Parallel SPARQL Query Processing Using Bobox

Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, and Filip Zavoral
Charles University in Prague, Czech Republic
{falt,cermak,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 compiler and execution plan optimizer, which were tailored specifically to work with the Bobox parallel framework. The experiments described in this paper show that such a parallel approach to RDF data processing has a potential to provide better performance than current serial engines.**

Keywords*-*SPARQL; Bobox; query optimization; parallel.

I. Introduction

SPARQL [2] is a query language for RDF [3] (Resource Definition Framework) widely used in semantic web databases. It contains capabilities for querying graph patterns along with their conjunctions and disjunctions. SPARQL algebra is similar to relational algebra; however, there are several important differences, such as the absence of NULL values. As a result of these differences, the application of relational algebra into semantic processing is not straightforward and the algorithms have to be adapted so it is possible to use them.

As the 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 semantic tools, such as different RDF triplestores [3] accessed using SPARQL. Semantic tools are still in development and a lot of effort is given to the research of effective storing of RDF data and their querying [4]. 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 [5], JENA [6], Virtuoso [7], OWLIM [8] or RDF-3X [9], that is currently considered to be one of the fastest single node RDF-store [10]. These stores support parallel computation of multiple queries; however, they mostly do not use the potential of parallel computation of particular queries.

The Bobox framework [11], [12], [13] was designed to support the development of data-intensive parallel computations. The main idea behind Bobox is to divide a large task into many simple tasks that can be arranged into a non-linear pipeline. The tasks are executed in parallel and the execution is driven by the availability of data on their inputs. The developer does not have to be concerned about problems such as synchronization, scheduling and race conditions. All this is done by the framework. The system can be easily 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 tool for efficient parallel querying of RDF data [14] using SPARQL build on top of the Bobox framework [1], [15]. The data are stored using an in-memory triple store. We provide a description of query processing using SPARQL-specific parts of the Bobox and provide results of benchmarks. Benchmarks were performed using the SP^{2}Bench [16] query set and data generator.

The rest of the paper is structured as follows: Section II describes the Bobox framework. Models used to represent queries and a description of query processing is contained in Section III. Data representation and the implementation of operators using Bobox framework is described in Section IV. Section V presents our experiments and a discussion of their results. Section VI compares our solution to other contemporary parallelization frameworks. Section VII describes future research directions and concludes the paper.

II. Bobox Framework

A. Bobox Architecture

Bobox is a parallelization framework which simplifies writing parallel, data intensive programs and serves as a testbed for the development of generic and especially data-oriented parallel algorithms.

Bobox provides a run-time environment which is used to execute a non-linear pipeline (we denote it as the *execution plan*) in parallel. The execution plan consists of computational units (we denote them as the *boxes*) which are connected together by directed edges. The task of each box is to receive data from its incoming edges (i.e. from its *inputs*) and to send the resulting data to its outgoing edges (i.e. to its *outputs*). The user provides the execution plan (i.e. the implementation of boxes and their mutual connections) and passes it to the framework which is responsible for the evaluation of the plan.

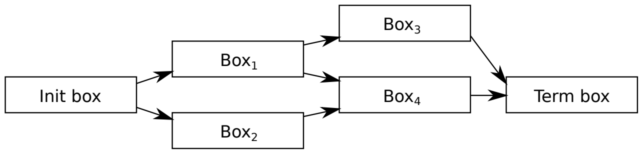

Figure 1. Example of an execution plan

Figure 1 shows an example of an execution plan. Each plan must contain two special boxes:

- *init box* – this is the first box (in a topological order) of the plan which is executed.
- *term box* – this is the last box and denotes that the execution plan was completely evaluated.

The implementation of boxes is quite straightforward and simple, since Bobox provides a very powerful and easy to use interface for their development. Additionally, the source code is expected to be strictly single-threaded. Therefore, the developer does not have to be familiar with parallel programming. Although this requirement on the source code may seam limiting, the framework is especially targeted to a development of highly scalable applications [17].

The only communication between boxes is done by sending *envelopes* (communication units containing data) along their outgoing edges. Each envelope consists of several columns and each column contains a certain number of data items. The data type of items in one column must be the same in all envelopes transferred along one particular edge; however, different columns in one envelope may have different data types. The data types of these columns are defined by the execution plan.

The number of data items in all columns in one envelope must be always the same. Therefore, we may define the list of i-th items of all columns in one envelope as its i-th *data line*. The Figure 2 shows an example of an envelope.

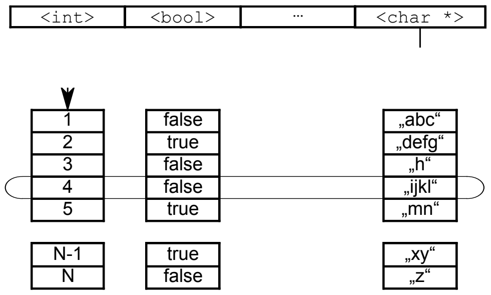

Figure 2. The structure of an envelope

The total number of data lines in an envelope is chosen according to the size of cache memories in the system. Therefore, the communication may take place completely in cache memory. This increases the efficiency of processing of incoming envelopes by a box.

Currently, only shared-memory architectures are supported; therefore, the only shared pointers to the envelopes are transferred. This speeds up operations such as broadcast box (i.e., the box which resends its input to its outputs) significantly since they do not have to access data stored in envelopes.

There is one special envelope (so called *poisoned pill*) which is sent after the last regular envelope to close the output of a source box. For the receiver of the poisoned pill it is a signal that all data were already received on that input.

In fact, the only work which is done by the init box is sending the poisoned pill to its output and the only responsibility of the term box is to terminate the evaluation of the execution plan when it receives the poisoned pill on its input.

The interface of Bobox for box development is very flexible; therefore, the developer of a box may choose between multiple views on the data communication:

- The communication is a stream of envelopes. This is useful for efficient implementation of boxes which do not have to access data in envelopes such as broadcast box or stream splitter (see Section IV-D) or implementation of boxes which process their inputs by envelopes.
- The communication is a stream of data lines. This is useful for easier implementation of boxes which manipulate with data lines one by one such as filter box (see Section IV-C2).
- The combination of both views. For example, the sort box (see IV-C3) processes input by envelopes, but produces the output as a stream of data lines.

Although the body of boxes must be strictly single-threaded, Bobox may introduce three types of parallelism:

1) Task parallelism, when independent streams are processed in parallel.
2) Pipeline parallelism, when the producer of a stream runs in parallel with its consumer.
3) Data parallelism, when independent parts of one streams are processed in parallel.

The first two types of parallelism are exploited implicitly during the evaluation of a plan. Therefore, even an application which does not contain any explicit parallelism may benefit from multiple processors in the system (see Section V-A). Data parallelism must be explicitly stated in the execution plan by the user (see IV-D); however, it is still much easier to modify the execution plan than writing parallel code by hand.

B. Flow control

Each box has only limited buffer for incoming envelopes. When this buffer becomes full, the producer of the envelopes is suspended until at least one envelope from the buffer is processed. This strategy increases the performance of the system since the operators which produce data faster than their consumers are able to process are suspended to not to consume the CPU time uselessly. This time may be used to execute other boxes. Additionally, this method yields to lower memory consumption, since there is only a limited number of unprocessed slots which occupy the memory at a time.

On the other hand, this flow control may sometimes yield to a deadlock (see Section V-C) or may limit the level of parallelism (see Section IV-C6 for an example).

C. Box scheduling

Scheduling of boxes is a very important factor which significantly influences the performance of Bobox. The scheduling strategies are described in a more detail in [12].

During the initialization of Bobox, a same number of worker threads as the number of physical processors is created. Only these worker threads may execute the code of boxes. The scheduler has two main data structures:

- Each worker thread has its own double ended queue of *immediate tasks*.
- Each execution plan which is being evaluated has its own queue of *deferred tasks*.

There are three cases when a box is scheduled:

- When a new execution plan is about to evaluate, a new queue of deferred tasks for that plan is created and its init box is put to the front of that queue.
- When a box sends an envelope to another box, the destination box is put to the front of the queue of immediate tasks of the thread which is executing the source box.
- When a box stops to be suspended because of flow control, it is put to the queue of deferred tasks of the corresponding plan.

When the working thread is ready to execute a box, it choose the first existing box in this order:

1) The newest box in its queue of immediate tasks. This box receives an envelope created by this thread recently. Therefore, it is probable that this envelope is completely hot in a cache so accessing its data is probably much faster than accessing other envelopes.
2) The oldest box in the queue of deferred tasks of the oldest execution plan. This ensures that scheduling of deferred tasks of one execution plan are scheduled fairly. However, the execution plans are prioritized according to their age – the older the execution plan is, the higher priority it has. Each evaluation of an execution plan needs some resources (such as memory for envelopes); therefore, the more plans are being evaluated at a time, the more resources are needed for them. This strategy ensures that if there is a box to execute from plans which are currently being executed, no new evaluation is started.
3) The oldest box in the queue of immediate tasks of another worker thread. Worker threads with shared cache memory are prioritized. This avoids suspending of a worker thread despite the fact that there are boxes to execute. Moreover, the oldest box has the lowest probability to have its input hot in a cache memory of the thread from which the box was stolen. Therefore, *stealing* this box should introduce less performance penalty than stealing the newest box in the same queue.

If there is no box to execute, the worker thread is suspended until some other box is scheduled.

Besides the SPARQL compiler described in this paper, the Bobox framework is used in several related projects - model visualization [18], semantic processing [19], [20], query optimization [21], and scheduling in data stream processing [12], [22].

III. Query Representation and Processing

One of the first Bobox applications was SPARQL query evaluator [19]. Since running queries in Bobox needs an appropriate execution plans, SPARQL compiler for Bobox was implemented to generate them from the SPARQL code.

During query processing, the SPARQL compiler uses specialized representation of the query. In the following sections, we mention models used during query rewriting and generation of execution plan.

A. Query Models

Pirahesh et al. [23] proposed the Query Graph Model (QGM) to represent SQL queries. Hartig and Reese [24] modified this model to represent SPARQL queries (SQGM). With appropriate definition of the operations, this model can be easily transformed into a Bobox pipeline definition, so it was an ideal candidate to use.

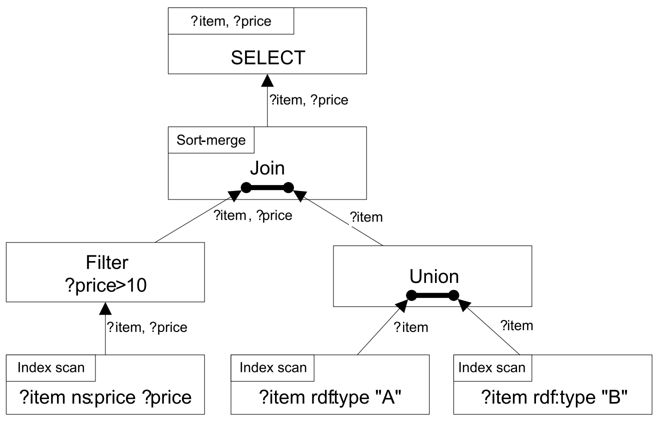

Figure 3. Example of SQGM model.

SQGM model can be interpreted as a directed graph (a directed tree in our case). 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 3 shows an example of a simple query represented in the SQGM model. This model is created during an execution plan generation and is used as a definition for the Bobox pipeline.

In [25], 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 [2] 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 4.

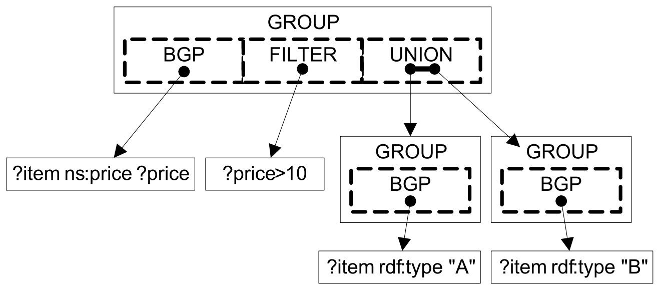

Figure 4. Example of SQGPM model.

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.

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.

B. Query Processing

Query processing is performed in a few steps by separate modules of the application as shown in Figure 5. The 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 generated 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.

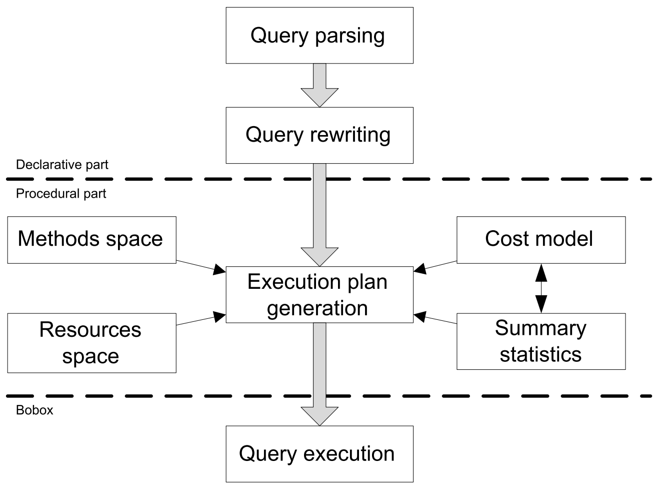

Figure 5. Query processing scheme.

C. Query Parsing and Rewriting

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

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. Therefore, 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

During this step, it is necessary to check applicability of each operation with regards to the SPARQL semantics before it is used to preserve query equivalency [25].

D. Execution Plan Generation

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

The query execution plan (e.g., the execution plan of query q5a is depicted in Figure 6) 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. The list of available atomic operations (e.g., the different types of joins) and their properties are provided by the *Methods Space* module.

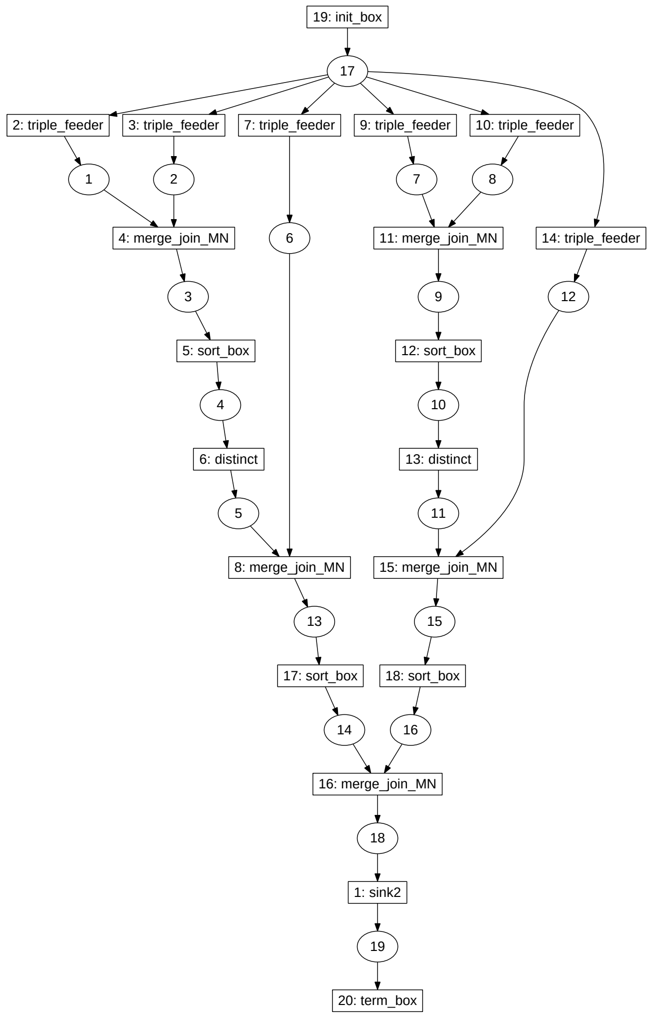

Figure 6. Query execution plan q5a.

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 using 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.

The search space of all execution plans could be extremely large; we used heuristics to reduce the complexity of the search. Only left-deep trees of join operations are considered. This means that right operand of a 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 unused operations is used to build separate tree recursively (using the same algorithm) and the 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.

The final execution plan is represented using SQGM model which is serialized into a textual form and passed to the Bobox framework for evaluation.

IV. Evaluation of SPAQRL queries using Bobox

When the compiler finishes the compilation, a query execution plan is generated. This plan must be transformed into a Bobox execution plan and then passed to Bobox for its evaluation. This basically means that operators must be replaced by boxes and they should be connected to form a pipe. Additionally, an efficient representation of data exchanged by boxes must be chosen to process the query efficiently.

A. Data representation

1) Representation of RDF terms: RDF data are typically very redundant, since they contain many duplicities. Many triples typically share the same subjects or predicates. To reduce the number of memory needed for storing the RDF data, we keep only one instance of every unique string and only one instance of every unique term in a memory. Besides the fact that this representation saves the memory, we may represent each term unambiguously by its address. Therefore, for example in case of a join operation, we can test equality of two terms just by a comparison of their addresses.

Additionally, if we need to access the content of a term (e.g. for evaluation of a filter condition) the address can be easily dereferenced. This is faster than the representation of terms by other unique identifiers which would have to be translated to the term in a more complicated way.

2) Representation of RDF database: The database consists of a set of triples. We represent this set as three parallel arrays with the same size which contain addresses of terms in the database. In fact, we keep six copies of these arrays sorted in all possible orders – SPO, SOP, OPS, OSP, PSO and POS. This representation makes implementation of index scans extremely efficient (see IV-C1).

3) Format of envelopes: The format of envelopes is now obvious. It contains columns which correspond to a subset of variables in the query in a form of an address of a particular RDF term. One data line of an envelope corresponds to one possible mapping of variables to their values.

B. Transformation of query execution plan

The output of the compiler is produced completely in a textual form. Therefore, the Bobox must deserialize the query plan first. Despite the fact that this serialization and deserialization have some overhead, we chose it because of these benefits:

- When distributed computation support is added, the text representation is safer than a binary representation where problems with different formats, encodings or reference types may appear.
- The serialization language has a very simple and effective syntax; serialization and deserialization are much faster than (e.g.) the use of XML. Therefore, the overhead is not so significant.
- The 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 earlier testing of the compiler during the development process.
- The query plan may be easily visualized to check the correctness of the compiler. Moreover, the plan might be written by hand which makes the testing of boxes easier. Altogether, this enables debugging of a compiler and Bobox independently on each other.

When the plan is deserialized, the operators in the query execution plan must be replaced by boxes and connected together. The straightforward approach is that each operator in the query execution plan is implemented by exactly one box. Even this approach yields to a parallel evaluation of the plan since pipeline parallelism and task parallelism might be exploited (the query plan has typically a form of a rooted tree with several independent branches). However, it is still usually insufficient to utilize all physical threads available and the most time consuming operations such as nested loops join becomes a bottleneck of the plan. Therefore, they have to be parallelized explicitly. We describe this modification in IV-D.

C. Implementation of query plan operators

1) Index scan: The main objective of a scan operation is to fetch all triples from the database that match the input pattern. Since we keep all triples in all possible orders, it is easy for any input pattern to find the range where all triples which match the pattern are. To find this range, we use binary search. To avoid copying triples from the database to the envelopes, we use the fact that they are stored in parallel arrays. Therefore, we may use the appropriate subarrays directly as columns of output envelopes without data copying.

2) Filter: A filter operation can be implemented in Bobox very easily. The box reads the input as a stream of data lines, evaluates the filter condition on each line and sends out the stream of that data lines which meet the condition.

The evaluation of the filter condition is straightforward since each data line contains addresses of respective RDF terms and by dereferencing them it gets full info about the term such as its type, string/numeric value etc.

3) Sort: Sort is a blocking operation, i.e. it must wait until all input data are received before it starts to produce output data. To increase the pipeline parallelism, we implemented two phase sorting algorithm [17] inspired by external merge sort.

In the first phase, every incoming envelope is sorted independently on other envelopes. This phase is able to run in parallel with the part of an execution plan which precedes the sort box. The second phase uses a multiway merge algorithm to merge all received (and sorted) envelopes into the resulting stream of data lines. In contrary to the first phase, this phase may run in parallel with the part of the execution plan which succeeds the sort box.

4) Merge join: Merge join is a very efficient join algorithm when both inputs are sorted by the common variables. Moreover, the merge join is the algorithm which is suitable for systems like Bobox since it reads both inputs sequentially allowing both input branches to run in parallel (in contrary to hash join, see Section IV-C6).

5) Nested loops join: The SPARQL compiler selects nested loops join when the inputs have no common variable and the result is determined only by the join condition. The implementation is straightforward; however, in order to increase the pipeline parallelism, the box tries to process envelopes immediately as they arrive, i.e. it does not read the whole input before processing the other.

6) Hash join: Hash join is used when the inputs have some common variables which are not sorted in the same order. In order to increase pipeline and task parallelism, we decided not to implement this algorithm. The problem with hash join is that it must read the whole one input first before processing the second one. However, the branch of the plan which produces data for the second input may be blocked because of flow control (see Section II-B) until the first input is completely processed.

Therefore, instead of hash join we implemented sort-merge join. The sort operation is used to transform the inputs to be usable by merge join.

7) Optional joins: Optional join works basically in the same way as regular join. The only difference is that data lines from the left input which do not meet the join condition (i.e., they are not joined with any data line from the right input), are also passed to the output and the variables which come from the right input are set as unbound.

This modification can be easily done when exactly one data line from the left is joined with exactly one data line from the right. In other cases we must keep information about data lines from the left which were already joined and which were not. To do this, each incoming envelope from the left input is extended by one column of boolean values initially set to `false`. When a data line from the left is joined with some data line from the right, we set corresponding boolean value to `true`. When the algorithm finishes, we know which left data lines were not joined and should be copied to the output.

8) Distinct: Operator distinct should output only unique data lines. We implemented this operator by the modification of a sort operator. The first phase is completely the same; however, during the merging in the second phase, the duplicated data lines are omitted from the output.

9) Other operators: The rest of operators is implemented very straightforwardly. Therefore, we do not describe them here.

D. Explicit parallelization of nested loops join

With the set of boxes described in Section IV-C, we can evaluate the complete SP^{2}Bench benchmark (see Section V). Despite the fact that the implicit parallelization speeds up the evaluation of several queries, this speed up does not scale with the number of physical cores in the host system.

Therefore, we focused on the most time-consuming operation – nested loops join – and tried to explicitly parallelize it using Bobox.

The task of nested loops join is to evaluate the join condition on all pairs of data lines from the left input and data lines form the right input.

This operation can be easily parallelized, since we can create N boxes which perform nested loops join (N denotes the number of worker threads used by Bobox). We pass one N-th of one input and the whole second input to each of these boxes and join their outputs together. It can be easily seen that this modification is valid since all pairs of data lines are still correctly processed. The whole schema of boxes is depicted in Figure 7.

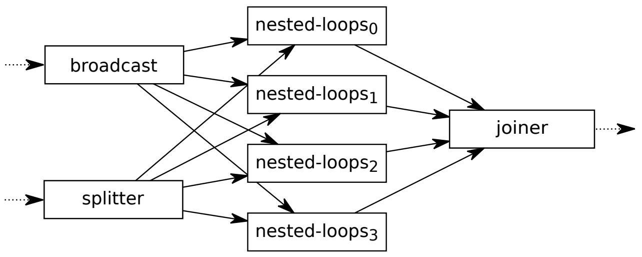

Figure 7. Parallelized nested loops join

The box *splitter* splits its input envelopes to N parts and sends these parts to its outputs. The implementation of this box must be careful since rounding errors may cause that splitted streams do not have the same length. The box *broadcast* just resends its every incoming envelope to its outputs and the box *joiner* resends any incoming envelope to its output.

Since all these three boxes are already implemented in Bobox as standard boxes, the parallelization of nested loops join is very simple.

V. Experiments

We performed a number of experiments to test functionality, performance and scalability of the SPARQL query engine. The experiments were performed using the SP^{2}Bench [16] query set since this benchmark is considered to be a standard in the area of semantic processing.

Experiments were performed on a server running Redhat 6.0 Linux; server configuration is 2x Intel Xeon E5310, 1.60Ghz (L1: 32kB+32kB L2: 4MB shared) and 8GB RAM. It was dedicated specially to the testing; therefore, no other application were running on the server during measurements. SPARQL front-end and Bobox are implemented in C++. Data were stored in-memory.

A. Implicit parallelization

In the first experiment, we measured the speed up caused by the implicit parallelization exploited by Bobox. To measure it, we chose some queries and evaluted them with an increasing number of worker threads. We did not use parallelized version of nested loops join in this experiment and we measured only runtime of evaluation of execution plan, i.e. we did not include the time spent by compilation of the query. The results are shown in Figure 8.

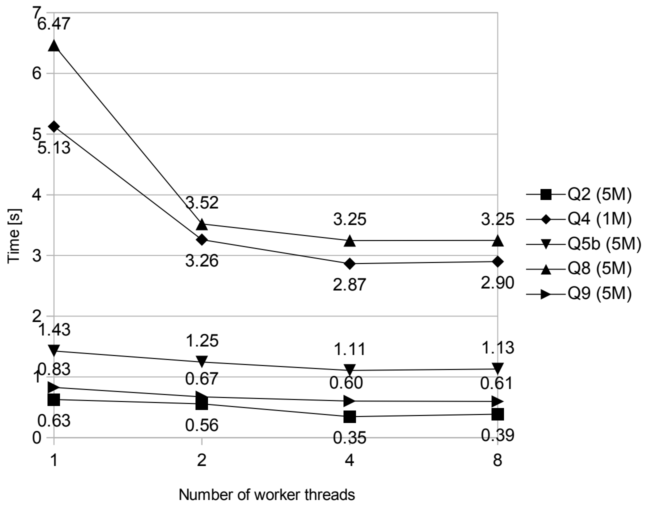

Figure 8. The speed up obtained by implicit parallelization

The results show that for some queries the speed up is quite significant; however, it does not scale with the increasing number of worker threads. This is caused by the fact that the level of parallelism is implicitly built in the execution plan which does not depend on the number of worker threads.

The query Q4 and Q8 benefits from the parallel evaluation most, since the last sort box (or distinct box respectively) runs in parallel with the rest of the execution plan. That is not the case of Q9 which contains distinct box as well; however, the amount of data processed by this box is too small to fully exploit the pipeline parallelism.

	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.

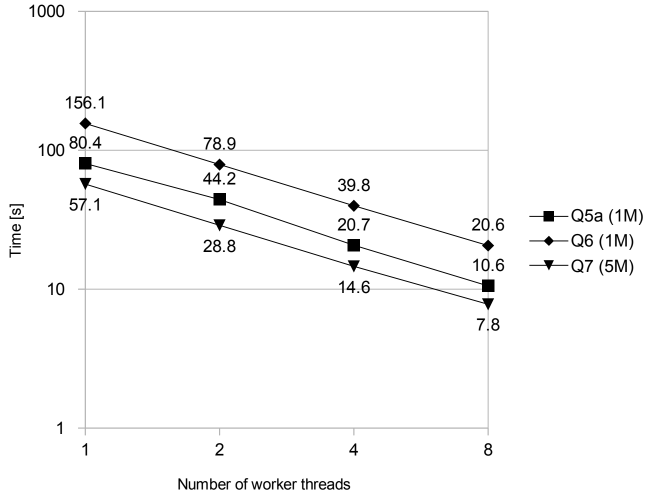

Figure 9. The speed up obtained by explicit parallelization of nested loops join

B. Explicit parallelization

In the second experiment, we focused on the speed up caused by the explicit parallelization of nested loops join. We selected the most time consuming queries with the nested loops joins. As in the first experiment, we performed multiple measurements with the increasing number of worker threads. In this experiment we also did not include the time needed by the query compilation since we focused on the runtime.

The results are shown in Figure 9. According to our expectations, data parallelism increases the scalability and causes a significant almost linear speed up on multiprocessor systems.

C. Comparison with other engines

The last set of experiments compares the Bobox SPARQL engine to other mainstream SPARQL engines, such as Sesame v2.0 [5], Jena v2.7.4 with TDB v0.9.4 [6] and Virtuoso v6.1.6.3127 (multithreaded) [7]. They follow client-server architecture and we provide sum of the times of client and server processes. The Bobox engine was compiled as a single application; we applied timers in the way that document loading times were excluded to be comparable with a server that has data already prepared.

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.

The query execution times are shown in Figure 10. The y-axes are shown in a logarithmic scale and individual plots scale differently. In the following paragraphs, we discuss some of the queries and their results. In contrary to previous experiments, we did include the time spent by the compiler in order to be comparable with other engines.

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 Engine scales well, even though it creates execution plans shaped as a left-deep tree. This is due to the parallel stream processing of merge joins. The reason why our solution is slower on 10k and 50k of triples is that the compiler takes more than 1s to compile and to optimize the query.

The variants of Q3 (labelled *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 a simple implementation of statistics used to estimate the selectivity of the filter.

Q4 (Figure 11) 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.

Queries Q5 (Figure 11) test implicit (Q5a) join encoded in a FILTER condition and explicit (Q5b) variant of joins. On explicit join both engines used fast join algorithm and are able to produce result in a reasonable time. On implicit join both engines used nested loops join which scales very badly. However, Bobox outperforms both Sesame and Jena since it is able to use multiple processors to get the results and is able to compute also documents with 250k, 1M and 5MB

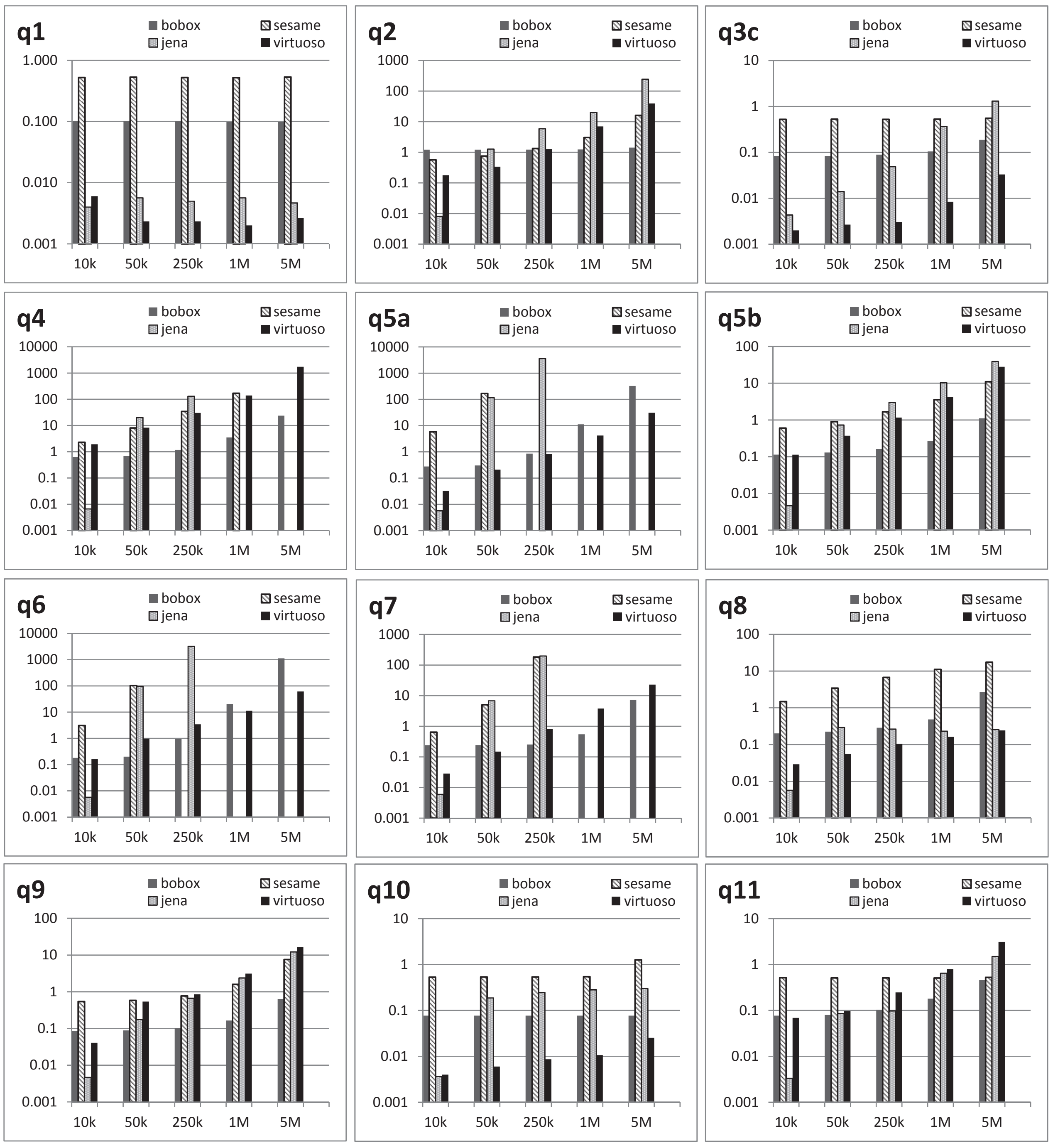

Figure 10. Results (time in seconds) for 10k, 50k, 250k, 1M, and 5M triples.

triples before the 30 minute limit is reached. On the other hand, Virtuoso outpeforms Bobox mainly due to particular query optimizations [16].

Queries Q6, Q7 and Q8 produce bushy trees; their computation is well handled in parallel, mainly because of nested loops join parallelization. As a result of this, Bobox outperforms Sesame and Jena in Q6 and Q7 and outperforms Virtuoso in Q7, being able to compute larger documents until the query times out. The authors of the SP^{2}Bench suggest [16] reusing graph patterns in a description of the queries Q6, Q7 and Q8. 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 (see Section II-B).

```
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 11. Examples of the benchmark queries.

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.

Q10 can be processed very fast because of our database representation. Therefore, only resulting triples are fetched directly from the database.

In contrary to Sesame, time of Q11 depends on the size of database. This is caused by the fact that we do not have any optimization for queries with LIMIT or OFFSET modifiers. In that case, the whole results set is produced which naturally slows down the evaluation.

Overall, the results of the benchmarks indicate very good potential of Bobox when used for implementation of RDF query engine. Our solution outperforms in all measurements Sesame, in most cases significantly and in most measurements Jena. The performance of Bobox and Virtuoso is comparable; Bobox outperforms Virtuoso namely in computing and data intensive queries.

VI. Related work

A. Parallel Frameworks and Libraries

The most similar to the Bobox run-time is the TBB library [26]. It was one of the first libraries that focused on task level parallelism. Compared to the Bobox, it is a low-level solution – it provides basic algorithms like parallel for cycle or linear pipeline and a very efficient task scheduler. The developers are able to directly create tasks for the scheduler and create their own parallel algorithms. But the tasks are designed in a way that makes it very hard to create a non-linear pipeline similar to the one Bobox provides. Such pipeline may be necessary for complex data processing [27]. Bobox also provides more services for data passing and flow control.

The latest version of OpenMP [28] also provides a way to execute tasks in parallel, but it provides less features and less control than TBB. The OpenMP library is mainly focused on mathematical computations – it can execute simple loops in parallel really fast, it can also run blocks of code in parallel, but it is not well suited for parallel execution of a complex structure of blocks. Unlike TBB or Bobox, it is a language extension and not just a library; the compiler is well aware of the parallelization and optimize the code better, but it also enables OpenMP to provide features that cannot be done with just a library, like defining the way variables are shared among threads with a simple declaration. In TBB such variable has to be explicitly passed to an appropriate algorithm by the programmer. In Bobox, it must either be explicitly passed to the model or sent using an envelope at run-time.

Some of the architectural decisions could be implemented in a different manner. One way would be to create a thread for each box and via in the model instance. This would also ensure that each box or via is running at most once at any given time. However, this is considered a bad practice [29]. There are two main reasons for not using this architecture. First, it creates a large number of threads, usually much larger than the number of CPU cores. Although it forces the operating system to switch the threads running on a core, it may not impact the overall performance that badly, since it can be arranged that the idling threads (those assigned to a box or via that is not processing any data at the moment) are suspended and do not consume any CPU time. The second problem is that when data (envelopes) are transfered from one box to another, there is very little chance that it would still be hot in the cache, since the thread that corresponds to the second box is likely to be scheduled to a different CPU, that does not share its cache with the original one. The concept of tasks used by TBB and Bobox avoid these problems and the use of thread pool, fixed number of threads and explicit scheduling gives developers of the libraries better control of parallel execution.

Besides these low-level techniques of parallel data processing, the *MapReduce* approach gained significant attention. While it is often considered a step back [30], there are application areas where MapReduce may outperform a parallel database [31]. Although MapReduce was originally targeted to other environments, it was also studied in shared-memory settings (similar to Bobox) [32], [33]. Unlike MapReduce, Bobox is desiged to support more complicated processing environment, namely nonlinear pipelines.

B. Parallel Databases

In a relational database management system, parallelism may be employed at various levels of its architecture:

- *Inter-transaction parallelism.* Running different transactions in parallel has been a standard practice for decades. Besides dealing with disk latency, it is also the easiest way to achieve a degree of parallelism in shared-memory or shared-disk environment. Although it is not considered a specific feature of parallel databases, it must be carefully considered in the design of parallel databases since parallel transactions compete for memory, cache, and bandwidth resources [34], [35].
- *Intra-transaction parallelism.* Queries of a transaction may be executed in parallel, provided they do not interfere among themselves and they do not interact with external world. Since these conditions are met rather rarely, this kind of parallelism is seldom exploited except for experiments [36].
- *Inter-operator parallelism.* Since individual operators of a physical query plan have well-defined interfaces and mostly independent behavior, they may be arranged to run in parallel relatively easily. On the other hand, the effect of such parallelism is limited because most of the cost of a query plan is often concentrated in one or a few of the operators [37].
- *Intra-operator parallelism.* Parallelizing the operation of a single physical operator is the central idea of parallel databases. From the architectural point of view, there are two different approaches:
  - a) *Partitioning* [38] – this technique essentially distributes the workload using the fact that many physical algebra operators are distributive with respect to union (or may be rewritten using such operators).
  - b) *Parallel algorithms* – implementing the operator using a parallel algorithm usually offers the freedom of control over the time and resource sharing and machine-specific means like atomic operations or SIMD instructions. However, designing, implementing, and tuning a parallel algorithm is an extremely complex task, often producing errors or varying performance results [39]. Moreover, the evolution of hardware may soon make a parallel implementation obsolete [40]. For these reasons, parallelizing frameworks are developed [41].

The central principle of Bobox allows parallelism among boxes but prohibits (thread-based) parallelism inside a box. This is similar to inter-transaction and inter-operator parallelism; however, a box does not necessarily correspond to a relational operator. In particular, Bobox allows the same approach to partitioning as in parallel databases, using transformation of the query plan.

Bobox does not allow parallel algorithms to be implemented inside a box (except of the use of SIMD instructions). Therefore, individual single-threaded parts of a parallel algorithm must be enclosed in their boxes and the complete algorithm must be built as a network of these boxes. This is certainly a limitation in the expressive power of the system; on the other hand, the communication and synchronization tasks are handled automatically by the Bobox framework.

VII. Conclusions and Future Work

In the paper, we presented a parallel SPARQL processing engine that was built using the Bobox framework with a focus on efficient query processing: parsing, optimization, transformation and parallel execution. We also presented the parallelization of nested loops join algorithm to increase parallelism during the evaluation of time consuming queries. Despite the fact that this parallelization is very simple to be done using Bobox, the measurements show that it scales very well in a multiprocessor environment.

To test the performance, we performed multiple sets of experiments. We have chosen established frameworks for RDF data processing as the reference systems. The results seem very promising; using SP^{2}Bench queries we have identified that our solution is able to process many queries significantly faster than other engines and to obtain results on larger datasets. Therefore, such a parallel approach to RDF data processing has a potential to provide better performance than current engines. On the other hand, we also identified several issues:

- We are working on improvements of our statistics used by the compiler to generate more optimal query plans.
- The pilot implementation of the compiler is not well optimized which is problem especially in Q1 and Q2.
- Our heuristics sometimes result in long chains of boxes. Streamed processing and fast merge joins minimize this disadvantage; however, it is better to have bushy query plans for efficient parallel evaluation.
- Also, some methods proposed in SP^{2}Bench, such as graph pattern reuse, are not efficiently applicable in the current Bobox version.
- The query Q4 is very time consuming and does not benefit much from the fact that the system has multiple processors. Therefore, we must parallelize besides the nested loops join also merge join, which is the bottleneck of this query.
- Currently, we support only in-memory databases. In order to have engine usable for processing of really large RDF databases such as BTC Dataset (Billion triple challenge) [42], we must keep the database in external memory.

Because of these issues, we are convinced that there is still space for optimization in parallel RDF processing and we want to focus on them and improve our solution.

Acknowledgments

The authors would like to thank the GAUK project no. 28910, 277911 and SVV-2012-265312, and GACR project no. 202/10/0761, which supported this paper.

References

[1] 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.

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

[3] 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/

[4] 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.

[5] 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.

[6] "Jena – a semantic web framework for Java," http://jena.sourceforge.net. [Online]. Available: http://jena.sourceforge.net, retrieved 10/2012

[7] "Virtuoso data server," http://virtuoso.openlinksw.com, retrieved 10/2012

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

[9] 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.

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

[11] Z. Falt, D. Bednarek, M. Cermak, and F. Zavoral, "On Parallel Evaluation of SPARQL Queries," in *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2012, pp. 97–102.

[12] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Data-Flow Awareness in Parallel Data Processing," in *6th International Symposium on Intelligent Distributed Computing - IDC 2012*. Springer-Verlag, 2012.

[13] "The Bobox Project - Parallelization Framework and Server for Data Processing," 2011, Technical Report 2011/1. [Online]. Available: http://www.ksi.mff.cuni.cz/bobox, retrieved 12/2012

[14] 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.

[15] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Bobox: Parallelization Framework for Data Processing," in *Advances in Information Technology and Applied Computing*, 2012.

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

[17] Z. Falt, J. Bulanek, and J. Yaghob, "On Parallel Sorting of Data Streams," in *ADBIS 2012 - 16th East European Conference in Advances in Databases and Information Systems*, 2012.

[18] J. Dokulil and J. Katreniakova, "Bobox model visualization," in *14th International Conference Information Visualisation*. London, UK: IEEE Computer Society, 2010, pp. 537–542.

[19] 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.

[20] J. Galgonek, "Tequila - a query language for the semantic web," in *DATESO 2009*, ser. CEUR Workshop Proceedings, K. Richta, J. Pokorný, and V. Snášel, Eds., vol. 471. Czech Technical University in Prague, 2009, pp. 105–118.

[21] M. Krulis and J. Yaghob, "Revision of relational joins for multi-core and many-core architectures," in *Proceedings of the Dateso 2011*. Pisek, Czech Rep.: FEECS, 2011.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] D. Bednárek, "Bulk evaluation of user-defined functions in XQuery," Ph.D. dissertation, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2009.

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

[29] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007.

[30] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes?" *Commun. ACM*, vol. 53, pp. 64–71, 2010.

[31] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. USA: ACM, 2010, pp. 975–986.

[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[33] G. Kovoor, "MR-J: A MapReduce framework for multi-core architectures," Ph.D. dissertation, University of Manchester, 2009.

[34] F. Morvan and A. Hameurlain, "Dynamic memory allocation strategies for parallel query execution," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2002, pp. 897–901.

[35] Z. Zhang, P. Trancoso, and J. Torrellas, "Memory system performance of a database in a shared-memory multiprocessor," 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1924, retrieved 10/2012

[36] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, "Optimistic intra-transaction parallelism on chip multiprocessors," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 73–84.

[37] A. N. Wilschut, J. Flokstra, and P. M. G. Apers, "Parallel evaluation of multi-join queries," in *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1995, pp. 115–126.

[38] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[39] J. Aguilar-Saborit, V. Muntes-Mulero, C. Zuzarte, A. Zubiri, and J.-L. Larriba-Pey, "Dynamic out of core join processing in symmetric multiprocessors," in *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 28—35.

[40] 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, no. 2, pp. 1378–1389, 2009.

[41] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye, "Automatic contention detection and amelioration for data-intensive operations," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. New York, NY, USA: ACM, 2010, pp. 483–494.

[42] "Billion triple challenge." [Online]. Available: http://challenge.semanticweb.org, retrieved 10/2012