XQuery Algebra

Petr Lukáš[*]
Faculty of Electrical Engineering and Computer Science
Department of Computer Science
Technical University of Ostrava
17. listopadu 15/2172, 708 33, Ostrava
petr.lukas@nativa.cz

Abstract
This work deals with design and implementation of a processor of the XQuery computer language used for searching data in tree organized XML documents and databases. The goal of the work is to design and create a processor based on algebraic operators which can give better performance in evaluation of the input queries than direct interpretation. The result of the work is a real functional prototype comparable to other commonly used implementations. Those are usually written in high-level languages like Java or .NET Framework. Our prototype is written completely in the C++ so we are not limited by using automatic memory management. The final part of this paper is focused on the time comparisons.

Categories and Subject Descriptors
H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; D.3.4 [**Programming Languages**]: Processors

Keywords
XML, XQuery, data query languages, processor, algebra, optimization

1. Introduction
Data query languages have been developed for many years and they are focused mostly on the relational databases. However tables or more technically relations are not the only way how to organize data. The trend of the modern IT life is to use tree structures in the shape of the XML documents or whole databases. The main advantage of these databases lies in the dynamic schema so the real objects or situations can be described more precisely and flexibly. This fact leads to the need of the development of efficient query tools over these tree organizations.

[*]Master study programme in field Software Engineering. Supervisor: Dr. Radim Bača. Faculty of Electrical Engineering and Computer Science, Department of Computer Science, Technical University of Ostrava.

Lukáš, P. XQuery Algebra. Information Sciences and Technologies Bulletin of the ACM Slovakia, Special Section on the ACM Student Project of the Year 2012 Competition, Vol. 4, No. 4 (2012) 49-50

2. Query processing
XQuery is a representative of compiled languages so the schema of query processing is derived from the schema of a typical compiler.

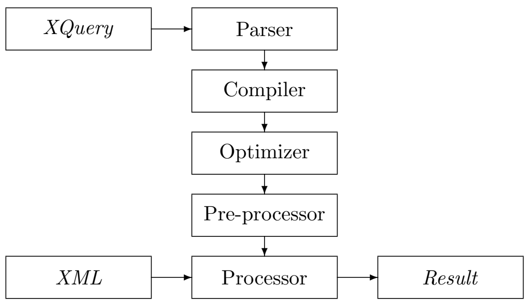

Figure 1: Block structure of the processor

The first step is to analyse the input query by the parser and build up a syntax tree based on the XQuery grammar. The grammar is standardized by the W3C organization. After that compiler translates the syntax tree to the query plan. The query plan is a tree structure where the nodes represent operators of an algebra described in [3]. It is a special algebra operating on two different types of sets – set of tuples (relations) and set of data items (sequences of XML nodes or atomic values). An example of a simple query and its query plan is shown on figure 2.

```
for $i in //items
where $i/@id = "456"
return $i/name
```

```
MapToItem { TreeJoin[child;name] ( IN#i ) }
(
  Select { Call[eq] ( TreeJoin[attribute;id]( IN#i ); Scalar[456] ) }
  (
    MapFromItem { [i:IN] }
    (
      TreeJoin[descendant-or-self;items] ( Call[root] )
    )
  )
)
```

Figure 2: An example of XQuery and its plan

The compile rules build the plan so that the result of the query is correct but it is not ensured that the evaluation is really efficient. The next step after compilation is the phase of optimization. The optimization techniques make such modifications in the compiled query plan as the result is preserved but the evaluation is more effective.

Before performing the result evaluation, the optimized query plan is statically preprocessed. The nodes of the query plan can perform some pre-calculations dependent on the purpose of a particular operator but independent of any intermediate result of the evaluation. Finally the optimized and preprocessed query plan is evaluated.

2.1 Compiling without normalization

In the standard chain of processing XQuery queries there should be one more step called normalization. W3C defines a subset of XQuery known as XQuery Core. It contains only such necessary subset of constructions as any query can be rewritten to them. For instance, the Core grammar does not allow arithmetic expressions – those are rewritten to built-in functions. The advantage of this approach is in simple compilation rules of the normalized query. However there are many simple queries that have to be rewritten to quite complicated structures of the Core constructions. XPath[1] expressions are a typical example. Those have to be normalized to nested FLWOR [*read as flower*] expressions.

The compilation rules in this work are designed to translate the input queries without the need of normalization so the query plans are more simple.

There are some problems resulting from compilation without normalization. The compilation rules have to ensure the availability of the context variables (variables valid in a part of a query) in all parts of the query plan where the variables could be accessed. The problem is that the variables could be accessed but in many situations they are not really accessed so the operators ensuring the availibility are redundant. The work shows how to detect and remove these redundant operators.

3. Evaluation

The goal in the implementation of the evaluating engine is to focus on all algorithms of the operators and find all operations that are not dependent on any itermediate result of the query, so that they can be performed only once in the static preprocessing phase.

A typical situation is a query calling a function. All function calls are compiled into the *Call* operators. The compiler passes the name of the called function to the node of this operator and sets up sub-operators evaluating the arguments. When the node of the query plan is evaluated it is needed to find the particular algorithm of the function according to the name. Just imagine a function call in the boolean expression of a selection operator[2]. The function call is evaluated for all tuples of the input table. It is obvious that the searching of the function algorithm can be performed only once during the static evalutation.

It is also desirable to avoid memory allocations during the evaluation - those are the most time expensive operations. The work describes a simple instance pooling mechanism which can give us a possibility to reuse allocated objects of the data model. The processor can process more queries and the second run of the same query is usually noticeably faster.

[1]XPath is a subset of XQuery language
[2]Selection is a standard operator of relational algebra

4. Time comparisons

The final part of the work compares our processor to some other existing implementations. Saxon and XmlPrime processors were chosen for the experimental evaluations. 13 testing queries were processed twice on each processor. It is obvious that all implementations have an algorithm to reuse allocated memory objects. The average relative time results comparing our processor to Saxon are found on figure 3.

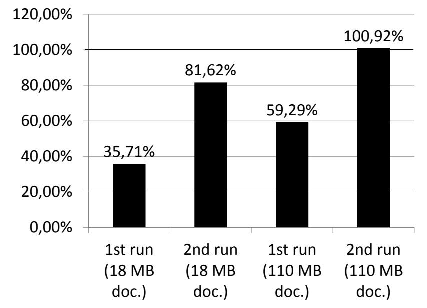

Figure 3: Our implementation compared to Saxon

At the point when the work was published the implementation was at the state of prototype. The implementation is still being developed, so the results on figure 3 come from the actual state of the work. The latest tests were processed over two (18 MB and 110 MB) XML documents. The results are calculated as gerometrical means of all relative comparisons (quotients of the times of our processor and the times of Saxon). The times were measured with an accuracy of 10 ms.

5. Conclusion

As we can see on figure 3, our processor is 64 % faster in the 1st runs and 18 % faster in the 2nd runs over the 18 MB testing XML document and 41 % faster in the first runs over the 180 MB document. The second runs over the 180 MB document lasted approximetely the same amount of time.

There are still possibilities how to increase the speed – e.g. using the indexed XML databases, unnesting optimizatios [3] or profiling the code.

The database research group of the Technical University of Ostrava is developing an indexed native XML database engine. The engine uses tree pattern queries (TPQ [1, 2]) as the main querying tool. We are working on the integration of this processor and its algebra to the indexed engine. The challenge is to detect TPQs in the compiled query plan and let them evaluate by the indexed XML database.

References

[1] I. Manolescu, Papakonstantinou, and V. Vassalos. Xml tuple algebra. 2009.
[2] P. Michiels, G. A. M. ailă, and J. Siméon. Put a tree pattern in your algebra. 2007.
[3] C. Ré, J. Siméon, and M. Fernández. A complete and efficient algebraic compiler for xquery. 2006.