs, five query executions were performed, and the average results determined the final result for a node in a configuration set. The result plots are structured as follows: The x axis contains all 150 storage nodes by their ID, the left y axis denotes the required time for each query ($+$ marker), and the right y axis describes the amount of triples returned ($\times$ marker). In general, a very low response time with a high amount of results returned are desirable for every storage node.

Fig. 3 shows the read results for different local cluster sizes, both for triple storage and pheromone clustering. The upper configuration used a `CLUSTER_LIMIT` value of 40, while the lower configuration used a value of 400. A higher amount of pheromone clusters leads to a more precise path selection and thus

node location, which reduces read times and increases result counts due to the smaller amount of hops required for a successful read operation.

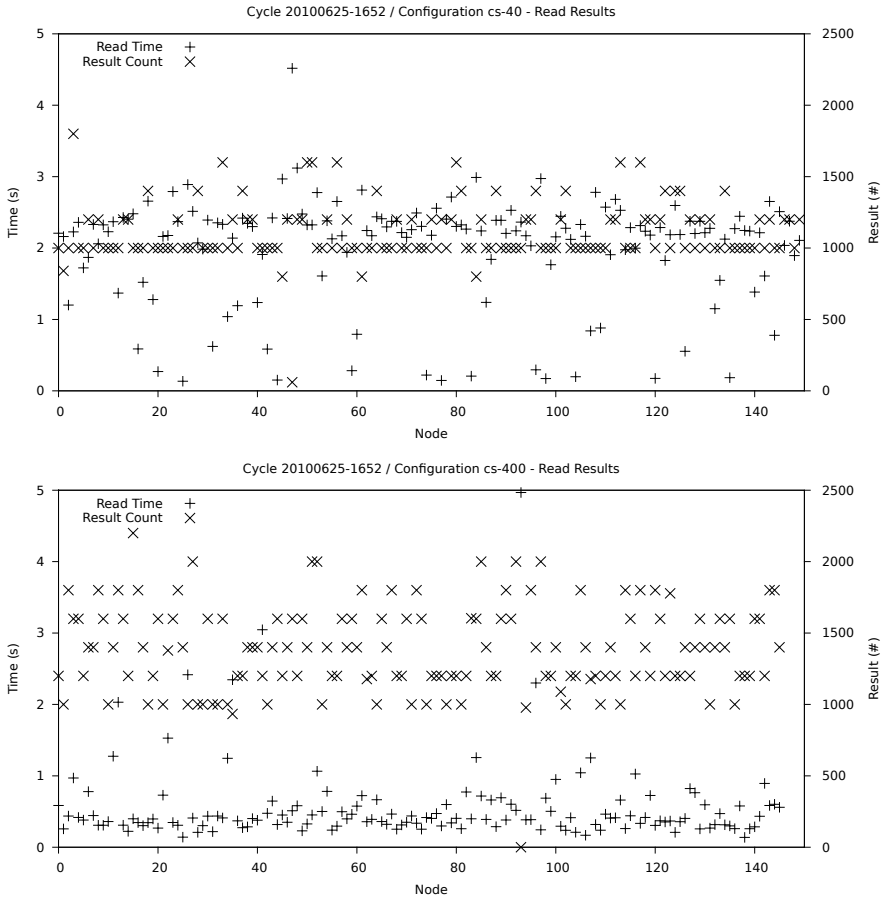


Fig. 3. Read results for `CLUSTER_LIMIT` parameter variations

The read operations are also greatly influenced by the amount of hops a single operation is allowed to take. Fig. 4 shows a test run with a `MAX_STEPS` setting of 5 on the top and 25 on the bottom. The upper configuration clearly shows a quick response time for the majority of nodes, but only the minimum amount of triples returned. This expected behaviour is confirmed by the bottom configuration: Not only are results delivered at a comparable speed, but the average amount of triples returned is also increased.

Contrary to in-memory executions of a swarm algorithm, the amount of steps allowed for a swarm individual has a huge impact on overall performance in a distributed system. This is due to the high costs of the transition operation between the storage nodes. Therefore, for any system built on swarm intelligence, this parameter has to be adjusted carefully.

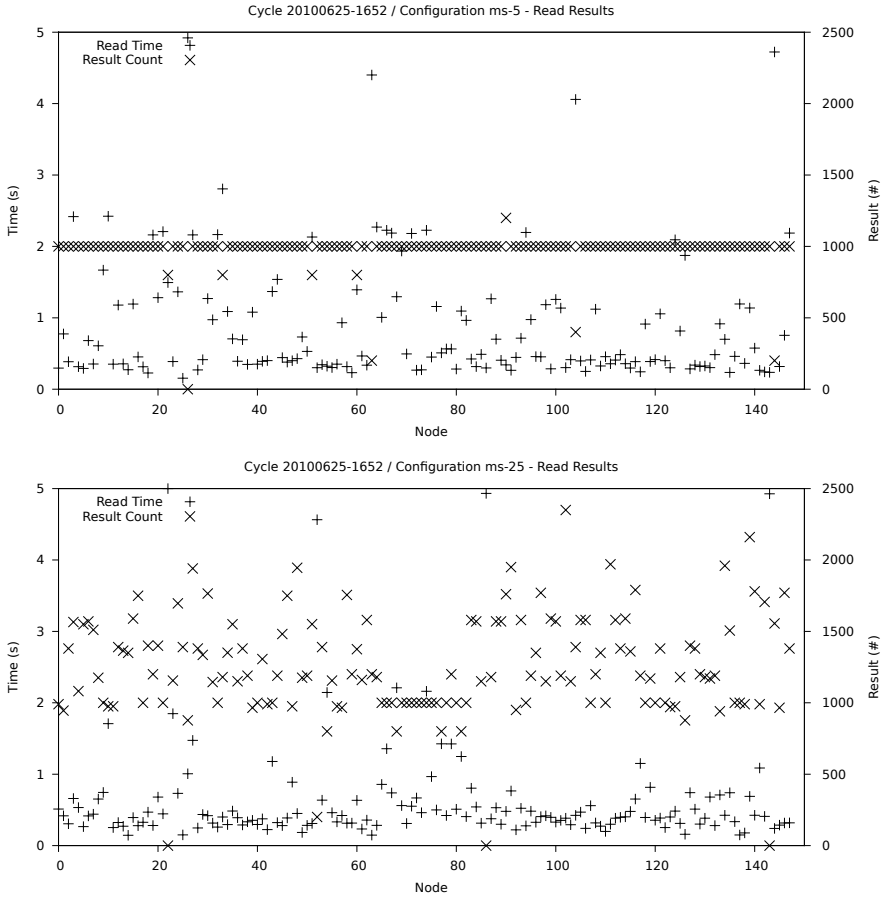


Fig. 4. Read results for `MAX_STEPS` parameter variations

Network setup from our bootstrapping algorithm is restricted to a limited amount of neighbor nodes. This behaviour is controlled by the `NEIGHBOR_LIMIT` parameter. In Fig. 5 two test runs with a neighbor limit of six on the top and a neighbor limit of 20 on the bottom are displayed. The configuration with the smaller limit performs better, the amount of triples returned is in gen-

eral higher, and the response times are consistently below one second with very few exceptions. This coincides with the observation of the read performance in the preceding section, where a smaller value for the neighbor limit also brought improvements in system performance. This may be due to a random error factor considered for the decision which neighbor to visit next. If there are more neighbor nodes, this error factor could have an increasingly disadvantageous influence, which will be evaluated in our further work.

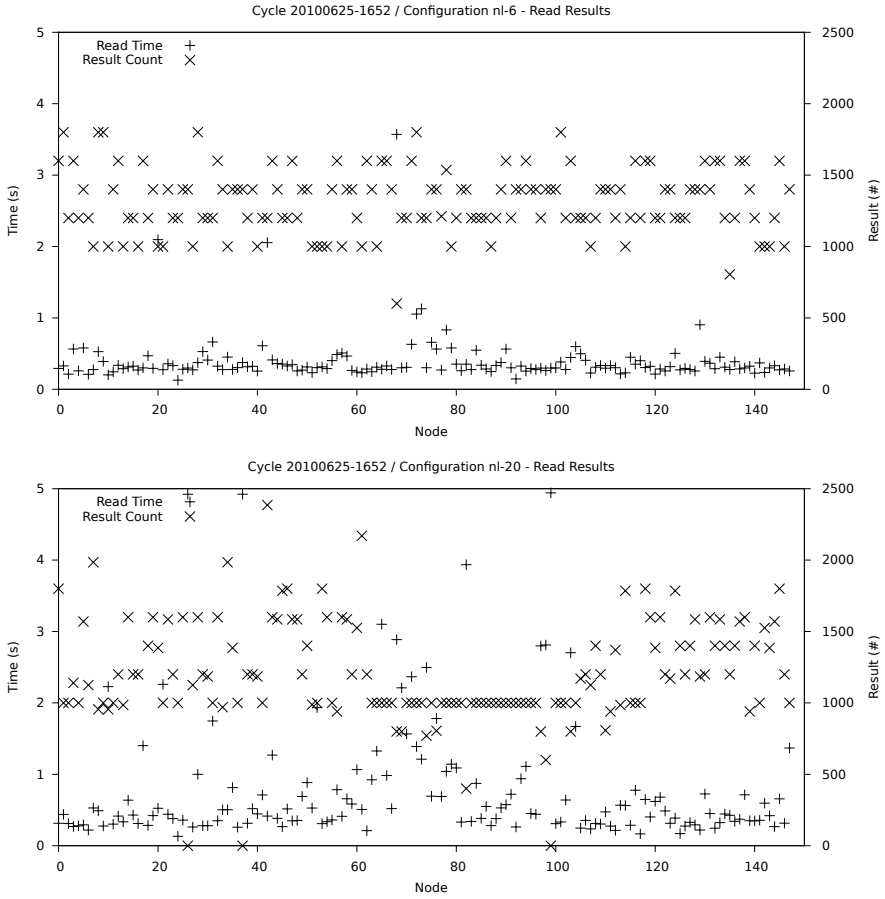


Fig. 5. Read results for `NEIGHBOR_LIMIT` parameter variations

The amount of decay in the intensity of the virtual pheromone also plays a role during read operations. Fig. 6 displays two test runs with a decay rate of 10% on the top and 1% on the bottom. As in the storage evaluation, the re-

sults again did not meet our expectations for the influence of this parameter. The test run with the increased decay shows very quick responses on nearly all nodes with a large amount of results, while the test run with the lower value shows considerably increased response times. This suggests that the introduction a mechanism to adjust this particular parameter is necessary for a distributed system based on swarm algorithms.

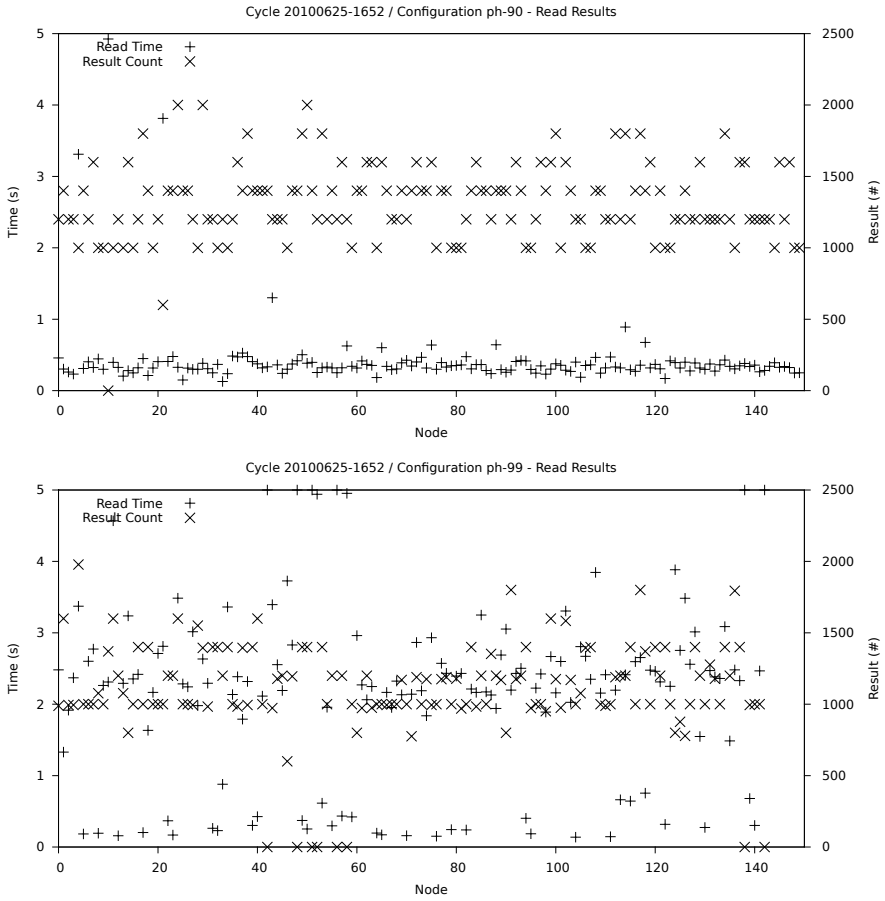


Fig. 6. Read results for `DECAY_RATE` parameter variations

5 Conclusion and Future Work

This paper started with outlining previous work in the application of swarm-based algorithms on distributed storage as well as distributed semantic storage

systems in general and current distributed reasoning approaches. We then continued introducing the relevant concepts of our Self-Organized Semantic Storage Service (S4), which uses self-organizing algorithms found in the behaviour of several ant species. A number of relevant configuration parameters for this system was identified and their expected impact on the system performance was described. Our preliminary evaluation of our implementation of the S4 system then showed the actual impact of the identified parameters in exemplary test runs using different values for those parameters in their configuration.

Our evaluation of basic storage and retrieval capabilities was set in an environment as close as possible to a real-world setting, as the desired performance of a swarm-based self-organizing system cannot be shown using formal proofs, but only by collecting statistical data. Therefore, simulations were not deemed to be satisfactory for our purpose. Successful test runs for a part of the configuration sets continue to show the general feasibility of building a larger-scale distributed system using swarm algorithms to facilitate self-organization. Tuning the various parameters was identified to be one of the main issues of swarm-based self-organizing systems, and tweaking the parameters to values where the storage network exhibits the desired results can be a tedious process, comparable to the work of a specialized database administrator. In some cases, heuristics could be used to adjust a particular parameter. However, special care has to be taken for these heuristics not to require global knowledge. This would contradict one of the basic concepts for swarm-based algorithms.

In our future work we would like to use our lab network to further advance our S4 implementation, and continue with evaluations using a variety of data sets, queries, network structures and configuration sets. We will implement the swarm-based distributed reasoning approach, and evaluate it as well. Further advancements of the S4 concepts are expected to be the design of additional auxiliary algorithms to support the basic algorithms, with the hope of achieving a system supporting its users by adjusting as many parameters as possible by itself.

Acknowledgments

We would like to thank our reviewers for their insightful comments. This work has been partially supported by the “DigiPolis” project funded by the German Federal Ministry of Education and Research (BMBF) under the grant number 03WKP07B.

References

1. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Santa Fe Institute Studies in the Sciences of Complexity Series, Oxford Press (July

1999)
2. Cai, M., Frank, M.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: WWW '04: Proceedings of the 13th international conference on World Wide Web. pp. 650–657. ACM, New York, NY, USA (2004)
3. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An infrastructure for peer information management. IEEE Internet Computing 11(5), 36–44 (2007), http://doi.ieeecomputersociety.org/10.1109/MIC.2007.108
4. Dentler, K., Gueret, C., Schlobach, S.: Semantic web reasoning by swarm intelligence. In: Proceedings of Nature inspired Reasoning for the Semantic Web, ISWC 2009 (2009)
5. Dorigo, M., Stützle, T.: Ant Colony Optimization. The MIT Press, Cambridge, Massachusetts (2004)
6. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7, 80–112 (1985)
7. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem 3(2-3), 158–182 (2005), http://dx.doi.org/10.1016/j.websem.2005.06.005
8. Harasic, M., Augustin, A., Tolksdorf, R., Obermeier, P.: Cluster mechanisms in a self-organizing distributed semantic store. In: Proceedings of Web Information System and Technologies, WEBIST 2010 (2010)
9. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In: The Semantic Web, ISWC 2007, Busan, Korea, November 11-15, 2007. Lecture Notes in Computer Science, vol. 4825, pp. 211–224. Springer (2007), http://dx.doi.org/10.1007/978-3-540-76298-0_16
10. Mamei, M., Menezes, R., Tolksdorf, R., Zambonelli, F.: Case studies for self-organization in computer science. J. Syst. Archit. 52(8), 443–460 (2006)
11. Menezes, R., Tolksdorf, R.: A new approach to scalable linda-systems based on swarms. In: Proceedings of ACM SAC 2003. pp. 375–379 (2003)
12. Mühleisen, H., Harasic, M., Tolksdorf, R., Teymourian, K., Augustin, A.: A self-organized semantic storage service (2010), submitted to the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS2010), preprint available at http://digipolis.ag-nbi.de/preprint/iiwas2010-s4-preprint.pdf
13. Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmr, M., Risch, T.: EDUTELLA: A P2P networking infrastructure based on RDF. Proceedings of the eleventh international World Wide Web Conference (Jan 01 2002), http://wwwconf.ecs.soton.ac.uk/archive/00000306/;http://wwwconf.ecs.soton.ac.uk/archive/00000306/01/index.htm
14. Obermeier, P., Augustin, A., Tolksdorf, R.: Towards swarm-based federated web knowledgebases (2009), submitted to Nature inspired Reasoning for the Semantic Web (NatuReS09), preprint available at http://digipolis.ag-nbi.de/preprint/natures2009-sr-preprint.pdf
15. Oren, E., Kotoulas, S., Anadiotis, G.: MaRVIN: A platform for large-scale analysis of semantic web data. In: Proceeding of the WebSci'09: Society On-Line (March 2009)
16. Tolksdorf, R., Augustin, A.: Selforganisation in a storage for semantic information. Journal of Software 4 (2009)
17. Tolksdorf, R., Augustin, A., Koske, S.: Selforganization in distributed semantic repositories. Future Internet Symposium 2009 (FIS2009) (2009)
18. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable distributed reasoning using MapReduce. In: The Semantic Web - ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Lecture Notes in Computer Science, vol. 5823, pp. 634–649. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04930-9

Scalable In-memory RDFS Closure on Billions of Triples

Eric L. Goodman[1] and David Mizell[2]

[1] Sandia National Laboratories, Albuquerque, NM, USA
elgoodm@sandia.gov
[2] Cray, Inc., Seattle, WA, USA
dmizell@cray.com

Abstract. We present an RDFS closure algorithm, specifically designed and implemented on the Cray XMT supercomputer, that obtains inference rates of 13 million inferences per second on the largest system configuration we used. The Cray XMT, with its large global memory (4TB for our experiments), permits the construction of a conceptually straightforward algorithm, fundamentally a series of operations on a shared hash table. Each thread is given a partition of triple data to process, a dedicated copy of the ontology to apply to the data, and a reference to the hash table into which it inserts inferred triples. The global nature of the hash table allows the algorithm to avoid a common obstacle for distributed memory machines: the creation of duplicate triples. On LUBM data sets ranging between 1.3 billion and 5.3 billion triples, we obtain nearly linear speedup except for two portions: file I/O, which can be ameliorated with the additional service nodes, and data structure initialization, which requires nearly constant time for runs involving 32 processors or more.

Keywords: Semantic Web, RDFS Closure, Cray XMT, Hashing

1 Introduction

Semantic web data in its most common format, the Resource Description Framework (RDF), has two primary means for defining ontologies: RDF Schema (RDFS) and the Web Ontology Language (OWL). Ontologies are useful mechanisms for organizing domain knowledge, allowing explicit, formal descriptions of data to be defined and shared. Also, ontologies allow reasoning about the domain, exposing new facts through application of the ontology to existing data.

In this paper we examine the simpler of the ontology languages, RDFS, and perform RDFS reasoning on billions of triples completely in-memory on a highly multithreaded shared-memory supercomputer, the Cray XMT. To our knowledge, no one has performed such large RDFS reasoning in a single global shared address space, nor has anyone previously achieved the inferencing rates we report in this paper.

The rest of the paper is organized as follows. Section 2 describes the Cray XMT, its unique characteristics, and why it may be well suited to RDFS closure

and many semantic web applications in general. Section 3 describes the algorithm we employ to perform closure. Sections 4 and 5 describe the experimental setup and the results. We then conclude in sections 6 and 7 with a comparison to other approaches and our path forward.

2 Cray XMT

The Cray XMT is a unique shared-memory machine with multithreaded processors especially designed to support fine-grained parallelism and perform well despite memory and network latency. Each of the custom-designed compute processors (called *Threadstorm* processors) comes equipped with 128 hardware threads, called *streams* in XMT parlance, and the processor instead of the operating system has responsibility for scheduling the streams. To allow for single-cycle context switching, each stream has a program counter, a status word, eight target registers, and thirty-two general purpose registers. At each instruction cycle, an instruction issued by one stream is moved into the execution pipeline. The large number of streams allows each processor to avoid stalls due to memory requests to a much larger extent than commodity microprocessors. For example, after a processor has processed an instruction for one stream, it can cycle through the other streams before returning to the original one, by which time some requests to memory may have completed. Each *Threadstorm* processor can currently support 8 GB of memory per processor, all of which is globally accessible. The system we use in this study has 512 processors and 4 TB of shared memory. We also employed 16 Opteron nodes of the service partition, which directly perform file and network I/O on behalf of the compute processors.

Programming on the XMT consists of writing C/C++ code augmented with non-standard language features including generics, intrinsics, futures, and performance-tuning compiler directives such as pragmas.

Generics are a set of functions the Cray XMT compiler supports that operate atomically on scalar values, performing either `read`, `write`, `purge`, `touch`, and `int_fetch_add` operations. Each 8-byte word of memory is associated with a full/empty bit and the read and write operations interact with these bits to provide light-weight synchronization between threads. Here are some examples of the generics provided:

- $readxx$: Returns the value of a variable without checking the full-empty bit.
- $readfe$: Returns the value of a variable when the variable is in a full state, and simultaneously sets the bit to be empty.
- $writeef$: Writes a value to a variable if the variable is in the empty state, and simultaneously sets the bit to be full.
- int_fetch_add: Atomically adds an integer value to a variable.

Besides generics, there are also intrinsic functions that expose many low-level machine operations to the C/C++ programmer.

Parallelism is achieved explicitly through the use of futures, or implicitly, when the complier attempts to automatically parallelize for loops. Futures allow programmers to explicitly launch threads to perform some function. Besides

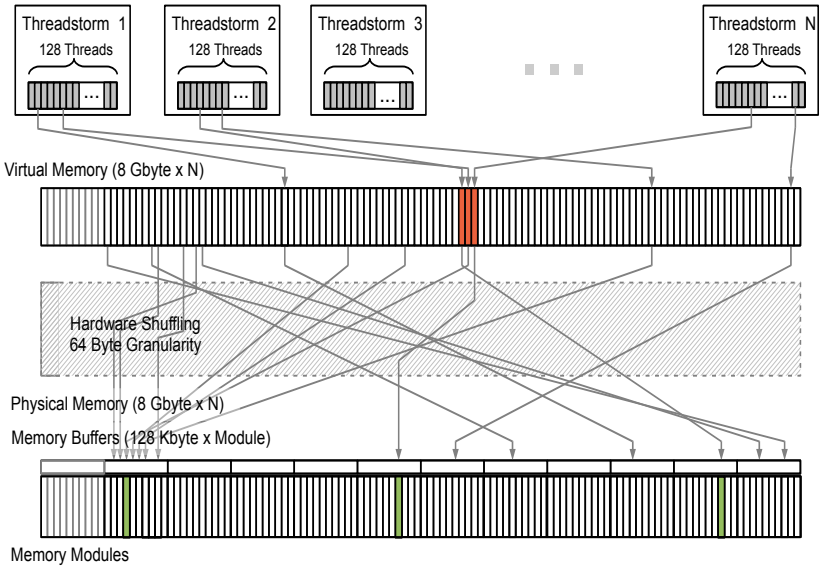


Fig. 1. Cray XMT Threadstorm memory subsystem: Threadstorm processors access a virtual memory address space that is mapped through hardware shuffling to actual physical locations to mitigate memory access hotspots.

explicit parallelism through futures, the compiler attempts to automatically parallelize for loops, enabling implicit parallelism. The programmer can also provide pragmas that provide hints to the compiler on how to schedule iterations of the for loop to various threads, whether it be by blocks, interleaved, or dynamically, or supply hints on how many streams to use per processor, etc. We extensively use the `#pragma mta for all streams i of n` construct that allows programmers to be cognizant of the total number of streams that the runtime has assigned to the loop, as well as providing an iteration index that can be treated as the id of the stream assigned to each iteration.

2.1 Applicability of the XMT to the Semantic Web

The XMT and its predecessors have a significant history of performing quite well on graph algorithms and on applications that can be posed as graph problems. Bader and Madduri [2] report impressive execution times and speedup for fundamental graph theory problems, breadth-first search and *st*-connectivity. Later, Madduri et al. [8] use the Δ-stepping algorithm, performing single source shortest path on a scale-free, billion-edge graph in less than ten seconds on 40 processors. More recently, Chin et al. [3] examine triad census algorithms for social network analysis on graphs with edges in the hundreds of millions. Also, Jin et al. [6] perform power grid contingency analysis on the XMT by posing the

problem as finding important links using betweeness centrality of the edges as a metric.

To see how this relates to the semantic web, consider that the semantic web amounts to a large, sparse graph. The predicate of an RDF triple can be thought of as a directed edge between two nodes, the subject and the object. Several have proposed thinking of the semantic web as a graph, and have suggested extensions to SPARQL to enable advanced graph queries otherwise not available in standard SPARQL, including SPARQ2L [1], nSPARQL [10], and SPARQLeR [7]. Also, Stocker et al. [12] present an optimization scheme for SPARQL queries based on the graph nature of RDF data. They state one limitation of the approach is due to its implementation in main memory. However, this is not as much of an issue for the XMT with its scalable global shared memory. Indeed, for the machine we use in this study, its 4 TB of shared memory is sufficient for the data of many semantic web applications to reside completely in memory. The XMT is well suited for applications with (a) abundant parallelism, and (b) little or no locality of reference. Many graph problems fit this description, and we expect that some semantic web applications will as well, especially those that involve more complex queries and inferencing.

3 Algorithm

The algorithm we present here performs incomplete RDFS reasoning, and calculates only the subset of rules that require two antecedents, namely rdfs rules 2, 3, 5, 7, 9, and 11 (for reference, see Table 1). This is fairly common in practice, as the other rules are easily implemented and parallelized and generally produce results that are not often used or can be inferred later at run time for individual queries. The flow of the algorithm (see Figure 2) is similar to the one outlined by Urbani et al. [13] in that only a single pass over the rule set is required (excluding rare cases where the ontology operates on RDFS properties) for full and correct inferencing. However, we move processing of the transitivity rules, 5 and 11, to the beginning. These rules accept no inputs except for existing ontological triples, and thus can be executed safely at the beginning. Also, we do not remove duplicates because our algorithm produces no duplicates. We use a global hash table described by Goodman et al. [4] to store the original and inferred triples. Thus, duplicate removal is accomplished *in-situ*, i.e. insertion of any triples produced during inferencing that already exists in the table results in a no-op.

As a preprocessing step, we run the set of triples through a dictionary encoding that translates the resource and predicate strings into 64 bit integers. We again use the hash table described earlier to store the mapping and perform the translation. As the triples are now sets of three integers, we can make use of the following hash function during the RDFS closure algorithm:

$$\text{hash}(t) = ((t.s + t.p \cdot B + t.o \cdot B^2) \cdot C) \mod S_{table} \quad (1)$$

where t is a triple, $C = 31,280,644,937,747$ is a large prime constant (taken from [4]), S_{table} is the size of the hash table, and B is a base to prevent the sub-

Rule	Condition 1	Condition 2 (optional)	Triple to Add
lg	s p o (o is a literal)		s p _:n
gl	s p _:n		s p o
rdf1	s p o		p type Property
rdf2	s p o (o is a literal of type t)		_:n type t
rdfs1	s p o (o is a literal)		_:n type Literal
rdfs2	**p domain x**	**s p o**	**s type x**
rdfs3	**p range x**	**s p o**	**o type x**
rdfs4a	s p o		s type Resource
rdfs4b	s p o		o type Resource
rdfs5	**p subPropertyOf q**	**q subPropertyOf r**	**p subPropertyOf r**
rdfs6	p type Property		p subPropertyOf p
rdfs7	**s p o**	**p subPropertyOf q**	**s q o**
rdfs8	s type Class		s subClassOf Resource
rdfs9	**s type x**	**x subClassOf y**	**s type y**
rdfs10	s type Class		s subClassOf s
rdfs11	**x subClassOf y**	**y subClassOf z**	**x subClassOf z**
rdfs12	p type ContainerMembershipProperty		p subPropertyOf member
rdfs13	o type Datatype		o subClassOf Literal

Table 1. This table lists all the rules that are part of RDFS entailment [11]. The rules in bold are the ones we implemented for our closure algorithm.

Algorithm: RDFS Closure

1: Read data from service nodes
2: Create and populate ontology data structures
3: Create and populate multimaps
4: Apply transitivity rules, **rdfs5** and **rdfs11**
5: Replicate multimap data structures.
6: Insert original triples into hash table
7: Add matching triples to queues
8: **rdfs7** - Subproperty Inheritance
9: Add matching triples to domain and range queues
10: **rdfs2** - Domain
11: Add matching triples to subclass queue
12: **rdfs3** - Range
13: Add matching triples to subclass queue
14: **rdfs9** - Subclass Inheritance

Fig. 2. Overview of RDFS Closure algorithm on the XMT.

ject, predicate, and object ids from colliding. The current dictionary encoding algorithm assigns ids to resources and properties by keeping a running counter, and a triple element's id is the value of the counter at the time the element was seen during processing. Thus, the set of integers assigned to the subject, predicate, and objects overlap. The base helps the hash function to avoid collisions due to this overlap. For our experiments, we used $B = 5e9$, near the maximum value assigned to any triple element. This hash function was decided upon after some experimentation, but probably deserves further attention.

The first step (line 1 of Figure 2) of the RDFS closure process on the XMT is to transfer the RDF triple data from the service nodes to the compute nodes. We use the Cray XMT snapshot library that allows programs to move data back and forth from the Lustre file system. On the service nodes a Linux process called a file service worker (fsworker) coordinates the movement of data from disk to the compute nodes. Multiple file service workers can run on multiple service nodes, providing greater aggregate bandwidth.

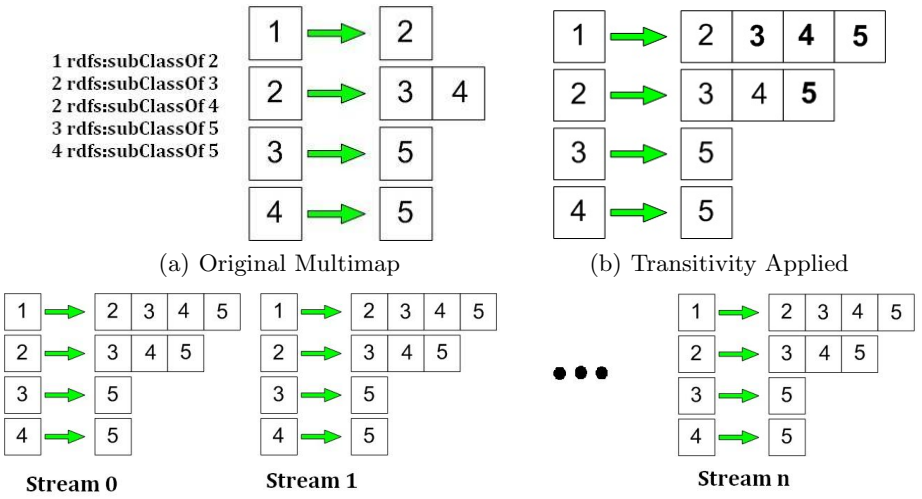
The next step (lines 2-5 of Figure 2) is to load the ontological data into multimap data structures. All of the rules under consideration involve a join between a

- data triple or an ontological triple with an
- ontological triple,

and in all cases, the subject of the latter ontological triple is matched with a subject, predicate, or object of the former. To speed processing of the rules and the associated join operations, we create four multimaps of the form $f : Z \rightarrow Z^*$, one for each of the predicates **rdfs:subPropertyOf**, **rdfs:domain**, **rdfs:range**, and **rdfs:subClassOf**, that maps subject ids to potentially multiple object ids. For example, the **rdfs:subClassOf** multimap is indexed according to class resources and maps to all stated super classes of those resources. After initially populating the multimap with known mappings as asserted in the original triple set (see Figure 3(a)), we then update each of the **rdfs:subPropertyOf** and **rdfs:subClassOf** multimaps with a complete listing of all super properties and super classes as dictated by inference rules 5 and 11 (see Figure 3(b)).

After populating the multimaps and applying the transitivity rules, we then replicate the entirety of this data and the accompanying data structures for each and every stream (see Figure 3(c)). The reason for this is to avoid read-hotspotting. The ontology is relatively small compared with the data to be processed, so many streams end up vying for access if only one copy of the ontology is available to all streams. Replicating the data circumvents these memory contention issues.

The next two steps (lines 6-7 of Figure 2) involve iterating over all of the original triples and adding them to the hash table, and also populating four queues, also implemented as hash tables, that store triples matching rules 2, 3, 7, and 9. Ideally, this could be solved with one pass through the original triple set, but the XMT favors tight, succinct loops. Processing all steps at once results in poor scalability, probably because the compiler is better able to optimize smaller, simpler loops. Thus, each of these individual steps receives its own for loop.



(a) Original Multimap (b) Transitivity Applied

(c) Replicated

Fig. 3. Subfigure (a) shows a set of triples containing *rdfs:subclassOf* information. The integers represent URIs. The displayed mulitmap represents the state after Step *3* of the RDFS Closure algorithm. Subfigure (b) shows the state after Step *4*, where each class has a complete listing of all superclasses. Subfigure (c) illustrates that after Step *5*, all of the multimaps are replicated for dedicated stream usage.

The last four steps (lines 8-14 of Figure 2) complete the calculation of RDFS closure. In general, each of these steps follow the `ComputeRule` procedure outlined in Figure 4. There are three parameters: an array of triples that match the given rule, a multimap that stores subject/object mappings for a given predicate, and a buffer to store new triples. We use the `for all streams` construct, allowing us to know the total number of streams, *num_streams*, and a stream identifier, *stream_id* to use within the outer for loop. Line 5 partitions the matching triples into nearly equal-sized chunks for each stream. The for loop from lines 7 to 10 determines the number new triples that will be created. This first pass at the data allows us to call `int_fetch_add` once for each stream instead of l times, where l is the length of the matching triple array. In essence, when we call `int_fetch_add` in line 11 we claim a block of indices in the buffer array that can be iterated over locally instead of requiring global cooperation by all streams. This helps to avoid hot-spots on the *total* variable. Lines 12 through 17 iterate over the stream's set of triples and then iterate over the set of matching rules in the multimap to create new inferred triples. After application of `ComputeRule`, inferred triples in the buffer are added both to the main hash table and to appropriate rule queues. Again, these two steps are done separately to maintain scalability.

Overall, the scalability and success of this algorithm largely rests on three main factors:

```
1: procedure ComputeRule(matching_triples, multimap, buffer)
2:     total ← 0
3:     for all streams stream_id of num_streams do
4:         l ← length(matching_triples)
5:         beg, end ← determine_beg_end(l, num_streams, stream_id)
6:         local_total ← 0
7:         for i ← beg, end do
8:             num ← num_values(matching_triples[i], multimap)
9:             local_total ← local_total + num
10:        end for
11:        start = int_fetch_add(total, local_total)
12:        for i ← beg, end do
13:            for j ← 1, num_values(matching_triples[i]) do
14:                buffer[start] ← new_triple(matching_triples[i], multimap, j)
15:                start ← start + 1
16:            end for
17:        end for
18:    end for
19: end procedure
```

Fig. 4. This figure outlines the general process for computing RDFS rules 2, 3, 7, and 9.

- Extensive use of hash tables: Hash tables proved effective for the implementation of RDFS closure by allowing the algorithm to be conceptually simple and to maintain scalability over large numbers of processors. The global nature of the hash table also removed the time-consuming deduplication phase, a necessary step for many algorithms targeting distributed memory platforms.
- Multiple tight loops: We favored small succinct loops, even if that meant iterating over the same data twice, as it allowed scalable performance.
- Replicating the ontology: A single instance of the ontology created a hotspot as many streams were accessing the same data simultaneously. Replicating the ontology removed memory contention issues. This is likely generalizable to other situations when many streams must use small, read-only data structures.

4 Data Sets

We used the Lehigh University Benchmark (LUBM) [5]. Specifically, we generated the LUBM10k and LUBM40k data sets, approximately 1.33 and 5.34 billion triples, respectively. Both increase in size after closure by about 25%, resulting in 341 million and 1.36 billion inferences.

To validate our approach, we applied a fixed-point algorithm written in python[3] to smaller LUBM instances, commenting out rules we did not implement in our algorithm.

The code we wrote for RDFS closure on the XMT is open source and publically available. Most of the code is published as part of the MapReduceXMT code base[4]. Our initial implementation first looked at porting the MapReduce implementation of Urbani, et al. [13] to the XMT, but to obtain better performance we adopted an implementation that was more closely tailored for the XMT. The main class is `rdfs_closure_native.cpp` located in the `test` directory. We also made use of functionality provided by the MultiThreaded Graph Library[5], primarily `mtgl/xmt_hash_table.hpp`.

5 Experiments

Figure 5 shows the times for computing RDFS closure on LUBM10k and LUBM40k. For all runs we use 16 service nodes each running one fsworker. We conducted runs using between 32 and 512 processors. We did not go smaller than 32 because the snapshot libraries require at least $2f$ processors, where f is the number of fsworkers.

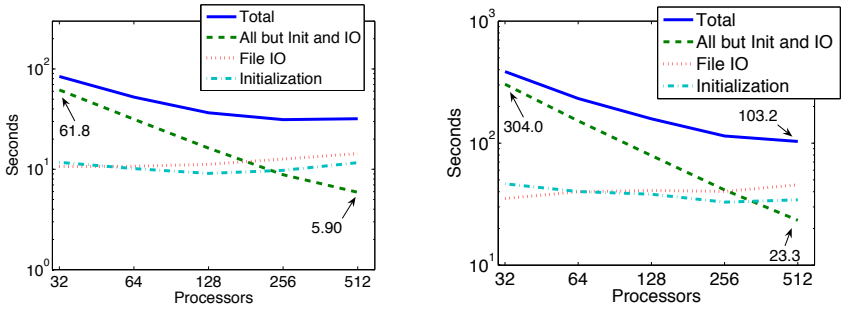
The file I/O times reported below include only the time to read in the file. As the XMT's most common use case will likely be a memory-resident semantic database, we include only the load time as any writes will likely be asynchronous.

Also, the initialization times reported below only include the time for the construction of the main hash table, the four smaller queues, and two large triple buffers. Other data structure initialization calls were made, but they were significantly smaller and included in times of the other phases.

The algorithm shows good scalability for everything but file I/O and initialization, which are nearly constant for a given problem size and irrespective of the number of processors. Comparing the times of the 32 and 512 processor runs on LUBM40k, we achieve about 13 times speedup for the non-I/O/init portion, where linear speedup in this processor range is 16. On LUBM10k, the speedup is lower, about 10, indicating that the problem size of LUBM10k is not sufficient to utilize the full 512 processor system as well as LUBM40k.

Figure 6 shows the inference rate in inferences/second each method obtained for each of the data sets. Initially, better rates are achieved on LUBM10k for smaller numbers of processors. However, for large processor counts, the algorithm's better scalability on LUBM40k wins out, obtaining a rate of 13.2 million inferences per second with 512 processors.

[3] http://www.ivan-herman.net/Misc/PythonStuff/RDFSClosure/Doc/RDFSClosure-module.html
[4] https://software.sandia.gov/trac/MapReduceXMT
[5] https://software.sandia.gov/trac/mtgl



(a) LUBM10k (b) LUBM40k

Fig. 5. Times recorded for LUBM10k and LUBM40k.

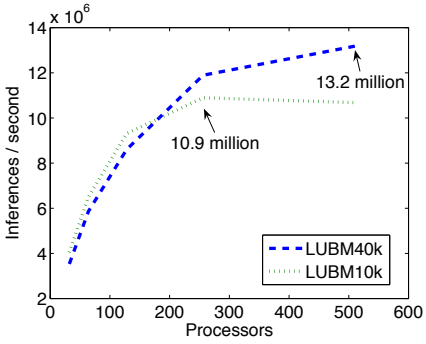


Fig. 6. Inference rates obtained on LUBM10k and LUBM40k.

5.1 Modifying the number of service nodes

We also examined how the number of fsworkers affects aggregate I/O throughput to the compute nodes. We kept the number of processors constant at 128 and varied the number of fsworkers between 1 and 16, each fsworker running on a single node of the service partition. We tested the read time on LUBM40k. Figure 7 shows the rates obtained and the ideal rates expected if performance increased linearly with the number of fsworkers. Overall, we experienced about a 12.4 increase in I/O performance for 16 fsworkers over 1 fsworkers, indicating that if I/O is a bottleneck for particular applications, it can be ameliorated with additional service nodes and fsworkers.

6 Comparison to other Approaches

There are two main candidates for comparison with our approach. The first is an MPI-based implementation of RDFS closure developed by Weaver and Hendler [16], which we will refer to simply as *MPI*. The second is WebPIE, a MapReduce style computation developed by Urbani et al. The authors first focused on

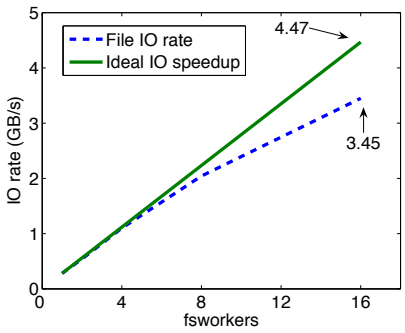


Fig. 7. I/O rates achieved for various number of fsworkers.

RDFS closure [13] and then expanded their efforts to include a fragment of OWL semantics [14].

Weaver and Hendler developed an embarrassingly parallel algorithm for RDFS closure by replicating the ontology for each process and partitioning up the data. They utilized a 32 node machine with two dual-core 2.6 GHz Opteron processors per node. They achieved linear scalability but with the side effect of producing duplicates. The individual processes do not communicate and do not share data. Thus, it is impossible to know if duplicates are being produced.

Besides the fundamental algorithmic difference between our implementation and theirs concerning duplicate creation (they create duplicates when we do not), there are several other factors which make a direct comparison difficult. For one, they operate on raw triple data rather than integers. Also, they perform all of the rules listed in Table 1 while we compute a subset. Finally, they include both reading the data in from disk and writing the result out, where we only report the read time.

The work of Urbani et al. offers a much more direct comparison to our work, as they first perform a dictionary encoding algorithm [15] as a preprocessing step and also perform a similar subset of RDFS rules as we do in this work. They employ a series of a MapReduce functions to perform RDFS closure. Like the Weaver and Hendler approach, their algorithm produces duplicates during processing; however, they account for that and have a specific stage dedicated for removal of duplicates. Their experiments were conducted on a 64 node machine with two dual-core 2.4GHz Opteron processors per node. It should be noted that they've shown results computing closure on 100 billion triples. Our approach, since we limit ourselves to in-memory computations, cannot reach that scale, at least given a similar number of processors. With a specialized version of the hash table code with memory optimizations not available in the more general implementation, we estimate the largest data set we could handle within 4 TB of memory to be about 20 billion triples. However, the next generation XMT system, due around the end of 2010, is expected to have as much as eight times as much memory per processor.

	Inferences/second
MPI (32 nodes)	$574e3$
WebPIE (64 nodes)	$\sim 700e3$
XMT (512 proc.)	$13.2e6$
XMT (256 proc.)	$11.9e6$
XMT (128 proc.)	$9.43e6$
XMT (64 proc.)	$6.50e6$
XMT (32 proc.)	$4.04e6$

Table 2. Listed above are the best rates achieved on LUBM data sets for various platforms, sizes, and approaches. MPI is the Weaver and Hendler approach [16]. The WebPIE number is an estimate for the RDFS closure portion of a larger OWL Horst calculation [14].

	Threadstorm processors to nodes
MPI	7.05
WebPIE	9.28

Table 3. This table reports the speedup achieved by the RDFS closure algorithm on the XMT versus the two top competing approaches.

Table 2 shows the rate of inferences/second each method achieves for varying platforms sizes. Table 3 shows the speedup in inferences/second we obtained over these other methods, again with the caveat that this comparison is inexact due to differences described above. We compare the number of *Threadstorm* processors to an equal number of Opteron cores (i.e. a socket-to-socket comparison). For both cases we compare against results on LUBM, though the generated sizes vary from 345 million to 100 billion. The numbers on LUBM for Urbani we take from their OWL paper [14], using the fact that the first computation in OWL inferencing for their algorithm is an RDFS closure. Since RDFS closure on LUBM data sets, regardless of size, generally produce $\frac{X}{4}$ inferences, where X is the original number of triples (at least for the set of rules in bold in Table 1), and using times reported in conversations with the authors for the initial RDFS closure stage, we calculate the estimated inference rate for this stage of their algorithm to be about 700 thousand inferences a second.

7 Conclusions and Future Work

We have presented an RDFS closure algorithm for the Cray XMT that achieves inference rates far beyond that of the current state of the art, about 7-9 times faster when comparing *Threadstorm* processors to commodity processor cores. Part of our approach's success is due to the unique architecture of the Cray XMT, allowing us to store large triple stores entirely in memory, avoiding duplication of triples and also costly trips to disk.

There are several obvious avenues for future work. In this paper we examined only artificially generated data sets with a relatively simple ontology. We would like to examine real-world data, and especially explore the effect that more complex ontologies have on our algorithm and make adjustments as necessary.

We also want to expand this work to encompass OWL semantics, probably focusing first on the Horst fragment [9], as it is among the most computationally feasible subsets of OWL. We believe the XMT will again be well suited to this problem, maybe even more so than RDFS. For instance, some rules in OWL Horst require two instance triples to be joined with one schema triple. This poses a challenge for distributed memory machines in that you can no longer easily partition the data as with RDFS, which had at most one instance triple as an antecedent. The XMT can store all of the instance triples in its large global memory, and create indices as needed to quickly find matching triples.

Also, there are some tricks to try that might help eliminate some of the constant overhead we see with file I/O (for a fixed number of service nodes) and data structure initialization. File I/O and initialization both require low numbers of threads, so it may be possible to overlay them, using a future to explicitly launch initialization just before loading the data.

One limitation of the approach presented in this paper is the use of fixed table sizes, forcing the user to have accurate estimates of the number inferences that may be generated. Using Hashing with Chaining and Region-based Memory Allocation (HACHAR), also presented in [4], may alleviate this requirement. It allows for low-cost dynamic growth of the hash table and permits load factors far in excess of the initial size; however, the current implementation exhibits poorer scalability than the open addressing scheme we used for this paper.

Finally, we wish to explore a fairer comparison between our work on a shared memory platform to a similar in-memory approach on distributed memory machines. The work presented by Weaver and Hendler [16] is the closest we can find to in-memory RDFS closure on a cluster, but the algorithmic differences make it difficult to draw definite conclusions. The *MapReduce-MPI Library* [6], a MapReduce inspired framework built on top of MPI, permits the development of entirely in-memory algorithms using standard MapReduce constructs. As such, it is an ideal candidate for producing a more comparable RDFS closure implementation for distributed memory platforms.

Acknowledgments. This research was funded by the DoD via the CASS-MT program at Pacific Northwest National Laboratory. We wish to thank the Cray XMT software development team and their manager, Mike McCardle, for providing us access to *Nemo*, the 512-processor XMT in Cray's development lab. We also thank Greg Mackey and Sinan al-Saffar for their thoughtful reviews of this paper.

[6] `http://www.sandia.gov/~sjplimp/mapreduce.html`

References

1. Anyanwu, K., Maduko, A., Sheth, A.P. SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In Proceedings of WWW. Banff, Alberta, Canada (2007)
2. Bader, D.A., Madduri, K. Designing Multithreaded Algorithms for Breadth-First Search and *st*-connectivity on the Cray MTA-2. In Proceedings of the 35th International Conference on Parallel Processing. Columbus, OH, USA (2006)
3. Chin, G., Marquez, A., Choudhury, S., Maschoff, K. Implementing and Evaluating Multithreaded Triad Census Algorithms on the Cray XMT. In Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing. Rome, Italy (2009)
4. Goodman, E.L., Haglin D.J., Scherrer, C., Chavarría-Miranda, D., Mogill, J., Feo J. Hashing Strategies for the Cray XMT. In Proceedings of the IEEE Workshop on Multi-Threaded Architectures and Applications. Atlanta, GA, USA (2010)
5. Guo, Y., Pan, Z., Heflin, J. LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3) (October 2005) 158-182
6. Jin, S., Huang, Z., Chen, Y., Chavarría, D.G., Feo, J., Wong, P.C. A Novel Application of Parallel Betweeness Centrality to Power Grid Contingency Analysis. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium. Atlanta, GA (2010)
7. Kochut, K.J., Janik, M. SPARQLeR: Extended Sparql for Semantic Association Discovery. In Proceedings of the 4th European Semantic Web Conference. Innsbruck, Austria (2007)
8. Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In Proceedings of the Workshop on Algorithm Engineering and Experiments. New Orleans, LA, USA (2007)
9. ter Horst, H.J. Completeness, Decidability, and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL Vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3) (October 2005) 79-115.
10. Pérez, J., Arenas, M., Guiterrez, C. nSPARQL: A Navigational Language for RDF. In Proceedings of the 7th International Semantic Web Conference. Springer, Karlsruhe, Germany (2008)
11. RDF Semantics, W3C Recommendation, 10 Februrary 2004, `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In Proceedings of WWW. Beijing, China (2008)
13. Urbani, J., Kotoulas S., Oren, E., van Harmelen, F. Scalable Distributed Reasoning using MapReduce. In Proceedings of the 8th International Semantic Web Conference. Springer, Washington D.C., USA (2009)
14. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In Proceedings of the 7th Extended Semantic Web Conference. Heraklion, Greece (2010)
15. Urbani, J., Maaseen, J., Bal, H. Massive Semantic Web data compression with MapReduce. In Proceedings of the MapReduce workshop at High Performance Distributed Computing Symposium. Chicago, IL, USA (2010)

16. Weaver, J., Hendler, J.A. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In Proceedings of 8th International Semantic Web Conference. Washington, DC, USA (2009)

SPARQL to SQL Translation Based on an Intermediate Query Language

Sami Kiminki, Jussi Knuuttila, and Vesa Hirvisalo

Aalto University School of Science and Technology

Abstract. We present a structured approach to SPARQL to SQL translation using AQL—a purpose-built intermediate query language. The approach produces a single SQL query for a single SPARQL query. Using AQL, we revisit the semantic mismatch between SPARQL and SQL and present query transformations on AQL presentation which enable the correct translation of some difficult corner cases. By using explicit expression type features in AQL, we also present type inference for expressions. We demonstrate the benefit of type inference as a basis for semantically correct optimizations in translation.

1 Introduction

We present a flexible approach to translating SPARQL queries into SQL queries, and discuss the properties of the approach based on our experimental implementation.

SPARQL is a query language for RDF graphs [14]. RDF data consists of triples expressing relationships between nodes. RDF is semi-structured, in that it does not imply a schema for storage. On one hand, this makes RDF a very flexible mechanism and suitable for representing, *e.g.*, web-related meta-data or other arbitrarily structured information. On the other hand, making efficient queries to access such data is not easy.

Even if native RDF stores are arguably more promising in the long run, there exists a massive amount of data stored in SQL databases with associated technology, infrastructure and know-how. This cannot be ignored in discussions on large scale adoption. During the transition, it is attractive to consider storing the RDF data in SQL databases parallel with non-RDF data in existing systems. The existing data can then be provided as virtual RDF graphs to applications [4], providing unified access to all data.

We address query translation from SPARQL into SQL that enables the use of SQL databases with flexible storage schemas. Storing RDF data in an SQL database is not trivial. For example, there is no single SQL layout for RDF data that is the best in all cases [1, 13]. Because of this, a flexible approach where the SQL layout can be tuned on per-application basis is preferable. Similarly, query translation is not easy, as SQL and SPARQL differ significantly, and for some corner cases, even getting semantically correct translation is hard [6]. Further, producing SQL queries that can be executed efficiently by the SQL database is difficult.

Our approach to the translation is to produce a single SQL query for a single SPARQL query without the need for result post processing, except the presentation. Also, an important design goal has been to produce SQL with the support for using native SQL data types where possible and avoiding subselects. This approach minimizes the amount of communication round-trips and leaves more optimization opportunities for the SQL back-end [11].

To obtain these goals, we devised a translation design based on an intermediate language that we call AQL (Abstract Query Language). As is typical for intermediate languages, it has straightforward basic semantics and has the ability to attach information to support translation. Such properties make it easy to find a translation from the source language into AQL and enable finding efficient translations from AQL to the target language.

AQL targets only the query semantics for translation. It does not address other issues, such as the representation of results like many concrete query languages do (including SPARQL and SQL). AQL has been designed especially in the context of SPARQL to SQL translation. It is used to address the translation of queries into semi-structured data into relational database queries in general.

AQL has language features that enable the use of type information to support translations. Similar type-based static analysis and translation mechanisms have been used for programming languages. We demonstrate how such a methodology can be applied to support the translation of SPARQL queries into SQL queries.

We have created an experimental implementation of our translation approach to test its properties. Our implementation, *Type-ARQuE* (Type-inferring AQL-based Resource Query Engine) [15], is an optimizing SPARQL to SQL query translator. It supports the most important SPARQL language features in order to validate the design. Type-ARQuE is written in C++ and supports PostgreSQL and MySQL back-ends with different database layout options.

Based on the implementation, we show how some demanding cases of RDF queries can be translated into efficient SQL queries. Our demonstration cases underline the challenges raised by alternate variable bindings, variable scoping and determining the required value joins. Especially, we demonstrate how determining the required value joins can benefit from type inference.

We review some of the related background in Sec. 2. The translation design is covered in Sec. 3, containing an overview of the AQL language and the steps of the translation of SPARQL into SQL by using AQL as an intermediate. We give special attention to the use of AQL type information, as type inference is essential for our translation. In Sec. 4 we illustrate the translation with concrete examples. We continue by discussing the properties of our translation (Sec. 5) and end with brief conclusions (Sec. 6).

2 Background

SPARQL[14] is a query language for RDF graphs. It is an official W3C recommendation. SPARQL has syntactic similarity to SQL but with some important

differences. Whereas in SQL the query data set is specified by joining tables (`FROM` and `JOIN` clauses), graph match patterns are used in SPARQL.

Relational databases are an important back-end option for storing RDF graphs due to the wide user base of relational databases. A variety of SQL layouts for storing RDF graphs have been suggested, but it seems that no single layout is good for all purposes [1, 13].

Harris was one of the first to systematically consider SPARQL to SQL translation discussing various ways of organizing RDF triple stores and considers especially using SQL back-ends [8]. The opportunity for a number of optimizations is acknowledged and the problem underlined as nontrivial.

To our knowledge, one of the most fundamental works on the problem domain is presented in [6]. The technical report explains the SPARQL algebra with discussion on how to map the SPARQL algebra to the traditional relational algebra. In the report, difficult corner cases of translation are also analyzed.

A rather flexible translation approach is presented in [11]. They consider SPARQL query translation into SQL queries by using a facet-based scheme that is designed to handle filter expressions. They underline that it is desirable that a single SPARQL query is translated into a single SQL statement, and that comparison between results of different data types is useful. They also consider optimization strategies to reduce complexity of translated queries.

Hartig and Heese considered optimization by query translation at SPARQL algebraic level [9]. Their approach was based on translating the query in Jena ARQ into a custom representation (SQGM) for optimization, and then translating back into ARQ.

Lately, Chebotko et al. presented a method for translating a SPARQL query to a single SQL query with preservation of semantics [5]. Their method operates on SPARQL algebraic level, and relies on SQL subqueries on data set declaration.

Left-to-right variable binding semantics are an alternative to bottom-up semantics. This changes variable scoping, enabling queries containing filters in nested graph groups that depend on variables bound by their parents. For a discussion, see [6]. SPARQL utilizes currently the bottom-up semantics.

3 The Translator Design

The query translator in Type-ARQuE translates SPARQL queries into SQL. The translator was designed with three main goals in mind. First, the translator should produce a single SQL query for a single SPARQL query. Second, the type support of the SQL back-end should be utilized. Finally, the translator should not be fixed to some specific SQL schema and SQL dialect.

The translator is based on a multi-stage translation architecture [2], consisting of front-end, intermediate, and back-end translation stages. The front-end parses and translates SPARQL queries into intermediate queries (Sec. 3.2). The intermediate query language, AQL (Sec. 3.1), is specifically designed to stand between SPARQL and SQL. The intermediate translation stage (Sec. 3.3) consists of general query transformation and optimization passes (general preparation)

as well as back-end specific transformation passes (specialization), and utilizes type inference (Sec. 3.5). Finally, the translator back-end translates the AQL query into SQL using a specific target dialect (Sec. 3.4).

A main design goal for AQL, the Abstract Query Language, was to be compact with straightforward but high-level semantics. It is relational in nature. The join expressions in AQL are extended from the traditional relational algebra to cover both SPARQL and SQL join semantics.

3.1 The Abstract Query Language

AQL is an intermediate query language, representing the query semantics. In other words, it does not cover representation of the query results, such as SELECT vs. CONSTRUCT forms in SPARQL. Its intended use is machine-only. We begin by introducing the language and then defining the query evaluation constructs and semantics, and finally, we consider expressions in AQL.

An AQL query is represented by a query object, which contains the data set declaration, sort orders, result slicing, and select expressions. The data set declaration consists of a join tree, where each node contains a set of join names, possible child nodes, and join criteria. The data set declaration specifies the data that is used in the query process. The data is a list of query solutions, represented by a 2-dimensional array, columns as solution components and rows as different solutions. Order expressions specify the ordering of the data set. Result slicing (LIMIT and OFFSET) selects a specific range of rows. Select expressions are projections of a solution row to singular values, comparable with SQL select expressions.

A node in a join tree represents joining one or more columns to the data set using the attached join criteria. Child nodes represent nested joins. Each node may be of type INNER or LEFT OUTER join. The columns are named by the join names.

Joins manipulate the rows in the data set. Conceptually, this begins with creating a temporary data set by taking all the triples in the store, and raising the set to the Cartesian power of the number of triple names in the join node. The temporary data set is then joined to the result data set by Cartesian product. Then, the child nodes, if any, are joined. Finally, each join criterion is evaluated per row. The row is eliminated unless it meets all criteria. It is possible that a row that was originally in the result data set is eliminated if all new combinations fail to meet the criteria. In that case, if the join is LEFT OUTER join, the original row is retained and nulls are inserted to new columns.

There are differences in the join processes between the relational algebra and AQL, namely, in the order of operations. In the relational algebra, the data set is created by processing the nested joins first. Joining a table (or a set of tables) and evaluating the join condition is an atomic operation [7]. In AQL, a top-down approach is used instead. The child joins are joined to the parent recursively, and only after that the join criteria are processed. The AQL approach enables referencing more columns in the join criteria than what is possible in the SQL approach. In SQL, the child joins and parent may be referenced in the join

criteria. In AQL, the parent (recursive) and every earlier node using in-order join tree traversal may be used, in addition. This extension covers both the SPARQL and SQL semantics.

We now define the AQL and its evaluation semantics, borrowing notation used in SPARQL evaluation definition [14] where applicable.

The query object is defined as follows:

```
(aql-query join-name-group
           join*
           (criterion <expr>)
           sort-expr*
           (result-max-rows <integer>)?
           (result-row-offset <integer>)?
           select-expr*
           distinct?)
```

The parts are:

join-name-group — The join names for the root join node.
join* — Any number of join expressions.
criterion — The filter expression for the data set.
sort-expr — Any number of sort expressions defining the sequence for data set enumeration.
result-max-rows, result-row-offset — Slice specifiers.
select-expr* — The select expressions.
distinct? — Optional result modifier.

A join expression represents joining the set of all triples in the store one or more times into the data set. The join expression is defined as:

```
(join join-type join-name-group expr join*)
```

And the parts:

join-type — The join type, either `INNER` for inner join or `LEFT` for left outer join.
join-name-group — The set of join names.
expr — The join condition expression.
join* — Any number of nested joins.

The `aql-query` object, join expressions and AQL query criteria form a join tree which specifies the data set. The data set is a 2-dimensional array, consisting of solutions as rows and triples as named columns. For the query result data set definition, let T be the set of all triples in the store and I the identity for Cartesian product (I is an empty array of 1 rows, 0 columns).

The data set for a join subtree is produced by function `JoinNode`(D_0, n) where the current data set is denoted as D_0 and the root node of the subtree as n. Return value denotes the data set after joining n to D_0. The following steps define the evaluation of `JoinNode`(D_0, n):

1. Join the Cartesian product specified by join name group of the node:
   $D_r \leftarrow D_0 \bowtie D_1$, where $D_1 = T_{jn_1} \times \cdots \times T_{jn_N}$
2. Join children of n in left to right order into the data set D as:
   $D_r \leftarrow$`JoinNode`$(\cdots($
   `JoinNode`(`JoinNode`($D_0 \bowtie D_1$, n.child1), n.child2), $\cdots$n.childN)
3. Apply filter:
   $D_r \leftarrow filter(D_r, n.expr)$
4. If the join type of n is LEFT, add row d,ω into D_r for each row d removed from D_0 by the filter expression, where ω contains `null`s to fill the join columns.
5. Return D_r

The result data set D for the query is produced by `JoinNode`(I, `aql-query`). Fig. 1 illustrates the result data set with query result.

After the result data set is formed, order expressions are used to sort the data set. The first order expression is evaluated for each row and then the rows are ordered so that the rows with smaller order value are enumerated first in ascending order or vice versa in descending order. If two or more rows have the same order value, the second order expression is used to determine the ordering between these rows, and so on. The sort expressions are of form:

```
(sort ascending|descending <expression>)
```

Finally, select expressions are applied for each row in the sorted data set. The set of select expressions for a row produces a result row. When the select expressions are applied for each row in the data sequence, the result sequence is produced. In AQL, select expressions are of form:

```
(select <column-label> <expression>)
```

After the result sequence is produced, if the `distinct` modifier is set, all duplicate rows are eliminated.

The expressions in AQL are of three categories: typed literals, triple property expressions, and function expressions. Function and property expressions are assigned a set of possible types. The expression templates are below:

```
(literal <type> <value>)
(property <type-set> <join-name> subject|predicate|object)
(function <function-name> <type-set> <param-expr>*)
```

`literal` specifies a typed literal, such as 5 (int) or 'abc' (string). `property` specifies access to the subject, the predicate, or the object of a named triple in a solution. `type-set` in `property` makes an assumption of the property type: the type must belong to the assumed type set. `function` represents evaluation of a function returning a value of a type belonging to the type set.

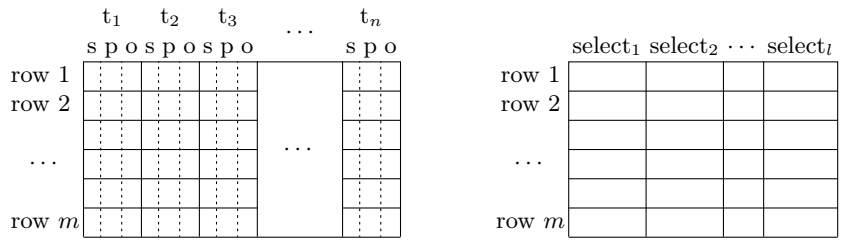


Fig. 1. Illustration of a solution set (left-hand table) and the query result (right-hand table) of an AQL query as a 2-dimensional arrays. Triple columns are added by in-order traversal of join groups, one column and three sub-columns (subject, predicate, object) per triple name. The rows are inserted and filtered by running the data set production algorithm described in Sec. 3.1. Each row represents a single solution to the query. Afterwards, the rows are ordered and sliced, and finally, the select expressions are evaluated per solution row to produce the query result array.

3.2 SPARQL to Abstract Query Language

The translation into AQL is mostly straightforward. First, the SPARQL query is parsed into an abstract syntax tree. After that, variable accesses are checked to conform to bottom-up binding unless left-to-right semantics are enabled. In bottom-up binding, variable may be used in FILTER expression if it is bound by a triple match pattern in the same or a nested graph group. Then, the abstract syntax tree is normalized by merging non-optional SPARQL graph groups with their parents.

For each graph group an AQL join node is created and unique join names are assigned to the triple match patterns in the graph group. Then the join names are inserted into the respective AQL join groups. The graph group hierarchy is naturally preserved in the AQL join group hierarchy.

Variable binding step addresses the mapping of SPARQL variable bindings to AQL property and function expressions. The SPARQL semantics require that variables are bound by the first matching triple match pattern. As non-optional match patterns always bind a variable, that variable can be mapped to the property of the triple join corresponding to the first non-optional match pattern. However, optional match patterns introducing variables require a bit more consideration.

Before a variable is encountered in a non-optional match pattern, the variable may be bound by optional graph groups containing a match pattern mentioning the variable. In this case, coalesce-expressions are used to select the first value-binding triple match visible at point of access.

After the variables are bound to property expressions, join conditions are constructed by triple match patterns. Literals in match patterns impose constraints to triple join properties. If a match pattern introduces a new variable, no condition is rendered. If a variable is already introduced, it is required that the property of the respective triple join is equal to the variable, if the variable

was bound. When it is not certain that the variable is bound, additional not-null conditions are added for non-optional match patterns. After translating match pattern conditions, filter expressions are translated as additional join conditions. Thus, match patterns and filters are unified.

Finally, selects and order expressions are translated. This completes the SPARQL to AQL translation.

3.3 Translation Passes on Abstract Query Language

The translation passes prepare an AQL query for SQL translation. The process is called *lowering*. Lowering consists of *general preparation* and *specialization* parts. General preparation is a sequence of generic transformation passes which simplify the AQL, and makes it easier to translate. Specialization part perform back-end-specific transformations such as replacing AQL property expressions with SQL access expressions for a specific layout. We list the passes in both parts briefly below.

General preparation:

- Inner join merge — joins inner joins with parents to simplify the query.
- Logical expression normalization — moves not-expressions inwards using De Morgan's laws and fuses not-expressions with comparisons, *e.g.*, $\neg(a > b \wedge c = d) \rightarrow a \leq b \vee c \neq d$
- Operators to functions — transforms comparison, typecast, and logical operators to equivalent function expressions.
- Type inference — infers possible types for expressions. Described in more detail in Sec. 3.5.
- Empty type sets to nulls — replaces possible empty type sets with null expressions. An expression with conflicting type requirements may only produce a null.
- Nested join flattening — transforms deep nested joins with many-levels-up accesses to a less deep form. Exemplified in Fig. 4.
- Comparison optimization — transforms equality and non-equality value comparisons to reference comparisons.
- Function variant selection — chooses the most appropriate variants of polymorphic functions in expressions.

Specialization:

- Property value requirer — adds not-null conditions whenever property expression could produce null. Null can be produced if value table(s) must be left-joined or typecast must be used to access value of a triple property.
- Property access resolver — rewrites property access expressions with lower-level back-end-specific equivalents. In the back-end-specific accesses table names, value and index columns are resolved and the required typecasts and `COALESCE`s are added whenever needed.
- Expression optimization — simplifies various expressions and performs common subexpression elimination. Having an explicit clean-up pass simplifies some of the previous passes.

– Function variant selection — this pass is run again to ensure that all variants of functions have been chosen after transformations.
– Typecast injection — inserts typecasts wherever needed in the expressions.
– Property access collection — collects all low-level property accesses. This is used to determine which value-joins are required in the final SQL.

3.4 Translation to SQL

After an AQL query is lowered, the translation into SQL is straightforward. The arrangement resembles code generation in compilers [3]. The AQL expressions in selects, orders, join conditions, and the query criteria are translated into SQL by traversing the expression trees.

The AQL join tree is then transformed into an SQL join tree with value joins and join expressions attached. For multiple triple joins in AQL join group, cross join is used in SQL. The SQL join tree is then serialized into string form.

Finally, all the translation results are inserted into an SQL query template. This completes the SPARQL to SQL translation process.

3.5 Type Inference

In our approach, type inference for all expressions is performed in AQL after the join tree and its expressions are normalized. The inferred information is used, *e.g.*, to optimize value accesses in complex SQL schemas.

We utilize equations on two levels to resolve the possible types for property and function expressions. The lower-level equations infer the possible types in the join conditions. The higher-level equations transfer the inferred type constraints between join conditions in the join tree.

Throughout this section, p denotes the property, *i.e.*, the triple component (subject, predicate, or object), S is the finite set of all known types and $C_{n,p} \subset S$ is the set of the possible types of a property p and expression node n. For example, $C_{n,\mathrm{t5.object}} = \{\mathrm{string}, \mathrm{int}\}$ specifies that the object of the named triple t_5 may only be of type string or int. n is the expression node (*i.e.*, function, literal or property expression) that scopes this constraint.

Type Inference on Join Condition Expressions. The normalized expression tree may contain function, property and literal expression nodes. Property and literal expression nodes are always leaf nodes. The children of function nodes represent function parameters.

For every expression node n, a set of possible expression types $R_n \subset S$ and a set of property type constraints $C_{n,p}$ are assigned. Then, based on expression node types, conditions to R_n and $C_{n,p}$ between adjacent nodes are set, as:

– if n is a literal of type t: $R_n = \{\mathrm{t}\}$, $C_{n,p} = C_{parent(n),p}$
– if n is a property expression p: $R_n = C_{n,p}$, $C_{n,p} = C_{parent(n),p}$

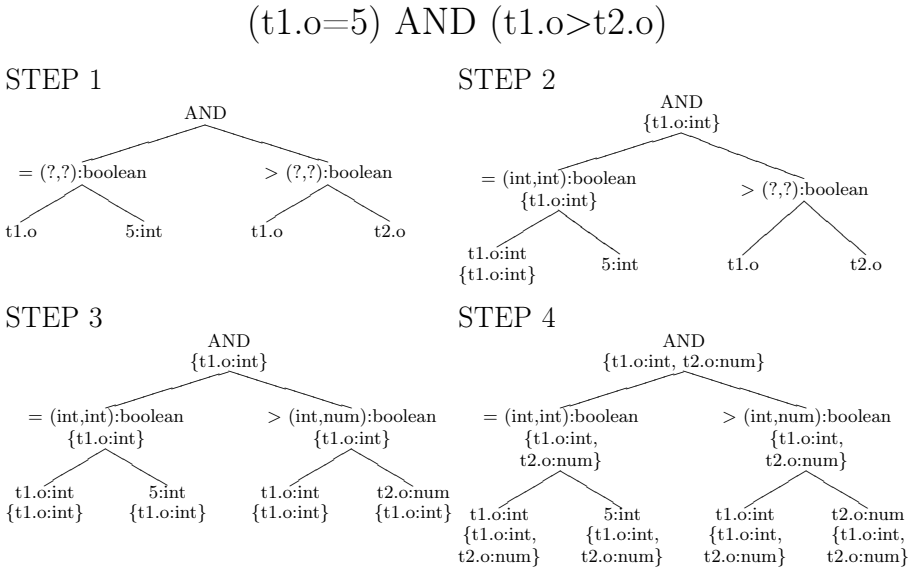


Fig. 2. Illustration of type inference for a simple expression. Using the inference procedure described in Sec. 3.5, the set of possible types (in braces $\{\cdots\}$, '?' represents all possible types) are reduced by analyzing expression types and function signatures, and propagating the sets using transfer functions. Property t1.o has been inferred as integral type and t2.o as general numeric type.

- if n is an expression node for the function f and m_i are the parameter nodes. Then, $R_n = F_f(R_{m_1}, R_{m_2}, \ldots)$ where F_f maps the set of possible parameter types to a set of possible return types. The constraint set propagation depends on the type of the function f as follows. If f is
  - or: $C_{n,p} = C_{parent(n),p} \cap (\bigcup_i C_{m_i,p})$
  - not: $C_{n,p} = C_{parent(n),p}$
  - any other function: $C_{n,p} = C_{parent(n),p} \cap (\bigcap_i C_{m_i,p})$

Expression type inference is illustrated in Fig. 2.

Constraint Propagation between Joins. The constraints are transferred between the roots of join condition expressions, as the effective Boolean value of the expression root determines whether the join contributes to the query solution. In the join tree, the constraint transfer between any two nodes must follow the following two simple rules:

1. **Every inferred type constraint of node n applies to every child node m_i of n**, *i.e.*, $C_{m_i,p} \subset C_{n,p}$. This is because the nested joins cannot contribute to the solution if n is null, and therefore, we can assume that inferred type restriction applies also to the nested joins without problems.

2. **Every inferred type constraint made in n for a property of a triple defined by n or some of its children applies everywhere.** This is because the property value can be non-null only if n is not null.

Note that by rule 1, all constraints inferred for the root join apply everywhere.

Solving the Equations. We take a conservative approach by assuming initially that objects of triples may be of any type. Subjects and predicates may only be IRIs in RDF graphs. Then we exclude non-viable type alternatives, *i.e.*, those that cannot appear. The exclusion is performed by iterating the two steps below until a fixed point is found. The steps correspond to the higher and lower level type inference equations. As the sets of viable type alternative constraints are initially finite and may only shrink, the fixed point is guaranteed to be reached. The descriptions of exclusion steps:

1) Condition expression step. First, all function expressions are analyzed in pre-order, separately for all join condition expressions. Based on parameter types and constraints on the return type the union of the viable function variants are computed. This may imply new constraints on function parameters, *e.g.*, function `a=abs(b)` requires that `b` is numeric. If function parameters are property expressions, the type set $C_{n,p}$ of the property is shrunk to reflect function parameter requirements.

After the constraint sets of properties are shrunk, they are propagated in post-order to adjacent expression nodes using the transfer rules described in *Type Inference on Join Condition Expression* above. Finally, every constraint set is intersected by constraints in expression root node.

2) Constraint propagation step. After the possible type sets in join condition expressions are shrunk, the sets are propagated between join conditions. This is straightforward using the two rules in *Constraint Propagation between Joins*.

4 Translation Examples

In this section, we demonstrate some of the techniques utilized by our query compiler. *Alternate binders* demonstrates a translation where a variable may bound by two alternate optional match patterns. *Two levels up access* demonstrates query flattening in AQL. This is required when there is a reference to a variable which is bound by a graph pattern residing two or more levels up in the join tree. *Expression type inference* demonstrates how type inference is used to eliminate superfluous joins to value tables in faceted storage layouts. The examples are translated using Type-ARQuE using the PostgreSQL target. *Alternate binders* and *two levels up access* are motivated by the corner cases discussed in [6].

In examples *alternate binders* and *two levels up access*, the triple graph is stored in a single table, `InlinedTriples`, consisting of three columns for subjects, predicates and objects. In *expression type inference* we use a central triple

table, `VI_Triples`, which consists of three index columns. The columns refer to value tables containing the actual property values, such as `VI_Strings` and `VI_BigStrings` for string-typed values.

Alternate binders query (Fig. 3) has two optional graph patterns, both which may bind variable `d`. We call this the alternate binders case. The variable dereference expression depends on where `d` is accessed. In the first optional match, we may assume that `d` is bound there. In the second match there are two alternatives: 1) the first match binds and the triple object must equal to this, or 2) the first pattern did not match and equality comparison should be omitted. In `SELECT` part, `COALESCE`-expression is used to select the binding expression.

Two levels up access example (Fig. 4) demonstrates working around the "two-levels-up" variable access by transforming the join tree. The nested join is translated as a sibling join with additional condition requiring that the former parent must match. In the figure, the join tree of the AQL is displayed before and after flattening. In this example, the extended left-to-right variable binding semantics are assumed. The query is not proper for bottom-up semantics, as in this case, the variable `c` is inaccessible in the filter.

In *Expression type inference* (Fig. 5) we demonstrate the use of type inference to determine the required value joins when using a facet-based storage layout. Without type inference, when dereferencing a variable, all value tables must be joined which can be costly. Using type inference, the number of joins and the complexity of respective coalesce expressions can often be reduced significantly.

5 Discussion

Our translation approach is somewhat different to the approach in popular SPARQL-enabled RDF stores, such as Jena/SDB[10] and Sesame[12], in that we use a purpose-built intermediate language in the translation. The use of an intermediate language enables an approach for translation using relatively small and straightforward passes.

AQL provides a new look into the problem of mismatching semantics between SPARQL and SQL, especially when left-to-right variable binding semantics are used. This is because AQL is more explicit than SPARQL algebra regarding query evaluation, and AQL is less complex than SPARQL algebra in what comes to the language features.

Contrary to SPARQL, AQL does not have query variables. In our approach, variables are translated into triple property access expressions, which refer to parts of the query solutions. Resolving variables to property access expressions is fairly straightforward and eliminates tedious translation problems related to variables altogether. Variables are especially difficult to translate directly into SQL expressions, as they may be bound by different parts of SPARQL query. We demonstrated this in the alternate binders case in Sec. 4.

When using the extended left-to-right variable binding, the "FILTER scope" problem poses a difficult corner case for translation [6]. The naïve SPARQL to SQL translation always fails, because in SQL, the variable bound by a graph

```
SELECT ?a ?d
WHERE {
  ?a ?b ?c
  OPTIONAL { ?a <http://test/surname> ?d }
  OPTIONAL { ?a <http://test/lastname> ?d }
}

SELECT tri_1_1.subj_value AS c0,
  COALESCE(tri_2_1.obj_value,tri_3_1.obj_value) AS c1
FROM InlinedTriples AS tri_1_1
LEFT JOIN InlinedTriples AS tri_2_1 ON
  tri_2_1.subj_value=tri_1_1.subj_value AND
  tri_2_1.pred_value='http://test/surname'
LEFT JOIN InlinedTriples AS tri_3_1 ON
  tri_3_1.subj_value=tri_1_1.subj_value AND
  tri_3_1.pred_value='http://test/lastname' AND
  (tri_2_1.obj_value IS NULL OR
   tri_3_1.obj_value=tri_2_1.obj_value)
```

Fig. 3. Variable `d` is bound by the first matching optional graph pattern. Variable dereference is translated as coalesce of alternate binders. In the latter optional graph pattern, the case where the first graph pattern binds `d` is taken into account by inserting additional null or equals condition.

pattern is not available in an optional join at a nesting distance of two or more levels. In our approach, in these cases the join tree is flattened by a semantically equivalent transformation, reducing the set of untranslatable queries. As a side product of variable elimination and "FILTER scope" workaround, the "Nested `OPTIONAL`s" problem in [6] is also remedied.

Using the left-to-right semantics, there are queries that are structurally untranslatable by our method. This class consists of queries where a parent graph group refers to a variable in nested graph group with a two-levels-up access. We present an archetype of this class, which we call the *2-up-1-down-access* query:

```
SELECT *
WHERE { ?a ?b ?c
        OPTIONAL { ?d ?e ?f
                   FILTER(?i='abc')
                   OPTIONAL { ?g ?h ?i
                              FILTER(?c='def') } } }
```

Within this class of queries, join tree flattening produces access expressions with forward references in the join conditions. In SQL, this is illegal. However, we believe that most of the practical queries do not belong to this class.

The data set construction semantics in AQL can be considered as a process of self-joining the set of all triples in the store with join conditions instead of applying triple match patterns. This unifies the handling of triple match patterns and filters. This also enables a clean isolation of triple store layout from the rest of the translation, as we can consider all the triples to reside in one virtual table, and translate the accesses to the virtual triple table to concrete layout-specific tables and columns.

```
SELECT ?a ?d ?e ?g
WHERE { ?a ?b ?c
 OPTIONAL { ?a ?d ?e
            OPTIONAL { ?e ?f ?g FILTER (?c=45.1) } } }
```

```
(aql-query ("tri_1_1") ...   # before flattening
  (join left ("tri_2_1")
    (function "builtin:comp-eq" (boolean)
              (property (IRI) "tri_2_1" subject)
              (property (IRI) "tri_1_1" subject))
    (join left ("tri_3_1")
      (function "builtin:and" (boolean)
        (function "builtin:comp-eq" (boolean)
          (property (IRI) "tri_3_1" subject)
          (property (IRI) "tri_2_1" object))
        (function "builtin:comp-eq" (boolean)
          (property (double) "tri_1_1" object)
          (literal double 45.1)))    ...)
```

```
(aql-query ("tri_1_1") ...   # after flattening
  (join left ("tri_2_1")
    (function "builtin:comp-eq" (boolean)
       (property (IRI) "tri_2_1" subject)
       (property (IRI) "tri_1_1" subject)))
  (join left ("tri_3_1")
    (and (function "builtin:and" (boolean)
      (function "builtin:comp-eq" (boolean)
        (property (IRI) "tri_3_1" subject)
        (property (IRI) "tri_2_1" object))
      (function "builtin:comp-eq" (boolean)
        (property (double) "tri_1_1" object)
        (literal double 45.1)))
      (function "builtin:is-not-null" ANY
        (property (reference) "tri_2_1" subject))))
...)
```

```
SELECT tri_1_1.subj_value AS c0, tri_2_1.pred_value AS c1,
       tri_2_1.obj_value AS c2, tri_3_1.obj_value AS c3
FROM InlinedTriples AS tri_1_1
 LEFT JOIN InlinedTriples AS tri_2_1 ON
   tri_2_1.subj_value=tri_1_1.subj_value
 LEFT JOIN InlinedTriples AS tri_3_1 ON
   tri_3_1.subj_value=tri_2_1.obj_value AND
   aqltosql_any_to_double(tri_1_1.obj_value)=45.1 AND
   tri_2_1.subj_value IS NOT NULL
```

Fig. 4. The "two-levels-up" variable access in filter condition is flattened by one level by a join tree transformation. The join `tri_3_1` is moved down to the same level with `tri_2_1` with additional condition that `tri_2_1` needs to be non-null. The original semantics are retained but the query becomes translatable into valid SQL. Non-interesting parts of the AQL queries are pruned away for brevity. This query requires the extended left-to-right variable binding semantics.

```
SELECT ?a ?c
WHERE { ?a ?b ?c. FILTER (?c=55 || ?c='David') }
ORDER BY (?c)

SELECT tri_1_1_subj_VC_IRIs.iri_value AS c0,
  COALESCE(tri_1_1_obj_VC_Strings.str_value,
    tri_1_1_obj_VC_BigStrings.text_value,
    CAST(tri_1_1_obj_VC_Integers.int_value AS TEXT)) AS c1
FROM VC_Triples AS tri_1_1
 LEFT JOIN VC_Strings AS tri_1_1_obj_VC_Strings ON
   tri_1_1_obj_VC_Strings.id=tri_1_1.obj
 LEFT JOIN VC_BigStrings AS tri_1_1_obj_VC_BigStrings ON
   tri_1_1_obj_VC_BigStrings.id=tri_1_1.obj
 LEFT JOIN VC_Integers AS tri_1_1_obj_VC_Integers ON
   tri_1_1_obj_VC_Integers.id=tri_1_1.obj
 INNER JOIN VC_IRIs AS tri_1_1_subj_VC_IRIs ON
   tri_1_1_subj_VC_IRIs.id=tri_1_1.subj
WHERE
  (tri_1_1_obj_VC_Integers.int_value=55 OR
   COALESCE(tri_1_1_obj_VC_Strings.str_value,
     tri_1_1_obj_VC_BigStrings.text_value)='David') AND
  (tri_1_1_obj_VC_Strings.str_value IS NOT NULL OR
   tri_1_1_obj_VC_BigStrings.text_value IS NOT NULL OR
   tri_1_1_obj_VC_Integers.int_value IS NOT NULL)
ORDER BY COALESCE(tri_1_1_obj_VC_Strings.str_value,
  tri_1_1_obj_VC_BigStrings.text_value,
  CAST(tri_1_1_obj_VC_Integers.int_value AS TEXT)) ASC
```

Fig. 5. Type inference is especially useful with faceted storage layouts. This becomes obvious when observing translation of variable `c`. It is inferred that `c` must be either integer or string for any solution to the query. Therefore, only string and integer value tables are required to be joined to obtain value for `c`. As variable `a` is used in subject, it is inferred as IRI.

6 Conclusion

We have presented an approach for SPARQL to SQL translation. The translation produces a single SQL query for a single SPARQL query, and does not rely on SQL result post-processing, except in data presentation. This approach reduces the amount of communication round-trips to the SQL server and allows the SQL server do more optimizations.

The translation is structured into three stages (front-end, intermediate, back-end) and the stages themselves are subdivided into self-contained passes. The intermediate stage operates on a purpose-built intermediate language, AQL. Using AQL, we have presented a way to target the queries for different SQL layouts for RDF data, and a strategy for query optimization.

As a basis for optimizations, we introduced type inference and provided an algorithmic view on its implementation. The implementation relies only on the constraints derived from the SPARQL query. Type inference is used to optimize expressions and SQL accesses. Ontology-awareness would likely enhance type inference in many practical scenarios, as, *e.g.*, known predicate of a triple often con-

strains type possibilities of the object. For triple with predicate `foaf:homepage`, we would expect IRI as the object type, for instance.

Using the extended join semantics of AQL, we provided intermediate-level query transformations for reducing the mismatch between SPARQL and SQL join semantics. The transformations enable translation of the corner cases presented in [6]. However, there is still a class of SPARQL queries that remain untranslatable by our approach, when the extended left-to-right variable binding semantics are used. Of this class, we presented a representative archetype.

To validate our design, we implemented Type-ARQuE, an experimental translator based on the presented design. The translator covers a representative subset SPARQL and demonstrates the translation in detail.

Acknowledgments. This work was supported by the DIEM project, funded by TEKES/Finland and the partner organizations of DIEM.

References

1. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: 33rd international conference on Very large data bases. pp. 411–422. VLDB Endowment (2007)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers—Principles, Techniques, & Tools. Addison-Wesley, 2nd edn. (2007)
3. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers (2002)
4. Bizer, C., Seaborne, A.: D2RQ — treating non-RDF databases as virtual RDF graphs (poster). In: ISWC (2004)
5. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data and Knowledge Engineering 68(10) (Oct 2009)
6. Cyganiak, R.: A relational algebra for SPARQL. Tech. rep., Hewlett-Packard (2005)
7. Groff, J., Weinberg, P.: SQL: The Complete Reference. McGraw-Hill, 2nd edn. (2003)
8. Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. In: Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q. (eds.) WISE Workshops, LNCS, vol. 3807, pp. 235–244. Springer, Heidelberg (2005)
9. Hartig, O., Heese, R.: The SPARQL query graph model for query optimization. In: The Semantic Web: Research and Applications. LNCS, vol. 4519, pp. 564–578. Springer, Heidelberg (2007)
10. Jena 2.6.2, `http://openjena.org/`
11. Lu, J., Cao, F., Ma, L., Yu, Y., Pan, Y.: An effective SPARQL support over relational databases. In: SWDB-ODBIS 2007. LNCS, vol. 5005, pp. 57–76. Springer, Heidelberg (2008)
12. Sesame 2.3.1, `http://www.openrdf.org/`
13. Sintek, M., Kiesel, M.: RDFBroker: A signature-based high-performance RDF store. In: The Semantic Web: Research and Applications. LNCS, vol. 4011, pp. 363–377. Springer, Heidelberg (2006)
14. SPARQL Query Language for RDF, `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
15. Type-ARQuE 0.2, `http://esg.cs.hut.fi/software/type-arque/`

Towards a better insight of RDF triples Ontology-guided Storage system abilities

Olivier Curé[1], David Faye[1,2], Guillaume Blin[1]

[1] Université Paris-Est, LIGM - UMR CNRS 8049, France
{ocure, gblin}@univ-mlv.fr
[2] Université Gaston Berger de Saint-Louis, LANI, Sénégal
dfaye@igm.univ-mlv.fr

Abstract. The vision of the Semantic Web is becoming a reality with billions of RDF triples being distributed over multiple queryable endpoints (e.g. Linked Data). Although there has been a body of work on RDF triples persistent storage, it seems that, considering reasoning dependent queries, the problem of providing an efficient, in terms of performance, scalability and data redundancy, partitioning of the data is still open. In regards to recent data partitioning studies, it seems reasonable to think that data partitioning should be guided considering several directions (e.g. ontology, data, application queries). This paper proposes several contributions: describe an overview of what a roadmap for data partitioning for RDF data efficient and persistent storage should contain, present some preliminary results and analysis on the particular case of ontology-guided (property hierarchy) partitioning and finally introduce a set of semantic query rewriting rules to support querying RDF data needing OWL inferences.

1 Introduction

The generally encountered use of ontologies consists in performing data inferences and validation using a Semantic Web compliant reasoner. The corresponding reasoning mechanism can be used to generate a set of queries executed over the appropriate data sets. For example, this approach was designed in a medical application [8] where inferences on chemical molecules were needed to highlight contra indications, side effects of pharmaceutical products. As mentioned in [8], results of queries with both inference on property (*i.e.* `rdf:property`) and concept (*i.e.* `rdf:class`) hierarchies are required by the application as well as by data quality or data exchange external tools.

In regards to large ontologies (e.g. OpenGalen or SNOMED in the medical domain) and data sets (e.g. Linked Data), providing efficient performances to reasoning dependent queries is an important issue. We believe that to enable efficient response time to such queries, one has to give a special attention to the storage system associated to the triples. In fact, RDF is basically a data model and its recommendation does not guide to a preferred storage solution. The related work about RDF data management systems can be subdivided into two

categories: the ones involving a mapping to a Relational DataBase Management System (RDBMS) and the ones that do not. In this paper, we do not focus on the latter one.

A set of techniques have been proposed for storing RDF data in relational databases. Several research groups think that this is likely the best performing approach for their persistent data store, since a great amount of work has been done on making relational systems efficient, extremely scalable and robust. Efficient storage of RDF data has already been discussed in the literature with different physical organization techniques based on partitioning (cf. Figure 1).

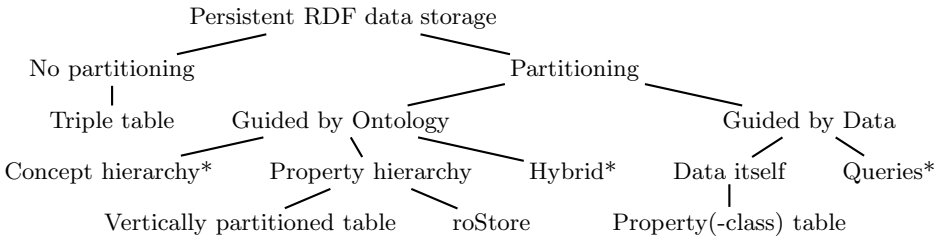


Fig. 1. Physical organization of RDF data based on partitioning. (*) no known study yet.

On one hand, there exists tools such as Sesame [5], Jena [18], Oracle [6] and 3store [10] which rely on a straightforward mapping of RDF into an RDBMS – called *triple table* approach. Each RDF statement of the form (*subject*, *predicate*, *object*) is stored as an entry of one large table with a three-columns schema (*i.e.* a column for each the *subject*, *predicate* and *object*). Indexes are then added for each of the columns in order to make joins less expensive. However, since the collection of triples are stored in one single table, the queries may be very slow to execute. Indeed when the number of triples scales, the table may exceed memory size (inducing costly disk-RAM transfers). Nevertheless, simple *statement-based* queries can be satisfactorily processed by such systems, although they do not represent the most important way of querying RDF data. Still, this storage system scales poorly since complex queries with multiple triple patterns require many self-joins over this single large table as pointed out in [18, 16, 12].

Whereas this specific approach does not use partitioning at all, on the other hand, some recent research highlighted two efficient main trends depending on the information one uses to guide the partitioning: guided by (1) the underlying ontology or (2) the data itself. Intuitively, one would expect that a well suited data partitioning will induce a better response time to queries (at least SELECT ones). Indeed, data partitioning will allow queries to be made on smaller sets of entries which, given an adapted RDF data clustering, should be faster. The counterpart of this storage system will be some possible worst performance for data updates.

Considering partitioning guided by the underlying ontology, most of the recent works focus only on the property hierarchy. Among others, the *vertical partitioning* approach suggested by Abadi et al. in [2] is to be mentioned. In this approach, using a fully decomposed storage model (DSM) [7], the RDF data is divided into n two columns (*subject,object*) tables where n is the number of unique *predicates* in the data. Each of these resulting tables represents a particular *predicate*, with an entry for each statement of the data containing the corresponding *predicate*. Sorting the tables according to the *subject* allows fast merge joins for reconstructing information about multiple *predicates* for subsets of *subjects*. The vertically partitioned approach offers a support for multi-valued attributes. Indeed, if a *subject* has more than one *object* for a given *predicate*, each distinct value is listed in a successive row in the corresponding table. For a given query, only the *predicates* involved in that query need to be read. Finally, the building of the two-columns tables can be done easily – without a clustering algorithm – by only browsing and relying on the property hierarchy of the data. As previously mentioned, as a counterpart, data updates/insertions may be slower in vertically partitioned tables rather than triple ones since multiple tables need to be accessed for statement about the same *subject*.

In [2], the authors described how a column-oriented DBMS [15] (*i.e.*, a DBMS designed especially for the vertically partitioned case, as opposed to a row-oriented DBMS, gaining benefits of compressibility [3] and performance [1]) can be extended to implement the vertically partitioned approach. Roughly, this is done by storing tables as collections of columns rather than collections of rows. The goal is to avoid transferring entire rows into memory from disk, like in row-oriented databases, if only a few attributes are accessed per query. Consequently, in column oriented databases only those columns relevant to a query will be read. Note that, in an independent evaluation [14] of the techniques presented in [2], the authors pointed out potential scalability problems for the vertically partitioned approach when the number of *predicates* in an RDF data set is high. With a larger number of *predicates*, the triple table solution manages to outperform the vertically partitioned one.

Depending on the type of reasoning dependent queries, it may be more efficient to consider an intermediate model between triple tables and vertical partitioning approaches. A first contribution of this work will be to provide an analysis of the effectiveness of an intermediate approach – also based on property hierarchy – where a table is created only for *top predicates*. Another interesting track (for future investigation), would be to consider partitioning the data regarding the concept hierarchy rather than the property one and/or considering both. To the best of our knowledge, such study has not yet been conducted. Considering data guided partitioning, one may distinguish two types of guides. First, one may consider that the RDF data itself may induce an efficient partitioning. The main achievement for this type of guide is the *property table* technique which was introduced later on [17] for improving RDF data organization by allowing multiple triple patterns referencing the same *subject* to be retrieved without an expensive join. In this model, RDF tables are physically stored in a representation

closer to traditional relational schemas in order to speed up the queries over the triple stores [17, 6]. Indeed, each named table includes a *subject* and several fixed *predicates*. The main idea is to discover clusters of *subjects* often appearing with the same set of *predicates*.

A variant of the property table named *property-class table* uses the `rdf:type` of *subjects* to cluster similar sets of *subjects* together in the same table. The immediate consequence is that self-joins on the *subject* column can be avoided. However, the property table technique has the drawback of generating many `NULL` values since, for a given cluster, not all *predicates* will be defined for all *subjects*. A second disadvantage of property table is that multi-valued attributes, that are furthermore frequent in `RDF` data, are hard to express. In a data model without a fixed schema like `RDF`, it is common to seek for all defined *predicates* of a given *subject*, which, in the property table approach, requires scanning all tables.

Note that, in this approach, adding *predicates* requires also to add new tables; which is clearly a limitation for applications dealing with arbitrary `RDF` content. Thus the flexibility in schema is lost and this approach limits the benefits of using `RDF`. Moreover, queries with triples patterns that involve multiple property tables are still expensive because they may require many union clauses and joins to combine data from several tables and consequently complicate query translation and plan generation. In summary, property tables are poorly used because of their complexity and inability to handle multi-valued attributes. Another type of guide which may worth being studied is *queries*. Indeed, one may consider an evolutive storage schema in regards to recent queries made on the data. To the best of our knowledge, this track of research has also not been considered yet.

In this article, we will concentrate on giving some preliminary results (on medium-sized datasets) on an intermediate property hierarchy based approach (that will need to be pursued) – namely RDF Ontology-guided Storage system (`roStore`). After presenting the general approach (Section 2), we will, in Section 3, evaluate the efficiency difference with vertically partitioning on the LUBM benchmark over both row and column oriented databases and on some extra specific queries highlighting limits of vertically partitioning.

2 The `roStore` approach

As a first step to an efficient `RDF` storage road map, we propose an intermediate ontology-guided approach – namely `roStore` – which lies between the two extremes: triple and vertically partitioned tables. The aim of this approach is to try to analyse the efficiency of a compromise approach where less partitions are used. Intuitively, such physical organization will take benefits of requiring less joins in practical queries and with less risk of unmappable table in memory.

As already mentioned, we believe that there should not be a unique generic solution to `RDF` storage and that depending on the data itself, the underlying ontology, application queries, better performance may be obtained by considering alternative and several dedicated approaches. The major aim of `roStore` is to

provide some clue of this belief. We will demonstrate that `roStore` is one of them and may, in specific cases, induce better efficiency. In this context, we consider `roStore` as **one among other** interesting physical organizations based on property hierarchy that should be present in the `RDF` storage road map.

Our storage approach derives from the vertically partitioned one and extends this last by putting back together into a single table data related to a *top-property* of a property hierarchy. Given a hierarchy, we say that a *predicate* is a top-property if it is only an `rdf:subPropertyOf` of itself. For each such top-property P^T, a three-columns table is created by (1) merging all the two-columns tables corresponding to *predicates* being `rdf:subPropertyOf` P^T and (2) adding a third column indicating from which *predicate* the entry *(subject, object)* was retrieved (cf. Figure 2).

Let us first notice that, providing with this definition, any *predicate* that is not an `rdf:subPropertyOf` of a top-property will still be stored in a two-columns table. This induces an insignificant expense of the space complexity of this novel approach. Moreover, in case of a cyclic *property* hierarchy, all *predicates* are necessarily all semantically equivalent. Hence selecting a single canonical *predicate* and rewriting triples accordingly is sufficient. Despite the fact that considering top-property seems to be the most natural, one may, depending on the topology of the hierarchy, define other physical organizations inducing better performance too for specific cases. Our preliminiary results demonstrate that an evolutive physical organizations guided by the queries may be efficient. The main impact of merging some tables is obtaining better performance of queries requiring joins over *predicates* belonging to the same "sub-hierarchy" of the property hierarchy. This is typically the case when one wants to retrieve all the information concerning a family of *predicates* of the property hierarchy; since they will be quite related. In the following, we will denote by `vpStore` (resp. `roStore`) the vertically partitioned (resp. our) approach.

Example 1: Let us consider a small data set (Figure 2b) defined over a given property hierarchy (Figure 2a). With `vpStore`, the triples would be distributed over six different tables as displayed in Figure 2c. Comparatively, in `roStore`, one obtains only two different tables (Figure 2d): a single relation named after the top-property `pa` and a relation named after the property `pf`.

Thus, if we consider an ontology consisting of `n` (e.g. 2 in our example) property hierarchies with an average of `k` (e.g. 3 in our example) properties in each hierarchy, the `roStore` approach will store `k` times less tables than a `vpStore` approach. Moreover, with this approach it is very unlikely to generate tables with no tuples (e.g. `pd` with `vpStore` in Example 1). Moreover, the set of tuples stored is the same as in `vpStore` and only their distribution over database tables is modified (*i.e.* physical organization).

We now consider the following query: one wants to retrieve all *objects* involved in a triple with a *predicate* of the `pa` hierarchy. Considering `vpStore`'s physical design, the following `SQL` query is needed:

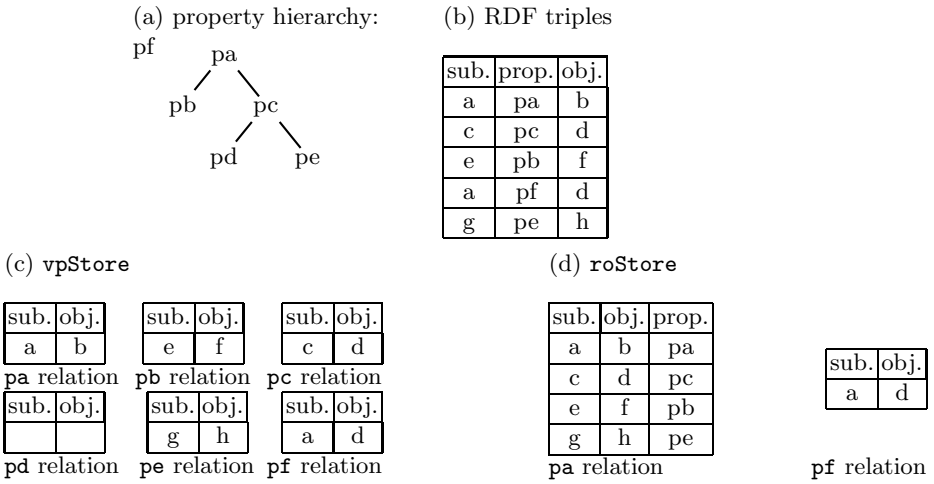


Fig. 2. Storage comparison of `vpStore` and `roStore`

```
SELECT object FROM pa UNION (SELECT object FROM pb UNION (SELECT
object FROM pc UNION (SELECT object FROM pd UNION (SELECT object FROM
pe))));
```

while the same query is answered far more efficiently considering `roStore`'s physical design with:

```
SELECT object FROM pa;
```

Such example highlights the kind of (1) reasoning dependent queries and (2) corresponding improvement one can obtain by using an intermediate physical organization such as `roStore` over `vpStore` when property hierarchies are present in the ontology.

In order to analyse more deeply the corresponding efficiency of `roStore` approach, we will first compare it to the `vpStore` approach on the LUBM benchmark and on some specific queries that highlight limits of vertically partitioning. In this work, as a first contribution, we focus only on SELECT queries. We are currently investigating the possibly negative impact of partitioning on UPDATE queries. As far as we went on this track, it seems that this impact is reasonable. First, we will discuss how to take benefits of the ontology based structure of the data without the needs of heavy inference mechanisms.

Indeed, compared to classical table schema, the ontology is far more meaningfull and can thus be used to enhance the performance even without needing knowledge inference. We propose an efficient use of Semantic Query Rewriting (SQR) adaptable to and usable by most of the data storage approaches. Our semantic query rewriting aims are, first, to guarantee the exhaustiveness of results returned when requiring data that should include `rdf:subClassOf` and `rdf:subPropertyOf`; and, a query validation mechanism simply based on the

domain and range information related to *predicates*, *i.e.* resp. `rdfs:domain` and `rdfs:range`. One has to note that the mechanisms we propose are not really needing heavy reasoning nor data inference mechanisms. Indeed, they can be considered as an efficient use of the right-away available information of the ontology.

The semantic aspect of this rewriting is provided by a thorough usage of the `OWL` entailment mechanism, on one hand, to detect if the answer set of a query will be empty or not, on the other hand, to optimize query in order to guarantee exhaustiveness of the solution returned. The rules can be decomposed into two sets: (i) a set of rules, denoted `subsume`, dealing with concept and property subsumptions; (ii) a set of rules, denoted `propertyCheck`, dealing with the `rdfs:range` and `rdfs:domain` of a given *predicate*. The rules processed by the `subsume` procedure are using the `OWL` inferences to compute all the sub-concepts (resp. sub-properties) of a given concept (resp. property). In fact, the query studied in Example 1 was already using the `subsume` procedure.

Example 2: Consider that the `rdfs:range` of the *predicate* `pb` of Example 1 is of `rdf:type ClassA` which is the top-concept in the following concept hierarchy:

$$\texttt{ClassC} \sqsubseteq \texttt{ClassA},\ \texttt{ClassB} \sqsubseteq \texttt{ClassA}\ \textit{and}\ \texttt{ClassC} \sqsubseteq \neg\texttt{ClassB}$$

That is `ClassA` has two sub-concepts which are disjoint. Consider a query asking for all *subjects* and *objects* of triples where `pb` is the *predicate* and all *subjects* belong to the `ClassA` hierarchy. Using `subsume`, the query can be translated in the following `SQL` query:

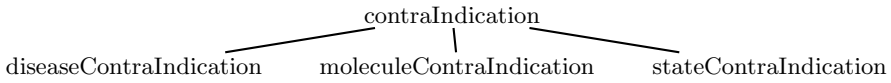
```
SELECT subject, object FROM pa, type WHERE type.subject =
pa.subject AND pa.property = 'pb' AND type.object IN
('ClassA','ClassB', 'ClassC');
```

Thus this approach enables to generate a singe `SQL` query whatever the size of the concept hierarchy is. Note that it also applies to the property hierarchy.

The rules of `propertyCheck` are being processed as follows: first the `SPARQL` query is parsed and for each *predicate* explicitly mentioned in the query with a typed (`rdf:type`) *subject* or *object*, we store a structure containing the *predicate* name and the `rdf:type` of the *subject* and/or *object*. Then for each *subject* (resp. *object*) in the structure, we search if there is a direct or indirect (via subsumptions) correspondence with the type of the `rdfs:domain` (resp. `rdfs:range`) defined in the ontology for this property.

Example 3: Let us consider the property hierarchy of Figure 3, dealing with contra indications and the corresponding `roStore` organization.

Moreover, consider the following ontology axioms: (1) `rdf:range` of `disease` `ContraIndication` is an instance of the `Disease` concept, (2) $\texttt{Disease} \sqsubseteq \texttt{Top}$, (3) $\texttt{Molecule} \sqsubseteq \texttt{Top}$ and (4) $\texttt{Disease} \sqsubseteq \neg\ \texttt{Molecule}$. Intuitively, axioms (2) to (4) state that the `Disease` and `Molecule` concepts are a sub-concept ot the `Top` concept and are disjoint. Consider the following `SPARQL` query:



subject	object	property
Ibuprofen	Ticlopidin	moleculeContraIndication
Ibuprofen	Clopidrogel	moleculeContraIndication
Ibuprofen	Breast feeding	stateContraIndication
Ibuprofen	Pregnant	stateContraIndication
Ibuprofen	Hypertensive heart	diseaseContraIndication

Fig. 3. Sample of the contraIndication relation

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.
?o rdf:type :Molecule.}
```

which asks for *subjects* and *objects* involved in triples where the *predicate* is `diseaseContraIndication` and the *object* has a `rdf:type Molecule`. Clearly the answer set to this query is empty since the `rdf:domain` of the *predicate* can not be a `Molecule` in this ontology.

Example 4: Consider the following query in the context of Example 3:

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.
?o rdf:type :Disease.}
```

The query is satisfiable since there is a model where its answer set is not empty. Anyhow, the query can be optimized. In fact, it is not necessary to check the `rdf:type` of the *object* because it corresponds exactly to the one defined as `rdf:range` in the ontology. Thus this query is rewritten in:

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.}
```

which once translated into `SQL` does not require any join and will thus perform far more efficiently than the orginal query. Note that this simplification does not work for property with multiple-range/domain. Those examples demonstrate that it is worth to efficiently use the basic knowledge available directly in the concept and property hierarchies.

3 Evaluation

3.1 Experimental settings

All our experiments have been conducted on four synthetic databases. They all have been generated from the Lehigh University Benchmark (LUBM) [9] which has been developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The `RDF` data sets generated with LUBM all commit to a single realistic ontology dealing with the university domain. This ontology is composed of 43 concepts, 25 object properties (*i.e.* relating objects to objects) and 7 data type properties (relating objects to literals).

This ontology serves as the schema underlying the four data sets we have created. This is an important requisite for our evaluation since our set of queries will be executed on all data sets in order to provide information on scalability issues. Table 1 summarizes the main characteristics of these data sets in terms of overall number of triples, number of concept and property instances.

Table 1. Synthetic data sets

DB name	# Universities	# Concept instances	# Property instances	# Triples
lubm1	1	15195	60859	100868
lubm2	2	62848	189553	236336
lubm5	5	114535	456137	643435
lubm10	10	263427	1052895	1296940

The `RDF` data sets are later translated into the different physical organization models we would like to evaluate. They are decomposed into the two main approaches `vpStore` and `roStore`. In order to emphasize the efficiency of our solution on queries needing reasoning, we had to test these settings in a context similar to [2]. More precisely, we evaluated each approach on a row store and a column store RDBMS. This yields the four following approaches: `vpStore` resp. on a row (**vpRStore**) and column (**vpCStore**) store and `roStore` resp. on a row (**roRStore**) and column (**roCStore**) store. Hence a total of sixteen databases are generated (each data set is implemented on each physical approach).

We have selected postgreSQL and MonetDB as the RDBMS resp. for the row-oriented and the column-oriented databases. We retained MonetDB instead of C-store (the column store used for evaluation in [2]) essentially due to (1) the lack of maintenance of the latter one, (2) the open-source licence of MonetDB and (3) the fact that MonetDB is considered state of the art in column-oriented databases. The tests were run on MonetDB server version 5 and postgreSQL version 8.3.1. The benchmarking system is an Intel Pentium 4 (2.8 GHz) operated by a Linux Ubuntu 9.10, with 512 Mbytes of memory, 1MB L2 cache and one disk of 60 Gbyte spinning at 7200rpm. The disk can read cold data at a rate of approximatively 55MB/sec.

For the `vpRStore`, there is a clustered B+ tree index on the *subject* and an unclustered B+ tree on the *object*. Similarly, for the `roRStore`, a clustered B+ tree index is created on the *property* column and unclustered B+ trees on the *subject* and *object*. As noted in [14], MonetDB does not include user defined indices. Hence, we relied on the ordering of the data on *property*, *subject* and *object* values. More precisely, any two columns table of `roCStore` and `vpCStore` is ordered on *subject* and *object*; while any three columns table (of `roCStore`) is ordered on *property*, *subject* and *object*.

Our evaluation contains fifteen queries out of which eleven are coming from the LUBM benchmark and four tackling the LUBM ontology to evaluate some particular aspects of `roStore`. An interesting aspect using LUBM Benchmark queries is that do not aim to emphasize on the performances of a given storage

model. Moreover, these queries tackle a wide range of possibilities on volume of input (number of tuples retrieved) and selectivity rate (*i.e.* number of conditions in the `WHERE` clause of a query). Among the eleven evaluated queries, three do not require any form of reasoning (#1, #2 and #14) and the eight remaining queries can be divided in two groups whether they are involving reasoning on the concept hierarchy (#3,#4,#6,#7,#9,#10) or both concept/property hierarchies (#5,#8). We now present the purpose of each of these queries:

Q1: retrieves instances of the `GraduateStudent` class who have taken the course `http://www.Department0.University0.edu/GraduateCourse0`.

Q2: retrieves three instances of respectively the `GraduateStudent`, `University` and `Department` concepts for those students that are member of a department, this department is a sub-organization of a University and this student has an undergraduate degree from this university.

Q3: selects all kinds of publications which have been authored by a given assistant professor.

Q4: retrieves all kinds of professors, their name, email address and telephone number for those professors working for a given department.

Q5: the result contains instances of the `Person` concept hierarchy for those persons that are related to a given department by either the `memberOf`, `workingFor` or `headOf` properties.

Q6: displays URIs of instances of the `Student` concept hierarchy.

Q7: retrieves instances of all kind of students and all kinds of courses for courses that are related by the `takesCourse` property for those courses that are taught by a given professor.

Q8: displays instances of all kinds of students with their email addresses and department instances of a given university these students are member of.

Q9: the retrieved dataset contains instances of the `Student`, `Faculty` and `Course` concept hierarchies for those students that are advised by faculties, have taken some courses taught by those faculties.

Q10: selects instances of all the `Student` class hierarchy who have taken a given course.

Q14: selects all undergraduate students.

We have introduced **Q15** to emphasize `roStore` performances when values are needed for a property hierarchy. In fact, it retrieves all *subjects* involved in triples where the *predicate* is one of the properties of the `memberOf` property hierarchy, i.e. `memberOf`, `headOf` and `worksFor`. This query is similar to Q5 but does not refer to any concepts.

Finally, the following three queries aim to highlight the efficiency of our SQR approach. **Q16:** selects the *subject* and *object* in triples where the *predicate* is `teacherOf` and *subject* is of `rdf:type` AdministrativeStaff. This query returns an empty answer set since the `rdfs:domain` of `teacherOf` is the `Faculty` concept which is disjoint with `AdministrativeStaff`. In the next section, we confront the performances of this query to the simple detection of unsatisfiability of our SQR solution.

Q17: selects the *subject* and the *object* in triples where the *predicate* is `teacherOf` and *subject* is of `rdf:type Faculty`. This query requires a join.

Q18: has the same purpose as Q17 but exploits one of our rewriting rules to improve its performances. In fact, the join in Q17 is not necessary if one knows that the `rdfs:domain` of `teacherOf` is the concept `Faculty`.

In the experiments, we will store the LUBM ontology in main-memory and perform reasoning using the Jena framework. We provide more details concerning the experimental settings and results on the following web site:
`http://sites.google.com/site/wwwrostore`.

3.2 Experimental results

The results presented in this section correspond to the average of 5 hot runs (i.e. repeated runs of the same query without stopping the DBMS) of real time (i.e. execution time of a query defined as the wall clock passed between the server receiving the query and before returning the results to the client) executions. All performance times, except for query Q16 and Q17, include the time needed to perform the inferences. Finally, in order to highlight the differences in terms of performances between the various approaches, we either present the results in bar or line diagrams.

Analysis of Q1. Not surprisingly, column stores outperform row stores. Indeed, the results will only contain a unique column which will clearly benefit column store advantage. Moreover, since the query does not involve sub-properties, the performances of `vpStore` and `roStore` are quite similar.

Analysis of Q2. This time, the row stores are more efficient than the column ones. The results require, in this case, to retrieve two columns of three tables, hence in a row store both columns will be transfered from the hard drive to main memory in a single step while two transfers will be needed for column stores. Moreover, two out of these three tables corresponds to *predicates* being part of a group of related predicates in the property hierarchies, namely `memberOf` and `undergraduateDegreeFrom`. Since we voluntarily decided to perform no inferences on these two *predicates*[3] (*i.e.* not including sub-properties in the query), it is not surprising that `vpStore` outperforms `roStore` since each *predicate* corresponds in the `vpStore` to a table.

Analysis of Q3. Despite the fact that this query has a similar structure as Q1 (*i.e.* only two triples are present in the `WHERE` clause), it requires to retrieve all concepts of a wide hierarchy. Due to the ordered organization of tuples, column stores outperform row ones which rely on indices and on a less effective I/O transfers. In a similar manner to Q1, the difference between `vpStore` and `roStore` is not significant.

Analysis of Q4. Once again, in this query, we do not use inference on the `worksFor` *predicate* to include results on sub-properties of this last. This is motivated by the will to emphasize on the weaknesses of the `roStore` approach. As expected, `vpStore` is, in this context, outperforming `roStore`. In fact, even

[3] since it will be specifically considered in query Q15.

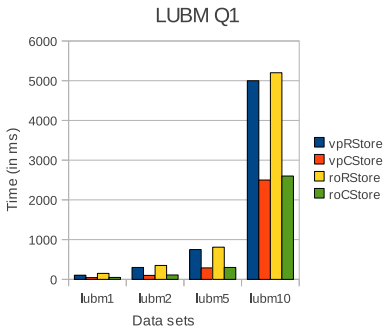


Fig. 4. Performance results for Q1

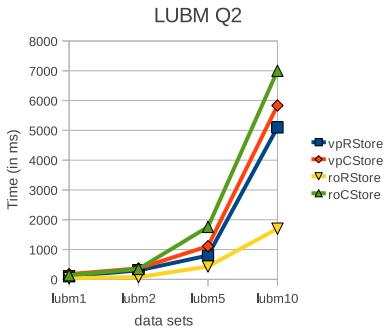


Fig. 5. Performance results for Q2

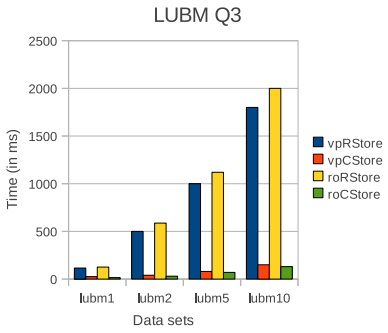


Fig. 6. Performance results for Q3

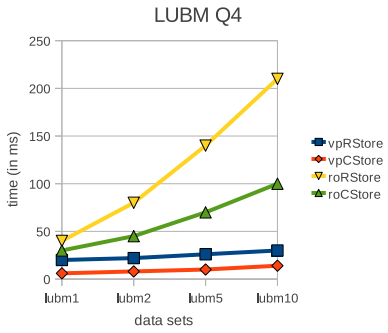


Fig. 7. Performance results for Q4

`vpRStore` is outperforming `roCStore`; which can be induced by the high selectivity nature of the query (four attributes in result set).

Analysis of Q5. Due to the exploitation of the sub-properties of the *predicate* `memberOf` in this query, it is not surprising that `roStore` outperforms `vpStore`. Indeed in `vpStore`, the results of the query comes from the union of three distinct queries (one for each *predicates* involved) while `roStore` only requires a single query.

Analysis of Q6. This query retrieves the subjects from a two columns table (i.e. type). Because the column stores primarily order these relations on the subject, they are more efficient than their row store counterparts. This is due to better I/O efficiency. Similarly to Q3, the `roStore` approach outperforms `vpStore`.

Analysis of Q7. Again query processes in `roStore` (resp. column store) is more efficient than `vpStore` (resp. row store). The reasons are similar to the ones for Q3.

Analysis of Q8. The analysis of the results for this query confirm the ones of Q5.

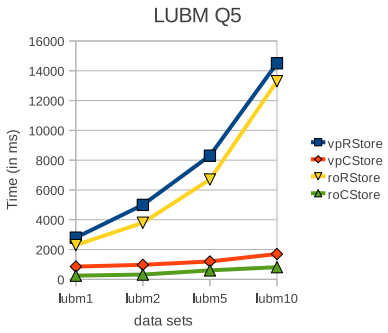


Fig. 8. Performance results for Q5

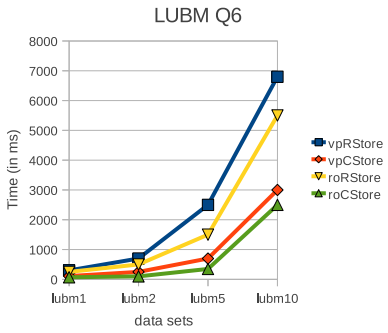


Fig. 9. Performance results for Q6

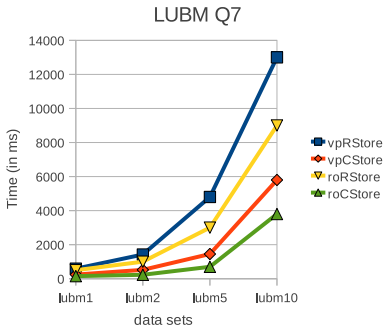


Fig. 10. Performance results for Q7

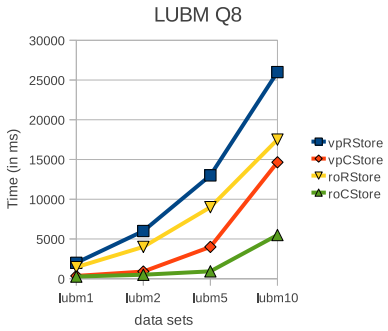


Fig. 11. Performance results of Q8

Analysis of Q9. This query does not require inferences on property hierarchies but some on several concept ones. As seen previously, in this situation column stores is more efficient than row stores. On column stores, `vpStore` and `roStore` have close performance results, with `roStore` slighty better than `vpStore`.

Analysis of Q10. The results are similar to Q9.

Analysis of Q14. This query has large input and low selectivity with no inferences. As expected, `roCStore` is faster than `vpCstore` which is more efficient than `roRStore`; the less effective being `vpRStore`. Note that this is due to distinguished variable being placed at the *subject* position of the only triple of the `WHERE` clause. A similar query pattern with the distinguished variable mapped to the *object* position of a triple would emphasize the superiority of the `vpStore` approach.

Analysis of Q15. This query clearly demonstrates the efficiency of `roStore` over `vpStore`. Even the row oriented `roStore` outperforms the column oriented `vpStore`. This is due to the presence of `UNION SQL` operators in the queries executed on the `vpStore` while `roStore` only requires a complete scan of the tuples of one table.

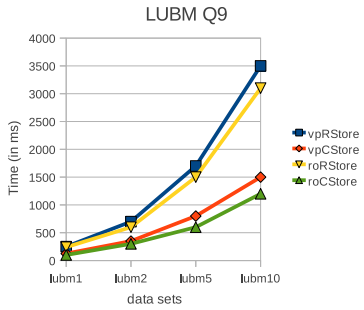


Fig. 12. Performance results for Q9

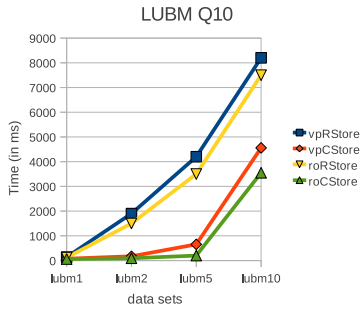


Fig. 13. Performance results of Q10

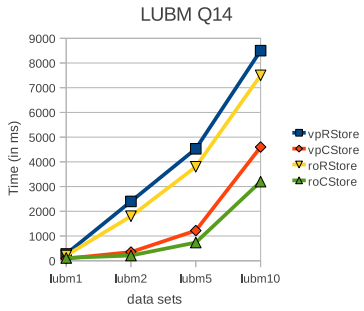


Fig. 14. Performance results for Q14

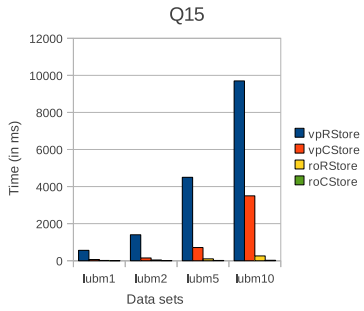


Fig. 15. Performance results of Q15

Analysis of Q16, Q17 and Q18. Finally, queries Q16, Q17 and Q18 emphasize the importance of reasoning over the ontology before executing queries over any of the store solutions. Figure 16 displays the duration times for all databases, ranging from approximately 42ms (column store with 1 university) to 1450ms (row store with 10 universities). This can be considered rather long to propose an empty answer set since, according to the ontology, the query is incoherent. Comparatively, the `propertyCheck` method we have implemented needs an average time of 1ms to reply that the query is coherent or not. Hence, a system implemented on top of an `OWL` compliant reasoner is able to determine almost instantly if the answer set is empty.

Moreover, it could also provide some explanations concerning the inconsistency of the query. We believe that such optimization are quite useful especially when end-users are not confident with all the details of a given ontology. The performance results of Q17 and Q18 are provided together in Figure 17 in order to highlight their comparisons. The purpose of Q17 and Q18 is to emphasize the importance of analyzing *predicate* `rdfs:domain` and `rdfs:range` in a property table approach. The execution of Q17 does not perform any optimization while Q18 checks that the concept `Faculty` is the `rdfs:domain` of the `teacherOf` *predicate* and hence a join to the `rdf:type` relation is not necessary.

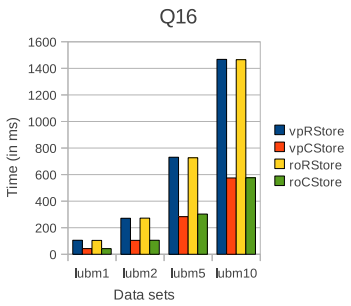


Fig. 16. Performance results for Q16

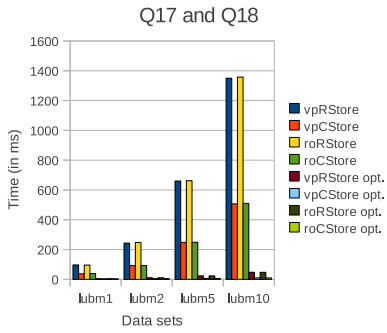


Fig. 17. Q17 and Q18 performance results

Summary: Several conclusions can be drawn from our evaluation. Considering the adoption of a database solution, we confirm the evaluations of [2] and [14] stating that column stores outperform row stores for `RDF` triple storage. The only exception in our experiments consists in Q2 which is rather due to the partitioning approach.

Concerning the partitioning approach, all our intuitions were confirmed by this evaluation. That is `roStore` outperforms `vpStore` whenever queries retrieve information from triples where properties belong a given property hierarchy (e.g. Q5 and Q15). On the contrary, `vpStore` is more efficient than `roStore` where only a subset of the properties of a property hierarchy are necessary to reply to a query (e.g. Q2). This result was expected since the `roStore` approach then requires to add additional conditions on the properties one wants to retrieve from a 'top property' relation.

Finally, the SQR approach seems to be quite useful since it does not slow down the processing of satisfiable queries and enables to detect unsatisfiable queries efficiently (e.g. Q17 and Q18). Anyhow, we consider that more evaluations need to be conducted on larger ontologies to confirm these results.

4 Conclusion

The first contribution of this paper is to show that depending on the type of applications and queries asked to the RDF triple stores, different partitioning approaches can be considered. Between the two extremes of triple and vertical partitioning, we introduced the `roStore` approach which is particularly advantageous for a certain class of queries, *i.e.* those relying on deep property hierarchies (e.g. the OpenGalen ontology contains a property hierarchy of depth 6). Moreover, this novel approach is a simple extension to the existing RDF column store work and can thus be easily adopted by other RDF stores. A second contribution of this work is to propose a semantic query rewritting solution that can be adopted by most of the RDF triples we have presented in this paper (triples tables, vertical partitioning, `roStore`, property-class tables). This approach seems

promising since it can be quite useful to detect unsatisfiable queries and optimizing other queries by analyzing property domains and ranges.

Our list of future works is large since we consider that several investigations need to be performed to complete the road map on efficient and persistent RDF triple storage. The first directions we would like to follow are ontology schema evolution in `roStore` (e.g. a new property hierarchy emerges or is removed from the ontology) and the consideration of concept hierarchies at the storage and querying levels.

References

1. Abadi, D.J., Myers, D.S., DeWitt, D.J., Samuel, R.M. : Materialization Strategies in a Column-Oriented DBMS. ICDE'07, 466-475, 2007
2. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K. : Scalable semantic web data management using vertical partitioning. VLDB '07, 411-422, 2007
3. Abadi, D.J., Madden, S., Ferreira, M. : Integrating compression and execution in column-oriented database systems. SIGMOD '06, 671–682, 2006.
4. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (Feb. 2004) `http://www.w3.org/TR/rdf-schema/`
5. Broekstra, J., Kampman, A., Van Harmelen, F. : Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC'02. 54–68 2002
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J. : An efficient SQL-based RDF querying scheme. VLDB'05. 1216–1227, 2005
7. Copeland, G.P., Khoshafian, S.N. : A decomposition storage model. SIGMOD'85, 268–279, 1985
8. Curé, O.: Semi-automatic Data Migration in a Self-medication Knowledge-based System. Wissensmanagement'05, 323-329
9. Guo Y., Pan Z., Heflin J. : LUBM: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005
10. Harris, S., Gibbins, N. : 3Store Efficient bulk RDF storage. PSSS'03, 1–20, 2003
11. Hayes, P.: RDF Semantics (Feb. 2004). `http://www.w3.org/TR/rdf-mt/`
12. Kolas, D., Emmons, I., Dean, M. : Efficient Linked-list RDF Indexing in Parliament. SSWS'09, 17-32, 2009
13. Prud'hommeaux, E., Seaborn A.: SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`
14. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S. :Column-store support for RDF data management: not all swans are white. VLDB'08, 1553-1563
15. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P., Rasin, A., Tran, N., Zdonik, S. : C-store: a column-oriented DBMS. VLDB'05, 553-564, 2005
16. Weiss, C., Karras, P., Bernstein, A. : Hexastore : sextuple indexing for semantic web data management. VLDB'08, 1008-1019, 2008
17. Wilkinson, K. : Jena property table implementation. SSWS'06, 2006
18. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D. : Efficient RDF Storage and Retrieval in Jena2. SWDB'03, 131-150, 2003

Avalanche: Putting the Spirit of the Web back into Semantic Web Querying

Cosmin Basca and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{lastname}@ifi.uzh.ch

Abstract. Traditionally Semantic Web applications either included a web crawler or relied on external services to gain access to the Web of Data. Recent efforts have enabled applications to query the entire Semantic Web for up-to-date results. Such approaches are based on either centralized indexing of semantically annotated metadata or link traversal and URI dereferencing as in the case of Linked Open Data. By making limiting assumptions about the information space, they violate the openness principle of the Web – a key factor for its ongoing success. In this article we propose a technique called Avalanche, designed to allow a data surfer to query the Semantic Web transparently without making any prior assumptions about the distribution of the data – thus adhering to the openness criteria. Specifically, Avalanche can perform "live" (SPARQL) queries over the Web of Data. First, it gets on-line statistical information about the data distribution, as well as bandwidth availability. Then, it plans and executes the query in a distributed manner trying to quickly provide first answers. The main contribution of this paper is the presentation of this open and distributed SPARQL querying approach. Furthermore, we propose to extend the query planning algorithm with qualitative statistical information. We empirically evaluate Avalanche using a realistic dataset, show its strengths but also point out the challenges that still exist.

1 Introduction

With the introduction of the World Wide Web, the way we share knowledge and conduct day to day activities has changed fundamentally. With the advent of the Semantic Web, a Web of Data is emerging interlinking ever more machine readable data fragments represented as RDF documents or queryable semantic endpoints. It is in this ecosystem that unexplored avenues for application development are emerging.

While some application designs include a Semantic Web (SW) data crawler, others rely on services that facilitate access to the Web of Data (WoD) either through the SPARQL protocol or various APIs (i.e. Sindice or Swoogle). As the mass of data continues to grow – currently Linked Open Data (LOD) [1] accounts for 4.7 billion triples – the scalability factor combined with the Web's uncontrollable nature and its heterogeneity will give raise to a new set of challenges.

A question marginally addressed today is: How to query the Web of Data on-demand, without hindering the flexible openness principle of the Web – seen as the ability to query independent un-cooperative semantic databases, not controlling their distribution, their availability or having to adhere to fixed publishing guidelines (i.e. LOD). The underlying assumptions of WoD, as with the WWW, are that (a) there exists no distribution pattern of information onto servers, (b) there is no guarantee of a working network, (c) there is no centralized resource discovery system, (d) there exists a standard (HTTP) for the retrieval of information, and (e) the size of RDF data no longer allows us to consider single-machine systems feasible. With the serendipitous nature of Semantic Web [12], querying the global information space gives rise to new possibilities unthought of before.

Several approaches that tackle the problem of querying the entire Web of Data have emerged lately. One solution provides a centralized queryable endpoint for the Semantic Web that caches all data. This approach allows searching for and joining potentially distributed data sources. It does, however, incur the significant problem of ensuring an up-to-date cache and might face crucial scalability hurdles in the future, as the Semantic Web continues to grow.

Other approaches use the guidelines of LOD publishing to traverse the linked data cloud in search of the answer. Obviously, such a method produces up-to-date results and can detect data locations only from the URIs of bounded entities in the query. Relying on URI structure, however, may cause significant scalability issues when retrieving distributed data sets, since (1) the servers dereferenced in the URI may become overloaded, and (2) it limits the possibilities of rearranging (or moving) the data around by binding the id (i.e. URI) to storage location.

Finally, traditional database federation techniques have been applied to query WoD. They rely on statistical information from queryable endpoints that are used by a mediator to build efficient query execution plans. Their main drawback is that some query execution engine is aware of the data distribution ex-ante (i.e., before the query execution). Moreover, in most cases, data sources even need to register themselves at the query execution engine with detailed information about the data they contain.

In this paper, we propose Avalanche, a novel approach for querying the Web of Data that (1) *makes no assumptions about data distribution, availability, or partitioning*, (2) provides *up-to-date results*, and (3) is *flexible as it makes no assumption about the structure of participating triple stores*. Consequently, it addresses the shortcomings of previous approaches. To query WoD Avalanche provides a novel technique via means of a two-phase protocol: a discovery step, i.e. gathering statistical information about data distribution from involved hosts, and a planning optimization step over the distributed SPARQL endpoints. Hence, the main contributions of our approach are:

- on-demand transparent querying over the Web of Data, without any prior knowledge about its distribution
- a formal description of our approach, together with possible optimizations for each step

– a novel planning strategy and cost model for dealing with towards Web scale graph data
– a reference implementation of the AVALANCHE technique

In the remainder we first review the relevant related work of the current state-of-the-art. Section 3 provides a detailed description of AVALANCHE. In Section 4 we evaluate several planning strategies and estimate the performance of our system. In Section 5 we present several future directions and optimizations, and conclude in Section 6.

2 Related work

Several solutions for querying the Web of Data over distributed SPARQL endpoints have been proposed before. They can be grouped into two streams: (a) distributed query processing and (b) statistical information gathering over RDF sources.

Research on distributed query processing has a long history in the database field [18, 9]. Its traditional concepts are adapted in current approaches to provide integrated access to RDF sources distributed over the Web. For instance, Yars2 [6] is an end-to-end semantic search engine that uses a graph model to interactively answer queries over structured and interlinked data, collected from disparate Web sources. Another example is the DARQ engine [15], which divides a SPARQL query into several subqueries, forwards them to multiple, distributed query services and, finally, integrates the results of the subqueries. Finally, Rdfpeers [3] is a distributed RDF repository that stores three copies of each triple in a peer-to-peer network, by applying global hash functions to its subject, predicate and object. Virtuoso [4], a data integration software developed by OpenLink Software, is also focused on distributed query processing. The drawback of these solutions is, however, that they assume total control over the data distributions—an unrealistic assumption in the open Web. Similarly, SemWIQ [11] provides a mediator that distributes the execution of SPARQL queries transparently. Its main focus is to provide an integration and sharing system for scientific data. Whilst it does assume control over the instance distribution they assume perfect knowledge about it. Addressing this drawback some [20, 17] propose to extend SPARQL with explicit instructions where to execute certain sub-queries. Unfortunately, this assumes an ex-ante knowledge of the data distribution on part of the query writer. Finally, Hartig et al. [7] describe an approach for executing SPARQL queries over spaces structured according to the Web of Linked Data rules [1]. Whilst they make no assumptions about the openness of the data space the LOD rules requires them to place the data on the URI-referenced servers—a limiting assumption for example when caching/copying data.

Research on query optimization for SPARQL includes query rewriting [8], triple pattern optimization based on selectivity estimations [13, 19, 14], and on other statistical information gathering over RDF sources [10, 5]. RDFStats [10] is an extensible RDF statistics generator that records how often RDF properties

are used and feeds automatically generated histograms to SemWIQ. Histograms on the combined values of SPO triples have proved to be especially useful to provide selectivity estimations for filters [19]. For joins, however, histograms can grow very large and are rarely used in practice. Another approach is to compute ahead frequent paths (i.e., frequently occurring sequences of S, P or O) in the RDF data graph and keep statistics about the most beneficial ones [13]. It is unclear how this would work in a highly distributed scenario. Finally, Neumann et. al [14] claim that selectivity estimation is a worthwhile solution for tens of millions of RDF triples, but unsuitable for billions of triples, because the size of the data and the increasing diversity in property names lead to poor estimations, thus misguiding the query optimizer.

3 Avalanche — System Design and Implementation

In this section, we describe the overall design of AVALANCHE and the underlying philosophy of the distributed query execution across large datasets spread over multiple, uncooperative servers.
The major design difference between AVALANCHE and previous systems is that *it assumes that the distribution of triples to machines participating in the query evaluation is unknown prior to query execution.* Hence, our approach follows neither a federated nor a peer-to-peer model, instead the statistical discovery phase that traditionally is reserved for the (parallel) mediator component in clustered approaches, has become an individual step during each query execution phase. In the remaining part of this section, we will first illustrate our approach using a motivating example. This will lead the way towards thoroughly describing the AVALANCHE components and its novelty.

The system consists of six major components working together in a parallelized pipeline: the AVALANCHE *endpoints Web Directory* or *Search Engine*, the *Statistics Requester*, the *Plan Generator*, *Plan Executor* instances, *Plan Materializer* instances and the *Query Stopper* component as seen in Figure 1.

AVALANCHE comprises of two phases: the *Query Preprocessing* phase and the parallel *Query Execution* phase. During *Query Preprocessing*, participating hosts are identified via means of a `Search Engine` such as Sindice[1] or `Web Directory`. A lightweight endpoint-schema inverted index can also be used. Ontological prefix (the shorthand notation of the schema – i.e. `foaf`) and schema invariants (i.e. predicates, classes, labels, etc) are appropriate candidate entries to index. After query parsing, this information is immediately available and used to quickly trim down the number of potential endpoints. Then, all selected AVALANCHE endpoints are queried for the cardinality (number of instances) of each unbounded variable — statistical information that triple-stores generally posses.

In the *Query Execution* phase, first the query is broken down into the superset of all **molecules**, where a molecule is a subgraph of the overall query graph.

[1] http://sindice.com/

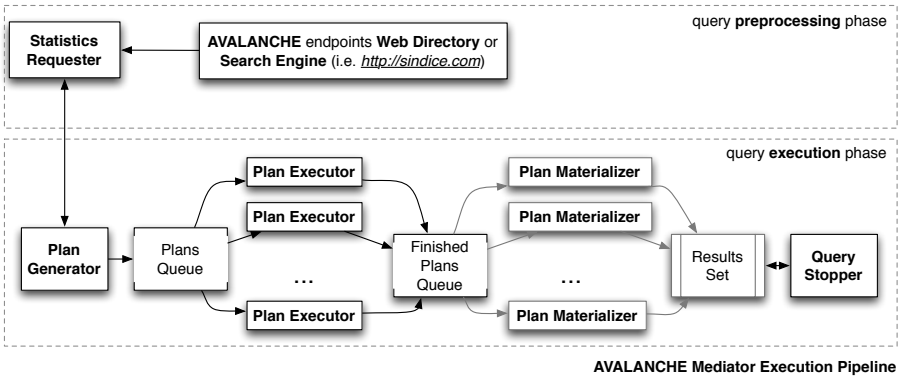


Fig. 1. The Avalanche execution pipeline

A combination of minimally overlapping molecules covering the directed query graph is referred to as a **solution**. Binding all molecules in a given solution to physical hosts (Avalanche endpoints) that may resolve them, transforms a solution into a **plan**. Given the size of the Web and the unknown distribution of the RDF data, Avalanche will try to optimize the execution of the query to quickly find the **first K** results. The proposed planning system and algorithm, though complete, will normally not be allowed to exhaust the entire search space since the cost of doing so is prohibitively expensive. Instead, the planner component strives to execute the most "promising" query plans first, while being monitored by the `Query Stopper` for termination conditions. To further reduce the size of the search space, a windowed version of the search algorithm can be employed – i.e. with each exploratory step only the first M molecules are considered, thus sacrificing completeness.

As shown in Figure 1 the `Plan Generator` relies on statistics about the data contained on the different hosts from the `Statistics Requester`. Any generated plan gets put in the `Plans Queue` regardless if the planner finished its overall tasks of exploring the plan space or not. Plans in the `Plans Queue` are fetched by `Plan Executors` that execute them generating partial results in parallel and put them in the `Finished Plans Queue`. There, they get fetched by one of the parallel executing `Plan Materializers`, who will merge and materialize the partial results.

To put Avalanche into perspective consider the following **motivating query** that executes over Linked Open Datasets describing movies and actors:

```
SELECT ?title ?photoCollection ?name WHERE {
?film dc:title ?title; movie:actor ?actor;  owl:sameAs ?sameFilm.
?actor a foaf:Person; movie:actor_name ?name .
?sameFilm dbpedia:hasPhotoCollection ?photoCollection.
?sameFilm dbpedia:studio ‘‘Producers Circle’’; }
```

The goal of AVALANCHE is to return the list of all movie titles, their photo collections and the names of starring actors, that have been produced at "Producers Circle" studios – considering that the required information is spread with an unknown distribution over several LOD endpoints.

At a given moment during the execution of a plan, a `Plan Executor` instance may find itself in the state depicted in Figure 2 (in depth description in Section 3.2). The plan is comprised of three molecules: `M1, M2, M3` and three hosts are involved: `host A, host B` and `host C`. Molecule `M1` was reported to be highly selective on host A (holding Linked Movie[2] data), while the remainder of the plan: molecule `M2` and `M3`, is distributed between hosts B and C (both holding DBPEDIA[3] data). Given that we operate in an environment where bandwidth cost is non-trivial we should not "just" transport all partial results to one central server to be joined. Instead we start with executing the highly selective (or in this case: with the lowest cardinality) molecule `M1` on host A and then limit the execution space on host B by sending over the results from host A. The process repeats itself given the number of molecules in the plan and is finalized with a merge/update operation in reverse join order.

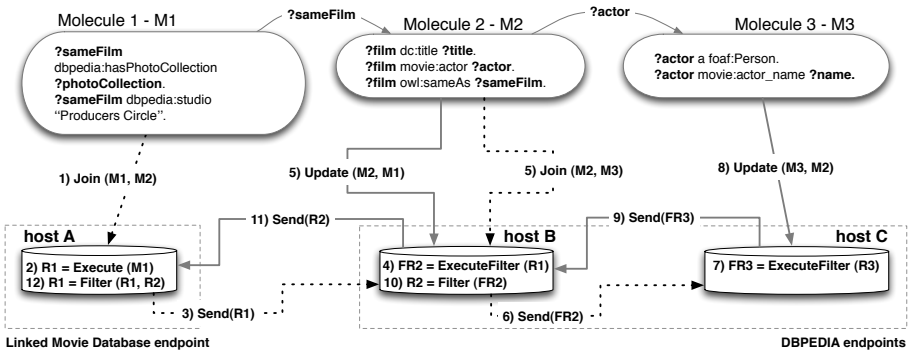


Fig. 2. Distributed Join and Update operations for a Simple Plan

It is important to note that to execute plans, hosts will need to share a common id space – a given in Semantic Web via URIs. Naturally, using RDF strings can be prohibitively expensive. To limit bandwidth requirements, we chose to employ a single global id space in the form of the `SHA` family of hash functions on the URIs.

The remainder of this section will detail the functionality of the most important elements: the `Plan Generator`, `Plan Executor` and `Plan Materializer` as well as explain how the overall pipeline stops.

[2] http://www.linkedmdb.org/
[3] http://dbpedia.org/About

3.1 Generating Query Plans

The planner's main focus is to generate query plans that are likely to produce results fast with a minimum of cost. As shown in Algorithm 1 the planner will try to optimize the construction of plans using a multi-path informed (best-first) search strategy by maximizing the OBJECTIVE function of a plan. Therefore, all plans are generated in descending order of their objective function.

Algorithm 1 The plan generator algorithm

```
PLAN-GENERATOR(Molecules, Hosts, Cardinalities)
 1  fringe = []
 2  for each molecule M ∈ Molecules, host H ∈ Hosts
 3       partialPlan = {M, H, NULL, Cardinalities}
 4       APPEND(fringe, partialPlan)
 5  SORT(fringe)
 6  while !fringe.empty() // Loop through fringe
 7       best = GETFIRSTELEMENTWITHPOSITIVEOBJECTIVE(fringe)
 8       if PLANISCOMPLETE(best) // all molecules assigned to host
 9            SORT(fringe)
10            yield GETPLAN(best) // returns results but continues planning
11       else // plan is incomplete
12            remMol = GETREMAININGCONNECTEDMOLECULES(Molecules, best)
13            planFringe = []
14            for each molecule M ∈ remMol, host H ∈ Hosts
15                 partialPlan = {M, H, best, Cardinalities}
16                 APPEND(planFringe, partialPlan)
17            SORT(planFringe)
18            CONCATENATE(fringe, planFringe)
```

In defining the OBJECTIVE function we use the statistical information gathered beforehand (result set cardinality). To ensure the generation of most productive plans, our function models the chance of finding a solution, utility U, divided by the cost of executing the query, C. Hence:

$$Objective = \frac{U}{C} \tag{1}$$

An emergent challenge from preserving the openness of the query process and the flexibility of semantic data publishing, is denoted by the exponential complexity class of the plan composition space. Thus the space complexity of the problem is $O(N^3)$, considering that the problem size increases by $M * H$ with each step towards a complete plan, where H represents the total number of hosts involved and M is a measure of the query complexity (i.e. the number of unique *molecules* that can be extracted from the given query graph). A simple calculation for the scenario where 1000 hosts are involved and a rather large

query ($\approx$ 15 unbounded variables) might generate 500 molecules with the average depth of a plan of 10 (molecules), results in 5 million possible combinations to form *plans*. Not all combinations produce viable plans, so pruning low or no utility plans early is desired as seen in line 7 of the planning algorithm.

We follow the assumption that selective molecules – with low cardinalities – will help the plan to converge faster. In the bootstrap phase the utility of the first plan node is equal to the inverse of its cardinality: CNT_{N1} (where $N1$ is node 1 and CNT is the cardinality) factored by the size of the plan ($Edges(N1)$). Further on, we consider a join where the best-case cardinality is the minimum of the involved result set cardinalities (see Equation 2). We define the cost, C for executing queries in Equation 3. The cost of the first node is assumed to be constant. For all other nodes we combine:

- the network latency L (between two nodes)
- a measure of the time required to send the results from node $N1$ to node $N2$ given the bandwidth B
- the cost of executing on $N1$ and $N2$ as approximated by their cardinalities

Finally, we scale this result with a measure of the current molecule size (molecule assigned to $N2$) relative to the size of the whole solution, in order to encourage the choice of nodes that aid convergence.

$$U_{N1,N2} = \begin{cases} \frac{Edges(N1)}{CNT_{N1}}, & \text{first node} \\ min(CNT_{N1}, CNT_{N2}), & \text{otherwise} \end{cases} \tag{2}$$

$$C_{N1,N2} = \begin{cases} 1, & \text{first node} \\ (L + \frac{CNT_{N1}}{B} + CNT_{N1} + \frac{CNT_{N2}}{CNT_{N1}}) \frac{Edges(Solution)}{Edges(N2)}, & \text{otherwise} \end{cases} \tag{3}$$

Extended Utility Function The main drawback of this utility function is that it assumes the lower cardinality of the two nodes is representative—an assumption that is quite wrong when searching for "rare" results given a large number of "promising" hosts. Therefore it disregards the actual join probabilities. Consider the previous example query that goes out to two almost disjoint RDF servers: one with DBPEDIA data and another with public social network data. Assuming we found an actor through some other host, the utility function will not be able to favor DBPEDIA over the other host, as it cannot evaluate the actual number of joins. Hence, if the public social network host happens to be using a better network connection, the planer will be lead astray. To overcome this effect we need a measure of join-quality. Following [16] we employ *bloom-filters*, which are space-efficient set representation bit vectors composed of multiple hash functions. As stated by [2] bloom-filters allow for a statistically solid estimation of the cardinality of the join between two sets:

$$JOIN_{BF_1,BF_2} \approx -\frac{1}{k} \frac{\ln(m \frac{Z_1 + Z_2 - Z_{12}}{Z_1 Z_2})}{\ln(1 - \frac{1}{m})} \tag{4}$$

where BF is a bloom filter, m is the number of bits in the bloom filter, k represents the number of hash functions, Z_i represents the number of zero bits in BF_i, and Z_{12} represents the number of zero bits in the magnitude of their inner product.

Since computing bloom filters for large sets is a costly operation, we propose the use of bloom filters as an extension to the previously proposed *utility* function only for highly selectivity molecules — where the cardinality is below a manually set threshold. Given implementation specific, execution considerations we empirically set the threshold to 1000 partial results (ids) for the given set. Consequently the *extended utility EU* is now defined as follows:

$$EU_{N1,N2} = \begin{cases} w1 \cdot JOIN_{BF(N1),BF(N2)} + w2 \cdot U_{N1,N2}, & N1,\ N2 \text{ selective} \\ w2 \cdot U_{N1,N2}, & \text{otherwise} \end{cases} \quad (5)$$

where $w1$ and $w2$ are weights that define the importance of the employed estimation methods. We chose $w1 = 0.8$ and $w2 = 0.2$ for our experiments, which means that for selective molecules we favor a more expensive, but more realistic estimation.

Algorithm 2 The plan execution algorithm

```
EXECUTE-PLAN(Plan)
 1  nodes = SortedList() // initialize
 2  update = Queue()
 3  for each node N ∈ Plan
 4      PUSH(nodes, N) // Note: sorting according to selectivity gets preserved
 5  while !nodes.empty() // While joins to perform, do so
 6      best = POP(nodes)
 7      if nodes.empty()
 8          break
 9      for N ∈ nodes, where GETMOLECULE(best) ∩ GETMOLECULE(N) ≠ ∅
10          joinVariables = GETMOLECULE(best) ∩ GETMOLECULE(N)
11          selectivity = DOJOIN(join[0], best, N)
12          APPEND(update, [join[0], N, best])
13          if selectivity == 0
14              exit and stop
15          else
16              N.selectivity = selectivity
17              UPDATE(nodes, [N.selectivity, N])
18  REVERSE(update) // Now inform all hosts of elements without join partners
19  for every [join, N1, N2] ∈ update
20      UPDATE(join, N1, N2)
```

Algorithm 3 Materialing a resolved plan

```
MERGE-MATERIALIZE(Plan, Solution, Query)
 1  graph = GETGRAPH(Solution) // the molecule graph
 2  resultVariables = GETPROJECTIONS(Query) // the result variables
 3  resolved = [] // the bound result variables
 4  results = [][] // the final table of results
 5  while !resultVariables.empty()
 6      v1 = POP(resultVariables) // get next unbound result variable
 7      if resolved.empty() // no currently bound result variables
 8          v2 = GETNEARESTRESULTVARIABLE(v1, resultVariables, graph)
 9          REMOVE(projections, v2); PUSH(resolved, v2)
10      else // there are currently bound result variable
11          v2 = GETNEARESTRESULTVARIABLE(v1, resolved, graph)
12          PUSH(resolved, v2)
13      resultsTable = GETMERGETABLE(v1, v2, graph) // merge partial results (id's only)
14      if results.empty()
15          results = resultsTable
16      else
17          results = EXTEND(results, resultsTable)
18      REMOVEDUPLICATES(results)
19  MATERIALIZE(results) // turn id's into actual strings
20  return results
```

3.2 Executing Plans

Specifically, following Algorithm 2 we start by executing the most selective molecule in the plan (steps 1 and 2 in Figure 2). To perform the join (lines 10-12 in Algorithm 2) we send the results to host B and execute the join there (steps 3 and 4 in Figure 2). Similarly we join the remainder of the molecules. After all join operations have ended, we need to let hosts A and B know of all the elements that did not have a join-partner by updating its structure (lines 18-20 in Algorithm 2; steps 8 to 12 in Figure 2).

To increase execution performance, since many plans contain overlapping subqueries, we employ a *memoization* strategy by keeping partial results on the respective hosts for the duration of the query execution, while at the same time database caching strategies are in effect. As a further improvement, site-level memory caches can be employed, bypassing the database altogether for "popular" result sets.

3.3 Materializing Plans

Once a plan has finished its execution, the `Plan Executor` monitoring the process will signal the AVALANCHE mediator by pushing the executed plan onto the `Finished Plans` queue. Note that the executed plans do not contain the results yet, since the matches are kept as partial tables on their respective hosts. Hence,

plans in the `Finished Plans Queue` will be handled by a `Plan Materializer` that materializes the partial results as described in Algorithm 3. First, we get an unbound result variable $v1$ (line 6). We then try to find the next possible result variable that will produce the lowest number of merge operations (procedure getNearestResultVariable in lines 8 or 11). Having chosen the next result variable we create a partial result table (line 13) and merge it with the global result table (lines 14-17). We finish by removing duplicates and replacing all ids with the actual strings (lines 18 and 19). To further reduce the overhead of sending the results between hosts, we use RLE compression.

3.4 Stopping the query execution

Since we have no control over distribution and availability of RDF data and SPARQL endpoints, providing a complete answer to the query is an unreasonable assumption. Instead, the `Query Stopper` monitors for the following *stopping conditions*:

- a global timeout set for the whole query execution
- returning the *first K* unique results to the caller
- to avoid waiting for the timeout when the number of results is $\ll$ K we measure relative result-saturation. Specifically, we employ a sliding window to keep track of the last n received result sets. If the standard deviation (σ) of these sets falls below a given threshold then we stop execution. Using Chebyshev's inequality we stopped when $1 - \frac{1}{\sigma^2} > 0.9$.

4 Preliminary Evaluation

In this section we describe the experimental evaluation of the Avalanche system. We first succinctly introduce the setup and then discuss the two evaluated properties: the query execution and plan convergence.

4.1 Experimental setup

We tested Avalanche using a five-node cluster. Each machine had 2GB RAM and an Intel Core 2 Duo E8500 @ 3.16GHz. We chose this small number of nodes to better illustrate Avalanche's query handling strategies, but did not measure its ability to scale.

The data was gathered directly from the LOD cloud. Specifically, we employed the IEEE (66K triples), DBLP (22 millions) and ACM (13 millions) publication data. The datasets were distributed over a five-node cluster, split by their origin and chronological order (i.e. ACM articles till 2003 on host A) as shown in Table 4.1. Recall that as stated above Avalanche makes no assumptions about the data distribution over the nodes.

For the purpose of evaluating Avalanche we selected 3 SPARQL queries as listed in Appendix A. The queries were chosen in increasing order of complexity

Host	# Triples	# S / O	# DBLP P	# ACM P	# IEEE P
Host A	7058949	1699554	0	18	0
Host B	6549326	1554767	0	18	14
Host C	6547513	2153509	20	0	17
Host D	8319504	2773740	19	0	0
Host E	7399881	2680160	19	0	0

Table 1. Number of triples, unique subject S, object O, and predicate P distributions on the hosts. Predicates are shown by dataset.

(in terms of number of unbound variables and triple patterns). We conducted all query executions with the following parameters: 1) *timeout* set to 20 seconds, 2) a *stop sliding window* of size 5, 3) a *saturation threshold* of 0.9, and 4) a *selectivity threshold* for bloom filter construction of 1000 while searching for a maximum of 200 results.

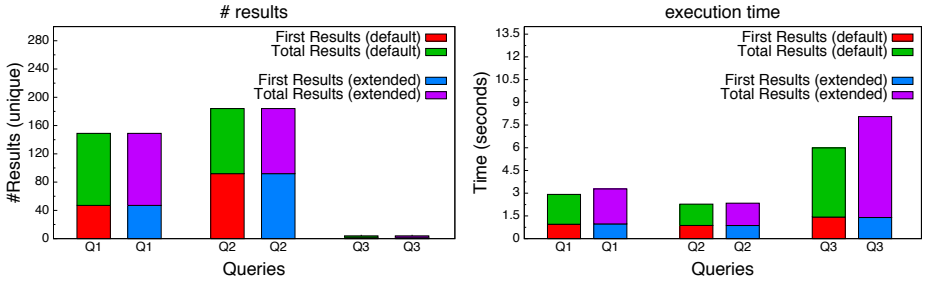


Fig. 3. Number of retrieved results and query execution times

Query execution Figure 3 graphs the number of query results (left) and the execution time (right) for both the default utility U and the extended utility EU introduced in Section 3. Note that the query execution time for the extended utility is somewhat higher (lower than the timeout), but it does find more answers to the queries. The time used for the extended utility is higher since gives the better plans a higher priority and executes them earlier. The execution of "useful" plans does take longer, since a non-useful plan is stopped as soon as an empty join is discovered. Hence, the saturation condition will stop the default utility earlier after having executed fewer useful plans. Given a large number of hosts we expect that the overhead of cancelling non-useful plans will overcome the cost of executing useful plans. Hence, the extended utility planer should converge faster.

As we see in this experiment, Avalanche is able to successfully execute query plans and retrieves many up-to-date results without having any prior

knowledge of the data distribution. We, furthermore, see that different objective functions have a significant influence on the outcome and should play a critical role when deployed on the Semantic Web.

Planner convergence A second issue we planned to investigate is the usefulness of the convergence criteria introduced in Section 3.4. Figure 4 graphs the total number of results against the number of new results where the data points represent newly arriving—possibly empty—answer sets whilst disabling the stopping condition.

As an example consider query $Q1$. At first, the number of new results grows to a certain level. But, after having gathered ≈ 140 results, no more new results are received. A similar behavior can be seen for each of the three queries. Hence, given the experimental results the choice of a stopping condition is pertinent. The current stopping conditions would stop both queries $Q1$ and $Q3$ at the right point when the correct plateau is reached. When considering the number of results found (see also Figure 3), query $Q2$, however, is stopped somewhat early in one of the local plateaus.

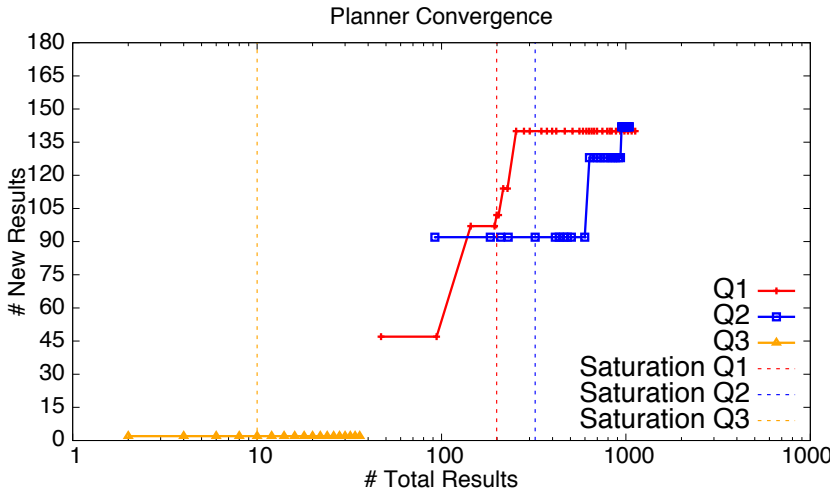


Fig. 4. Query planner convergence

5 Limitations, Optimizations and Future Work

The Avalanche system has shown how a completely heterogeneous distributed query engine that makes no assumptions about data distribution could be implemented. The current approach does have a number of limitations. In particular,

we need to better understand the employed objective functions for the planner, investigate if the requirements put on participating triple-stores are reasonable, explore if Avalanche can be changed to a stateless model, and empirically evaluate if the approach truly scales to large number of hosts. Here we discuss each of these issues in turn.

The core optimization of the Avalanche system lies in its cost and utility function. The basic utility function only considers possible joins with no information regarding the probability of the respective join. The proposed utility extension UE estimates the join probability of two highly selective molecules. Although this improves the accuracy of the objective function, its limitation to highly selective molecules is often impractical, as many queries (such as our example query) combine highly selective molecules with non-selective ones. Hence, we need to find a probabilistic distributed join cardinality estimation for low selectivity molecules. One approach might be the usage of bloom-filter caches to store precomputed, "popular" estimates. Another might be investigating sampling techniques for distributed join estimation.

In order to support Avalanche existing triple-stores should be able to:

- report statistics: **cardinalities**, bloom filters, other future extensions
- support the execution of **distributed joins** (common in distributed databases), which could be delegated to an intermediary but would be inefficient
- share the same **same key space** (can be URIs but would result in bandwidth-intensive joins and merges)

Whilst these requirements seem simple we need to investigate how complex these extensions of triple-stores are in practice. Even better would be an extension of the SPARQL standard with the above-mentioned operations, which we will attempt to propose.

The current Avalanche process assumes that hosts keep partial results throughout plan execution to reduce the cost of local database operations and that result-views are kept for the duration of a query. This limits the number of queries a host can handle. We intend to investigate if a stateless approach is feasible. Note that the simple approach—the use of REST-ful services—may not be applicable as the size of the state (i.e., the partial results) may be huge and overburden the available bandwidth.

We designed Avalanche with the need for high scalability in mind. The core idea follows the principle of *decentralization*. It also supports *asynchrony* using asynchronous HTTP requests to avoid blocking, *autonomy* by delegating the coordination and execution of the distributed join/update/merge operations to the hosts, *concurrency* through the pipeline shown in Figure 1, *symmetry* by allowing each endpoint to act as the initiating Avalanche node for a query caller, and *fault tolerance* through a number of time-outs and stopping conditions. Nonetheless, an empirical evaluation of Avalanche with a large number of hosts is still missing—a non-trivial shortcoming (due to the lack of suitable, partitioned datasets and the significant experimental complexity) we intend to address in the near future.

6 Conclusion

In this paper we presented Avalanche , a novel approach for querying the Web of Data that (1) makes no assumptions about data distribution, availability, or partitioning, (2) provides up-to-date results, and (3) is flexible since it assumes nothing about the structure of participating triple stores. Specifically, we showed that Avalanche is able to execute non-trivial queries over distributed data-sources with an ex-ante unknown data-distribution. We showed two possible utility functions to guide the planning and execution over the distributed data-sources—the basic simple model and an extended model exploiting join-estimation. We found that whilst the simple model found some results faster it did find less results than the extended model using the same stopping criteria. We believe that if we were to query huge information spaces the overhead of badly selected plans will be subdued by the better but slower plans of the extended utility function.

To our knowledge, Avalanche is the first Semantic Web query system that makes no assumptions about the data distribution whatsoever. Whilst it is only a first implementation with a number of drawbacks it represents a first important towards bringing the spirit of the web back to triple-stores—a major condition to fulfill the vision of a truly global and open Semantic Web.

Acknowledgements This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000. We would like to acknowledge Cathrin Weiss and Rob H. Warren for their help and contribution in the development and evolution of the ideas behind Avalanche.

References

1. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - The story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.
2. A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.
3. M. Cai and M. R. Frank. Rdfpeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *13th International World Wide Web Conference (WWW)*, pages 650–657, 2004.
4. O. Erling. Virtuoso. In *http://openlinksw.com/virtuoso/*.
5. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *19th International World Wide Web Conference (WWW)*, May 2010.
6. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. In *6th International Semantic Web Conference (ISWC)*, pages 211–224, 2007.
7. O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL queries over the Web of linked data. In *8th International Semantic Web Conference (ISWC)*, page 293309, October 2009.
8. O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *4th European Semantic Web Conference*, June 2007.

9. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
10. A. Langegger and W. Wöß. RDFStats - An extensible RDF statistics generator and library. In *8th International Workshop on Web Semantics, DEXA*, September 2009.
11. A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web middleware for virtual data integration on the web. In *5th European Semantic Web Conference*, June 2008.
12. O. Lassila. Programming Semantic Web applications: a synthesis of knowledge representation and semi-structured data Doctoral. *Doctoral dissertation*, 2007.
13. A. Maduko, K. Anyanwu, and A. Sheth. Estimating the cardinality of RDF graph patterns. In *16th International World Wide Web Conference (WWW)*, May 2007.
14. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *36th International Conference on Management of Data (SIGMOD)*, June 2010.
15. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. *The Semantic Web: Research and Applications*, pages 524–538, 2008.
16. S. Ramesh, O. Papapetrou, and W. Siberski. Optimizing distributed joins with bloom filters. In *ICDCIT '08: Proceedings of the 5th International Conference on Distributed Computing and Internet Technology*, pages 145–156, 2009.
17. S. Schenck and S. Staab. Networked graphs:a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *17th International World Wide Web Conference (WWW)*, April 2008.
18. A. P. Sheth and J. A. Larson. Federated databases systems for managing distributed, heterogeneous and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.
19. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *17th International World Wide Web Conference (WWW)*, April 2008.
20. J. Zemánek, S. Schenk, and V. Svatek. Optimizing SPARQL queries over disparate RDF data sources through distributed semi-joins. In *7th International Semantic Web Conference (ISWC)*, October 2007.

A Appendix

Query 1:
```
SELECT  ?title ?author ?date   WHERE {
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?paperDBLP <http://www.aktors.org/ontology/portal#has-date> ?date .
        ?author <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
    }
```
Query 2:
```
SELECT ?name ?title WHERE {
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?author <http://www.aktors.org/ontology/portal#full-name> ?name .
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?avi .
        ?paper <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?avi <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
    }
```
Query 3:
```
SELECT ?title ?date WHERE {
        ?author <http://www.aktors.org/ontology/portal#full-name> "Abraham Bernstein" .
        ?paper <http://www.aktors.org/ontology/portal#has-author> ?author .
        ?paper <http://www.aktors.org/ontology/portal#has-title> ?title .
        ?paper <http://www.aktors.org/ontology/portal#has-date> ?date .
        ?paper <http://www.aktors.org/ontology/portal#article-of-journal> ?journal .
        ?journal <http://www.aktors.org/ontology/portal#has-title> "ISWC/ASWC".
    }
```

RDFMatView: Indexing RDF Data using Materialized SPARQL queries

Roger Castillo, Christian Rothe, and Ulf Leser

Humboldt University of Berlin
{castillo,rothe,leser}@informatik.hu-berlin.de
http://www.hu-berlin.de/

Abstract. The Semantic Web aims to create a universal medium for the exchange of semantically tagged data. The idea of representing and querying this information by means of directed labelled graphs, i.e., RDF and SPARQL, has been widely accepted by the scientific community. However, even when most current implementations of RDF/SPARQL are based on ad-hoc storage systems, processing complex queries on large data sets incurs a high number of joins, which may slow down performance. In this article we propose materialized SPARQL queries as indexes on RDF data sets to reduce the number of necessary joins and thus query processing time. We provide a formal definition of materialized SPARQL queries, a cost model to evaluate their impact on query performance, a storage scheme for the materialization, and an algorithm to find the optimal set of indexes given a query. We also present and evaluate different approaches to integrate materialized queries into an existing SPARQL query engine. An evaluation shows that our approach can drastically decrease the query processing time compared to a direct evaluation.

Key words: SPARQL, Indexing, RDF, Materialized Queries, Semantic Web, Query Processing

1 Introduction

The *Semantic Web* as an evolution of the *World Wide Web* aims to create a universal medium for the exchange of data where data can be shared and processed by automated tools as well as by people. The basis for this proposal is a logical data model called Resource Description Framework (RDF) [1]. An RDF data set is a collection of statements called *triples*, of the form *(s,p,o)* where s is a subject, p is a predicate and o is an object. Each triple states the relation between subject and object. A set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes. The SPARQL query language is the official standard for searching over RDF repositories [2].

The increasing amount of RDF data has motivated the development of approaches for efficient RDF data management. Therein, SPARQL implementations have been built either over relational database technology or using an ad-hoc storage system (e.g. Jena [3], 3Store [4, 5], Sesame [6]). Furthermore, very

large scale systems have been proposed using the common paradigm of a triple table normalized using two or more tables (4Store [7], YARS [8]). Consequently, in these systems, joins are still used extensively to answer queries. Optimizing these joins is one of the critical issues to obtain scalable SPARQL systems.

In relational databases, query processing using materialized views is a well established method to achieve scalability [9]. Here, we propose the use of materialized SPARQL queries to speed-up queries. We target large data sets and SPARQL queries consisting of many basic graph patterns producing a set of results. Examples of huge data sets are for instance, UniProt containing more than 600 million triples [10] or the W3C SWEO Linking Open Data Community with more than 4 billion triples [11]. With such datasets, executing a query with many graph patterns becomes a problem.

Listing 1. Example SPARQL query to gather information about Hexokinase enzyme [12]

```
SELECT * WHERE {
   ?s1 ?p1 ?o1 .
   ?o1 ?p2 "hexokinase" .
   ?s1 rdf:type ?type1 .
   ?s1 rdfs:comment ?comment1 .
   ?s1 rdfs:label ?label1 .
   ?s1 rdfs:comment ?comment2 .
   ?s1 rdfs:label ?label2 .
}
```

Consider the query in Listing1. Executing this query on a conventional SPARQL processor over a large triple table results in the computation of six self-joins. However, one can safely assume that the types, labels, and comments of an object are used together very often. Therefore, similar to [3], which create tables to group properties that tend to be defined together we suggest to cluster frequently used *triple patterns* by materializing and storing the results inside the system. If this information were available, the query could be computed with only three joins, as the materialized query would help to retrieve the information for $s1$.

Our indexing method aims to fully exploit the RDF graph-structure. We do not index single attributes or triples, but fractions of queries that occur frequently in an expected workload. Therefore, our approach is a *native RDF/SPARQL indexing* method whose concepts are viable for all possible implementations of RDF stores. Our method can be seen as an orthogonal indexing solution, which may be used in conjunction with other indexing methods.

Such an approach requires to solve several problems. First, selected queries must be materialized and the results stored such that efficient retrieval is possible. Second, a given query at runtime must be analyzed to identify the materialized query or the combination of materialized queries that offers the highest speed-up for this query. This requires a query rewriting algorithm and a cost model. Third, the query processing itself must be modified to be able to retrieve materialized results and to combine them with those parts of the original query that are not covered by the indexes.

Here, we present solutions to these problems. We first discuss related work in Section 2. Section 3 presents the fundamental principles of *RDFMatViews*. We describe different ways to introduce materialized queries into an existing SPARQL processor in 4. Section 5 gives an evaluation of our method. We conclude in Section 6.

2 Related work

In the following, we discuss those works that are most related to our main contribution, i.e., using indexes to speed up SPARQL queries.

Some approaches have proved to be very efficient to query SPARQL queries based either on relational database technology or following a native data scheme. For instance, in [13] Abadi et al. propose a vertical partition approach for Semantic Web data management. An enhancement of this approach is proposed by Weiss et al. in [14]. Therein, RDF data is indexed in six possible ways, i.e., an index for each possible ordering of the three RDF elements. Each instance of an RDF element is associated with two vectors; each such vector gathers elements of one of the other types, along with lists of the third-type resources attached to each vector element. This scheme is capable of speeding up single joins tremendously, but storage requirements are very high, which becomes a serious issue when using huge data sets.

Neumann and Weikum developed *RDF-3X*, a SPARQL engine pursuing a RISC-style architecture – a streamlined architecture – with specific-designed data structures and operations [15]. The authors overcome the "giant-triples-table" [13] bottleneck by creating a set of indexes and a fast way for processing merge joins. Similar to [14], RDF-3X maintains six possible permutations of subject, predicate and object in six separate indexes. The authors also present a compression algorithm to decrease the space consumption.

All these approaches have in common that they focus on indexing the relational representation of the RDF data. When faced with queries consisting of multiple basic graph patterns, they still have to compute multiple joins (although every single join is faster). In contrast, our work specifically targets the speed-up of complex queries consisting of many basic graph patterns by indexing complete query patterns which occur in other queries.

There is some other work along this line. In [16] the authors present *GRIN*, a lightweight indexing mechanism for RDF data. The idea is to draw circles around selected *center* vertices in the graph where the circle would comprise those vertices in the graph that are within a given distance of the "center" vertex. Basically, GRIN is a binary tree where the set of leaf nodes form a partition of the set of triples in the RDF graph. An interior node represents the set of all vertices in the RDF graph that are within a specific distance. To evaluate a query, GRIN derives a set of inequality constraints from the query. These constraints are evaluated against the nodes of the GRIN index.

A similar indexing approach is presented in [17]. This work proposes a set of indexes of precomputed joins created from all possible join combinations be-

tween triple patterns. As [16], this approach creates a general purpose set of indexes based on joined triple patterns, but the number of indexes to manage is impractical when the number of joined triples is >= 3.

The two systems just described index larger portions of the RDF data set and not just single triples. However, they propose to apply their techniques to all RDF triples, while we only build user-chosen indexes. Our work fundamentally is based on the assumption that some patterns are combined more frequently than others, and that only indexing those combinations promises to provide large speed-ups at manageable space and maintenance cost.

The differences between our ideas and that of other RDF indexing schemes can be described by drawing a parallel to B*-indexes in relational databases [18]. Nobody would suggest to speed up queries by indexing every attribute; instead, systems assume that developers have a rough idea about the types of queries that need to be answered and therefore index only the relevant attributes. Furthermore, optimal speed-up can only be achieved when also combinations of attributes can be indexed, and not only single attributes. In this sense, the former approaches index every single attribute, the latter indexes every possible combination of attributes, and we suggest to index only selected combinations of attributes.

Note that we do not claim that our current implementation of RDFMatView offers a particular fast SPARQL-processor, compared to systems such as [13–15]. Instead, we present a new technique to speed up query execution with SPARQL that is applicable to any SPARQL query processor. We showcase its potential and compare different ways to integrate it into query processing using one particular system (namely ARQ [19]), which was been chosen because of its widespread use. An extended version of this paper can be found in [20].

3 The RDFMatView Approach

3.1 Indexes and Covers

A SPARQL query Q is represented by a simple graph pattern P and is denoted by $P(Q)$. A mapping is a function that maps the symbols of one pattern into the symbols of another pattern. Our notion of mappings is based on the SPARQL-Standard [2] and its definition of *pattern solutions*. However, while in the SPARQL standard such solutions are only searched in the data graph, we also permit that variables are mapped to other variables. This generalization allows us to search occurrences of patterns in other patterns, in particular, occurrences of indexes in a query. We say that a pattern P_1 occurs in a pattern P_2 if there is a mapping function S such that $S(P_1) \subseteq P_2$. Extending occurrences and mappings also to RDF triples, we define an index over an RDF data graph as follows.

Definition 1 (Index). *An* index *I over a data graph G is a pair $I = (P, O)$, where P represents a pattern and O represents the set of all occurrences of P in G.*

Indexes are precomputed queries suitable to speed up other queries when the index pattern is "contained" in the query pattern. An index I is eligible for a query Q when the patterns set of I occurs in the pattern set of Q. However, it would be more advantageous when query processing uses more than one index. For those cases, we require that indexes "overlap". Overlapping indexes are good candidates for reducing query processing because the query engine can combine occurrences of these indexes and generate solutions without matching against the RDF dataset.

We define two ways in which indexes overlap. Two indexes overlap intensionally iff there *could* exist a triple pattern in which their materialization would overlap. Two indexes overlap extensionally if their materializations overlap on a concrete data graph. Thus, intensional overlap relies only on the index patterns and is independent of a concrete data graph, while extensional overlap needs to consider the actual data graph.

Definition 2 (Overlapping Indexes). *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph G.*

- *I_1 and I_2 intensionally overlap iff there exists mapping functions S_1, S_2 such that*

$$S_1(P_1) \cap S_2(P_2) \neq \emptyset$$

- *I_1 and I_2 extensionally overlap in G iff*

$$\exists o_1 \in O_1, o_2 \in O_2 : o_1 = o_2$$

where o_i is a concrete occurrence in O_i.

Here, we only consider intensional overlapping since it is independent from the data graph and can be efficiently implemented. Using intensionally overlapping indexes, we define a cover. Think of set E which contains all the eligible indexes as graph nodes, connected by an edge when they overlap. For a given query Q, we call every subset of E a *cover of Q*, for which the induced subgraph has a single component. Furthermore, we are only interested in *maximal covers*, i.e., those covers which cannot be extended further by adding new indexes.

3.2 Example

Finding covers requires to analyze the set of indexes and the given query. The idea is to find mappings between index and query patterns. This process is performed using a query containment algorithm [21] adapted for SPARQL queries. Details of the algorithm can be found in [20]. Essentially, we find all mappings between any index pattern and the query pattern by enumerating all possible cases. If a mapping exists, the index is eligible for that query and we store the mapping. Note that, for a given index, there are potentially many different ways to be eligible, i.e., different mappings between index and query patterns. Consider a SPARQL query and two indexes described in Table 1.

Table 1. SPARQL query returning universities and their departments, Index1 computes places and their names and Index2 computes universities with their departments.

```
Query: SELECT * WHERE {
       ?university  rdf:type ub:University;
         ub:name ?university_name.
       ?ub_department rdf:type ub:Department;
         ub:name ?ub_name_department;
       ub:subOrganizationOf ?university . }
Index1: SELECT * WHERE {
         ?place  rdf:type ?place_type;
             ub:name ?place_name. }
Index2: SELECT * WHERE {
         ?ub_department rdf:type ub:Department;
             ub:name ?ub_name_department;
             ub:subOrganizationOf ?university;
         ?university  rdf:type ub:University.}
```

Index1 and Index2 are eligible for the query, using the mappings in Table 2. Note that Index1 has two mappings. Each mapping represents an occurrence of the patterns of Index1 in the query pattern. Index occurrences generated from the previous mapping functions overlap in the triple pattern *?university rdf:type ub:University*, using the first mapping function of Index1. Hence, partial results can be joined to completely cover the query.

Table 2. Mappings of Index1 and Index2

Index1: Mapping 1			**Mapping 2**		
?place	⇒	?university	?place	⇒	?ub_department
?place_type	⇒	ub:University	?place_type	⇒	ub:Department
?place_name	⇒	?university_name	?place_name	⇒	?ub_name_department
Index2: Mapping 1					
?ub_department	⇒	?ub_department			
?ub_name_department	⇒	?ub_name_department			
?university	⇒	?university			

Assume an RDF data stored in a RDBMS within a *triple* table (without indexes), for instance, *Triple(subj, prop, obj)* [3, 15]. Hence, query in Table 1 could be answered by the SQL query in Listing 2 requiring four self joins. However, using pre-computed tables, Index1 and Index2, requires only one join as shown in Listing 3.

Listing 2. SQL representation of SPARQL query in Table 1

```
SELECT t1.subj AS a0, t2.obj AS a1, t3.subj AS a2,t4.obj AS a3
FROM Triple AS t1, Triple AS t2, Triple AS t3,
     Triple AS t4, Triple AS t5
WHERE t1.prop= 'type' AND t1.obj= 'University' AND
      t2.prop= 'name' AND t3.prop= 'type' AND
      t3.obj= 'Department' AND t4.prop= 'name' AND
      t5.prop= 'subOrganizationOf' AND
      t1.subj = t2.subj AND t3.subj = t4.subj AND
      t3.subj = t5.subj AND t1.subj = t5.obj;
```

Listing 3. SQL representation of SPARQL query in Table 1 using RDFMatView indexes

```
SELECT index2.university, index1.place_name AS university_name,
       index2.ub_department, index2.ub_name_department
FROM index1, index2
WHERE index1.place = index2.university;
```

This example illustrates advantages to use materialized queries as indexes to process SPARQL queries. Note that this case does not require to query against the data graph, because all query patterns are covered and the partial results are materialized. Other cases would require to extend the results of the covered patterns with the results of query patterns which are not covered using indexes.

3.3 Cost Model

Previous sections define which indexes and which sets of indexes are eligible for a given query. In the following, we define a model to estimate which cover brings more savings in time to query execution. Our model is based on the definition of *selectivity* of an index. Selectivity defines the relation of the number of index occurrences in a given graph to the possible total number of index occurrences in the graph. To this end, we need the *size and frequency of the index pattern* (number of triples in the index pattern and number of tuples for the index pattern in the data graph) and the *size of the data graph* (total number of triples) represented by $|I|$, $\#(I)$ and $|G|$ respectively. Hence, we define selectivity as follows.

Definition 3 (Selectivity of an index). *Let I be an index over a data graph G. The selectivity $s(I)$ of an index I is defined as:*

$$s(I) = \frac{\#(I)}{|G|^{|I|}}.$$

We can estimate selectivity of two indexes based on the overlapping of their index patterns, i.e., they can completely, partially or not overlap at all. This leads to estimate the selectivity of a cover.

Definition 4 (Selectivity of a cover). *Let C be a cover for a query Q consisting of indexes $I_1, I_2, \ldots I_n$. The selectivity $s(C)$ of the cover C is defined as:*

$$sel(C) = sel(I_1 \cup I_2 \cup \ldots \cup I_n) \leq \frac{min|O_1|, ..., |O_n|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

Having the selectivity of all maximal covers, the query optimizer determines which cover is the best for query processing.

4 Implementation

We describe the implementation of our approach into a SPARQL query processor by using the ARQ system [19]. However, we want to stress that the general process would be the same for any other SPARQL query processors. We differentiate two main phases in our approach. At offline-time, indexes are created and materialized. At query-time, queries are answered using indexes (or covers). We describe our implementation regarding these phases.

4.1 RDFMatView Index processing

Each index is preprocessed as a table in the underlying database. We create its schema by materializing all variables of the index and not only those mentioned in the SELECT clause. This strategy enables the materialized data to be eligible also for those queries requesting variables that were not selected in the original index. Occurrences of the index in the dataset are stored as values for these fields. Each attribute of a tuple represents a binding for the respective variable. To avoid large requirements of storage space, we use an *RDF Data Dictionary*, which maps each resource to a unique identifier. Thus, instead of storing large literal values, we store only numeric identifiers in the index structure. Table 3 summarizes the space required to store 10 indexes for each RDF dataset. During the processing of an index we also calculate and store index properties, for instance size and frequency, which are later used to evaluate the query execution plans. These tasks are executed only once per index and can be used to process any SPARQL query.

Table 3. Storage required for a set of 10 RDFMatView indexes over 4 different databases (BSBM) [22] (K = 1000, M = 1 Million triples).

	$250K$	$500K$	$1M$	$10M$
index storage	12Mb	18Mb	34Mb	363Mb
total storage	379Mb	616Mb	1.2Gb	12Gb
storage ratio	0.03	0.02	0.02	0.03

4.2 Executing a query using RDFMatView indexes

Query processing using *RDFMatViews indexes* usually combines results of multiple indexes. However, it is not always possible to cover all query patterns. The set of uncovered patterns is referred to as *residual part of a query*. This breaks down into the following steps: i) Analysis of the query to find all maximal covers ii) Selection of the most suitable cover to answer the query given our cost model iii) Rewriting of the query using the chosen cover iv) Extension of the results of the cover to results of the query. Steps one and two were already discussed in Sections 3.1 and 3.3 respectively. Here, we concentrate on the two last steps, the query rewriting. We developed three strategies to fulfill this task:

- Our first strategy uses ARQ to process the residual part of the query. RDFMatView extends the results of the chosen cover by joining the partial solutions with the solutions of the residual patterns.
- The second strategy is based on a SPARQL-to-SQL algorithm to translate SPARQL queries into SQL queries. The idea is to directly access the native Jena storage tables and to combine those results with the index tables to generate the final solution.
- The last strategy is built from a combination of the previous two strategies, i.e. ARQ and database execution engine.

These strategies are explained in detail in the next sections.

Method 1: MatView-and-ARQ Engine

Rewriting engine built on top of the Jena Framework. Given a query and a cover, it computes the set of uncovered residual patterns of the query and uses ARQ to execute this (sub-)query. Furthermore, it computes the result of the cover by joining the respective index tables according to the variable mappings between the query and the indexes forming the cover. Results are also joined with the data dictionary to obtain RDF values, and joined to the result of the ARQ query to produce the complete answer for the original query. This engine encapsulates the logic for the execution of the cover and provides total independence from the underlying database.

Method 2: MatView-to-SQL Engine

Rewriting engine which, unlike our first method, translates the residual part of the query into a SQL query using an algorithm proposed by Chebotko in [23]. The SQL query is executed by the RDBMS which evaluates the query using the Jena tables. The result set is processed using our dictionary and combined with the results of the cover. The complete query processing is performed by the database execution engine using a stored procedure.

Method 3: Hybrid Engine

A mixture of MatView-and-ARQ and MatView-to-SQL. As in Method 1, after rewriting the query, this engine transfers the residual patterns to the query execution engine of ARQ. The second part of the process combines the results of the residual patterns with the resulting set of the covered part of the query patterns. Contrary to Method 1, this engine is database-dependent since this task is performed inside the database execution engine, as in Method 2.

5 Evaluation

We evaluate our approach using two well known SPARQL benchmarks: the Berlin SPARQL Benchmark (BSBM) [22] and the SPARQL Performance Benchmark (SP^2B) [24]. We use the ARQ/Jena RDF Storage System (version 2.5.7) on Postgres 8.2 as framework in which we integrated our solution. We generated eight RDF datasets with sizes ranging from 250K to 10M triples and tested the impact of the indexes on six different queries (three queries for each benchmark). For each query, we manually defined a set of indexes, leading to covers composed of one to three indexes. Our intention here is not to find the best set of indexes given a workload (generally called index selection, see, e.g., [25–27]); instead, we study to which degree indexes that use different processing schemes speed up the execution of queries.

5.1 Dataset and queries

For each benchmark, we create four datasets containing 250K, 500K, 1M and 10M triples, respectively. As these datasets have identical value distributions but different sizes, our evaluation concentrates on the scalability of our methods in different domains. Based on the number of triple patterns we chose three queries for each benchmark. We transformed the query patterns into simple graph patterns and removed most bindings to variables. Bounded variables incur high selectivity resulting in the retrieval of only a handful of triples. Such queries are well supported by existing index structures and do not require the type of join-optimization that is achieved with our optimization technique. Therefore, performance gains would be only marginal. Our test queries are described in Listings 4 and 5.

Listing 4. Test queries derived from BSBM

```
Query1: Finds products for a given set of generic features.
Query2: Retrieve basic information about products.
Query3: Retrieve in-depth information about products including
offers and reviews.
```

Listing 5. Test queries derived from SP^{2}B

```
Query4: Extract all information about inproceedings documents.
Query5: Select all pairs of articles of an author that have been
published in the same journal.
Query6: Return for each year, the set of all publications including
the name of the authors.
```

From the queries described in Listings 4 and 5, we derive two sets of indexes containing 10 and 8 indexes respectively. Each index covers two to six patterns from at least one query. However, none of them completely covers a query. We focus to evaluate covers containing either a combination of indexes and possible a residual part of the query since most real-life SPARQL queries would comply with this case.

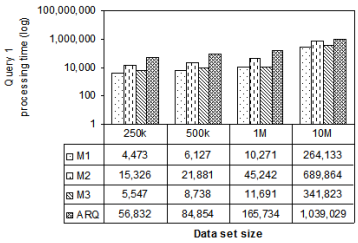
5.2 Results

For each benchmark we evaluated three queries over four data sets using our three RDFMatView methods and plain ARQ (without indexes), which amounts 36 different configurations. We refer to the approaches to query execution as M1 for MatView-and-ARQ, M2 for MatView-to-SQL, M3 for the hybrid approach, and ARQ for plain ARQ. The experiments use the optimal cover and evaluate the real and estimated costs of different covers for the same query. All queries were executed 5 times and average execution times are reported[1]. Furthermore, we evaluated all different covers generated for Query1 and Query2 (BSBM) to show the performance of our cost model in the selection of the query execution plan. Figure 1 illustrates the average processing time for each query. Clearly, processing time significantly improves in both domains when using MatView-and-ARQ (M1) and Hybrid (M3). However, processing time does not significantly improve when using MatView-to-SQL (M2) (see Fig. 1). The reason for this is the Jena native storage schema. Since the values are encoded following the Jena layout, our process needs to parse the stored values and extract the required information, which increases the processing time.

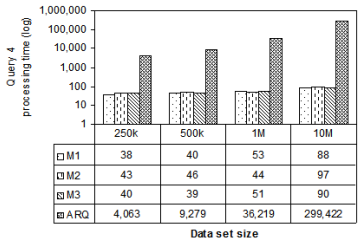
Figure 2(a) and Figure 3(a) show the evaluation of real and estimated cost for different covers for Query1 and Query2 (BSBM). Additionally, Figure 2(b) and Figure 3(b) show the relation between the estimated costs of a cover, its indexes and number of covered and uncovered patterns from the given query. Note that our system selects as optimal Cover 6 in Figure 2(a) (for Query1) and Cover 3 in Figure 3(a) (for Query2).

Figure 2(a) and Figure 3(a) show the costs estimated by our model together with the real processing time. In all cases our model manages to prevent the selection of exceptionally bad plans, and all plans improve the total execution times when compared to those without using indexes. However, the figures also show that our model can be improved further, as total real and estimated costs do not correlate well. Especially, our model does not yet reflect the fact that, in

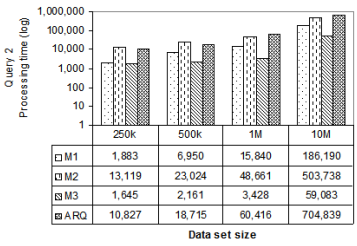
[1] Except for Query5 without indexes over a 10 Million triples dataset, which did not finish after 24 hours



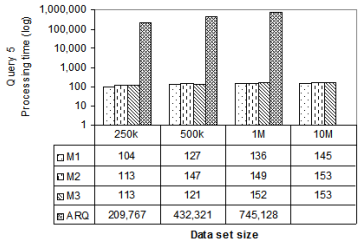
(a) Query1 (BSBM)



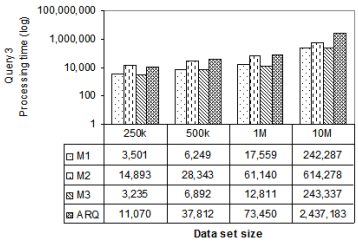
(b) Query4 (SP^{2}B)



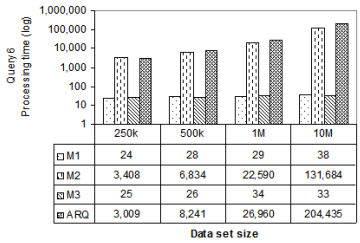
(c) Query2 (BSBM)



(d) Query5 (SP^{2}B)

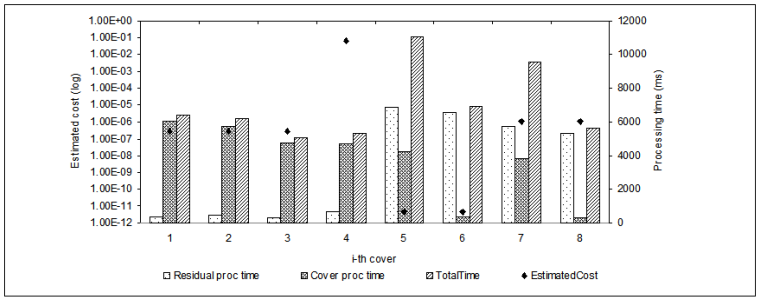


(e) Query3 (BSBM)

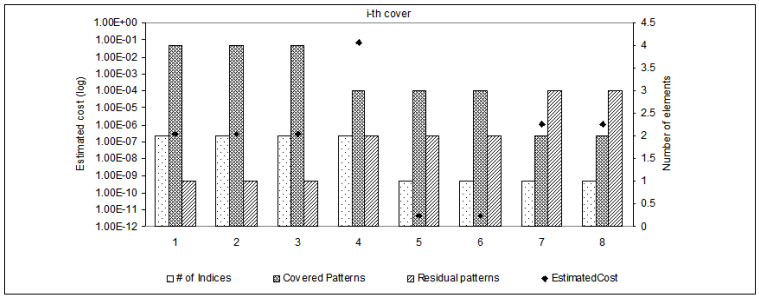


(f) Query6 (SP^{2}B)

Fig. 1. Processing time for test queries using BSBM and SP^{2}B. Each query was processed on four data sets using three rewriting methods. M1: MatView-and-ARQ; M2: MatView-to-SQL; M3: Hybrid; ARQ: plain ARQ (time in milliseconds).



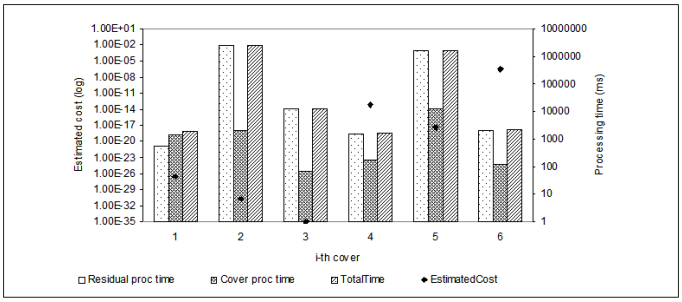
(a) Estimated cost vs. real processing time



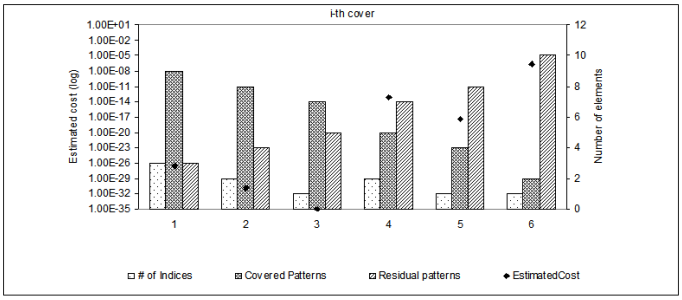
(b) Estimated cost vs elements processing time

Fig. 2. Figure 2(a) shows estimated cost, total real processing time, cover processing time, and residual processing time of Query1. Values are plotted on log-scale. Note that total real processing time virtually equals real processing time for the covers for larger covers. Figure 2(b) shows estimated costs for each cover based on intensional dependency between indexes for the same query. Costs are based on the model introduced in Section 3.3 and are presented in relation to the size for each cover, number of participating indexes and the size of the residual part of the query. This analysis shows the influence of these elements in the selection of an optimal cover.

a setting with two covers both covering the same number of patterns, but with a different number of indexes, it is usually advantageous to chose the cover with less indexes as this requires less joins at runtime. Figure 2(b) illustrates that plans with fewer indexes have a superior performance than plans with the same number of covered patterns, but consisting of more indexes. Thus, the number of necessary joins between indexes is a natural next factor to consider in future work. In Figure 2(b), Covers 5 and 6 have the best estimated costs according to our model. However, the residual part of the query (2 triple patterns) incurs an undesirable overhead, which is not yet properly reflected in our model. An interesting fact can be observed for those covers covering larger patterns using two indexes (see covers 1, 2 and 3). These cases show the reduction of processing time when joining two indexes. At the end, more patterns are covered and the number of patterns to match against the data set decreases. Though their cost



(a) Estimated cost vs. real processing time



(b) Estimated cost vs elements processing time

Fig. 3. Figure 3(a) shows estimated versus real cost for Query2. Estimated costs correlate with cover real processing time however, residual processing time consumes most of the real processing time. Figure 3(b) shows for Query2 estimated cost versus number of covered patterns and number of indexes; for explanation, see Figure 2(b).

estimation is not the best, their processing times are significantly better than those of covers with a better estimated cost. We attribute this behavior to the join (between indexes) and the processing of the residual part of the query which decreases the fewer are the patterns.

As for Query1, Figure 3(a) shows that the estimated costs and the real processing time for Query2 approximately correlate. Additionally, the graphic shows that the residual part of the query should be considered as an important factor when selecting an optimal cover, since residual processing time nearly spans the complete total processing time. Figure 3(b) supports this conclusion showing the details for the generated covers, i.e., covering a larger number of query patterns using as few indexes as possible decreases the real processing time.

6 Conclusions and future work

In this article we proposed a logical framework and a prototype implementation for answering SPARQL queries using materialized queries as indexes. At runtime, queries are analyzed to see whether their execution can be sped-up by using one

or more of those precomputed partial results. The subsequent query rewriting and integration of precomputed results into the overall result generation was implemented following three different approaches on a standard SPARQL query processor. Initial experiments with different queries, different indexes, and different data sets showed that the performance gains in query processing can be considerable.

However, a closer look reveals that our cost model needs to be improved in several aspects. In particular, it needs to model the influence of the number of used indexes, the size of the covers, and the number of residual query patterns. A more accurate estimation of the impact of these elements and its inclusion in the cost model is important to select an better cover.

Up to now, we assume a predefined set of indexes suitable for a given workload of SPARQL queries. A natural extension to this assumption is to study ways of finding the optimal set of indexes under some resources constraints, given a workload. We report on some initial results in this direction in [28], but these also need to be improved by using a better cost model.

References

1. Manola, F., Miller, E.: RDF Primer (February 2004) W3C Recommendation.
2. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (April 2008) W3C Recommendation.
3. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: Proc. First International Workshop on Semantic Web and Databases. (2003)
4. Stephen Harris, N.G.: 3store: Efficient Bulk RDF Storage. In: 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003)
5. Harris, S.: SPARQL Query Processing with Conventional Relational Database Systems. In: International Workshop on Scalable Semantic Web Knowledge Base System. (2005)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: International Semantic Web Conference. (2002) 54–68
7. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). (2009)
8. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: LA-WEB '05: Proceedings of the Third Latin American Web Congress, Washington, DC, USA, IEEE Computer Society (2005) 71
9. Goldstein, J., Larson, P.A.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In: SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (2001) 331–342
10. Dataset, U.R.: http://dev.isb-sib.ch/projects/uniprot-rdf/
11. Project, W.S.C.: Linking Open Data on the Semantic Web.
12. Bio2RDF. http://bio2rdf.org/ (2009)

13. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management using Vertical Partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
14. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
15. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. **1**(1) (2008) 647–659
16. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A Graph Based RDF Index. In: AAAI. (2007) 1465–1470
17. Groppe, S., Groppe, J., Linnemann, V.: Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In Cardoso, J., Cordeiro, J., Filipe, J., eds.: Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI), Funchal, Madeira, Portugal, INSTICC (June 12 - 16 2007) 13–20
18. Connolly, T.M., Begg, C.E., Strachan, A.D.: Database Systems: A Practical Approach to Design, Implementation and Management. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1996)
19. ARQJena: ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/ (2010)
20. Castillo, R., Leser, U., Rothe, C.: RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report 234, Humboldt Universitaet zu Berlin (2010)
21. Halevy, A.Y.: Answering Queries Using Views: A Survey. The VLDB Journal **10**(4) (2001) 270–294
22. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems, 2009 (2009)
23. Chebotko, A., Lu, S., Jamil, H.M., Fotouhi, F.: Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, Department of Computer Science, Wayne State University (2006)
24. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2Bench: A SPARQL Performance Benchmark. Data Engineering, International Conference on **0** (2009) 222–233
25. Comer, D.: The Difficulty of Optimum Index Selection. ACM Trans. Database Syst. **3**(4) (1978) 440–445
26. Caprara, A., Fischetti, M., Maio, D.: Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. IEEE Transactions on Knowledge and Data Engineering **7**(6) (1995) 955–967
27. Chaudhuri, S., Narasayya, V.R.: An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In: VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997) 146–155
28. Castillo, R., Leser, U.: Selecting Materialized Views for RDF Data. In: Semantic Web Information Management Workshop (SWIM 2010). (2010)

B+Hash Tree: Optimizing query execution times for on-Disk Semantic Web data structures

Minh Khoa Nguyen, Cosmin Basca, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{lastname}@ifi.uzh.ch

Abstract. The increasing growth of the Semantic Web has substantially enlarged the amount of data available in RDF format. One proposed solution is to map RDF data to relational databases (RDBs). The lack of a common schema, however, makes this mapping inefficient. Some RDF-native solutions use B+Trees, which are potentially becoming a bottleneck, as the single key-space approach of the Semantic Web may even make their $O(log(n))$ worst case performance too costly. Alternatives, such as hash-based approaches, suffer from insufficient update and scan performance. In this paper we propose a novel type of index structure called a B+Hash Tree, which combines the strengths of traditional B-Trees with the speedy constant-time lookup of a hash-based structure. Our main research idea is to enhance the B+Tree with a Hash Map to enable constant retrieval time instead of the common logarithmic one of the B+Tree. The result is a scalable, updatable, and lookup-optimized, on-disk index-structure that is especially suitable for the large key-spaces of RDF datasets. We evaluate the approach against existing RDF indexing schemes using two commonly used datasets and show that a B+Hash Tree is at least twice as fast as its competitors – an advantage that we show should grow as dataset sizes increase.

1 Introduction

The increasing growth of the Semantic Web has substantially increased the amount of data available in RDF[1] format. This growth necessitates the availability of scalable and fast data structures to index and store RDF. Traditional approaches store RDF in relational databases. Mapping RDF to a relational database typically follows one of the following approaches: (1) all triples are mapped to a single three column table – an approach which will result in numerous inefficient self-joins of that table, (2) every property gets mapped to its own three column table [1] – resulting in a high number of Unions for property-unbound queries and a table creation for every newly encountered property type, or (3) draws upon domain-knowledge to map properties to a relational database schema – forgoing some flexibility when adding new properties. Moreover, according to Abadi and Weiss, storing dynamically semi-structured data such as RDF in relational databases may cause a high number of NULL values in the tables, which imposes a significant computational overhead [15, 1]. As a consequence, many native RDF databases have been proposed [15, 10].

[1] `http://www.w3.org/RDF/`

Most native RDF databases propose mapping the RDF-graph to some existing indexing scheme. The most straightforward approach, RDF-3X [10] essentially proposes to store all possible subsets of the triple keys (i.e., s, p, and o from every $< subject, predicate, object >$ triple) as composite keys in a traditional B+Tree structure. This approach results in 15 B+Trees, each of which having large-keyspace (e.g., sizes of $|s| \cdot |p| \cdot |o|$, $|s| \cdot |o|$, etc.) and many entries. Given the $O(log(n))$ access time for single key lookup, this can result in a considerable time overhead for some queries. Consequently, given the ever increasing amount of data to be stored in RDF stores, traditional approaches relying on B+Trees in the sprit of RDF-3X have the potential of becoming a main bottle-neck to scalability [14]. Taking the adaptation to RDF structures to the extreme, Weiss and colleagues [15] propose a specialized index consisting of 3-level cascades of ordered lists and hashes. This approach provides a constant (i.e., $O(c)$) lookup time but has the drawback that updating hashes can become quite costly. Whilst the authors argue that updates in the Semantic Web are oftentimes rare, they are, however, common and should not be dismissed.

In this paper we propose a novel type of index structure called a B+Hash Tree, which combines the strengths of traditional B+Trees (i.e., ease of use and updatability) with the speedy lookup of a hash-based structure. Our main research idea is to enhance the B+Tree with a Hash Map to enable constant retrieval time instead of the common logarithmic one of the B+Tree. The result is a **scalable**, **updatable** and **lookup-optimized**, on-disk index-structure that is especially suitable for the large key-spaces of RDF datasets. Consequently, the main contribution of this paper is the presentation, formalization, and evaluation of the B+Hash Tree.

This paper is structured as follows. In Section 2 we set the stage with a discussion of related work, its benefits and drawbacks. Section 3 then introduces the B+Hash Tree, provides a formalization as well as a cost model thereof, and discusses some its limitations. In Section 4 we empirically compare the B+Hash Tree to the RDF-3X like approach storing the key in a B+Tree. In the final section we summarize our conclusions and discuss future work.

2 Related work

Several architectures for storing Semantic Web data have been proposed. Many of them use relational databases to index RDF data. Row store systems such as *Jena* [16, 17] map RDF triples into a relational structure, which results in creating a giant three column $< subject, predicate, object >$ table. Having a single large table, however, oftentimes results in expensive self-joins; in particular if the basic graph patterns of a query are not very selective. To counter this problem, Jena creates *property tables*, which combine a collection of properties of a resource in one table. Whilst this approach reduces the number of self-joins it (1) assumes that the RDF actually has some common exploitable structure that does not change often over time and (2) has the potential to result in a large number of NULL values where properties are missing from some resources in a table [1]. Hence, this inflexibility and the NULL values may lead to a significant computational overhead [15].

An alternative approach to the property table solution are column stores such as *SW-Store* [1]. For each unique property of the RDF dataset, SW-Store creates a two column table containing the *subject* and the *object*. Assuming a run-length encoding of the column, this provides a compact storage mechanism for RDF data that allows efficient joins, as only the join columns are retrieved from disk (as opposed to the table in row-stores). Nevertheless, if a graph pattern has an unbound property (e.g., $< s, ?p, ?o >$) then an increased number of joins and unions are inevitable [15].

A recent approach is *Hexastore* [15], which stores RDF data in a native disk vector-based index. Hexastore manages all six possible orderings of the RDF triple keys in their own index. In each of the six indices, a triple is split into its three levels: All levels are stored in native on-disk sorted vectors. A lookup of the triple $< s, p, o >$ would, hence, result in a hash-lookup in the first level of s, the result of which would point to a second-level hash that would be used to lookup p, which would point to an ordered list containing o. Given that a hash-lookup can be achieved in constant time, Hexastore provides a constant-time lookup for any given triple. Its six-fold indexing allows a fast lookup of any triple pattern at the cost of a worst-case five fold space cost. The biggest drawback, however, is that Hexastore in its "pure" implementation does not support incremental updates, as inserts would require resorting the vectors of each of the six indices – a time consuming process.

RDF-3X [10] avoids this drawback by storing the data in B+Trees instead of vector lists. Specifically, it stores every possible subset combination of the three triple keys in a separate B+Tree. This approach allows updates and has an $O(log(n))$ complexity for retrieval, updates, and deletion. Note, however, that this approach leads to a huge key-space for some B+Trees (at worst $|s| \cdot |p| \cdot |o|$), as the composite key-space grows in the square of the number of nodes of the RDF graph. Hence, even $O(log(n))$ can become a bottleneck. A similar approach to RDF-3X or Hexastore is *YARS2* [5], which indexes a certain subset of triples in 6 separate B-Trees or Hash Tables. By not treating all possible $< s, p, o >$ subset combination equally, the missed indexes must be created by joining other indexes, which can be a time-consuming process.

A more recent approach is *BitMat* [2], which is a compact in-memory storage for RDF graphs. BitMat stores RDF data as a 3D bit-cube, where each dimension represents the subjects, the objects, and the predicates. When retrieving data the 3D bit-cube can be sliced for example along the "predicates-dimension" to get a 2D matrix. In each cell of the matrix, the value 1 or 0 denotes the presence respectively the absence of a subject and object bounded by the predicate of that matrix. Since bitwise operations are cheap, the major advantage of the BitMat index is its performance when executing low-selectivity queries. Nevertheless, BitMat is constrained by the available memory and, as the authors have shown in their evaluation, traditional approaches such as RDF-3X or MonetDB [8] outperform BitMat on high-selectivity queries.

In real-time systems, *Hybrid Tree-Hashes* [11] have been proposed to provide a fast in-memory access structure. To index data, the Hybrid Tree-Hash combines

the use of a T-Tree and a Chained Bucket Hash. Lehman describes the T-Tree [7] as a combination of the AVL-Tree and the B-Tree: similarly to B+Trees, nodes contain multiple elements whilst the binary search strategy of the AVL-Tree is employed for retrieval. To enhance the retrieval time the keys in the nodes of the T-Tree are hashed and the offset to that node is stored as the value. Data retrieval is accomplished by a lookup in the Chained Bucket Hash, which retrieves the offset value for the given search key. Then, the node that holds the data is accessed directly without traversing the T-Tree. Whilst T-Trees perform well as an in-memory data structure, their usage as an on-disk structure is problematic: First, the use of a binary tree results in deep trees, which in turn results in many disk pages being accessed. Second, the binary tree nature of the T-Tree makes it cache oblivious: range queries are highly expensive, as one has to continuously "jump" up and down the tree for traversal, leading to costly disk-seek operations. This is in contrast to the B+Tree, where the data is stored in the leaves as a linked list, resulting in fewer disk page seeks and cache awareness. To the best of our knowledge the Hybrid Tree-Hash is the most similar structure to our proposed B+Hash Tree index. The main difference is that we optimized our index structure for disk-based operations, whereas Hybrid Tree-Hashes were optimized for in-memory retrieval.

3 B+Hash Tree

In this section we introduce our B+Hash Tree, a scalable, updatable and lookup-optimized, on-disk index-structure combining the strengths of B+Trees and Hashes. First, we describe the structure of the B+Hash Tree and elucidate its operations. Second, we provide a time and space complexity analysis of the relevant B+Hash Tree operations. Finally, we discuss some limitations of the B+Hash Tree and suggest appropriate solutions.

Note that throughout this section we propose to index a RDF graph akin to the RDF-3X approach. In other words, we propose a separate index for each possible subset combination of the $< s, p, o >$ triples. Hence, a $s_1p_3o_2$ triple is stored in level 1 with the key s_1, in level 2 with the composite key s_1p_3, and finally in level 3 using the composite key $s_1p_3o_2$. This structure allows retrieval of all triple patterns with a single lookup [15, 10]. In contrast to RDF-3X, we propose to use B+Hash Trees as opposed to B+Trees. Like all other approaches we also propose to dictionary encode all literals.

3.1 B+Hash Tree Description

The architecture of the B+Hash Tree comprises two core elements: A B+Tree and the Hash Map. Here, we first explain how these elements are combined to form a joint index and then elaborate on the main operations.

We use the standard B+Tree as the basis for our B+Hash Tree. Recall that B+Trees are optimized for disk access. In particular, nodes of the trees are adapted to the size of a disk page to facilitate caching and limiting disk access. Additionally, all values are stored in the leaf nodes of the tree, which are interlinked, allowing *fast index-range queries*. More information about B+Trees can be found in [4].

As Figure 1 illustrates, the B+Hash Tree combines the B+Tree with a *Hash Map*. Specifically, to improve retrieval performance the leaf nodes of the B+Tree are being hashed. Each bucket entry in the Hash Map contains a key (or ID), an offset containing the address of the element designated by the key on disk and the count of distinct elements sharing the same prefix. The prefix being the anterior part of the key. As an example consider the $s_2p_1o_4$ triple: in a level 2 index the prefix should be s_2p_1; in a level 1 index s_2.

A retrieval operation in a B+Hash Tree starts with a lookup of the offset in the Hash Map using the key and then accesses the node holding the search key directly without traversing the tree. The count aids both the query optimizer (e.g., to gauge selectivity) and the execution of range scans (indicating the number of elements to be read). As exemplified in Figure 1, "Bucket 1" indicates that there are two predicates for subject S_1 ("count = 2").

Note that not every leaf node needs to be hashed. Usually, only the leaf nodes containing the smallest suffix for a given prefix have to be hashed – an approach which we refer to as *overall hashing*. For example, on level 1 of the *spo* index only the leaf nodes where the S key changes need to be hashed, as illustrated in Figure 1.

Alternatively, the hash can be tuned to contain the most popular keys – an approach which we refer to as *cached hashing*, as it employs a Hash Map akin to a cache of the B+Hash Tree's contents. Cached hashing can be tuned to reduce space consumption compared to overall hashing. This space saving comes at a cost of slowing down non-frequent accesses. Therefore, the empirical evaluation in this paper focuses on *overall hashing*.

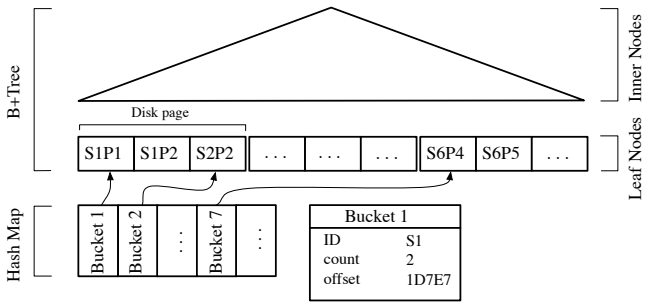


Fig. 1. Architecture of the B+Hash Tree: spo index level 1

3.2 Basic Operations

The B+Hash Tree has three basic operations: get, insert, and delete. To enable the same model interface as Hexastore [14], we split the get operation into two distinct ones:

getIdx(a) Given a triple pattern a, look up the offset and the count of elements in the Hash Map: $getIdx(a) : \text{offset}_a,\ \text{count}_a$

getSet(offset, count) Given an offset and a count, retrieve $count_a$ elements from the leaf node:

$$getSet(\text{offset}_a,\ \text{count}_a) : \text{set}_a$$

Hence, a traditional lookup would be composed of $getSet(getIdx(a))$, a very common index range-query, not to be mistaken with a SPARQL range-query. Furthermore, in contrast to Hexastore, we have the following data changing operations:

insert(a) Insert triple a into the B+Hash Tree:

$$insert(a) : \text{void}$$

delete(a) Delete triple a from the B+Hash Tree:

$$delete(a) : \text{void}$$

Note, that an insert, respectively a delete operation may cause a rebalancing of the B+Tree. If this occurs then the keys in the newly created leaf node have to be verified if an update of their page offset's in the Hash Map is required – a process whose cost depends on the on-disk implementation of the B+Tree.

Index range-scans are quite common operations in Semantic Web applications. Just consider retrieving a list of all predicates that connect subject s with an object o. Such an operation results in the triple pattern $< s, ?p, o >$. In a B+Tree, neighboring leaf nodes are connected to each other, enabling sequential iteration through the pertinent leaf nodes. Hence, the logical way to retrieve the answers for this triple pattern in a RDF-3X like index is to iterate through the level 2 *sop* index starting from the "smallest" (or first indexed) p_s. Hence, the B+Tree is first being traversed from the root node down to the leaf node to find the node for sop_s and then sequentially iterating through the leaf nodes until a different object is encountered. When using a B+Hash Tree, in contrast, we first lookup the key *so* in the Hash Map of the appropriate index followed by the B+Tree traversal like in the traditional tree. Hence, we reduce the tree traversal down from the root node to the first leaf node – a $O(log(n))$ operation – to a single hash lookup – a constant time ($O(c)$) operation. From a certain data size the B+Hash Tree will, hence, outperform the B+Tree. We elaborate this fact by doing a simple complexity analysis of the most important operations in Section 3.3.

3.3 Time and Space Complexity Considerations

In this section we provide a formalization for the time and space complexity of the most important B+Hash Tree operations. Note that since we are talking about a disk-based index structure, the hard drive access times, as the slowest component, are likely to dominate in-memory operations. Hence, the time complexities of B+Tree operations are measured by the number of page reads. Given that the B+Tree only stores the actual data in the leaves (the inner nodes of the trees are "only" used to organize the index) and that the data for a single key typically fits into one page, the number of page reads for any simple lookup is solely dependent on the tree height. Assuming that we denote the order (i.e., the # of index elements per inner node) of the B+Tree as d and the number of entries as n, Comer [4] elucidates the height of a B+Tree as:

$$height = log_d\,(n) \tag{1}$$

Time complexity: Given that most queries do not solely rely on simple key lookups, but actually retrieve multiple elements (e.g., find all objects for a subject) we also need to account for the number of leaf pages that need to be read. Assuming that the values fit on s pages then the number of page reads for a query can be defined as:

$$Reads_{B+Tree} = log_d(n) + s \tag{2}$$

Note that this formula assumes that the values are (i) stored on consecutive pages and (ii) that the leaf-pages are interlinked, which the B+Tree guarantees.

A logarithmic complexity is, obviously, excellent and has served the IT community well in many applications such as relational databases. If the key-space, however, grows enormously and the number of separate accesses for any given query is large – both of which are especially true for SPARQL queries – then even a logarithmic complexity may slow the execution down. The main rationale behind the B+Hash Tree is to cut down the time complexity of these reads using a Hash Map with its constant-time accesses. As a result, the complexity of a simple lookup is 2 – one access to retrieve the bucket hash entry and another one to access the leaf node. In addition, as before, when performing index range-scans we need to read all the pages which contain the data, resulting in:

$$Reads_{HashMap} = 2 + s \tag{3}$$

Consequently, using the Hash Map results in fewer disk page reads than the B+Tree, if the height of the B+Tree exceeds two levels. With Equations 2 and 3 we can calculate the number of disk page accesses for a set retrieval (range query). To estimate the total retrieval time we multiply the number of disk page reads with the average disk page access time of a common hard disk. However, in reality there are three different types of disk page reads: `random read`, `sequential read`, and `cache read`. Random disk page reads are the slowest kind, as the seek operation requires the HDA (Head Disk Assembly) to jump to another track. Sequential reads (i.e., reading some data from the same track) consists of waiting until the required disk page arrives under the HDA, which depends mainly on the rotational speed of the hard disk. The fastest form of access is the cache read, i.e. when a previously read page is found in the on-board disk cache and no mechanical action is required to retrieve the data.

In contrast to B+Trees – on-disk optimized data-structures enabling efficient (sequential) scans – Hash Map data lookup and retrieval is usually random, due to the lack of locality. Again, this is dependent on the actual hash implementation.

Inserting and deleting in the B+Hash Tree adds an additional level of complexity. Assuming that the Hash Map has a sufficient number of free buckets, then insert/delete operations in a B+Hash Tree add – in theory – just one more write operation over the B+Tree: the update of the Hash Map. However, if a leaf node in the B+Tree has to be split or merged during an insert, respectively delete operation, then the page offsets of the keys in the affected leaf nodes may have to be updated in the Hash Map. Depending on the B+Tree implementation,

usually, only the keys where the prefix changes in the newly created or merged node, may need an update. Worst case, this is an $O(n)$ operation where n denotes the number of keys with different prefixes in the affected node.

If the Hash Map is full, however, or there are too many collisions, then the HashMap needs to be reorganized (rehashed) resulting in a higher cost operation [6, 9].

Space complexity: The space consumption of a B+Tree depends on the number of nodes. In practice the size of the node is chosen as such to match the size of a disk page (usually 4, 8 or 16 KB), therefore the space occupancy of a B+Tree is the number of inner nodes plus the number of data containing leaf nodes times the size of a disk page:

$$Size_{B+Tree} = Size_{Page}(\sum_{level=0}^{h} Page_{level} + Page_{leafs}) \quad (4)$$

where $Page_{level}$ denotes the number of nodes respectively disk pages on level l of the tree and $Page_{leafs}$ denotes the number of data containing leaf-nodes.

The space consumption of a B+Hash Tree *additionally* adds the size of the Hash Map, which can be expressed by the number of chunks holding hash buckets. Chunks in turn, are typically sized to match a disk page. Consequently, the size of the Hash Map is:

$$Size_{HashMap} = Size_{Page} \cdot \#Chunks \quad (5)$$

where $\#Chunks$ denotes the number of chunks needed for the Hash Map.

Summarizing, we find that the B+Hash Tree provides a better complexity for reads compared to B+Trees. This advantage comes at the cost of additional space shown in Equation 5 and some time to maintain the Hash Map. We would argue that the cost in most cases is relatively small for Semantic Web applications. Addressing the former, we believe that given the price of disk space the additional space complexity for the Hash Map is negligible. Addressing the latter, it can be argued that, assuming a sufficiently large hash and a higher ratio of reads than writes/updates, the frequency of hash map reorganization operations can be limited to a few instances.

Database space complexity: Consequently, given the multi-ordering multi-level index structure chosen, the total space consumption of a full database index (all possible index orderings for triples) is:

$$Size_{index} = \sum_{ord}^{ORDS} \sum_{lvl=0}^{2} (Size_{ord,lvl}(B+Tree) + Size_{ord,lvl}(HashMap)) \quad (6)$$

where *ord* represents the current index type (i.e. `SPO`, `OPS`, etc. $\in ORDS$), while *lvl* denotes the current index level.

3.4 Limitations and Solutions

The *overall hashing* technique enhances the retrieval time of the B+Tree at the cost of space consumption as described previously. Considering empirical evidence such as Kryder's Law [13], space consumption entails a rapidly decreasing economical cost versus the high cost of query answering in today's DBMS

(Database Management Systems), where real time or near real time response is often required.

For high update rate scenarios, the extra overhead induced by the B+Hash Tree structure can be nullified by considering a parallel architecture where the traditional B+Tree part of the index would reside on one disk unit while the Hash Map would be served/updated on another disk. Furthermore, if the update cost of the Hash Tree is higher than that of the B+Tree at one time, one can still serve queries by reverting to the $O(log(N))$ worst case performance of the B+Tree with no time penalty versus traditional indexing approaches. Hence, in such a setup the worst-cost complexity of a B+Hash Tree is equal to a B+Tree (whenever the Hash Map is reorganized). In most cases (when the Hash Map is available), however, the retrieval complexity would be linear (as shown in equation 3).

If space is a hard constraint (e.g. in an embedded system) one can change the policy of updating all keys (*overall hashing*) to *cached hashing*, where similar to cache policies, only "popular" keys are stored. In general, if incremental updates are rare, then using the *overall hashing* approach is recommended.

System cache impact: As argued above, the Hash Map significantly improves access to leaf nodes in comparison to a "pure" B+Tree. Nonetheless, in reality any modern computing system employs a hierarchy of cache systems starting with the *disk cache* at the lowest level.

When considering the disk cache, there are four possible situations: (i) none of the structures are in cache – in this case the the B+Hash Tree will provide the highest performance as described previously, (ii) the Hash Map of the B+Hash Tree is in cache while the B+Tree is not – the B+Hash Tree will outperform the former, (iii) the B+Tree is cached and will gain the highest performance, and (iv) both structures are in cache, which is the most likely scenario.

Nowadays, the typical size of the disk cache varies between 8 and 64 MB. Performing a simple space consumption estimation for a B+Tree holding 30 million triples, the total number of inner nodes can be approximated to 0.5 million assuming a standard page size of 4KB. This results in an approximately 2 GB inner node index of the B+Tree. Given usual cache eviction approaches (such as *Least Recently Used*) it is likely that only the higher inner node levels of such a B+Tree will be cached while the lower levels will mostly reside on disk. In this case, the B+Tree structure will have to read $h - k$ pages from disk to reach the queried leaf page, where h is the height of the tree and k represents the number of levels in the cache. When the dataset is large enough then $h - k > 2$ (where 2 is the lookup cost in the B+Hash Tree). In these cases, neglecting the OS filing system cache, the B+Hash Tree will outperform the traditional approach. Due to the growth of the Semantic Web we expect that this gap will grow.

4 Evaluation

To evaluate the performance of the B+Hash Tree compared to a B+Tree we created a prototype for both indices, which we used in conjunction with

an in-memory simulation of the on-disk structures. The advantage of the disk simulation is that it can monitor actual disk page accesses regardless of wall-clock-time confounding factors such as disk-cache and other operating system processes. The main disadvantage of this method is that all evaluations were constrained by the available memory (72GB). Hence, the largest dataset we could run contained 31 million triples.

Given that the B+Hash Tree is solely an index and not a full-fledged triple store, we used the query optimizer of a triple store called TokyoTyGR[2] to obtain the index access traces for a query and then ran the traces on the B+Hash Tree. Since we solely evaluate the index and not the overall triple store in this paper, we limited our measurements to the retrieval time of the B+Hash Tree (or B+Tree) index structure and not complete query answering such as parsing of the SPARQL query or selectivity estimations – these measures would have been the same for both index structures. Moreover, we do not monitor the additional sequential page reads for range scans, because as discussed in Equations 2 and 3, the number of additional page-accesses due to the scan is identical (i.e., s) for both considered approaches. Therefore we only track the differentiating parts of the equations, which excludes the scans. To maximize disk access performance we set the B+Tree page size to the size of the disk page size (4 KB).

Consequently, each result presented in Figures 2a–f shows the number of disk page reads of a whole index access trace during a single query execution and not just a single lookup of an element in the index. All test simulations use the *overall hashing* technique.

For the traditional B+Tree approach, we discriminate between sequential and random disk page reads in the diagrams. In the case of the B+Hash Tree, we present only "all reads", as most reads from a Hash Map are random.

The space consumption was calculated by applying the formulas provided in our complexity analysis of the B+Hash Tree.

In the remainder of the section we first present the two datasets—the Berlin SPARQL Benchmark dataset and Yago—with their associated queries as well as performance measures, and then discuss the results and their limitations.

4.1 The Berlin SPARQL Benchmark Dataset

The first dataset, the Berlin SPARQL Benchmark[3] (BSBM), is a synthetic dataset. It simulates an enterprise setting, where a set of products are offered by different vendors and clients can review them [3]. The dataset can be generated using the available data generator and the BSBM provides twelve distinct SPARQL queries.

Some of these SPARQL queries contained "REGEX", "OFFSET", "UNION", "DESCRIBE", and other expressions, which TokyoTyGR does not support. Therefore, we selected a subset of 5 queries without these elements from which

[2] The TokyoTyGR is an extension of the Hexastore [15] triple store. It can easily accommodate inserts/deletes and has a state of the art query optimizer based on selectivity estimation techniques.

[3] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark`

we further removed "FILTER"-expressions and "OPTIONALS", as they would be handled in the exact same way by both the B+Hash Tree and the B+Tree. The result are the 5 queries (denoted as Query 1 to 5), which we list in Appendix A in ascending query complexity (in terms of variables and triple-patterns).

To compare the performance of the index structures with increasingly larger dataset sizes, we created five different datasets ranging from 1 million to 31 million triples. The details of these datasets are shown in Table 1. Note that the number of unique predicates in all datasets is the same while the number of unique subjects and objects increases. The results of running the five queries on

Dataset	# Triples	S	P	O
BSBM	1,075,597	99,527	40	224,032
	2,160,331	200,096	40	443,753
	4,969,590	452,507	40	966,120
	10,632,426	966,013	40	1,979,668
	31,741,096	2,863,525	40	5,297,862
Yago	16,348,563	4,339,591	91	8,628,329

Table 1. Number of triples, unique subjects, predicates, and objects in the datasets

the different dataset sizes are shown in Figure 2a–e. For every query and dataset, the B+Hash Tree is in each case faster (i.e., uses fewer disk reads) than the traditional B+Tree approach. Furthermore, the difference between the number of disk page reads for both structures increases with the size of the dataset. Therefore, for the 31 million triple dataset, the B+Hash Tree performs twice as fast as the B+Tree.

Figure 2g) shows the disk space consumption of both data structures. As expected, the B+Hash Tree consumes more space. We believe, however, that for most applications, trading-in 20% of the space against a halved access-time is a worthy trade-off.

4.2 Yago Dataset

To complement the synthetic BSBM dataset with a real-world representative, we employed Yago[4] as a second dataset, which consists of facts extracted from Wikipedia. Again, Table 1 shows the characteristics of the Yago dataset, which contains about 16 million distinct triples and almost 13 million resources.

The Yago dataset does not come with a defined set of queries. Therefore, we constructed three queries (numbered Queries 6–8; shown in Appendix A), which simulate a realistic information request such as "What actors play in American movies?" or "Which scientist is born in Switzerland?". In addition, we ensured that two queries (Q6 and Q7) have a low selectivity and, therefore, "touch" a lot of the data, and one query (Q8) is highly selective and is, therefore, expected to "touch" fewer disk pages.

To simplify the comparison, and as Yago has only one dataset size, we graphed all results in a single plot (Figure 2f). Also, given the large number distribution, the plot employs a *logarithmic scale* on the x-axis.

[4] `http://www.mpi-inf.mpg.de/yago-naga/yago/n3.zip`

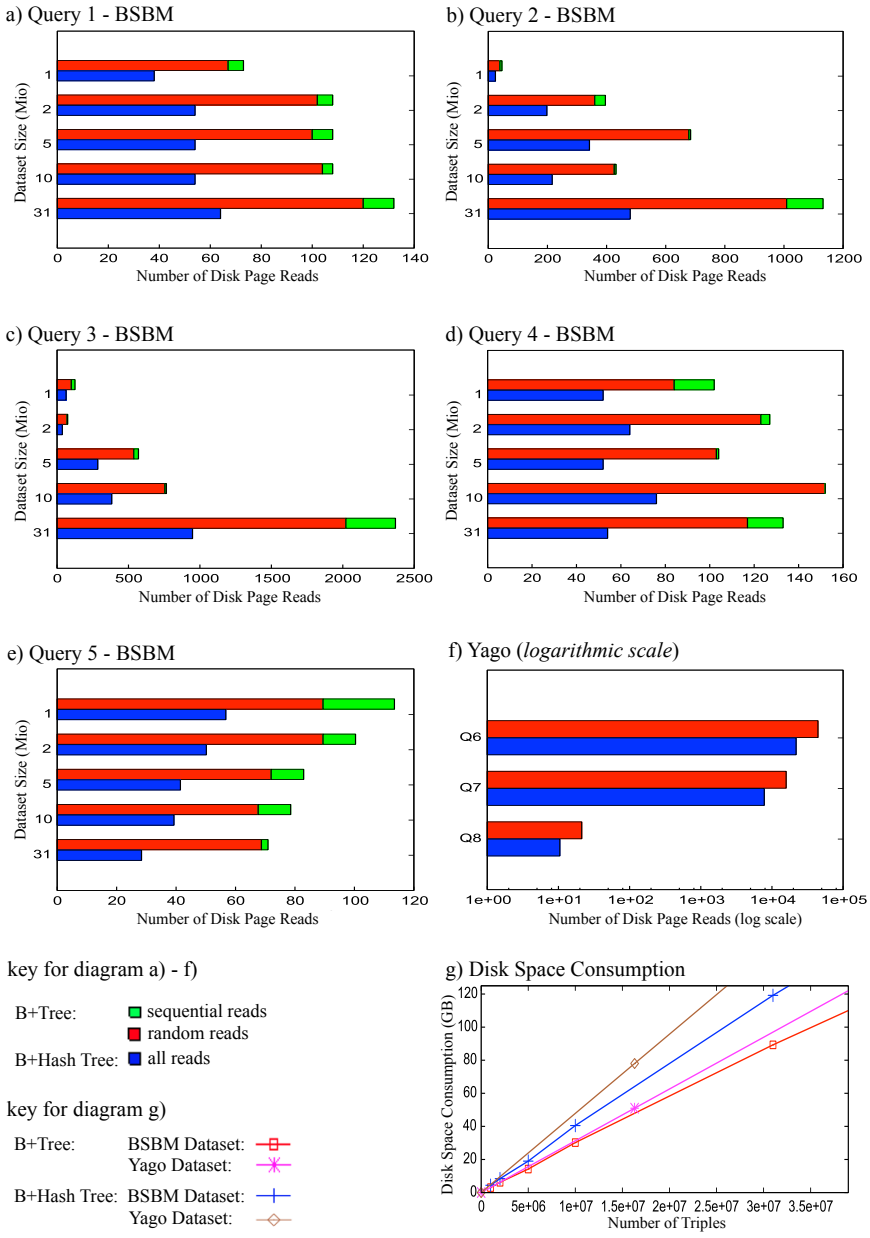


Fig. 2. Number of disk reads for the queries and disk space consumption

Again, the B+Hash Tree outperforms the traditional B+Tree by accessing about half the pages. As expected, Query 8 reads fewer disk pages. It is note-

worthy to observe that the performance improvement seems independent of the query's selectivity.

Figure 2g again graphs the space consumption. Note that the higher space consumption of Yago compared to BSBM can be attributed to the number of distinct values for URIs and literals: while the 31 million BSBM dataset has 8 million distinct values, Yago's 16 million triples have 13 million distinct values.

4.3 Discussion and Limitations

Our results confirm the theoretical analysis that the performance improvement of the B+Hash Tree compared with the normal B+Tree increases with the size of the dataset. To further illustrate the result, Figure 3 graphs the speed-up factor against the dataset size. Observably, the speed-up factor increases with the size of the number of triples inserted and therefore confirms Equations 2 and 3 in practice. Given that Equation 2 grows logarithmically and Equation 3 is constant (ignoring the scan element) we would expect the difference to grow logarithmically with dataset size.

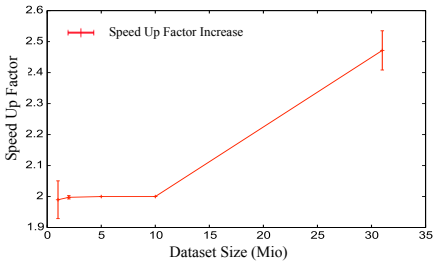


Fig. 3. Speed Up Factor vs. Dataset Size (error bars show results for different queries)

We also investigated, if the query complexity in terms of number of query variables (which varies from 2 to 11) and triple patterns (which varies from 2 to 12) affects the speed performance. Visually, Figure 2 indicates no such difference. Indeed, a pairwise, two-tailed t-test confirmed that the speed-up between queries remains constant with a signifiance of far below 0.1%.

This B+Hash Tree prototype consumes a considerable amount of space which can be traced to four main reasons. First, by storing all possible subset combinations of a triple we gain speed in query answering, as highlighted in the Hexastore project. Second, we set the size of a disk page to 4KB which entails B+Trees containing more inner nodes, thus consuming more space. Third, we use 20 byte instead of 8 byte keys typically used in DBMS, as we wanted a global rather than a "table-local" key space. And last, we have not yet considered index compression, further reducing the consumed space while still maintaining access speed, as shown by Neumann [10].

Building the RDF index in the B+Hash Tree from scratch can increase the build time significantly compared to the B+Tree depending on the Hash Map implementation. A more thorough investigation of this issue is still open.

Limitations: We see three major limitations in our evaluation; not of our proposed approach. First, all our empirical calculations are based on an in-memory simulation of an on-disk B+Hash Tree structure. To mitigate this problem we ensured that our hard-disk model was as accurate as possible and we parameterized it with present-day hard disk parameters. In addition, we measured disk page accesses rather than wall-clock time, essentially focusing on the most time-intensive element of the queries. Consequently, we are confident that our findings generalize to the on-disk setting.

Second, our simulation disregards disk caches. Disk-caches in modern day operating systems are intricate structures that any on-disk index would share with other disk-accessing processes. This makes an adequate simulation a highly-complex issue and may mislead evaluations in a real, on-disk setting. As discussed in Section 3.4, however, we would expect a B+Hash Tree to outperform a traditional B+Tree even in the presence of disk-caches.

Last, we used the TokyoTyGR RDF store to obtain index-access traces for each of the experimental queries. It could be argued that the results of our experiments are biased towards its query optimizer. Given that TokyoTyGR uses a selectivity-based optimizer [12] like most other modern triple stores (such as Hexastore or RDF-3X) the danger of a systematic bias seems limited—especially since it seems unlikely that other query optimizations would lead to a vastly different access pattern between a B+Hash Tree and a B+Tree. Nonetheless, a completely different query optimization approach might require the re-evaluation of our results.

Note that even in light of these limitations, the use of a disk-simulation had several advantages: First, it allowed us to isolate the evaluation from confounding effects (e.g., by the operating system). Second, it allowed us to meticulously distinguish between different types of disk accesses—an undertaking that is non-trivial in a real on-disk structure. Nonetheless, a future on-disk evaluation will have to complement our current findings.

5 Conclusion and Future Work

In this paper we proposed the B+Hash Tree—a scalable, updatable, and lookup-optimized on-disk index-structure especially suitable to the Semantic Web with its large key-space. We showed that using a Hash Map to store the offsets of the leaf nodes successfully trades a slight increase in database size against significantly reduced retrieval time. When used in the context of existing index approaches such as Hexastore and RDF-3X, this will allow for effective retrieval of all possible triple patterns.

To evaluate the B+Hash Tree empirically, we benchmarked the number of page reads (and hence indirectly retrieval time) using two well-established Semantic Web test datasets. As the results show, the B+Hash Tree consistently requires approximately half the page reads of a B+Tree. Note that this difference is expected to grow with the logarithm of the dataset size.

The current implementation of the B+Hash Tree was only used in the simulated measurements. We, therefore, intend to implement a fully operational

disk-based version of the index and evaluate it with several "truly" large datasets. In this context we also want to investigate the interaction between the B+Hash Tree and the disk-cache. Last but not least, we intend to consider the use of index-compression to develop even more efficient index structures.

Research in index structures has come a long way, from the early days of simple re-use of traditional, relational, row-based data base indices to the construction of specialized structures such as Hexastore and RDF-3X. We believe that the B+Hash Tree provides a new quality to this exploration. It does not smartly reuse existing structures like its predecessors but investigates a Semantic Web data specific algorithmic extension. As such it calls for the exploration of index structures that exploit the structural and statistical idiosyncrasies of Semantic Web data. The result of this exploration should be truly web-scalable triple stores, which lie at the very foundation of the Semantic Web vision, and the B+Hash Tree can provide a major building block towards that foundation.

Acknowledgements This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

References

1. D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.
2. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th international Conference on World Wide Web*, pages 41–50, Raleigh, North Carolina, USA, 2010. ACM.
3. C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Sematnic Web and Information Systems*, 5:1–24, 2009.
4. D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121—137, 1979.
5. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. In *ISWC'07/ASWC'07: Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pages 211–224, Berlin, Heidelberg, 2007. Springer-Verlag.
6. J. Keller. Hashing and rehashing in emulated shared memory. In *Proceedings of the 3rd Workshop on Parallel Algorithms (WOPA)*, 1993.
7. T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303. Morgan Kaufmann Publishers Inc., 1986.
8. S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, 2009.
9. W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, 7(1):5–19, 1975.
10. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

11. C. Ryu, E. Song, B. Jun, Y. Kim, and S. Jin. Hybrid-TH: a hybrid access mechanism for Real-Time Main-Memory resident database systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, page 303. IEEE Computer Society, 1998.
12. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th International World Wide Web Conference*, pages 595–604, Beijing, China, 2008. ACM.
13. C. Walter. Kryder's law. *Scientific American*, Aug. 2005.
14. C. Weiss and A. Bernstein. On-disk storage techniques for semantic web data - are B-Trees always the optimal solution? In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, Oct. 2009.
15. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1:1008–1019, 2008.
16. K. Wilkinson. Jena property table implementation. In *Proceeding of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2006.
17. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of the First International Workshop on Semantic Web and Databases (SWDB)*, volume 3, pages 7–8, 2003.

A Appendix

The Berlin SPARQL Benchmark (BSBM) Dataset queries:

Query1:
```
SELECT ?product ?label WHERE { ?product <label> ?label;
<type> <Product>; <productFeature> <ProductFeature50>;
<productFeature> <ProductFeature580>; <productPropertyNumeric1> ?value1 . } LIMIT 10
```

Query2:
```
SELECT ?offer ?price WHERE { ?offer <product> <Product62>; <vendor> ?vendor;
<publisher> ?vendor . ?vendor <country> <US> . ?offer <deliveryDays> ?deliveryDays;
<price> ?price; <validTo> ?date . } LIMIT 10
```

Query3:
```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName WHERE {
?review <reviewFor> <Product197>; <title> ?title; <text> ?text; <reviewDate> ?reviewDate;
<reviewer> ?reviewer . ?reviewer <name> ?reviewerName . } LIMIT 20
```

Query4:
```
SELECT ?product ?productLabel WHERE { ?product <label> ?productLabel .
<Product613> <productFeature> ?prodFeature . ?product <productFeature> ?prodFeature .
<Product613> <productPropertyNumeric1> ?origProperty1 .
?product <productPropertyNumeric1> ?simProperty1 .
<Product613> <productPropertyNumeric2> ?origProperty2 .
?product <productPropertyNumeric2> ?simProperty2 . } LIMIT 5
```

Query5:
```
SELECT ?label ?comment ?producer ?productFeature ?propTextual1
?propTextual2 ?propTextual3 ?propNumeric1 ?propNumeric2 WHERE {
<Product2227> <label> ?label; <comment> ?comment; <producer> ?p .
?p <label> ?producer; <publisher> ?p; <productFeature> ?f . ?f <label> ?productFeature .
<Product2227> <productPropertyTextual1> ?propTextual1;
<productPropertyTextual2> ?propTextual2; <productPropertyTextual3> ?propTextual3;
<productPropertyNumeric1> ?propNumeric1; <productPropertyNumeric2> ?propNumeric2 . }
```

The YAGO Dataset queries:

Query6:
```
SELECT ?actor ?p WHERE { ?actor <actedIn> ?p .
?p <type> <wikicategory_American_films>  . }
```

Query7:
```
SELECT ?scientist WHERE { ?scientist <type> <wordnet_scientist>; <bornIn> ?city .
?city <locatedIn> <Switzerland> . }
```

Query8:
```
SELECT ?person WHERE { ?person <graduatedFrom> <University_of_Zurich>
?person <hasWonPrize> ?price . }
```

Progressive Semantic Query Answering

Giorgos Stamou, Despoina Trivela, and Alexandros Chortaras

School of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou Campus, 15780, Athens, Greece
{gstam,achort}@cs.ntua.gr
despoina@image.ntua.gr

Abstract. Ontology-based semantic query answering algorithms suffer from high computational complexity and become impractical in most cases that OWL is used as a framework for data access in the Semantic Web. For this reason, most semantic query answering systems prefer to lose some possible correct answers of user queries rather than being irresponsible. Here, we present a method that follows an alternative direction that we call *progressive* semantic query answering. The idea is to start giving the most *relevant* correct answers to the user query as soon as possible and continue by giving additional answers with *decreasing relevance* until you find all the correct answers. We first present a systematic analysis that formalises the notion of answer relevance and that of query answering sequences that we call strides, providing a formal framework for progressive semantic query answering. Then, we describe a practical algorithm performing sound and complete progressive query answering for the W3C's OWL 2 QL Profile.

Keywords: scalable query answering, tractable reasoning, approximate query answering, semantic queries over relational data, OWL 2 Profiles

1 Introduction

The use of ontologies in data access is based on semantic query answering, in particular on answering user queries expressed in terms of a terminology (that is connected to the data) and usually represented in the form of conjunctive queries (CQs) [6, 1]. The main restriction of the applicability of the specific approach is that the problem of answering CQs in terms of ontologies represented in description logics (the underlying framework of the W3C's Web Ontology Language -OWL) has been proved to be difficult, suffering from very high worst-case complexity (higher than other standard reasoning problems) that is not relaxed in practice [6]. This is the reason that methods and techniques targeting the development of practical systems mainly follow two distinct directions. The first suggests the reduction of the ontology language expressivity used for the representation of CQs vocabulary, while the second sacrifices the completeness of the CQ answering process, providing as much expressivity as possible. Methods of the first approach mainly focus on decoupling CQ answering into (a) the

use of terminological knowledge provided by the ontology (the reasoning part of query answering) and (b) the actual query execution (the data retrieval), thus splitting the problem into two independent steps [1, 4, 10]. During the first step (usually called *query rewriting*) the CQ is analysed with the aid of the ontology and expanded into a (hopefully finite) set of conjunctive queries, using all the constrains provided by the ontology. Then, the CQs of the above set are processed with traditional query answering methods on databases or triple stores, since terminological knowledge is no longer necessary. The main objective is to reduce the expressivity of the ontology language until the point that the procedure guarantees completeness. Late research in the area, introduced the DL-Lite family of description logics, underpinning W3C's OWL 2 QL Profile [4, 3], in which the CQ answering problem is AC^0 w.r.t. data. Despite the obvious advantage of using the mature technology (more than 40 years research) of relational databases, there are also other major technological advantages of the specific approach, most of them connected with the use of first-order resolution-based reasoning algorithms [7][5, Ch.2]. The main restriction is that in the presence of large terminologies, the algorithm becomes rather impractical, since the exponential behaviour (caused by the exponential query complexity) affects the efficiency of the system.

The attempts to provide scalable semantic query answering over ontologies expressed in larger fragments of OWL introduced the notion of *approximation*. Approximate reasoning usually implies unsoundness and/or incompleteness, however in the case of semantic query answering most systems are sound. Typical examples of incomplete query answering systems are the well-known triple stores (Sesame, OWLIM, Virtuoso, AllegroGraph, Mulgara etc). The two main characteristics distinguishing incomplete semantic CQ answering systems is *how efficient* and *how incomplete* they are. The efficiency of semantic query answering is usually tested with the aid of real data of a specific application or using standard benchmarks [8]. Lately, a systematic approach of the study of incompleteness of semantic query answering systems has been also presented [9]. A major issue here is that the user should be aware of the *type* of lost correct answers, i.e. there should be a general deterministic criterion expressing a type of *relevance*, indicating how crucial is the loss of each correct answer.

Within the above framework, herein we present an alternative direction in scalably solving the problem of semantic query answering ensuring a safe approximation process that hopefully converges to a complete solution. The idea is to provide the user with correct answers as soon as they are derived and continue until all the correct answers are found, ensuring that the relevant correct answers will be first given. For example, in the case of the query rewriting approach this means that instead of clearly splitting the steps of query rewriting and query processing, whenever a new rewriting is found it can be evaluated against the database and the results can be presented to the user. In order for this idea to be successfully applied, several intuitive requirements should be fulfilled: the first correct answers should be given very fast; an important amount of correct answers should be found in a first small percentage of execution time; complete-

ness should be ideally reached (or at least approximated); correct answers should be given following a degree of importance, according to a semantic preference criterion; the results should not be widely replicated.

In the present paper, we provide a systematic approach formalising the above idea. We introduce *progressive semantic query answering* based on the notion of *CQ answering strides* that are flows of correct answers with specific properties that formalise the intuitive meaning of the above criteria. We then provide a practical progressive semantic CQ answering algorithm that has some nice properties and is complete in OWL 2 QL and present the results of its implementation and evaluation (we call the implemented system ProgRes). The algorithm is based on a query rewriting resolution-based procedure that computes a sequence of rewritings, the elements of which have a decreasing importance according to a query similarity criterion (measuring the similarity of the rewriting with the user CQ). The order is proved to be ensured under a specific resolution rule application strategy that ProgRes follows. It is interesting that although the results are ordered (ranked) and given during the execution, the efficiency of the algorithm is not worse than other similar ones (like the one implemented in Requiem [7]).

2 Preliminaries

The most relevant with the problem of query answering OWL QL Profile is based on DL-Lite$_R$ [1, 4]. A DL-Lite$_R$ *ontology* is a tuple $\langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is the *terminology* (usually called TBox) representing the entities of the domain and $\mathcal{A}$ is the assertional knowledge (usually called ABox) describing the objects of the world in terms of the above entities. Formally, $\mathcal{T}$ is a set of terminological axioms of the form $C_1 \sqsubseteq C_2$ or $R_1 \sqsubseteq R_2$, where C_1, C_2 are concept descriptions and R_1, R_2 are role descriptions, employing atomic concepts, atomic roles and individuals that are elements of the denumerable, disjoint sets $\mathbf{C}, \mathbf{R}, \mathbf{I}$, respectively. The ABox $\mathcal{A}$ is a finite set of *assertions* of the form $A(a)$ or $R(a, b)$, where $a, b \in \mathbf{I}$, $A \in \mathbf{C}$ and $R \in \mathbf{R}$. A DL-Lite$_R$-concept can be either an atomic one or $\exists R.\top$. Negations of DL-Lite$_R$-concepts can be used only in the right part of subsumption axioms. A DL-Lite$_R$-role is either an atomic role $R \in \mathbf{R}$ or its inverse R^-. A *conjunctive query* (CQ) has the form $q : Q(\mathbf{x}) \leftarrow \bigwedge_i^n C_i(\mathbf{x}; \mathbf{y})$, where $Q(\mathbf{x})$ is the answering predicate (the *head* of the CQ), employing a finite set of variables, called *distinguished*, and the conjuncts $C_i(\mathbf{x}; \mathbf{y})$ (forming the *body* of the CQ) are predicates possibly employing non-distinguished variables. We say that a CQ q is posed over an ontology $\mathcal{O}$ iff all the conjuncts of its body are concept or role names occurring in the ontology. A tuple of constants $\boldsymbol{a}$ is a *certain answer* of a conjunctive query Q posed over the ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ iff $\mathcal{O} \cup q \models Q(\boldsymbol{a})$, considering q as a universally quantified implication under the usual FOL semantics. The set containing all the answers of the query q over the ontology $\mathcal{O}$ is denoted with $\mathsf{cert}(q, \mathcal{O})$. A union of CQs q' is a *rewriting* of q over a TBox $\mathcal{T}$ iff $\mathsf{cert}(q', \langle \emptyset, \mathcal{A} \rangle) = \mathsf{cert}(q, \mathcal{O})$, with $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ and $\mathcal{A}$ any ABox. We will denote the set of all CQs in the rewriting of q over the TBox

$\mathcal{T}$ by $\mathsf{rewr}(q, \mathcal{T})$. With a little abuse of notation we write $\mathsf{rewr}(q, \mathcal{O})$, meaning $\mathsf{rewr}(q, \mathcal{T})$ ($\mathcal{T}$ is the TBox of $\mathcal{O}$).

3 Semantic query answering strides

Semantic CQ answering systems are based on sophisticated algorithms that try to find as many certain answers of CQs as possible. Formally, any procedure $\mathsf{A}(q, \mathcal{O})$ that computes a set of tuples $\boldsymbol{a}$ for a CQ q posed over an ontology $\mathcal{O}$ is a *CQ answering algorithm* (CQA algorithm). $\mathsf{A}(q, \mathcal{O})$ is *sound* iff $\mathsf{res}(\mathsf{A}(q, \mathcal{O})) \subseteq \mathsf{cert}(q, \mathcal{O})$ and *complete* iff $\mathsf{cert}(q, \mathcal{O}) \subseteq \mathsf{res}(\mathsf{A}(q, \mathcal{O}))$ ($\mathsf{res}(\diamond)$ is the result of any algorithm $\diamond$). Any procedure $\mathsf{R}(q, \mathcal{T})$ computing a set of rewritings of q over a TBox $\mathcal{T}$ is a *CQ rewriting algorithm*. $\mathsf{R}(q, \mathcal{T})$ can be the basis of a CQ answering algorithm $\mathsf{A}(q, \mathcal{O})$ with the aid of a procedure $\mathsf{E}(q, \mathcal{A})$ that evaluates the query and retrieves the data from the database. In this case, we write $\mathsf{A}(q, \mathcal{O}) = [\mathsf{R} \mid \mathsf{E}]\,(q, \mathcal{O})$. Obviously, it is $\mathsf{res}([\mathsf{R} \mid \mathsf{E}]\,(q, \mathcal{O})) = \mathsf{res}\,(\mathsf{E}(\mathsf{res}\,(\mathsf{R}(q, \mathcal{T})), \mathcal{A}))$. With a little abuse of notation, we freely write $\mathsf{A}(U, \mathcal{O})$, $\mathsf{R}(U, \mathcal{T})$ and $\mathsf{E}(U, \mathcal{A})$ for procedures computing answers to sets U of CQs. A natural question arising in cases where scalability is a major requirement is how to split the execution of a CQ query answering algorithm into parts enabling a progressive extraction of certain answers. Also, how to control the progress of the algorithm ensuring that certain answers extracted until any specific time have desirable characteristics. Let us now proceed to the definitions that form the framework of progressive CQ answering, covering the above intuition.

Definition 1. Progressive CQ answering (PCQA)
Any sequence $\mathsf{P}(q, \mathcal{O}; i) = \{\mathsf{A}_j\}_i$, $i \in \mathbb{N}$, $i > 1$ *where* A_j *are CQA algorithms for a query* q *posed over an ontology* $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, *is a* progressive CQ answering algorithm. *The elements of* P *are its* componets. *If* P *is finite, we write* $\mathsf{P}(q, \mathcal{O}) = [\mathsf{A}_1; \mathsf{A}_2; \cdots; \mathsf{A}_n]\,(q, \mathcal{O})$.

It is important to notice that the components of PCQA algorithms can freely change inputs and outputs in a forward manner, leaving the sequence $\mathsf{P}(i)$ to possibly have a recursive character. The restriction is that any component should be considered as a CQA algorithm meaning that, at any time point, we can ask for the result of any subset of the components being able to compute it (possibly using the output of the previous components). We say that $\mathsf{P}(q, \mathcal{O})$ is *sound* iff $\mathsf{res}(\mathsf{P}) \subseteq \mathsf{cert}(q, \mathcal{O})$ and *complete* iff $\mathsf{cert}(q, \mathcal{O}) \subseteq \mathsf{res}(\mathsf{P})$. Since our intention is to provide users with correct answers during the execution of P, we need refer to the result set of a subset of components of P stratifying the desired answer flow.

Definition 2. PCQA Strides
Let $\mathsf{P}(\mathsf{q}, \mathcal{O})$ *be a PCQA algorithm. A* stride $\mathsf{s}(\mathsf{P}; i : j)$, $i, j \in \mathbb{N}, i \leq j$ *of* P *(if it is infinite* j *can be equal to* ∞*) is the result of the execution of its components* A_i *to* A_j, *i.e.* $\mathsf{s}(\mathsf{P}; i : j) = \bigcup_{[i,j]} \mathsf{res}(\mathsf{A}_\mathsf{k})$.

Let $\Sigma(\mathsf{P})$ denote the set of all strides of P. Obviously, $\mathsf{res}(\mathsf{A}) \in \Sigma(\mathsf{P})$, for any component A of P and also $\mathsf{res}(\mathsf{P}) \in \Sigma(\mathsf{P})$. We will refer to the former as a *single*

stride and to the latter as the *total* stride. A *stratification* of P is a sequence $\mathsf{s}_1, \mathsf{s}_2, ..., \mathsf{s}_k$ of strides of P. Now, we turn our attention to the study of PCQA algorithms the strides of which provide answers with decreasing relevance to the user query posed. The first step is to rank the elements of the strides according to the query posed by the user. Let $\sigma(\mathcal{O}; \boldsymbol{a}, q)$ be a relevance measure expressing the importance of the tuple $\boldsymbol{a} \in \mathsf{s}$, with $\mathsf{s} \in \Sigma(\mathsf{P})$, i.e. the degree in which the specific tuple ideally (and with the less risk) fits to the user query.

The nature of the relevance measure may differ from one application to another. In the present paper we consider the case of a measure representing the *possibility of correctness* of a specific answer. Similar measures play an important role in information retrieval systems, in the process of ranking the results of query answering (see for example [12, 14–16]). Intuitively, here we consider that some axioms of the TBox are not always valid for the specific data, i.e. some exceptions may exist in the large data set making the specific axiom typically inconsistent. The effect is that some answers derived from the query answering process are not correct. This could be a result of untrusted knowledge or by the nature of the specific axiom that is only *usually* valid. For example, although we know that all professors are teaching some courses, there could be the case that some professors (for some reasons) are not teaching. The problem could be solved if we had some additional information about the correctness of the TBox axioms (the provenance of the specific axiom implying a degree of trust etc). If we do not have any additional information, the only rule that we could follow is that the answer is more safe if we use less knowledge to derive it, i.e. the less reasoning the most relevance. We will further clarify this later that we focus to query rewriting PCQAs.

Definition 3. Stride Ordering
Let $\mathsf{P}(\mathsf{q}, \mathcal{O})$ *be a PCQA algorithm and* $\Sigma(\mathsf{P})$ *the set of its strides after the execution of the query* q *over the ontology* $\mathcal{O}$*. Let also* $\sigma(\boldsymbol{a}, q)$ *be a relevance measure of the elements of the total stride and the CQs. We say that a stride* $\mathsf{s}_1(\mathsf{P}; i : j)$ σ-precedes *a stride* $\mathsf{s}_2(\mathsf{P}; i' : j')$ *for the query* q *and we write* $\mathsf{s}_1 \preceq^q_\sigma \mathsf{s}_2$ *iff for all* $\boldsymbol{a}_1 \in \mathsf{s}_1$, $\boldsymbol{a}_2 \in \mathsf{s}_2$ *we have* $\sigma(\boldsymbol{a}_1, q) \geq \sigma(\boldsymbol{a}_2, q)$*. If* $\sigma(\boldsymbol{a}_1, q) > \sigma(\boldsymbol{a}_2, q)$*, we say that* s_1 strictly σ-precedes s_2 *and we write* $\mathsf{s}_1 \prec^q_\sigma \mathsf{s}_2$*. If neither* $\mathsf{s}_1 \preceq^q_\sigma \mathsf{s}_2$ *nor* $\mathsf{s}_2 \preceq^q_\sigma \mathsf{s}_1$*, we have* $\mathsf{s}_1 \not\prec^q_\sigma \mathsf{s}_2$ *(they are* σ-irrelevant *for* q*).*

It is not difficult to see that $\Sigma(\mathsf{P})$ forms a lattice under any $\prec^q_\sigma$ operator, where $\emptyset$ is its infimum and $\mathsf{cert}(q, \mathcal{O})$ is its supremum (supposing that P is sound and complete). Now that we have an ordering of strides, we are ready to proceed to the definition of progressive algorithms the strides of which are ordered. Algorithms of this type ensure that the answer blocks are ordered according to a specific relevance measure. Therefore, they can be considered as approximation algorithms with deterministically controllable behaviour.

Definition 4. Sorted PCQA Algorithms
Let $\mathsf{P}(q, \mathcal{O})$ *be a PCQA algorithm,* $\epsilon = \mathsf{s}_1, \mathsf{s}_2, ..., \mathsf{s}_k$ *a stratification of* P *and* $\prec^q_\sigma$ *an ordering relation over* $\Sigma(\mathsf{P})$*. We say that* P *is* σ-sorted *under* ϵ *iff for any successive strides* $\mathsf{s}_i, \mathsf{s}_j$ *of* ϵ *it is* $\mathsf{s}_i \preceq^q_\sigma \mathsf{s}_j$*. It is* strictly σ-sorted *iff* $\mathsf{s}_i \prec^q_\sigma \mathsf{s}_j$ *otherwise it is* σ-unsorted *for* q*.*

Example 1. Consider the simple DL-Lite$_{\mathcal{R}}$ ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, *with*

$$\mathcal{T} = \{\mathsf{PhDStudent} \sqsubseteq \mathsf{Researcher}, \mathsf{Professor} \sqsubseteq \mathsf{ResDirector},\\ \mathsf{SeniorResearcher} \sqsubseteq \mathsf{ResCoordinator}, \mathsf{ResCoordinator} \sqsubseteq \exists \mathsf{advise.Researcher},\\ \mathsf{ResDirector} \sqsubseteq \exists \mathsf{advise.SeniorResearcher}, \mathsf{supervise} \sqsubseteq \mathsf{advise}\}$$

$$\mathcal{A} = \{\mathsf{Mary} : \mathsf{Researcher},\ \mathsf{Bill} : \mathsf{ResCoordinator},\ \mathsf{John} : \mathsf{ResDirector}\\ \mathsf{Alan} : \mathsf{Researcher},\ \mathsf{George} : \mathsf{PhDSudent},\ \mathsf{Peter} : \mathsf{SeniorResearcher},\\ \mathsf{Sofia} : \mathsf{Professor},\ \mathsf{Ema} : \mathsf{Professor},\ (\mathsf{Bill}, \mathsf{Mary}) : \mathsf{advise}, (\mathsf{John}, \mathsf{Bill}) : \mathsf{advise},\\ (\mathsf{Peter}, \mathsf{George}) : \mathsf{supervise}, (\mathsf{Alan}, \mathsf{Peter}) : \mathsf{advise}\}$$

$\mathcal{A}$ represents a materialisation of a relational database or a triple store. Let $q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{advise}(y, z)$. It is not difficult to see that $\mathsf{cert}(q, \mathcal{O}) = \{\mathsf{Alan}, \mathsf{John}, \mathsf{Sofia}, \mathsf{Ema}\}$. John is a direct result from the ABox, since it is explicitly given that he advises Bill, who advises Mary. It is a bit more difficult to derive Alan since some of the knowledge should be employed. Specifically, we should consider that Alan advises Peter that supervises George (who is a PhDStudent and thus a Researcher), which means that Peter advises George. More complex deductions lead to the conclusion that Sofia and Ema are also answers of the query. PCQA algorithms ensure that the results are given in a specific order that the user knows before the query answering process, according to a specific relevance criterion. For example, we could develop $\mathsf{P} = \mathsf{A}_1; \mathsf{A}_2; \mathsf{A}_3; \mathsf{A}_4; \mathsf{A}_5$ with a stratification $\mathsf{s}_1; \mathsf{s}_2; \mathsf{s}_3$, where $\mathsf{s}_1(\mathsf{P}; 1 : 1)) = \mathsf{res}(\mathsf{A}_1) = \{\mathsf{John}\}$, $\mathsf{s}_2(\mathsf{P}; 2 : 3) = \mathsf{res}(\mathsf{A}_3) \cup \mathsf{res}(\mathsf{A}_4) = \{\mathsf{John}, \mathsf{Alan}\}$ and $\mathsf{s}_3(\mathsf{P}; 4 : 5) = \mathsf{res}(\mathsf{A}_5 \cup \mathsf{A}_5) = \{\mathsf{John}, \mathsf{Sofia}, \mathsf{Ema}\}$. Here, we consider that data is more strong than the knowledge, meaning that there could be possible exceptions in some restrictions expressed as TBox axioms. Thus, an obvious solution is that John should be the first answer (explicitly given), Alan employs a bit more risk (some reasoning is needed), while Sofia and Ema are the most risky answers (for example it could be the case that Sofia does not advise anyone for some reasons). It is not difficult to see that P is sorted according to the above intuitive measure.

4 Practical progressive query answering

The problem of progressive query answering is more difficult than the non-progressive one, since the extracted results should be *ranked* according to a specific criterion and the ranking should be ensured *during* the query answering process (without knowledge of all the results). Obviously, the difficulty strongly depends on the expressivity of the ontology language and the specific relevance measure. Here, we focus on query rewriting PCQA algorithms for DL-Lite$_R$, i.e. on cases where the components of P are based on query rewriting algorithms. We follow a resolution-based FO rewriting strategy (more details can be found in [7]). Intuitively, algorithms of this category are based on the translation of the terminology into a set of FOL axioms and the repeated application of a set of resolution rules employing the query and the FOL axioms until no rule can

be applied. It is ensured that when the algorithm stops it has produced all the rewritings of the user query. The idea is to stratify the application of the resolution rules in order to ensure that the rewritings will be extracted according to a specific order, following a similarity criterion in comparison to the user query. Thus, we can evaluate against the database the fresh queries (rewritings) as they are derived and provide the user with a set of fresh answers. The proposed algorithm remains sound and complete and, although it solves a more difficult problem, in several cases it is more efficient than the respective non-progressive algorithm proposed in [7].

We will use a graph representation of the CQs that is more convenient for the definition of similarity measures. Let $q : Q(\mathbf{x}) \leftarrow \bigwedge_i^n C_i(\mathbf{x};\mathbf{y})$ posed over an ontology $\mathcal{O}$. Since the conjuncts of q are entities of the terminology, they can only employ one (if they are concepts) or two (if they are roles) variables (distinguished or not). A non-distinguished variable that appears only once in the query body is called *unbound*, otherwise it is bound. We represent a CQ as an undirected graph $\mathcal{G}_q(\mathcal{V}_q, \mathcal{E}_q, \mathcal{L}_{\mathcal{V}_q}, \mathcal{L}_{\mathcal{E}_q})$, where $\mathcal{V}_q$ is the set of nodes representing the variables of the query, $\mathcal{E}_q$ is the set of edges, $\mathcal{L}_{\mathcal{V}_q}$ is the set of *node labels* (representing the set of concept conjuncts employing each variable) and $\mathcal{L}_{\mathcal{E}_q} = \langle \mathcal{L}^+_{\mathcal{E}_q}, \mathcal{L}^-_{\mathcal{E}_q} \rangle$ is the tuple of sets of *edge labels* (representing the set of role conjuncts employing each variable pair in $\mathcal{L}^+_{\mathcal{E}_q}$ and its inverse in $\mathcal{L}^-_{\mathcal{E}_q}$). The nodes corresponding to unbound variables are called *blank nodes*. Moreover, let $\mathcal{V}_q^{(b)}$ ($\mathcal{V}_q^{(u)}$) denote the set of bound (unbound) variable nodes and $\mathcal{E}_q^{(b)}$ ($\mathcal{E}_q^{(u)}$) denote the set of edges employing two bound (at least one unbound) variable node(s). Obviously, $\mathcal{L}_{\mathcal{V}_q}(x) = \emptyset$ and $|\mathcal{L}^+_{\mathcal{E}_q}(x,y) \cup \mathcal{L}^-_{\mathcal{E}_q}(x,y)| = 1$, for each $x \in \mathcal{V}_q^{(u)}$, $y \in \mathcal{V}_q$. We can easily extend the above notations to represent general terms instead of simple variables. The only issue that needs more attention is that in this case a node is blank only if it represents a term employing functions and variables that do not appear in any other term. For simplicity, we can delete the blank nodes of the graph and add the role name that appears in $\mathcal{L}^+_{\mathcal{E}_q}(x,y)$ (or its inverse if it appears in $\mathcal{L}^-_{\mathcal{E}_q}(x,y)$) in the label of the node x that is connected with the specific blank node. In this case, the node label sets can also include role names (or inverse role names). We will make use of this simplifying convention in the sequel, in the description of our algorithm. There are several graph similarity measures (relations between graphs that are reflexive and symmetric) proposed in the literature, especially in the framework of ontology matching [11, 13]. Here, we will introduce a new measure that captures the intuitive meaning of the query similarity that is useful in query rewriting PCQA algorithms.

Definition 5. Query Similarity
Let q_1 and q_2 be two non-empty conjunctive queries and $\mathcal{G}_{q_1}(\mathcal{V}_{q_1}, \mathcal{E}_{q_1}, \mathcal{L}_{\mathcal{V}_{q_1}}, \mathcal{L}_{\mathcal{E}_{q_1}})$, $\mathcal{G}_{q_2}(\mathcal{V}_{q_2}, \mathcal{E}_{q_2}, \mathcal{L}_{\mathcal{V}_{q_2}}, \mathcal{L}_{\mathcal{E}_{q_2}})$ their graph representations. Their similarity can be defined as follows (up to variable renaming):

$$\sigma(q_1, q_2) = 1 - \frac{\frac{\lambda_v + \lambda_\epsilon}{v_{\min} + \epsilon_{\min}} + (\delta_v + \delta_\epsilon)}{|\mathcal{V}_{q_1}^{(b)}| + |\mathcal{V}_{q_2}^{(b)}| + |\mathcal{E}_{q_1}^{(b)}| + |\mathcal{E}_{q_2}^{(b)}| + 1} \tag{1}$$

where $v_{\min} = \min(|\mathcal{V}_{q_1}^{(b)}|, |\mathcal{V}_{q_2}^{(b)}|)$, $\epsilon_{\min} = \min(|\mathcal{E}_{q_1}^{(b)}|, |\mathcal{E}_{q_2}^{(b)}|)$, $\delta_v = |\mathcal{V}_{q_1}^{(b)} \triangle \mathcal{V}_{q_2}^{(b)}|$, $\delta_\epsilon = |\mathcal{E}_{q_1}^{(b)} \triangle \mathcal{E}_{q_2}^{(b)}|$ *(*$\triangle$ *computes the non-common elements of the two sets),* $\lambda_v = |\left\{x : x \in \mathcal{V}_{q_1}^{(b)} \wedge x \in \mathcal{V}_{q_2}^{(b)} \wedge \left(\mathcal{L}_{\mathcal{V}_{q_1}}(x) \neq \mathcal{L}_{\mathcal{V}_{q_2}}(x) \vee \mathcal{L}_{q_1}^{(u)}(x,y) \neq \mathcal{L}_{q_2}^{(u)}(x,y)\right)\right\}|$ *and* $\lambda_\epsilon = |\{(x,y) : (x,y) \in \mathcal{E}_{q_1}^{(b)} \wedge (x,y) \in \mathcal{E}_{q_2}^{(b)} \wedge \mathcal{L}_{\mathcal{E}_{q_1}}(x,y) \neq \mathcal{L}_{\mathcal{E}_{q_2}}(x,y)\}|$.

The first term of the numerator of the similarity measure (Eq. 1) captures the node and edge labeling differences, the second term the structure difference, while the denominator normalises the values between 0 and 1. The main intuition behind Eq. 1 is that the labeling differences should be counted as a secondary dissimilarity cause, while the primer one should be the difference in structure. Thus, the maximum value of the first term of the fraction should not exceed the value of the second term (cannot exceed the value 1, which is the lowest structural difference). The computation of differences ignores all the unbound variables (the blank nodes of the graph), that are only involved in the computation of λ_v (summarising the node label differences). The intuition behind this is that blank nodes are introduced only by unqualified existentials ($\exists R.\top$) and thus the specific unbound variable could be rejected without any problem if we just remember the role of the existential. It is important to notice that in the presence of role inverse, the bound variable could be either in the subject or in the filler part of the role; this is the reason why we consider undirected graphs and $\mathcal{L}_q^{(u)}$ (considering both $\mathcal{L}_q^{(u)+}$ and $\mathcal{L}_q^{(u)-}$) is involved in the computation of λ_v. Finally, we should notice that with a little abuse of notation, we introduced the similarity between two queries and not between queries and answers (as imposed in the previous section), implicitly meaning that the similarity is between the first query and the answers of the second one. In this way, we significantly simplified the notation in the case of query rewriting based PCQA algorithms.

Table 1. Translation of DL-Lite$_R$ axioms into clauses of $\Xi(\mathcal{O})$. *(Note: A different function f must be used for each axiom involving an existential quantifier.)*

Axiom	Clause	Type	Axiom	Clause
$A \sqsubseteq B$	$B(x) \leftarrow A(x)$	(1)		
$P \sqsubseteq R$ $R \sqsubseteq P$	$S(x,y) \leftarrow P(x,y)$	(2)	$P \sqsubseteq R^-$ $R^- \sqsubseteq P$	$S(x,y) \leftarrow P(y,x)$
$\exists P \sqsubseteq A$	$A(x) \leftarrow P(x,y)$	(3)	$\exists P^- \sqsubseteq A$	$A(x) \leftarrow P(y,x)$
$A \sqsubseteq \exists P$	$P(x, f(x)) \leftarrow A(x)$	(4)	$A \sqsubseteq \exists P^-$	$P(f(x), x) \leftarrow A(x)$
$A \sqsubseteq \exists P.B$	$P(x, f(x)) \leftarrow A(x)$ $B(f(x)) \leftarrow A(x)$	(4) (5)	$A \sqsubseteq \exists P^-.B$	$P(f(x), x) \leftarrow A(x)$ $B(f(x)) \leftarrow A(x)$

We are now ready to introduce a PCQA algorithm, which we call ProgResAns and is sorted according to $\sigma(q_1, q_2)$. In order to compute the query rewritings of a user query q, the algorithm employs a set of resolution rules. The main premise is always a query (q or a subsequently computed query rewriting) and the side

premise a clause of $\Xi(\mathcal{O})$. $\Xi(\mathcal{O})$ is obtained from ontology $\mathcal{O}$ as described in Table 1 [7]. The idea of the algorithm is the following: start from the user query; apply the resolution rule using side premises that preserve the structure of the query (we call this step *structure preserving resolution* or *sp-resolution*); apply the resolution rule using side premises that minimally change the structure of the query (we call this step *structure reducing resolution* or *sr-resolution*); apply anew sp-resolution to the query rewritings produced by sr-resolution, and so on; at each step evaluate the queries against the database and provide the user with the results. sp- and sr-resolution are performed by procedures sp-Resolve and sr-Resolve, respectively. sp-Resolve takes as input a query q and an element s of $\mathcal{G}_q$ (i.e. either a node or an edge), and computes all possible rewritings of q, by iteratively applying the resolution rule using as side premises the clauses of $\Xi(\mathcal{O})$ that are of type (1)-(4) (see Table 1). Initially, the main premise is q and the resolution rule is applied for all atoms in q that correspond to s. The same process is iteratively applied to all the resulting rewritings, until no more rewritings can be obtained. The clauses of type (4) are used only if the skolemized term $f(x)$ unifies with an unbound variable of the main premise. sp-Resolve preserves the query structure, since resolution with clauses of type (1) or (2) affects only the sets $\mathcal{L}_\mathcal{V}$ and $\mathcal{L}_\mathcal{E}$, respectively. Clauses of type (3) and (4) introduce and eliminate blank nodes.

sr-Resolve takes as input a query q and an element s of $\mathcal{G}_q$ (i.e. either a node or an edge) and computes all possible rewritings of q, by iteratively applying the resolution rule using as side premises the clauses of $\Xi(\mathcal{O})$ that are of type (4) and (5). Clauses of type (4) are used only if $f(x)$ unifies with a bound variable of the query. In terms of graphs, sr-Resolve deletes a node together with the edges that connect it to the rest of the graph. ProgResAns applies sr-Resolve only on selected atoms (the atoms that correspond to the elements s of $\mathcal{G}_q$ mentioned above) of q, called the *sink node atoms* of q. Sink nodes s correspond to non distinguished terms and are determined by the following condition concerning their labels: $\mathcal{L}^{\pm}_{\mathcal{E}}(s, y) = \emptyset$ and $\mathcal{L}^{\mp}_{\mathcal{E}}(s, y) \geq 1$. The selective application of the resolution rule only to the sink node atoms is proved to suffice for the production of eventually all the possible query rewritings which can be evaluated against the database (i.e. query rewritings that do not contain functional terms).

We now provide the full definition of ProgResAns. Its components are the query rewriting procedures (sp-Rewrite and sr-Rewrite) and the procedure Eval, which evaluates a set of rewritings against the database:

$$\mathsf{ProgResAns} = \mathsf{Eval}\ ;\ \{\{[\mathsf{sp\text{-}Rewrite}\mid \mathsf{Eval}]\}_{i=1}^{n_j}\ ;\ [\mathsf{sr\text{-}Rewrite}\mid \mathsf{Eval}]\}_{j=1}^{m}$$

sp-Rewrite and sr-Rewrite are defined by algorithms 1 and 2. sp-Rewrite employs sp-Resolve to perform exhaustive sp-resolution for all queries in the input set Q^{sp}_{in}. Therefore, the queries in res(sp-Resolve) have the same structure (up to blank nodes) with the queries in Q^{sp}_{in}, but each one of them is the result of the exhaustive application of sp-resolution on a single node or edge. The node or edge whose label is modified is annotated, so that by recursively applying sp-Resolve we can compute all possible rewritings that have the same number of different

labels. In particular, the queries computed by the k-th recursive application of sp-Resolve differ in k labels w.r.t the queries Q^{sp}_{in} of the first application. Finally, sr-Rewrite uses sr-Resolve to compute the rewritings that have a single structural change w.r.t. Q^{sr}_{in}.

Data: Set of annotated conjunctive queries Q^{sp}_{in}, ontology $\mathcal{O}$
Result: Set of annotated query rewritings
$Q := \{\}$;
foreach *query* q *in* Q^{sp}_{in} **do**
  foreach s *in* $\mathcal{V}_q \cup \mathcal{E}_q$ *that is not annotated* **do**
    $Q := Q \bigcup$ sp-Resolve(q, s, $\Xi(\mathcal{O})$);
  end
end
return Q;

Algorithm 1: Procedure sp-Rewrite

Data: Set of conjunctive queries Q^{sr}_{in}, ontology $\mathcal{O}$
Result: Set of query rewritings
$Q^{sr}_{in} :=$ filter(Q^{sr}_{in});
$Q := \{\}$;
foreach *query* q *in* Q^{sr}_{in} **do**
  $\mathcal{X} :=$ sink-nodes(q);
  foreach x *in* $\mathcal{X}$ **do**
    if $\mathcal{L}(x) = \{A_1, \ldots, A_n\}$ *(*$\mathcal{L}(x)$ *is a set of concepts)* **then**
      forall the *concepts* C *such that* $\forall i = 1 \ldots n \, \Xi(\mathcal{O}) \models A_i \leftarrow C$ **do**
        $Q := Q \bigcup$ sr-Resolve(q, $C(x)$, $\Xi(\mathcal{O})$);
      end
    else if $R \in \mathcal{L}^{\pm}(x, y)$ **then**
      $Q := Q \bigcup$ sr-Resolve(q, $R(x, y)$, $\Xi(\mathcal{O})$);
    end
  end
end
return Q;

Algorithm 2: Procedure sr-Rewrite

ProgResAns, after evaluating first the user query, enters an (outer) loop, part of which is the (inner) loop [sp-Rewrite | Eval]. The outer loop is executed (let's say m times) until sr-Rewrite can make no further structural change to the query. The j-th time the outer loop is executed, the inner loop is executed n_j times, where n_j is the number of nodes and edges of the queries in $Q^{sp}_{in_j}$ (all have the structure and $m \leq n_1$) and $Q^{sp}_{in_j}$ is the input of the first application of sp-Rewrite at the j-th iteration of the outer loop. $Q^{sp}_{in_j}$ contains only the user query when $j = 1$, otherwise it is equal to res(sp-Rewrite) obtained at iteration $j - 1$. The input $Q^{sr}_{in_j}$ of sr-Rewrite contains the rewritings that are computed by the execution of sp-Rewrite. Before entering its main body, sr-Rewrite calls procedure filter on $Q^{sr}_{in_j}$, which keeps only the rewritings in which sp-Rewrite

has changed only sink node atoms, so that, as mentioned before, sr-resolution is applied only on these atoms. As soon as sp-Rewrite or sr-Rewrite return, Eval computes cert(res(sp-Rewrite), $\mathcal{O}$) and cert(res(sr-Rewrite), $\mathcal{O}$), respectively. The following theorem can be proved:

Theorem 1. ProgResAns *terminates and it is sound, complete and σ-sorted.*

Proof. The proof is omitted due to restricted space.

Example 2. (continued) Table 2 summarises the results of applying ProgResAns to the input of Example 1 (all the single strides are shown).

Table 2. Results of ProgRes in the data of Example 1

Stride	Query rewritings	Similarity	Answers
1	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{advise}(y, z)$	1.000	John
2	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{advise}(y, z)$	0.952	
	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{ResCoordinator}(y)$	0.952	John
	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{ResDirector}(y)$	0.952	
	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{supervise}(y, z)$	0.952	Alan
	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{SeniorResearcher}(y)$	0.952	Alan
	$Q(x) \leftarrow \mathsf{advise}(x, y) \wedge \mathsf{Professor}(y)$	0.952	
3	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{ResCoordinator}(y)$	0.905	
	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{ResDirector}(y)$	0.905	
	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{supervise}(y, z)$	0.905	
	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{SeniorResearcher}(y)$	0.905	
	$Q(x) \leftarrow \mathsf{supervise}(x, y) \wedge \mathsf{Professor}(y)$	0.905	
4	$Q(x) \leftarrow \mathsf{ResDirector}(x)$	0.400	John
5	$Q(x) \leftarrow \mathsf{Professor}(x)$	0.400	Sofia, Emma

5 System evaluation

We now present an empirical evaluation of ProgRes, which implements the ProgResAns algorithm, assuming that ABoxes are stored in a relational database. Our goal is to evaluate the performance of ProgRes and investigate whether the ranked computation of the answers introduces a significant time overhead. Given the progressive nature of ProgRes, meaning that each rewriting should be evaluated against the database upon its computation, our implementation follows a producer/consumer approach: A thread computes the rewritings and adds them to an execution queue, while another thread implementing the Eval procedure, retrieves the rewritings from the queue, translates them into SQL, dispatches them to the database, and collects the answers. We compare ProgRes with Requiem (implementation of the algorithm in [7]), which is the most similar non-progressive DL-Lite$_\mathcal{R}$ query answering system. We should point out, however, that a fair comparison is not possible since we are comparing a ranking,

progressive algorithm with a non-ranking, non-progressive one. For an as fair as possible comparison, we reimplemented the greedy unfolding strategy of Requiem within our programming framework. This was mainly done for the comparison of the time performance to be done on equal terms and did not alter in any way the Requiem algorithm (although as we shall see in the sequel in one case our implementation performed significantly better than the original one). Nevertheless, we have to notice that the results should be interpreted only as indicative of the relative performance of the two systems.

An important issue our system had to face is the fact that, due to the real-time orientation of ProgRes, many redundant rewritings (i.e. rewritings that are structurally subsumed by subsequent, not yet computed rewritings) may be obtained. If all of them are dispatched to the database, performance may be significantly compromised. For this reason we slightly delay the addition of the new rewritings to the execution queue, by computing first all the rewritings of a stride, check for redundancies and add only the non-redundant ones to the queue. The structure of the Requiem algorithm does not allow for a directly comparable optimization strategy. In fact, the last step of Requiem is precisely a final query redundancy check step, in which any redundant queries are discarded all together. Therefore, in our evaluation of Requiem, following the original implementation, all the queries are added to the queue after the entire algorithm has finished and the final redundancy check has been performed.

We present the results for the A and S ontologies [7], as well as for P5S, a modified version of P5, in which we have included some subconcepts for the Path$X$ concepts of P5. Briefly, ontology A is an ontology that models information about abilities, disabilities and related devices developed for the South African National Accessibility Portal. Ontology S is an ontology that models information about European Union financial institutions. Ontology P5 is a synthetic ontology developed by the authors of [7] that models information about graphs (nodes and vertices) and contains classes that represent paths of length 1-5.

We evaluate ontology A with queries AQ4 and AQ5 (of size 3 and 5, respectively), ontology S with query SQ5 (of size 7) and P5S with queries PQ4 and PQ5 (of size 4 and 5, respectively). Due to restricted space we do not present the results of all ontologies and queries in [7], we choose the particular combinations as most representative of several diverse cases. We populated the ABoxes according to [9]. The results are shown in Fig. 1. In each row, the lhs graphs present the total number of retrieved answers vs time and the rhs graphs the evolution of similarity as new answers are retrieved. Execution time is total time, including both query rewriting computation and answer retrieval time. The small vertical bars at the two horizontal dotted lines on the top of the lhs graphs indicate the time points at which one or more new rewritings become available for execution. The upper line corresponds to ProgRes, the lower to Requiem. Table 3 presents the number of rewritings and inferences. Within parentheses are the no. of inferences during sp-resolution (which corresponds roughly to the unfolding of Requiem) and the no. of inferences during sr-resolution (which corresponds roughly to the saturation step of Requiem). Note that some of the inferences

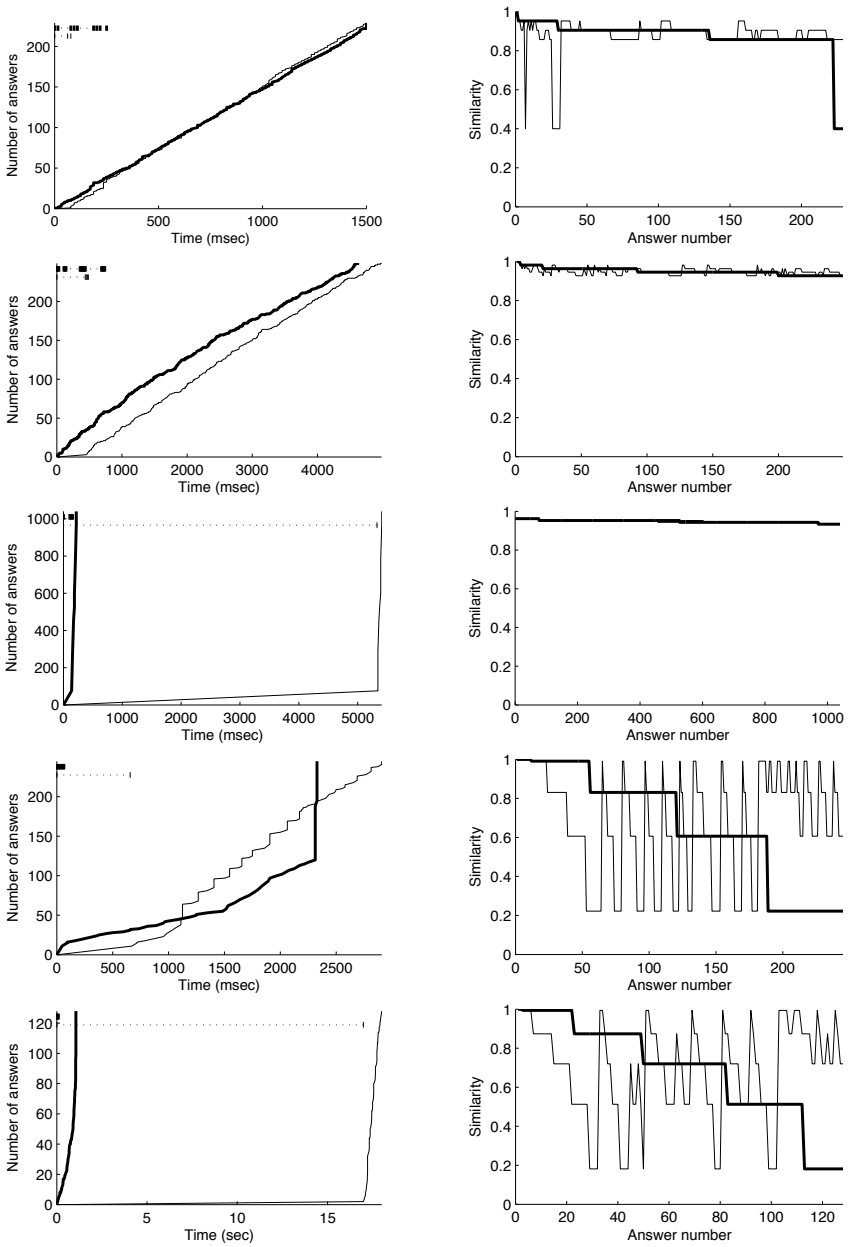


Fig. 1. Execution time and similarity. From top to bottom the graphs in each row correspond to the ontology-query pairs A-AQ4, A-AQ5, S-SQ5, P5S-PQ4 and P5S-PQ5. The thick and thin lines correspond to ProgRes and Requiem, respectively

that ProgRes performs during sp-resolution, in Requiem are performed during saturation. The last part of Table 3 presents the time required for the computation of the rewritings (not including the query execution) for ProgRes and Requiem. For Requiem two times are given: the first refers to our implementation of Requiem (which we used in all other parts of the evaluation), and the second one, within parenthesis, to the original implementation (obtained from `http://www.comlab.ox.ac.uk/projects/requiem`). We note that in one case, in particular for ontology S, our implementation performs significantly better. This is due to the fact that it handles more effectively and removes at an earlier stage atoms that happen to be redundant within a single rewriting; as we shall also mention in the sequel, SQ5 happens to be a query that produces a lot of rewritings that contain several redundant atoms.

Table 3. Number of rewritings, number of inferences and query rewriting time for ProgRes and Requiem.

ontology/ query		no. of rewritings		no. of inferences		rewriting time (ms)	
		ProgRes	Requiem	ProgRes	Requiem	ProgRes	Requiem
A	AQ4	320	224	532 (507/25)	540 (507/25)	188	78 (156)
A	AQ5	624	624	1326 (1324/2)	1017 (811/206)	750	484 (625)
S	SQ5	128	8	892 (891/1)	1457 (1433/24)	156	5250 (21062)
P5	PQ5	16	16	25 (5/20)	11350 (0/11350)	31	657 (687)
P5S	PQ5	76	76	95 (75/20)	11424 (74/11350)	47	17015 (16860)

Ontology A is to a large extent taxonomic. Both queries AQ4 and AQ5 produce many non-redundant rewritings, after a long sp-resolution/unfolding phase. In AQ4, ProgRes produces more queries (some redundant ones) and in AQ5 it performs more inferences than Requiem. This is due to the structured pattern followed in the computation of the rewritings, which as mentioned before may introduce more redundancies. This is more obvious in ontology S, where ProgRes produces a lot of redundant queries. This is however an extreme degenerate case: SQ5 is a bad query, as half of its atoms are essentially redundant. Back to ontology A, we note that although ProgRes needs slightly more time to compute the rewritings, it terminates faster. This is due to the progressive nature of ProgRes. While the producer thread computes the query rewritings, the consumer thread executes the already available ones, so there is no significant idle time for the consumer thread. In contrast, in Requiem the rewritings are obtained at the end, all together. This demonstrates the significant benefit from progressively computing and evaluating the query rewritings. The situation is quite different in ontology P5S, in which the inference procedure consist mainly of a chain of sr-reduction/saturation steps. Here, the strategy of ProgRes is much more efficient and scalable. It avoids a very large number of inference sequences that are guaranteed to give no new queries, by applying sr-resolution only on sink atoms. As a result, it finishes almost instantly, while for query PQ5 Requiem needs significantly more time and inferences. It is important to note, that in this case the

performance of Requiem is not scalable in the size of the query, as it is easy to see by comparing the results for queries PQ4 and PQ5 (in PQ5 we search for all graph paths of length 5, while in PQ4 for paths of length 4). In contrast to Requiem, almost the entire execution time of ProgRes is answer retrieval time. The synthetic nature of P5S allows us to comment also on the evolution of the similarity of the answers. In (the original) ontology P5, only sr-resolution/saturation steps are involved, hence we expect ProgRes and Requiem to compute the rewritings in the same order (but not the same fast). In ontology P5S, however, for each rewriting produced at a sr-reduction step, ProgRes computes immediately all its sp-resolutions, and so the progressive decrease of similarity is achieved. Requiem computes first all the rewritings resulting from the saturation step and then all the unfoldings, hence no ranking of the answers is possible. Similar is the situation for query AQ4. In the case of AQ5, all the rewritings have the same graph structure, so the slight fluctuations in the similarity graph for Requiem are due to the non-ordered computation of the unfoldings. A steep decrease in the similarity measure occurs only when the structure of a rewriting w.r.t the original query changes.

6 Conclusions and future work

We have presented a systematic approach to the problem of stratifying over time the execution of CQA algorithms. We introduced the notions of progressive CQA algorithms (sequences of CQAs - its components), of strides (result sets of subsets of the components), of ordering of strides measuring the relevance of the answers with the user query and of sorted progressive CQAs ensuring a controllable approximation of the correct answer set. We have also presented a practical algorithm for progressive CQA in DL-Lite$_{\mathcal{R}}$ that implements the above ideas and showed that it is possible to develop progressive algorithms that are efficient. Actually, it has been found that in the presence of large queries and TBoxes that are not simple taxonomies of atomic concepts the proposed algorithm can be much more efficient than the similar non-progressive ones (overcoming a strong limitation of some DL-Lite$_{\mathcal{R}}$ CQA systems). An obvious advantage of the specific approach is the ability to be responsive even in cases of huge databases under a predetermined approximation strategy. A second advantage is that in case of inconsistency and considering data as stronger than theory (in information retrieval this is reasonable), PCQAs ensure a decreased possibility of incorrect answers. Finally, PCQA have ideal structure for parallel processing. The main disadvantage is that it is difficult to reduce the redundancies, since the complete set of answers is not available before the output. The present work can be extended in several directions. We could take advantage of the ideas presented in [10] and dramatically improved the performance of DL-Lite$_{\mathcal{R}}$ CQAs. Also, we could develop PCQA algorithms and systems for more expressive DLs. Finally, we could try to stratify smaller strides in ProgRes avoiding wide replications by applying sophisticated redundancy checking.

Acknowledgements

This work has been funded by ECP 2008 DILI 518002 EUscreen (Best Practice Networks).

References

1. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. J. on Data Semantics, pp. 133–173 (2008)
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2007)
3. B. Motik et al (editors). OWL 2 web ontology language profiles. W3C Recommendation, 27 October (2009)
4. A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. Journal of Artificial Intelligence Research, pp. 36–69 (2009).
5. J. A. Robinson and A. Voronkov, editors. Handbook of Automated Reasoning (in 2 volumes). Elsevier and MIT Press (2001)
6. B. Glimm, I. Horrocks, C. Lutz, and U. Sattler. Conjunctive query answering for the description logic SHIQ. J. of Artificial Intelligence Research, 31:157–204 (2008)
7. H. Perez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for OWL 2. In: 8th International Semantic Web Conference (ISWC 2009), vol. 5823 of Lecture Notes in Computer Science, pp. 489–504. Springer (2009)
8. Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. Third International Semantic Web Conference, Hiroshima, Japan, LNCS 3298, Spinger (c), pp. 274-288 (2004)
9. G. Stoilos, B. Cuenca Grau, and I. Horrocks. How Incomplete is your Semantic Web Reasoner? In: 20th Nat. Conf. on Artificial Intelligence (AAAI) (2010)
10. Riccardo Rosati and Alessandro Almatelli, Improving Query Answering over DL-Lite Ontologies. In: Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010) (2010)
11. H. Bunke, Graph matching: theoretical foundations, algorithms, and applications. In: Vision Interface 2000, pp. 82–88. Montreal/Canada (2000)
12. M. Dehmer, F. Emmert-Streib, A. Mehler and J. Kilian, Measuring the structural similarity of web-based documents: a novel approach, J. Comp. Intell., pp. 1-7 (2006)
13. J. Euzenat and P. Shvaiko, Ontology Matching, Springer-Verlag (2007)
14. I. Ilyas, G. Beskales, and M. Soliman, A Survey of Top-k Query Processing Techniques in Relational Database Systems, ACM Computing Surveys, Vol. 40, No. 4, Article 11 (2008)
15. M. Soliman and I. Ilyas, Top-k Query Processing in Uncertain Databases, ICDE, pp. 896–905 (2007)
16. T. Lukasiewicz and U. Straccia, Top-k retrieval in description logic programs under vagueness for the Semantic Web, In Proc. SUM-2007, pp. 16–30 (2007)