Using Methods of Parallel Semi-structured Data Processing for Semantic Web

David Bednářek, Jiří Dokulil, Jakub Yaghob and Filip Zavoral
Faculty of Mathematics and Physics, Charles University in Prague
Email: {bednarek, dokulil, yaghob, zavoral}@ksi.mff.cuni.cz

Abstract—The state of the art in semi-structured data processing (and XML in particular) and Semantic Web repositories correspond to each other: the non-scalability of pilot implementations, the inability of optimizations, and the cost of the fully native implementation. Although there are successful implementations in each of the approaches, none of the methods may be considered universal. The Bobox framework proposed in this paper is a relational-like storage engine applicable both as a native XML database and as a Semantic Web repository. The main purpose of the engine is in experiments in both areas. The main stress is put to the performance of complex queries and transformations, and to the ability of parallel evaluation in particular.

Keywords—XML database; Semantic Web repository; query evaluation; parallelization

I. INTRODUCTION

Parallel and distributed processing have become two of the leading trends in the development of massive data processing systems. Current state of the art semantic frameworks and data repositories have not yet adopted this trend. Therefore they cannot utilize the potential of current advanced hardware architectures such as multiprocessor or distributed systems.

In this paper we propose an architecture for a task-based parallel and distributed data processing engine – the Bobox project. It was originally designed for parallel XQuery processing; as Section II-A shows, it can be adopted for Semantic Web data processing as well. So far, we have defined interfaces for the system that allows us to implement particular computational components that do the actual data processing. Currently, we have a serial version of the execution environment that can be used to test the functionality of the computational components. An efficient parallel and distributed version is currently under development.

The following section presents the related work, the Section III describes the application and goals of the Bobox framework. The following two sections describe the evaluation environment and the optimizations that are possible in it. The Section VI concludes the paper.

II. RELATED WORK

Semi-structured data, and XML in particular, formed one of the hot topics in both academic and industrial research. In the area of XML data bases, indexing, and querying, the research resulted in a number of experimental as well as commercial implementations. Majority of the designs share the following principles:

- The physical representation of a XML collection is based either on binary XML [15] or on shredding XML data into relational form [13].
- XML-specific indexing techniques are required – various forms of path/value indexes [7], interval encoding [12], or Dewey numbering [16] are used.
- Queries are translated to some form of algebra by the addition of XML-specific operators, including structural, path [8], or twig joins [10].

Many implementations, in particular the commercial ones, were created as an extension to an existing relational DBMS [13], others are built on top of an existing relational core [3], [9]. This approach allows reuse of solutions of a number of collateral problems, including transaction handling and concurrency. Despite the general success of such methods, there are still many areas open to new designs. We emphasize two problems:

- The application of the relational-like techniques of XQuery evaluation in a parallel environment.

The two problems are related by the fact that a stream (pipe) is a natural way of interconnection in parallel environment when shared memory is not available. Furthermore, in many environments including the exploration of the web, a XML database is continuously fed by a XML stream instead of individual updates.

A. Semantic Web

The Semantic Web initiative created a set of new applications of databases (in the broadest sense of this word). The new areas include the storage of the information gathered from the web by crawlers as well as the representation of the knowledge prepared with human assistance, including linguistic and ontological data. Research in the Semantic Web, social networks, and other areas that study the data gathered from the world-wide web, produces huge collections of data that require appropriate storage technologies. The semantic collections of these days contain about $10^{9}$ entries; however, significant growth is expected in the future.

There is already a number of implementations in the area of RDF repositories. The most common approaches are:
The following criteria apply:

- A dedicated main-memory engine (e.g. [4]). This approach allows easy implementation and high performance; however, it is limited by the main memory available to a single process.
- Using a relational or object-relational DBMS [14]. The use of a RDBMS offers the experience gained in the relational DBMS development - their reliability, scalability, and cost-based optimization techniques. Unfortunately, relational systems were designed with different applications in mind; consequently, some features like full transaction isolation or rollback ability are almost useless in a typical Semantic Web application. Furthermore, a typical semantic query, after translation to SQL, is a tough job for the relational query optimizer because such queries were rare in traditional relational settings.
- A specialized native engine, like [5]. Although still based on relational operations, such an engine offers better query performance because its query optimizer is specialized to the particular semantic query language (e.g., SPARQL) and the compilation to SQL is bypassed. Moreover, specialized algebra operators may be added to support specific tasks. The main disadvantage of this approach is the costly implementation, although a number of components may be reused from relational systems.

The state of the art in Semantic Web repositories corresponds to the situation in the area of XML processing, with a few more years of experience at the side of the XML. In the XML world, the above-mentioned approaches may be identified too: There are in-memory implementations of XSLT and XQuery, implementations based on shredding XML into a relational DBMS, and native XML databases. The problems are also similar: The non-scalability of an in-memory engine, the inability of a RDBMS to optimize a query translated from a different language, and the cost of the fully native implementation. Nevertheless, there are successful XML implementations of all three approaches. None of the methods may be considered universal – they perform differently in different areas of application.

III. APPLICATION AREA

The Bobox project is targeted at the application areas where the following criteria apply:

- The structure of the data cannot be matched to the models (like the E-R model) used in the area of relational databases.
- The schema of the data is variable over different sources, evolving in time, or unknown at all.
- The data often include recursive structures (XML documents, tree-banks) or graph-like connections (RDF).
- The repository is filled either by a continuous flow of data (e.g. by crawlers) or in large batches (e.g. RDF collections); the incoming data replace existing data in a versioning manner. Individual record update is rare.
- Transaction control is reduced to batch-update / snapshot-read behavior; a delay in the propagation of updates is considered tolerable.
- XML-based formats are used in the exchange of data at both the batch-input and the query-output side.
- Incoming data may require transformation to fit to the required repository format.
- Sophisticated queries are placed that process a large subset of the repository data. The performance of such queries may benefit from parallel evaluation.
- Queries are formulated in a domain-specific query language (XQuery, SPARQL); the language is at least partially translatable to the relational algebra.

These criteria are met in many applications of the Semantic Web like RDF repositories; furthermore, there are also areas of XML where the reduced update / transaction capabilities may be tolerated.

A. Main Bobox Goals

The goal of the Bobox project is an experimental implementation of a relational-like storage engine applicable both as a native XML database and as a Semantic Web repository. The main purpose of the engine is in experiments, in both the Semantic Web and XML areas. The main stress is put to the performance of complex queries and transformations, in particular to the ability of parallel evaluation.

The implementation is intended to become a base for a number of experiments, including the evaluation of the storage engine on its own, the comparison of different structural join algorithms or shredding strategies etc. The performance gains achieved by parallelization will be measured.

The engine is composed of the following layers:

- A query-language front-end compiles a domain-specific query language (XQuery, SPARQL, . . . ) into a domain-independent relational-like intermediate language.
- The intermediate representation is statically analyzed and optimized.
- The optimized query is executed using a parallel engine in a multiprocessor and/or cluster arrangement.
- The input to the engine is fed either from an input pipeline or from a partitioned persistent storage.
- The output is pushed to an output pipeline or stored back into the persistent storage.
- The input/output pipelines may consume/produce XML documents, including RDF, or other Semantic Web formats, including a feed from a web crawler.
- The persistent storage as well as the intermediate language is based on the relational paradigm and extended towards operations required in the XML/Web domains.

IV. EVALUATION ENGINE

A. The Architecture

The evaluation of a query (Fig. 1) starts by creation of an R-program – a plan of the evaluation of that particular
query. This is done by the front-end specific to the processed query language, e.g. XQuery, SPARQL, TriQ [6].

The back-end consists of expander and pipelined interpreter. The expander transforms the R-program to the actual computational components and connecting pipes. The pipelined interpreter then controls the computation and cleans up after it is complete.

The plan is the interface between the front-end and the back end. It defines the desired structure of components and pipes (see Fig. 1) and also contains the information about structure of the records at each pipe, required buffer sizes, and parallelization plans.

The computational components perform atomic data manipulations like selection, projection, joins, sorts, subgraph and path queries, etc. Some multiple-input structural join algorithms may be distributed over a set of components. The data is streamed among the components using pipes. Depending on the actual location of the components, the pipes can either be local or distributed over a network. The system is loosely synchronized, allowing out-of-order execution with respect to the canonical application language semantics.

The main component of the interpreter is a thread manager. There is a thread manager for each node in the system. Under the control of the thread manager, the components and pipes cooperatively evaluate the program in a multi-threaded distributed environment. Synchronization of the system is done only by the producer-consumer relationship between components connected by a pipe.

B. R-programs

One of the significant front-ends is the XQuery compiler. It works by translating a query to an R-program by a reversed evaluation approach [1]. Furthermore, static analysis methods [2] are implemented in the XQuery language front-end. The produced R-program is then easily translated into a Bobox query evaluation plan.

An R-program consists of a set of R-functions. The body of each R-function is described by a directed graph of (extended) relational algebra operators and R-function calls. Each R-function receives one or more relations as its input arguments and produces one or more relations at its output. Therefore, the traditional relational algebra can be considered as a special case of R-programs.

Since the language of R-programs does not offer any programmatic structures like conditions or loops, any recursive R-program would immediately fall into endless recursion. To give recursive R-programs their semantics, the notion of controlling argument is defined that allows to predict the output and to stop the recursion when the controlling arguments are empty. This mechanism corresponds to the case when recursion in an XQuery program is stopped by iterating over an empty set. Of course, termination of R-programs is not generally guaranteed just as the termination of XQuery programs is not, therefore the evaluation engine enforces a maximal level of recursion.

The Fig. 2 shows an example of R-program in a textual form; this R-program computes the transitive closure $m^+$ of its argument $m$.

First, only the plan for the main function is instantiated and the evaluation is started. While the evaluation is in progress, the $t_1$ component can receive data through one of the pipes. When this happens for the first time, the $t_1$ is replaced with instance of the plan for the body of the function. This new part contains $t_2$ which is replaced by the appropriate plan instance under the same circumstances.

C. RDF data

Most contemporary RDF query languages are table oriented. They combine graph pattern matching and some form of relational algebra. Since the graph pattern matching is used as a first step of the query to transform the graph structure of RDF data into relations, we only have to create the pattern matching components. The relational algebra can be reused.

On the other hand, the TriQ language is one of the graph based languages. Although the same framework can be used, most of the computational components have to be reimplemented.
V. R-PROGRAM OPTIMIZATION

In this section, we present the proposed methods and principles. Consider the sub-query “return all persons which were employed on a given date” used in the query “return number of employees for each date in the history”.

```xml
declare function local:employed($P as xs:date) {
  fn:doc("company")//employee
  [ @hired lt $P and @fired gt $P];
}

<report>
  for $D in fn:doc("history")//@date return
  <point date="{$D}"
    number="{fn:count(local:employed($D))}"/>
</report>
```

Figure 3. An XQuery function returning a sequence of nodes

In XQuery, such a sub-query may be represented by a function. The function `employed` in the Fig. 3 is parametrized by the date `$P` and return a sequence of matching employees.

A. Naive Evaluation

A naïve implementation calls the function `employed` for each date in the given history. Due to the nature of the condition placed on the `employee` nodes, value indexes on `@hired` and `@fired` can not reduce the number of scanned nodes significantly.

B. Optimized Evaluation

To enlarge the opportunity to optimize, we suggest the following arrangement shown in the Fig. 4: the function `employed` is statically transformed so that it processes all the values of the parameter `$P` in one call. The transformed function returns all the original return values in a single batch. In other words, the transformed function is called only once, instead of repeated calling in the naïve approach.

Bulk evaluation offers the ability to use more effective join techniques than nested-loop evaluation. In our example, a kind of theta-join is used to combine the set of parameter values with the set of employee nodes. To reduce cost, this theta-join may be implemented using range scans on a sorted materialization of the left operand – this arrangement would never be possible in the naïve implementation. Figure 5 shows the corresponding query plan.

The concept of view merging is a form of procedure integration known from compiler construction. In the merged query, subsequent transformations are unaware of the original boundary of the view, allowing aggressive optimization (like join reordering) across the hidden boundary. Of course, the preservation of function boundaries reduces the maneuvering space of subsequent optimization.

C. Reverse Optimized Evaluation

The Fig. 6 shows an improved version of the execution schema. Assuming that the attribute `@date` has an ordered index, the theta-join was implemented by a repeated range scan over the index. The Fig. 6 depicts its query plan.

If the function `employed` were integrated into the surrounding query, the shift from the Fig. 4 to the Fig. 6 would be a simple algebraic transformation. However, the bulk evaluation requires that the boundary of the function be still present. Therefore, such a transformation must be formalized as a transformation of the function interface.
The caller of the function is expected to perform a range-
based theta-join of the original sequence of dates with 
the intervals generated by the function through the output 
parameter. The attributes not involved in the join (marked 
here as emp) are just passed around.

Although it may seem that we are pulling out all joins 
from the function, it is not true — only those join conditions 
that may be implemented with a particular physical access 
method are worthy of extraction. Therefore, there is only a 
small number of transformations that may be applied to a 
function parameter.

The transformed function can no longer be evaluated in 
a simple call-return manner. Instead, the ability to pass the 
control to and from the function more than once is required. 
Moreover, the function must be able to retain their private 
data during the time the control is temporarily returned to 
its caller.

Pipelined execution is required to avoid unnecessary ma-
terialization of intermediate results. Therefore, a function 
may run effectively in parallel with its caller, returning the 
first data for output parameters sooner than the last data for 
input parameters arrive.

As illustrated in the example, the reversed data flow 
allows query reformulation in a way that is essentially 
equivalent to join reordering and similar techniques known 
from traditional RDBMS architectures. Such a replacement 
is necessary because traditional query rewrite methods are 
not directly applicable to R-programs due to the presence of 
functions and expression reuse.

D. Optimization Architectures

The optimizations can be performed statically during a 
compile time or dynamically during the run-time. Intra-
procedural optimization (denoted as static rewriting) and 
local plan selection may be applied as shown in the Fig. 
8. The resulting physical plan is again in the form of a dag 
of algebra operators and function calls; thus it is again an 
R-program, albeit using a different set of operators.

![Architecture with Static Optimization](image)

The effectiveness of intra-procedural cost-based optimiza-
tion is limited because the cost of function calls and cardin-
ality of their outputs is not known. This weakness may be 
addressed with the architecture depicted in the Fig. 9.

![Architecture with Dynamic Optimization](image)

In each cycle, the expanded R-program may be optimized 
by rewriting and transformed using cost-based plan selection 
to a physical R-program. Since the original R-functions 
were integrated into a single function, the optimization is 
in fact inter-procedural. Of course, this phase may alter 
only the newly appended R-function body because the 
previously integrated code is already being executed. On 
the other hand, the plan selection may make use of cost
and cardinality estimation computed throughout the whole integrated program. Therefore, it may produce better plans than in the case of local plan selection.

VI. Conclusion and Future Work

In this paper we propose the architecture of the Bobox native repository useful for semi-structured data and the Semantic Web. The system is designed to run in a parallel and distributed environment. It is currently the only system capable of providing the features required to evaluate R-programs, most notably the run-time recursive plan expansion. The proposed solution extends the traditional architecture of query engines with the ability to connect streaming data sources and sinks; in particular, it can be embedded into heterogeneous and service-oriented environments.

At the moment, we have an experimental implementation of key parts of the system. Their main purpose is to test basic principles and to help us refine the interfaces between various components of the system. The consecutive experiments will be focused on the evaluation of the system and the optimization of parallel and distributed access and query evaluation.

ACKNOWLEDGMENT

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 - XML Data Processing and 201/09/0983 - Agile systems and service-oriented software.

REFERENCES


