On Parallel Evaluation of SPARQL Queries

Zbyněk Falt, David Bednárek, Miroslav Čermák and Filip Zavoral
Department of Software Engineering
Charles University in Prague, Czech Republic
{*falt, bednarek, cermak, zavoral*}*@ksi.mff.cuni.cz*

***Abstract*—The Semantic Web databases are growing bigger and much effort is given to introduce new approaches to querying them effectively. Most of the current approaches are based on query optimization or their parallel or distributed run. However, they do not fully benefit from potential of the modern, multicore computers in parallel processing. Although the parallel relational database systems have been well examined, parallel query computing in Semantic Web databases has not been extensively studied. This paper follows our previous research in parallelization of evaluation of SPARQL queries by outlining the possibility of further parallelization of query operators themselves. As a result, we developed a parallel SPARQL query execution engine built over Bobox framework. We show that intraoperation parallelism yields to better performance.**

***Keywords*-parallel, SPARQL, Bobox**

I. Introduction

As prevalence of semantic data on the web is getting bigger, the Semantic Web databases are growing in size. There are two main approaches to storing and accessing these data efficiently: using traditional relational means or using sematic tools, such as different RDF triplestores [1] and SPARQL [2] language. Since semantic tools are still in development, a lot of effort is given to research of effective storing of RDF data and their querying [3]. One way of improving performance is the use of modern, multicore CPUs in parallel processing. Nowadays, there are several database engines which are capable of evaluating SPARQL queries, such as SESAME [4], OWLIM [5] or RDF-3X [6], which is currently considered to be one of the fastest single node RDF-store [7]. These stores support parallel computation of multiple queries; however, they do not use the potential of parallel computation of query itself, in contrary to the work [8]. It introduces implementation of RDF-store based on RDF-3X with parallelized join operators, but not full set of operators.

In our previous work, we chose the semantic approach and presented parallel SPARQL engine [9], that provides streamed parallel query execution on basic operation level. While the implementation of operations is based on sequential algorithms, there is considerable research on parallel versions, such as parallel joins [10], [11], [12]. Moreover, new architectures have been explored to improve performance. Gedik et al. [13] adapts join operation for the Cell processors. He et al. [14] uses GPUs for the processing of joins. Both papers try to exploit parallel nature of these architectures and show performance benefits over optimized CPU-based counterparts.

The SPARQL algebra is similar to the relational algebra; however, there are several important differences, such as absence of NULL values. As a result of these differences, the application of relational knowledge into semantic is not straightforward and the algorithms have to be adapted so it is possible to use them.

In this paper, we explore the effects of further decomposition of basic operations in our SPARQL engine in order to increase parallelism during the query evaluation. In pilot implementation, we focused on following basic operations: nested loops join, index scan and filter. This enables evaluation of some queries completely in parallel and reaching better scalability.

The rest of the text is organized as follows: In Section II, we present background information about Bobox framework we used. It also contains more detailed description of some of its parts, since it is important for understanding of the rest of the paper. Section III introduces two main concepts of parallelization of operation in Bobox. In Section IV, we describe parallelization of selected operations. The results of our experiments are introduced in Section V. Section VI concludes the paper and presents our future work.

II. Background

The platform we used to build our execution engine is Bobox [15]. Bobox is a parallel framework, which was designed to support development of data-intensive parallel computations. The main idea behind Bobox is connecting a large number of relatively simple computational components into a nonlinear pipeline. This pipeline is then executed in parallel, but the interface used by the computational components is designed in such way that they do not need to be concerned with the parallel execution issues such as scheduling, synchronization and race conditions.

This system may be easily used for database query evaluation. Since Bobox supports only a custom low-level interface for the definition of the structure of the pipeline, a separate query compiler and optimizer has to be created for required query language. Firstly, we developed SPARQL compiler for Bobox [16]. Its main task is to perform lexical, syntactic and semantic validation of the given query, to

 ISBN: 978-1-61208-185-4

perform static optimizations and to build optimal query execution plan using heuristics to reduce search space and statistics collected over stored data.

Traditionally (e.g. in relational databases), query execution plans have the form of directed rooted trees in which the edges indicate the flow of the data and all of them are directed to the root. The nodes of the tree are the basic operations used by the evaluation engine, such as full table scan, indexed access, merge join, filter etc. This plan corresponds to Bobox architecture, since the tree is a special case of the nonlinear pipeline supported by the system.

A. Evaluation of the plan in the Bobox

After the Bobox receives a plan for evaluation, it replaces the operations in the plan by the boxes, which are elementary execution units in Bobox, and connects them according to edges in the execution plan.

Each box is able to receive data, process them and send resulting data out to the boxes which are connected to its output. Data are processed by small parts, which, in Bobox, are called envelopes. The envelope contains a list of columns which contain the data. The columns may have arbitrary data types, but all columns have always the same number of data elements. Therefore, from other point of view, the envelopes are also lists of rows where each element of the row may have arbitrary data type. Additionally, each envelope may contain also some scalar data (for example integer or boolean value) which may be used for additional communication between the boxes.

The processing of envelopes inside the box is always single-threaded, because boxes are not allowed to create another threads. There are two reasons for this limitation:

- The development of the boxes is easier, since a developer does not have to take any parallelization into account.
- If the boxes were allowed to create its own threads, the total number of threads in the system would be out of control and may easily exceed the number of physical threads in the computer by several order of magnitude. This may cause significant slowdown of the execution.

The evaluation of the execution plan works as follows: when the box receives an envelope, the envelope is stored into its (limited) input buffer and the box is scheduled. When the scheduler, which is an important part of Bobox, decides to run the box, the box is executed and one envelope of its input buffer is processed. If the buffer is not empty yet, it is scheduled again. When the input buffer of the box is full, the preceding box which causes this state is paused as long as the buffer is full.

Since boxes are independent on each other and their state is determined only by the data they received, it is possible to run all boxes which have at least one envelope in the input buffer in parallel. This enables automatic parallelization of the plan evaluation.

Bobox also determines the size of envelopes according to hardware parameters of the computer. The optimal size of one envelope is chosen to be $\frac{1}{2}L2/N$ bytes, where L2 denotes the total size of L2 cache in the system and N the number of physical threads [17]. According to our experiments, this value gets the best results, since all envelopes which are being processed at a moment may be completely stored in L2 cache and there is still space for auxiliary data needed for the execution.

III. Basic concepts of parallelization of operations

The straightforward approach to the implementation of boxes is that each operation in the execution plan is represented by one box. Despite the fact that execution of one box is strictly single-threaded, there is space for parallelization. The main reasons are:

- The execution plan has typically a form of rooted tree, therefore, its branches may be executed in parallel.
- Parallel evaluation may be reached through pipeline processing, i.e. while one box is processing envelope number i, the consecutive box may be processing envelope number $i + 1$.

Because of these facts, even straightforward implementation of boxes yields to parallel evaluation and causes indispensable speedup of the plan evaluation. Unfortunately, it is still usually insufficient to utilize all physical threads in the computer.

In order to enhance parallelization, the operations should be implemented by more boxes than one. In this case, the number of threads which perform the operation is limited only by the number of boxes corresponding to the operation and by the number of physical threads in the system. Unfortunately, the decomposition of operation to multiple boxes may be difficult.

Generally, there are two types of operations:

- Stateless operations – their state does not depend on envelopes received so far, i.e. processing of one envelope is totally independent on processing of any other envelope. For example filter operation meets this condition.
- Stateful operations – processing of one envelope is influenced by the content of envelopes received before, therefore, their state depends on all data received so far.

A. Decomposition of stateless operations

Decomposition of stateless operations is quite simple. The box which performs the operation may be duplicated and each instance of the box may process only the proportional part of the data. Predecessors of these boxes should be some kind of dispatch box, which resends incoming envelopes to them. Their successor must be a box, which aggregates the resulting data together and passes them to another operation. This scheme is shown in Figure 1.

 ISBN: 978-1-61208-185-4

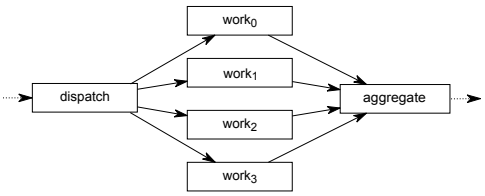

Figure 1. Decomposition of stateless operation

The `dispatch` box just forwards envelopes to the worker boxes in round-robin manner. This is very efficient, since we keep the size of envelope to be a constant. Therefore, the work boxes receive approximately the same amount of data for processing. In this case, the `aggregate` box is also simple, since it receives envelopes from work boxes in round-robin manner as well and resends them to the output.

It may seem, that `dispatch` and `aggregate` are bottlenecks of the algorithm, but they only resends the incoming envelopes, which is by several order of magnitude faster than accessing or processing their data.

B. Decomposition of stateful operations

Stateful operations are much harder to decompose, since in order to generate envelope number i, the box needs to know the state after generation the envelope number $i-1$. One possible way of dealing with this is to have two algorithms: the first algorithm P (*Processing algorithm*) performs the real operation and the second algorithm S (*Skipping algorithm*) computes only the state of the box after the performance of the operation. If the second algorithm exists and is much faster than the first, then the decomposition may follow the scheme depicted in Figure 2.

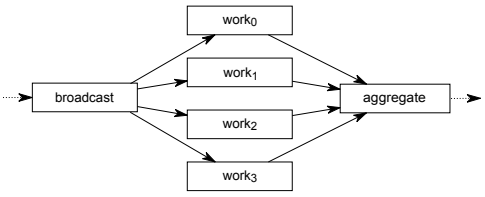

Figure 2. Decomposition of stateful operation

The `broadcast` box forwards all incoming envelopes to all work boxes. The work boxes keep a phase counter except their state. This counter is continually increasing during the evaluation and all work boxes must have these number synchronized. This may be easily done, since all work boxes receive the same input data. Boxes increase the phase counter after some event which might be reception of envelope, sending of output envelope etc. The work box number i performs algorithm P if phase counter mod N, where N is number of work boxes, is equal to i. Otherwise, it performs algorithm S which updates the internal state and also phase counter if needed.

The `aggregate` box might be more complicated, since it must know which work box is sending the next envelope. However, if the phase counter is increased on the sending of an envelope, the `aggregate` box works in the same way as for a stateless operation, i.e. it receives the data in round-robin manner. One example of this kind of decomposition is in Subsection IV-A.

IV. Implementation of SPARQL operations

Operations needed for SPARQL queries evaluation have typically one or two inputs and one output. The format of envelopes used for communication between two consecutive operations is as follows: columns correspond to variables and rows contain all allowed mappings of these variables before or after the operation depending on whether the envelope is incoming or outcoming.

In the rest of this section, we describe the decomposition of the operations, which are minium for performance of several basic experiments.

A. Decomposition of scan operation

The main objective of scan operation is to fetch from RDF database all triples that matches the input pattern. We keep six indexes to the database, which are simply the list of indexes of all triples sorted in all possible order (SPO, SOP, OPS, OSP, POS and PSO). Therefore, it is easy for any input pattern to find this range in corresponding index where all triples which match the pattern are. To find this range, we use binary search. Each work box may find this range independently. After that, the boxes start to send envelopes with requested data in round-robin manner. This is an example of a very simple stateful operation, since the only state of the box is position of the triple which was sent for the last time. Algorithm S is very simple, because it only adds value $C * (N-1)$ to the position where C is number of triples in one envelope and N is number of work boxes.

B. Decomposition of filter operation

Filter operation is a typical stateless operation, therefore, we may use the scheme for decomposition of stateless operation. Unfortunately, this straightforward approach has one drawback – the output envelope contains typically less rows than the input envelope. However, it is ineffective to work with half-empty envelopes since the manipulation with them brings some overhead such as box scheduling etc. If the envelopes are too small, this overhead may be higher than the useful work and it may yield to significantly slow down of evaluation. Therefore, we need to defragment the envelopes in order that the output envelopes of the filter operation have the optimal size.

 ISBN: 978-1-61208-185-4

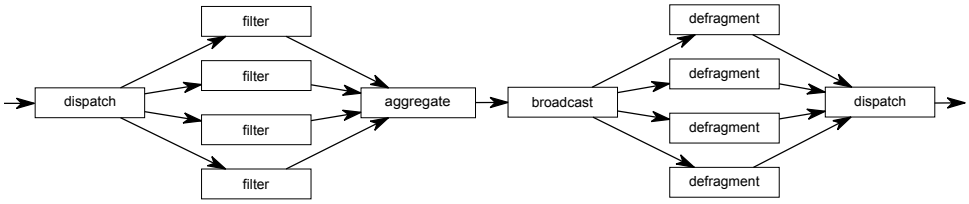

Figure 3. Decomposition of filter operation

The solution we chose is to create boxes which join the output envelopes of the filter boxes and copy their data into the envelopes with optimal size. The output envelopes from filter boxes are aggregated and broadcasted to the stateful defragment boxes. These boxes follow the scheme for stateful operation. The phase counter is increased when an output envelope is sent. Algorithm P is easy, because it only copies some subset of incoming data into the output envelope. Algorithm S is also simple, because the box knows the number of rows in incoming envelopes. According to this number, the algorithm decides to either skip the incoming envelope or process it.

Output envelopes from the defragment boxes are simply aggregated and sent to consecutive operation. The complete scheme is shown in Figure 3. On the other hand, the defragmentation may bring some slow down when the selectivity of the filter is very high. In this case, the defragmentation may become a blocking operation, which receives data, but does not send them into the output. The rest of the execution plan must wait at the worst case until the all data are filtered. The optimal solution of this problem would probably require some hints from the compilator, which we plan in future.

C. Decomposition of nested loops join

Nested loops join operation has two inputs – we denote them as left and right. The join tries all combination of one row from the left input and one row from the right input. If the combination is compatible, i.e. selected variables in the left row are compatible with selected variables in the right row, it evaluates given filter condition on the combination. If the condition is true, then this combination should be send to the output.

Because the input data are received in envelopes, we may modify the algorithm as follows: The operation tries all combination of left and right envelopes and for each such combination it tries all rows from left and right envelope. Since the output of the nested loops join may be in arbitrarily order, both these algorithms are equivalent.

A simple approach to implementation of the algorithm is an use of N work boxes. Work box number i is responsible for processing all, for example left, envelopes of which sequence number mod N is equal to i, i.e. the box generates all combinations of left envelopes it is responsible for and all right envelopes.

This approach, however, may lead to the situation, in which one box processes all data and the others do nothing. If the left input consists of very few envelopes and the opposite input of a large number of envelopes, then redundant work boxes do nothing and the employed boxes become bottleneck of the system.

To avoid this situation, the combinations of envelopes must be processed uniformly by the work boxes. Therefore, we numbered all combinations of left and right envelopes by ordinal numbers as shown in Table I. This numbering does not prefer neither left nor right input. The corresponding formula is $(L + R) * (L + R + 1)/2 + R$ where L is the number of the left envelope and R is the number of the right envelope. The work box number i processes combination only if its number mod N equals to i which causes that boxes are utilized uniformly.

Table I
NUMBERING OF PAIRS OF ENVELOPES

left \ right	0	1	2	3
0	0	2	5	9
1	1	4	8	
2	3	7		
3	6			

The nested loops utilize memory bus in a great deal because all pairs of rows must be tested and therefore fetched from the main memory. This slows down the evaluation, especially when the nested loops join is computed on multiple thread since the memory becomes the bottleneck of the system. Fortunately, the modified algorithm prevents this situation. The size of envelopes is optimal according to cache memory. When the box generates all combinations of rows from one envelope with another, the data of one envelope are kept hot in cache while the data of the other are read sequentially from the main memory, and sequential access to memory is cache optimal.

D. Optional nested loops join

Optional nested loops join works in two phases. In the first phase, it does the same operation as nested loops join. In the second phase, it sends to the output all rows from the left input which were not send to the output so far.

 ISBN: 978-1-61208-185-4

In order to implement the second phase, boxes need to know which rows from the left input were already sent and which not. To fulfil this requirement, we used the fact that the data in one envelope are shared among all boxes which received the envelope. Therefore, if we add one column of boolean variables to envelopes from the left input before their are broadcasted to work boxes, they can easily mark all left rows which were sent to the output. When the boxes finishes the first phase, they know which rows from the left should be sent to the output in the second phase.

V. EXPERIMENTS

To measure the influence of the decomposition of selected operations, we selected three queries from SP^{2}Bench benchmark [18] since they may be evaluated completely with the new decomposed operation. We chose this benchmark, since this is considered to be standard in the area of semantic processing. We used a computer with two Intel Xeon E5310, which run at 1,60GHz with 6MB shared L2 cache. Each processor has 4 cores, thus the system has 8 physical threads. The size of operating memory is 8 GB. The operating system is Red Hat Enterprise Linux Server 6.1 and we used g++ 4.4.5 with -O2 switch. The database was in-memory.

For each query, we performed four different tests to measure scalability of our solution. The tests differ in the number of parallel work boxes we used in decomposition of operations. We used 1,2,4 and 8 parallel boxes. Important fact is, that the first test (with 1 work box) is the same as if no decomposition was performed. Additionally, each test was evaluated twice – single-threaded and multithreaded, i.e. we used each physical thread in the system for the query evaluation.

We performed each experiment 5 times and we selected the median of all measurement. Since the database loading and query compilation time is not relevant for the experiments the results do not include them. On ther other hand, they include time spent by the operation decomposition, since such overhead is an integral part of the query execution.

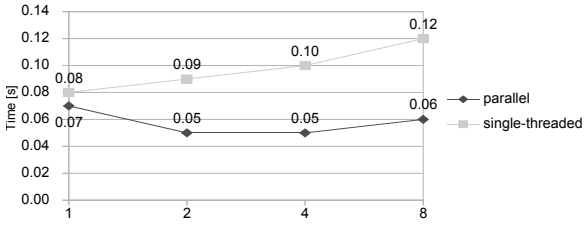

Figure 4. Results of query q1

The first query we used is the query q1 on a database with 5M triples. This query uses only index scan and nested loops join operation. The results are show in Figure 4. The second query is q3a on a database with 250k triples which utilizes both nested loops join and filter operation (Figure 5) and the last is query q6 on the same database as the query q3a which additionally uses optional nested loops joins (Figure 6).

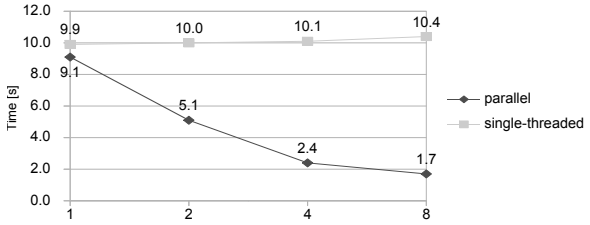

Figure 5. Results of query q3a

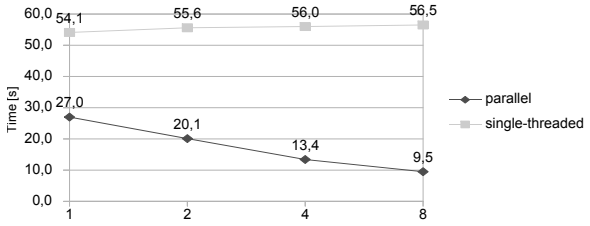

Figure 6. Results of query q6

The query q6 benefits from multithreading even in the case when no decomposition is performed. This is caused due to the pipeline processing and the fact that there are independent braches in the plan. Additional experiments show that increasing the number of work boxes in the decomposition increases the performance.

The query q3a benefits only a little from parallel environment when there is no decomposition. The execution plan contains one filter operation followed by the nested loops join and the small speed up is caused by the fact that filter operation is much faster operation then the nested loops join. Therefore, the evaluation of the filter operation in parallel with the join operation is not so profitable. However, the decomposition of the operations, mainly of the nested loops join, increases the performance noticeably.

The query q1 is a little bit atypical. The graph contains two consecutive nested loops joins, but the input data are very small. In fact, they fit in one or two envelopes. Therefore, the decomposition of problem does not cause significant speed up, since the whole operation is performed by only a subset of work boxes.

In each test, the single-threaded evaluation shows that with the increasing number of work boxes, the overhead caused by the manipulation of envelopes is slightly increasing.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented possibility of the decomposition of basic SPARQL operations to increase parallelism during the evaluation of execution plans when evaluated by Bobox. In our pilot implementation, we focused on subset

 ISBN: 978-1-61208-185-4

of basic operations - nested loops join, filter and index scan. These operations were sufficient to measure the influence of the decomposition on a query evaluation performance.

The tests were performed using SP^{2}Bench test queries that used implemented operations only. As follows from presented results, further decomposition of basic SPARQL operations can provide additional performance gain – for 8 physical threads in the system, the evaluation of time consuming queries is almost 6 times faster than single-threaded execution, therefore, our solution scales well.

We do not compare these results with other RDF stores, since the execution plans were not optimal because we forced the compiler to use only nested loops join instead of merge joins, with which the evaluation would be much faster. The comparison between version with optimal but not decomposed plans and the SESAME database may be found in our previous work [9]. On the other hand, for the database sizes, for which we performed the tests, the evaluation of the queries with decomposed operations is faster than the evaluation without the decomposition but with optimal operations.

In the future, we want to focus on paralellizing extended set of basic SPARQL operations such as sorting, union or faster join operations such as hash join, or merge join. After that, we will be able to perform more extensive and accurate performance tests and comparisons of parallel version with similar semantic engines.

ACKNOWLEDGMENT

The authors would like to thank the GAUK project no. 28910, 277911 and SVV-2011-263312, and GACR project no. 202/10/0761, which supported this paper.

REFERENCES

[1] J. J. Carroll and G. Klyne, *Resource Description Framework: Concepts and Abstract Syntax*, W3C, 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[2] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF*, W3C, 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

[3] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan, "Efficiently querying rdf data in triple stores," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 1053–1054.

[4] 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.

[5] A. Kiryakov, D. Ognyanov, and D. Manov, "Owlim a pragmatic semantic repository for owl," 2005, pp. 182–192.

[6] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, pp. 91–113, February 2010.

[7] J. Huang, D. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, 2011.

[8] J. Groppe and S. Groppe, "Parallelizing join computations of sparql queries for large semantic web databases," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 1681–1686.

[9] M. Cermak, J. Dokulil, Z. Falt, and F. Zavoral, "SPARQL Query Processing Using Bobox Framework," in *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*. IARIA, 2011, pp. 104–109.

[10] A. Brown and C. Kozyrakis, "Parallelizing the index-nested-loops database join primitive on a shared-nothing cluster," 1998.

[11] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, pp. 1378–1389, August 2009.

[12] H. Lu, K.-L. Tan, and M.-C. Shan, "Hash-based join algorithms for multiprocessor computers," in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 198–209.

[13] B. Gedik, P. S. Yu, and R. R. Bordawekar, "Executing stream joins on the cell processor," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 363–374.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 511–524.

[15] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Using Methods of Parallel Semi-structured Data Processing for Semantic Web," in *3rd International Conference on Advances in Semantic Processing, SEMAPRO*. IEEE Computer Society Press, 2009, pp. 44–49.

[16] M. Cermak, J. Dokulil, and F. Zavoral, "SPARQL Compiler for Bobox," in *SEMAPRO 2010, The Fourth International Conference on Advances in Semantic Processing*. IARIA, 2010, pp. 100–105.

[17] Z. Falt and J. Yaghob, "Task Scheduling in Data Stream Processing," in *Proceedings of the Dateso 2011 Workshop*. Citeseer, 2011, pp. 85–96.

[18] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL performance benchmark," *CoRR*, vol. abs/0806.4627, 2008.

 ISBN: 978-1-61208-185-4