SPARQL Query Processing Using Bobox Framework

Miroslav Čermák, Zbyněk Falt, Jiří Dokulil and Filip Zavoral
Charles University in Prague, Czech Republic
{*cermak,falt,dokulil,zavoral*}*@ksi.mff.cuni.cz*

***Abstract*—Proliferation of RDF data on the Web creates a need for systems that are not only capable of querying them, but also capable of scaling efficiently with the growing size of the data. Parallelization is one of the ways of achieving this goal. There is also room for optimization in RDF processing to reduce the gap between RDF and relational data processing. SPARQL is a popular RDF query language; however current engines do not fully benefit from parallelization potential. We present a solution that makes use of the Bobox platform, which was designed to support development of data-intensive parallel computations as a powerful tool for querying RDF data stores. A key part of the solution is a SPARQL complier and execution plan optimizer, which were tailored specifically to work with the Bobox parallel framework. The performance of the system is compared to the Sesame SPARQL engine.**

Keywords-SPARQL; Bobox; query optimization.

I. Introduction

SPARQL [1] is a popular RDF (Resource Definition Framework) query language. It contains capabilities for querying graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs.

The Bobox framework was designed to support development of data-intensive parallel computations [2], [3]. The main idea behind Bobox is to divide a large task into many simple tasks that can be arranged into a non-linear pipeline. These simple tasks are performed by *boxes*. They are executed in parallel and the execution is driven by the availability of data on their inputs. The developer of such boxes does not have to be concerned about problems such as synchronization, scheduling and race conditions. All this is done by Bobox itself. The system can easily be used as a database execution engine; however, each query language requires its own front-end that translates a request (query) into a definition of the structure of the pipeline that corresponds to the query.

In the paper, we present a way in which we used the Bobox framework to create a tool for effective parallel querying of RDF data [4] using SPARQL. The data are stored using an in-memory triple store which consists of one three-column table and a set of indexes. We provide a brief description of query processing using SPARQL-specific parts of the Bobox and provide results of benchmarks. Benchmarks were performed using the SP^{2}Bench [5] query set and data generator.

The rest of the paper is structured as follows: Sections II and III describe the Bobox framework and models used to represent queries during their processing. Section IV contains a description of the SPARQL compiler and steps performed during query processing. Bobox back-end processing and SPARQL specific boxes are discussed in the Section IV-D. Section V presents our experiments and a discussion of their results. Section VI describes future directions of research and concludes the paper.

II. Bobox Framework

A. Bobox Architecture

The Bobox parallelization framework has two primary goals: to simplify writing parallel, data intensive programs and to serve as a testbed for the development of generic parallel algorithms and data-oriented parallel algorithms. The main aspects that make writing parallel programs easier include the following: all synchronization is hidden from the user; most technical details (NUMA, cache hierarchy, CPU architecture) are handled by the framework; high-performance messaging is the only means of communication and synchronization; and it is built around easy-to-comprehend basic paradigms such as task parallelism and non-linear pipeline.

Bobox provides a run-time environment that is used to execute a non-linear pipeline in parallel. The pipeline consists of computational components provided by the user and connecting parts that are part of the framework. The structure of the pipeline is defined by the user, but the communication and execution of individual parts is handled by the run-time; a component is executed when it has data waiting to be processed on its inputs. This simplifies the design of the individual computational components, since communication, synchronization and scheduling are handled by the framework.

Compared to scientific workflows, the Bobox boxes are usually smaller than actors or other workflow elements and they never encapsulate user interaction or unreliable remote communication.

B. Task Level Parallelism

The environment with many simple components and pipeline-based communication is very suitable for task level parallelization. In this paradigm, the program is not viewed as a process divided into several threads. Instead, it is seen

 ISBN: 978-1-61208-175-5

as a set of small tasks. A *task* is a piece of data together with the code that should be executed on the data. Their execution is handled by a *task scheduler*. The scheduler maintains a pool of tasks to be executed and a pool of execution threads and allocates the tasks to the threads. At any given time, a thread can either be executing a task or be *idle*. If it is idle, the task scheduler finds a suitable task in the task pool and starts the execution of the task on the idle thread.

C. Run-time Architecture

One of the main differences between other parallelization frameworks and the Bobox architecture is the way the user's code interacts with Bobox. OpenMP [6] and TBB [7] are used to invoke parts of the code in parallel; MPI [8] provides means for communication between processes. Bobox is more similar to the first two systems; however, there are two key differences. First, it uses a declarative approach to describe the way in which elements of the computation are put together. Second, it provides more services to the user code (data transport, flow control etc.), but also imposes greater restrictions (only pipeline, no recursive calls, etc.).

The parallel execution environment is somewhat similar to that of TBB, since it contains a task pool and several threads that execute tasks from that pool. However, the way in which the tasks are created and added to the pool is completely different [9]. In TBB, this is controlled either directly by the user's code or by using a thin layer of parallel algorithms provided by the library.

In Bobox, the user first specifies a *model*. The model defines the way in which the individual computational components are connected. The model is then *instantiated* to produce a *model instance*. The elements of the model instance are used as tasks. When they are ready, they are *enqueued* – added to the task pool. Later, a thread takes a task from the pool, performs the action (*invokes* the task) and then the model instance element is returned and can be used again as a new task and added to the pool.

D. Scheduling

The Bobox system is well suited for a certain class of problems, due to the way in which the system decides what computational components should be executed. This is controlled by the flow of the data through the pipeline. The data must be passed in a way defined by the system, so that the system is aware of the fact that a component consumed or created some data. This simplifies the design of the individual computational components; they do not have to be concerned with controlling the execution and data flow.

The basic Bobox computational component is a *Box*. Boxes are used for the implementation of basic operations such as joins (see Section IV-D for a more details).

III. Query Representation

During query processing, our SPARQL compiler uses different representations of the query itself. They are chosen according to the needs of each processing step. In the following sections, we mention models used during query rewriting and generation of execution plan.

A. SQGM Model

Pirahesh et al. [10] proposed the Query Graph Model (QGM) to represent SQL queries. Hartig and Reese [11] modified this model to represent SPARQL queries (SQGM). With appropriate operations definition, this model can be easily transformed into Bobox pipeline definition, so it was ideal candidate to use.

SQGM model can be interpreted as a directed graph (in our case a directed tree). Nodes represent operators and are depicted as boxes containing headers, body and annotations. Edges represent data flow and are depicted as arrows that follow the direction of the data. Figure 1 shows an example of a simple query represented in the SQGM model.

This model is created during execution plan generation step and is used as a definition for the Bobox pipeline.

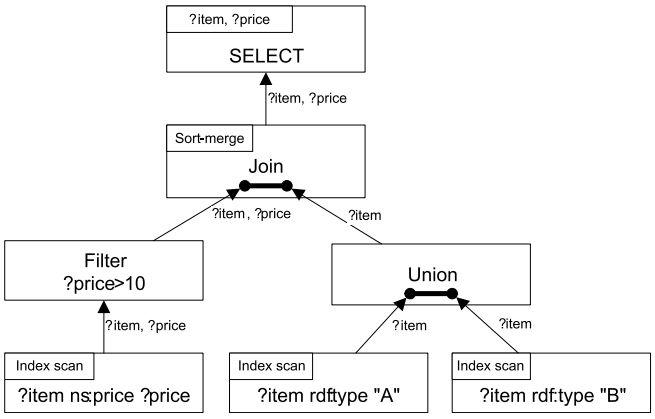

Figure 1. Example of SQGM model.

B. SQGPM Model

In [12], we proposed the SPARQL Query Graph Pattern Model (SQGPM) as the model that represents query during optimization steps. This model is focused on representation of the SPARQL query graph patterns [1] rather than on the operations themselves as in the SQGM. It is used to describe relations between group graph patterns (graph patterns consisting of other simple or group graph patterns). The ordering among the graph patterns inside a group graph pattern (or where it is not necessary in order to preserve query equivalency) is undefined. An example of the SQGPM model graphical representation is shown in Figure 2.

Each node in the model represents one group graph pattern that contains an unordered list of references to graph patterns. If the referenced graph pattern is a group graph pattern, then it is represented as another SQGPM node. Otherwise the graph pattern is represented by a leaf.

 ISBN: 978-1-61208-175-5

The SQGPM model is built during the syntactical analysis and is modified during the query rewriting step. It is also used as a source model during building the SQGM model.

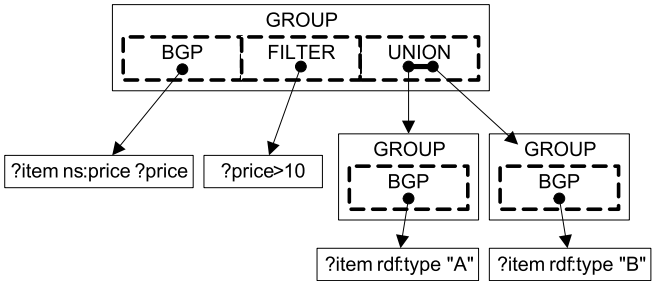

Figure 2. Example of SQGPM model.

IV. Query Processing

Query processing is performed in a few steps by separate modules of the application as shown in Figure 3. First steps are performed by the SPARQL front-end represented by compiler. The main goal of these steps is to validate the compiled query, pre-process it and prepare the optimal execution plan according to several heuristics. Execution itself is done by the Bobox back-end where execution pipeline is initialized according to the plan from the front-end. Following sections describe steps done by the compiler in a more detail way.

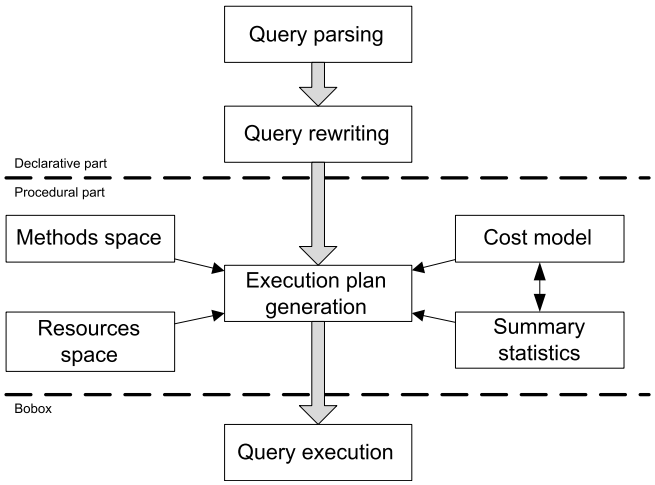

Figure 3. Query processing scheme.

A. Query Parsing

The query parsing step uses standard methods to perform syntactic and lexical analysis according to W3C recommendation. The input stream is transformed into a SQGPM model. Transformation also includes expanding short forms in query, replacing aliases and transformation of blank nodes into variables.

B. Query Rewriting

The second step is query rewriting. We cannot expect that all queries are written optimally (they may contain duplicities, constant expressions, inefficient conditions, redundancies etc.). So, the goal of this phase is to normalize queries to achieve a better final performance. We use the following operations:

- Merging of nested *Group graph patterns*
- Duplicities removal
- *Filter*, *Distinct* and *reduced* propagation
- Projection of variables

It is also necessary to check applicability of each operation with regards to the SPARQL semantics, before it is used to preserve query equivalency. Query representation is the same as in the previous step.

C. Execution Plan Generation

In the previous steps, we described some query transformations that resulted in a SQGPM model. However this model does not specify complete order of all operations. Main goal of the execution plan generation step is to transform the SQGPM model into an execution plan. This includes selecting from different join operation orderings, join types and selecting the best strategy to access the data stored in the physical store.

The query execution plan is built from the bottom to the top using dynamic programming to search part of the search space of all possible joins. This strategy is applied to each group graph pattern separately because the order of the patterns is fixed in the SQGPM model. Also, the result ordering is considered, because a partial plan that seems to be worse locally, but produces a useful ordering of the result may provide a better overall plan later. The list of available atomic operations (e.g., the different types of joins) and their properties are provided by the *Methods Space* module.

In order to compare two execution plans, it is necessary to estimate the *cost* of both plans – an abstract value that represents the projected cost of execution of a plan on the actual data. This is done with the help of the *cost model* that holds information about atomic operation efficiency and *summary statistics* gathered about the stored RDF data.

Search space of all execution plans could be extremely large, so we used heuristics to reduce the complexity of the search. At first, only left-deep trees of join operations are considered. This means that right operand of join operation may not be another join operation. There is one exception to this rule – avoiding cartesian products. If there is no other way to add another join operation without creating cartesian product, the rest of the unused operations is used to build separate tree recursively (using the same algorithm) and result is joined with the already built tree. This modification greatly improves plans for some of the queries we have tested and often significantly reduces the depth of the tree.

 ISBN: 978-1-61208-175-5

The final execution plan is represented using SQGM model and later transformed int a Bobox model. This transformation is completely straightforward.

D. Query Representation for Back-end

After the execution plan is generated, it is transformed into a serialized form and passed to the back-end. The back-end deserializes the plan and instantiates boxes provided by the runtime implementation. Boxes are connected according to the plan and computation may then be started. The serialization and deserialization is useful since it provides many benefits, such as:

- When distributed computation support is added, text representation is safer than (e.g., binary), where problems with different formats, encodings or reference types may appear.
- Serialization language has very simple and effective syntax; serialization and deserialization is much faster than (e.g.) the use of XML.
- Text representation is independent on the programming language; new compilers can be implemented in a different language.
- Compilers can generate plans that contain boxes that have not yet been implemented, which allows for earlier testing of the compiler during the development process.

E. Runtime

Another important part of the front-end on which the compiler depends is called *runtime*. It provides compiler-specific features in the (otherwise compiler independent) back-end. For example, it handles the instantiation of the boxes, since they are compiler-specific (e.g., the join operation used in SPARQL is slightly different from joins used in SQL). SPARQL runtime provides boxes that represent operations used in SPARQL evaluation. Examples of such boxes are *scan*, *join*, *union*, *filter* box etc. Some of the operations have different implementations. For example, scan box is implemented as full-table scan using direct access to the triples table but also as an indexed access to the table. Join boxes use two basic approaches: nested-loops join and merge-join (faster, but requires ordered inputs). Most other boxes use only one implementation.

V. Experiments

We performed a number of experiments to test functionality and performance of the SPARQL query engine. The experiments were performed using the SP^{2}Bench [5] query set, since this benchmark is considered to be standard in the area of semantic processing. The compiler output was visualized to check the correctness of the plans and the whole query engine was benchmarked against a set of test queries on differently sized data sets to determine. We also performed the same tests on the Sesame [13] SPARQL engine, so we can compare these two SPARQL query engines.

A. Set-up

Experiments were performed on a server running Redhat 6.0 Linux. Server configuration is 2x Intel Xeon E5310, 1,60Ghz (L1: 32kB+32kB L2: 2x4MB shared) and 8GB RAM. It was dedicated specially to the testing; therefore no other CPU or memory services were running on the server. As the benchmark framework (queries and data) we chose the SP^{2}Bench [5] framework that is targeted on testing SPARQL engines and provides a set of queries, and a data generator that creates DBLP-like publication database.

SPARQL front-end and Bobox are implemented in C++. Document data were stored in-memory. We also tested Sesame v2.0 engine using its in-memory data store. We report the total elapsed time that was measured by a timer.

For all scenarios, we carried out multiple runs over documents containing 10k, 50k, 250k, 1M, and 5M triples and we provide the average times. Each test run was also limited to 30 minutes (the same timeout as in the original SP^{2}Bench paper). All data were stored in-memory, as our primary interest is to compare the basic performance of the approaches rather than caching etc. The expected number of the results for each scenario can be found in Table I.

B. Discussion of the Benchmarks Results

The query execution times are shown in Figure 4. The y-axes are shown in logarithmic scale and individual plots scale differently. In following paragraphs, we discuss some of the queries and their results.

Q2 implements a bushy graph pattern and the size of the result grows with the size of the queried data. We can see that Bobox scales well, even though it creates execution plans shaped as a left-deep tree. This is due to the parallel stream processing of fast merge joins.

The variants of Q3 (labeled *a* to *c*) test FILTER expression with varying selectivity. We present only the results of Q3c as the results for Q3a and Q3b are similar. The performance of Bobox is negatively affected by the simple statistics implementation used to estimate the selectivity of the filter.

Q4 (Figure 5) contains a comparably long graph chain, i.e., variables ?name1 and ?name2 are linked through articles that (different) authors have published in the same journal. Bobox embeds the FILTER expression into this computation, instead of evaluating the outer pattern block and applying the FILTER afterwards and propagates the DISTINCT modifier closer to the leaves of the plan in order to reduce the size of the intermediate results. This provides better performance than Sesame.

Queries Q5a (Figure 5) and Q5b test implicit join encoded in FILTER condition (Q5a) and explicit (Q5b) variant of joins. While on explicit join (Q5b) both engines performs similarly, on implicit join (Q5a) Bobox outperforms Sesame since it is able to compute also documents with 250k and 1M triples before the 30 minute limit is reached. This is achieved by creating bushy execution plan (thanks to

 ISBN: 978-1-61208-175-5

	Q1	Q2	Q3a	Q3b	Q3c	Q4	Q5a/b	Q6	Q7	Q8	Q9	Q10	Q11
10k	1	147	846	9	0	23.2k	155	229	0	184	4	166	10
50k	1	965	3.6k	25	0	104.7k	1.1k	1.8k	2	264	4	307	10
250k	1	6.2k	15.9k	127	0	542.8k	6.9k	12.1k	62	332	4	452	10
1M	1	32.8k	52.7k	379	0	2.6M	35.2k	62.8k	292	400	4	572	10
5M	1	248.7k	192.4k	1.3k	0	18.4M	210.7k	417.6k	1.2k	493	4	656	10

Table I
QUERY RESULT SIZES ON DOCUMENTS UP TO 5M TRIPLES.

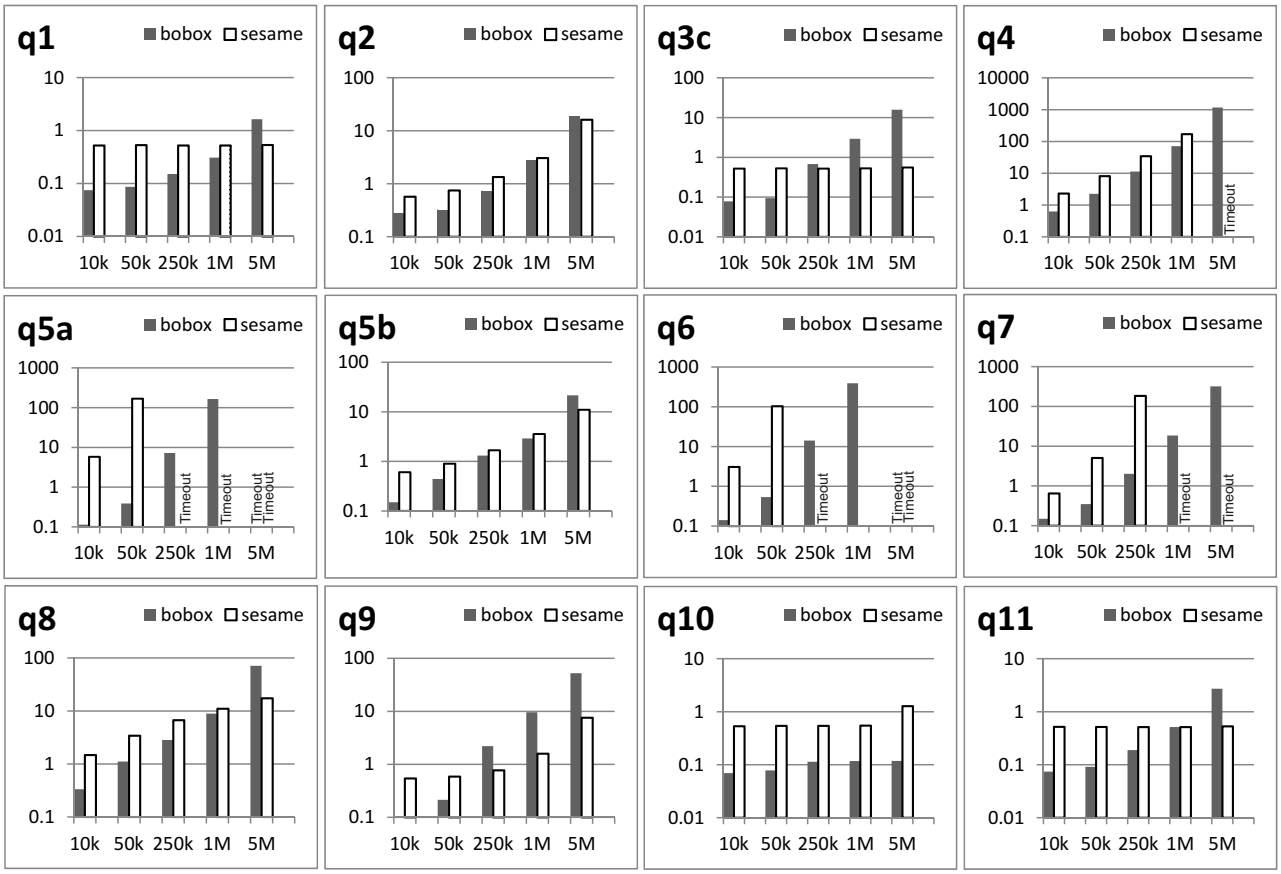

Figure 4. Results (time in seconds) for 10k, 50k, 250k, 1M, and 5M triples.

the rule of minimizing the number Carthesian products) whose execution scales well when executed in parallel. Also, incorporating FILTER operation into the final join, which would otherwise create a Carthesian product, reduces intermediate data size and speeds up query evaluation.

Queries Q6, Q7 and Q8 enable us to create bushy trees, so their computation is well handled in parallel. As a result of this, Bobox outperforms Sesame in Q6 and Q7, being able to compute larger documents until the query times out. The authors of the SP^{2}Bench suggest reusing graph patterns in description of the queries Q6, Q7 and Q8 [5]. However, this is problematical in Bobox. Bobox processing is driven by the availability of the data on inputs but it also incorporates methods to prevent the input buffers from being overfilled. Pattern reusing can result in the same data being sent along two different paths in the pipeline running at a different speed. Such paths may then converge in a join operation. When the faster path overfills the input buffer of the join box, the computation of all boxes on paths leading to the box is suspended. As a result, data for the slower path will never be produced and will not reach the join box, which results in a deadlock. We intend to examine the possibility of introducing a buffer box, which will be able to store and provide data on request. This way, the Bobox SPARQL implementation will be able to reuse graph patterns.

Overall, results of the benchmarks indicate good potential of the Bobox framework when used as an RDF query engine. It is often comparable to the Sesame framework and in

 ISBN: 978-1-61208-175-5

```
SELECT DISTINCT ?name1 ?name2                                    Q4
WHERE { ?article1 rdf:type bench:Article.
        ?article2 rdf:type bench:Article.
        ?article1 dc:creator ?author1.
        ?author1 foaf:name ?name1.
        ?article2 dc:creator ?author2.
        ?author2 foaf:name ?name2.
        ?article1 swrc:journal ?journal.
        ?article2 swrc:journal ?journal
        FILTER (?name1<?name2) }

SELECT DISTINCT ?person ?name                                    Q5a
WHERE { ?article rdf:type bench:Article.
        ?article dc:creator ?person.
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?person2.
        ?person foaf:name ?name.
        ?person2 foaf:name ?name2
        FILTER(?name=?name2) }
```

Figure 5. Examples of the benchmark queries.

some benchmarks it was able to process larger documents and/or outperform it. However, there are still some scenarios, in which Sesame performs better and we are working to improve our implementation to handle these cases better.

VI. Conclusion and Future Work

In the paper, we presented a parallel SPARQL processing engine that was built using the Bobox parallelization framework. Our main focus was on efficient query processing: parsing, optimization, transformation and parallel execution. To store the data, we implemented a simple in-memory triple store. To test performance of our pilot implementation, we performed multiple experiments. We have chosen an established framework for RDF data processing Sesame as the reference system.

The results seem very promising; using SP^{2}Bench queries we have identified that on simple queries we are in most cases comparable to Sesame. For more complicated queries like Q4, Q5, Q6 or Q7 we are able to process larger documents than Sesame. These queries let us produce richer execution plans; we are able to incorporate FILTER expressions into computation better and together with the use of fast merge joins their execution in parallel gives better performance. However, we also detected some bottle-necks. Our heuristics sometimes result in long chains but streamed processing and fast merge joins minimize this disadvantage. Also, some proposed methods, such as graph pattern reuse are not applicable in our system. During the benchmarking we also discovered some new ideas of how to increase performance of generated plans by query modification and also better use of statistics. We are, therefore, convinced that there is still space for optimization in RDF processing.

We proved that the parallel approach to RDF data processing using the Bobox framework has potential to provide better performance than current serial engines.

Acknowledgments

The authors would like to thank the GAUK project no. 28910, 277911 and SVV-2010-261312, and GACR project no. 202/10/0761, which supported this paper.

References

[1] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, 2008.

[2] D. Bednarek, J. Dokulil, and J. Yaghob, "Bobox: Parallelization framework for data processing," 2011, iET Software - submitted for publishing.

[3] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "The bobox project - parallelization framework and server for data processing," Charles University in Prague, Technical Report 2011/1, 2011.

[4] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel, "An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario," in *ISWC*, Karlsruhe, 2008, pp. 82–97.

[5] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Sp2bench: A sparql performance benchmark," *CoRR*, vol. abs/0806.4627, 2008.

[6] *OpenMP Application Program Interface, Version 3.0*, OpenMP Architecture Review Board, May 2008, http://www.openmp.org/mp-documents/spec30.pdf, retrieved 9/2011.

[7] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, no. 04, pp. 309–322, November 2007.

[8] *MPI: A Message-Passing Interface Standard, Version 2.2*, Message Passing Interface Forum, September 2009, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, retrieved 9/2011.

[9] Z. Falt and J. Yaghob, "Task scheduling in data stream processing," in *Proceedings of the Dateso 2011 Workshop*, 2011, pp. 85–96.

[10] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in starburst," *SIGMOD Rec.*, vol. 21, pp. 39–48, June 1992.

[11] O. Hartig and R. Heese, "The SPARQL Query Graph Model for query optimization," in *The Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science, E. Franconi, M. Kifer, and W. May, Eds. Springer Berlin / Heidelberg, 2007, vol. 4519, pp. 564–578.

[12] M. Cermak, J. Dokulil, and F. Zavoral, "Sparql compiler for bobox," *Fourth International Conference on Advances in Semantic Processing*, pp. 100–105, 2010.

[13] J. Broekstra, A. Kampman, and F. v. Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema," in *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*. London, UK: Springer-Verlag, 2002, pp. 54–68.

 ISBN: 978-1-61208-175-5