A Query Processing Approach for XML Database Systems

Christian Mathis, Theo Härder

University of Kaiserslautern
{mathis | haerder}@informatik.uni-kl.de

Abstract: Besides the storage engine, the query processor of a database system is the most critical component when it comes to performance and scalability. Research on query processing for relational database systems developed an approach which we believe should also be adopted for the newly proposed XML database systems. It includes a syntactic and semantic analyzation phase, the mapping onto an internal query representation, algebraic and cost-based optimization, and finally the execution on a record-oriented interface. Each step hides its own challenges and will therefore be discussed throughout this paper. Our contribution can be understood as a road-map that reveals a desirable set of functionalities for an XML query processor.

1 Introduction

After relational, network-based, hierarchical, object-oriented, object-relational, and deductive database systems, academic research and businesses raise their attention to database-driven processing of XML documents, resulting in a new kind of information system, namely the *(native) XML database system* (XDBS). This development is reasonable, because the *eXtensible Markup Language* nowadays plays an important role in various key technologies like content management systems, electronic data interchange, and data integration techniques. Furthermore, for the management of a possibly large collection of XML documents, the classical advantages of dedicated database systems over file systems still hold: convenient use of XML data through a standardized application programming interface (API); transactional warranties for all operations on XML data; processing of large volumes of data, measured in number of documents as well as document size. Further advantages of database systems like scalability with respect to the current transactional load, high availability and fault tolerance, as well as data and application independence shall be mentioned for completeness, though they are not XML specific.

In [1], Michael Haustein outlines the design and realization for a subset of these desirable concepts—a prototype for academic research and native XDBS, named *XML Transaction Coordinator* (XTC). Currently, XTC provides an internal node interface called taDOM, which includes the features of the Document Object Model enhanced with user transactions. Every sequence of DOM operations can be encapsulated by a transaction and can thus benefit from the ACID warranties. For declarative language access, an XQuery processor resides on top of this interface. Its implementation follows the concepts given in the XQuery formal semantics [2], thereby neglecting important optimization techniques, which were crucial for the success of relational database systems in the past thirty years.

Throughout this paper, we focus on the problem of query evaluation for declarative XDBS access using XQuery. Our contribution can be understood as a road-map that reveals a desirable set of functionalities for the XTC query processor.

2 Levels of Abstraction in XML query processing

To handle the complexity of query processing, several levels of abstraction between a declarative query expression and its procedural evaluation using a set of low-level operations can be identified. These levels are depicted in Table 1. To facilitate comprehension, the new XML-related concepts are compared to their well-known counterparts of relational query processing. The most abstract view of a query is its formulation in a way that only describes the desired result in a certain declarative language. The same query may be represented using an algebra expression, whose operators express the query in the *Logical Access Model*. Optimization techniques at this level only rely on the expression itself, but do not cope with system-specific information. In general, this is the task of the layer below—the *Physical Access Model*. Finally, the bottom layer accomplishing the storage of XML documents plays

Level of Abstraction	**XDBS**	**RDBS**
Language Model	XQuery	SQL
Logical Access Model	XML Query Algebra	Relational Algebra
Physical Access Model	Physical XML Query Algebra	Physical DB-Operators
Storage Model	XTC, Natix, Shredded Documents	Record-oriented DB-Interface

Table 1. XML Query processing abstraction levels

also an important role, because the efficiency of operations is critically dependent on the chosen storage structure. An explicit separation of this abstraction level helps to cope with mapping requirements when multiple heterogenous storage models are present. Each of the depicted layers has its own associated tasks for query evaluation and hides its own challenges. Therefore, we will elaborate on them.

2.1 The Language Model

So far, several declarative XML query languages have been proposed, among them Lorel, XML-QL, XML-GL, and, as the latest development, XQuery. A survey of these languages in [3] singles XQuery out as the most universal language, measured by the demands posed in [4][1]. Furthermore, XQuery is likely to be standardized by the W3C and will therefore presumably play a similar role for XML data as SQL does for relational data. These were the main arguments for choosing XQuery as the XTC query language. The language was designed to meet the demands of both the “document-centric” community—notably text search functionality and document-order awareness—and the “data-centric” community—expecting powerful selections and transformations[2]. The design efforts resulted in a very complex, strongly typed language allowing nested subexpressions at almost every position. Contribution [5] shows that XQuery has the same expressive power as μ-recursive functions, and is thus Turing-Complete. Because of an inherent trade-off between expressiveness and evaluation complexity, the question which “parts”, or more formally, which sublanguages of XQuery may effectively be evaluated by a query processor, gains significant importance. For XPath, Gottlob et. al. answered this question in [6] stating that XPath is evaluable in polynomial time and space. In addition, a certain part of XPath can even be processed in linear time and space. However, for special extensions of XPath towards complete XQuery, this complexity determination is still an open problem. In the same context, another interesting question is to decide on given problems whether they can be solved using XPath or whether they need XQuery constructs.

Furthermore, several practical problems regarding the language model arise: the syntactic and semantic analysis of a query and its transformation into a convenient internal representation to be used throughout the subsequent optimization steps. As observed in [3], XQuery may be syntactically analyzed using an LR parser. The semantic processing requires a specific phase for static type and reference checking to recognize user errors as early as possible. As demanded in [7], an internal query representation should be efficiently accessible, flexible w. r. t. subsequent transformation steps, and, furthermore, should reflect a kind of procedural evaluation strategy. So far, we have only dealt with operator trees and we may legitimately ask whether there is the need for a further refinement of the internal representation structure or whether operator trees are sufficient, in general. We conclude this section with our running XQuery example depicted in Figure 1. Given the departments and employees of an organization in the documents `dept.xml` and `emp.xml`, it returns a list of all persons who work in Kaiserslautern and earn more than 50000.

```
<result>{
    for $dep in doc('dept.xml')//departement,
        $emp in doc('emp.xml')//employee
    where $dep/@depnr = $emp/@depnr and
        $dep/location = 'Kaiserslautern'
        and $emp/salary > '50000'
    return
        <person>{
            $pers/name,
            $pers/salary
        }</person>
}</result>
```

Figure 1. Example Query

2.2 The Logical Access Model

After a query is transformed into its internal representation, query optimization can begin. In general, optimization goals are the reduction of query processing time or the maximization of throughput. The main obstacle occurring during the optimization process is the possibly large number of equivalent evaluation strategies for a given query, originating from varying operator orders, different operator implementations, the existence of indices, and so on.

[1] However, unlike Lorel, XQuery does not support document modification operations, which certainly will be added in future versions.

[2] These expectations resemble the different qualities represented by the object-oriented model for “vertical search” and the relational model for “horizontal search”

In a first step, the query may be optimized regarding only the logical query structure, neglecting all further system-specific issues. This process is called non-algebraic optimization or query restructuring. Its key idea is to minimize intermediary results by executing the most selective operations as early as possible. To achieve this goal, the query has to be transformed in a semantics-preserving way. A general approach to identify these transformations is the use of algebra expressions onto which queries are mapped.

To facilitate the operation tracking of our running example across the various abstraction levels, we will use our own algebra notation here, although several others have been published over the past years. Figure 2 contains the algebraic equivalence for the query in Figure 1. Each algebra operator relies on a set of ordered sequences of tuples, depending on the arity of the operator, and produces a single result sequence containing n-ary tuples. For example, a binary join operator processes two input sequences and produces one output sequence containing composed tuples. The expression in Figure 2 can be read as follows: The innermost operator S ("source") provides the document node of the specified documents in a singleton sequence. Then the *follow* operator (ϕ) evaluates the specified relative path expressions and its result sequence is bound (β) to the variables $\$dep$ and $\$emp$. Afterwards, the join operator can be evaluated, generating a sequence of binary tuples which obeys the join order. The *selection* operator filters these tuples by its predicate and sends the result to the *construct* operator (C), which wraps each tuple into a person element. Another construct operator builds the final result contained in a result element.

$$
\begin{gathered}
e_1 = \sigma_{[\phi_{location}(\$dept)=\text{"}KL\text{"} \wedge \phi_{salary}(\$emp)>50000]} \\
\Big[\beta_{\$dept:\phi_{//departement}}(S(\text{"}dept.xml\text{"})) \\
\bowtie_{[\phi_{@depnr(\$dept)}=\phi_{@depnr(\$emp)}]} \\
\beta_{\$emp:\phi_{//employee}}(S(\text{"}emp.xml\text{"}))\Big] \\
\\
C_{result}(C_{person,\phi_{name}(\$emp)\circ\phi_{salary}(\$emp)}(e_1))
\end{gathered}
$$

Figure 2. Algebraic Expression for Query of Figure 1

So far, various proposals for an XQuery algebra have not lead to a standardization. In fact, the published approaches differ a lot in the following features: underlying data model, operator input and output format, existence of a designated evaluation operator (like ϕ) for XPath, handling of order, expressive power, representation of XQuery expressions, treatment of query nesting, etc. Nevertheless, each approach identifies certain algebraic equivalence rules. Relying on those equivalences, an optimization heuristics may have the following course of action: general, n-ary joins are mapped onto binary ones; selections containing multiple predicates are decomposed and pushed towards the source operators (eventually integrated into path expressions, if possible); projections specified as XPath expressions are also pushed towards the source operators, their results are bound to a variable for later use (for example, in the construction process); finally, multiple adjacent selections may be composed again. Applying these actions to the expression in Figure 2 leads to the result in Figure 3.

Note that the child axis leading from *employee* to *salary* has to be evaluated twice. To save this overhead, the salary could be bound to a variable to be used in the construction operator. In this way, however, the placement inside the XPath expression would not be possible. This is an argument in favor of algebras which explicitly handle XPath expressions, because this memorization technique would work in these cases. Further issues regarding this stage of evaluation are query standardization, i. e., the transformation into some kind of canonical form, and query simplification, mainly by eliminating redundancies and logical transformations of selection predicates. These steps should be done before algebraic optimization takes place. However, because we do not explicitly elaborate on them, they appear at the end of this section.

$$
\begin{gathered}
e_1 = \Big[\beta_{\$dept:\phi_{//departement[location=\text{"}KL\text{"}]}}(S(\text{"}dept.xml\text{"})) \\
\bowtie_{[\phi_{@depnr(\$dept)}=\phi_{@depnr(\$emp)}]} \\
\beta_{\$emp:\phi_{//employee[salary>50000]}}(S(\text{"}emp.xml\text{"}))\Big] \\
\\
C_{result}(C_{person,\phi_{name}(\$emp)\circ\phi_{salary}(\$emp)}(e_1))
\end{gathered}
$$

Figure 3. Optimized algebraic expression

2.3 The Physical Access Model

At the next lower level of abstraction, system-specific issues become visible. Each operator of the logical algebra can be composed of one or more physical operators. Those operators embody a specific evaluation algorithm that possibly relies on the existence of indices, document structure, and element order. The overall goal during this step of optimization—called query transformation or non-algebraic optimization—is a query execution plan (QEP) for which appropriate physical operators have to be chosen and subsequently arranged in a sequential manner. As in the relational case, we expect the various system-dependent parameters to span a large search space for possible

QEPs. An approach towards an *optimal* QEP might look as follows: an enumerator lists all “interesting” QEPs according to a numbering strategy (fully enumerative, limited enumerative, random). Then, the costs of each QEP are estimated using a tailored cost model. Finally, the cheapest plan is chosen and evaluated. Because of the possible scale of the search space, it is advisable to interleave the numbering of a plan and its cost estimation. Furthermore, a well adjusted plan may serve as input for the next QEP candidate to be enumerated.

In Figure 3 two critical parts of the non-algebraic optimization can be identified: the path expressions and the join operator. Because XPath is a little bit older than XQuery[3], it has been the subject of more intense research. Several evaluation strategies can be pointed out:

- **Pure algorithmic evaluation** [6]. This approach of Gottlob et. al. relies on a dynamic programming technique avoiding duplicate nodes, which may be produced by most axes[4] and, thus, may cause the repeated evaluation of certain path steps. The presented algorithm is the basis for a complexity estimation which reveals that XPath may be evaluated in combined complexity of $O(|D|^4 * |Q|^2)$, where $|D|$ is the size of the input document and $|Q|$ the size of the query. This strategy may be chosen, when no indices are present. However, it remains to be explored whether or not the presented algorithm can benefit from index support.

- **Index-based evaluation** [8]. The *T-Index* tries to evaluate a path expression at once, i. e., without further decomposition. Because path expressions can extremely vary and it seems utopian to support each type of expression by a single index, the appearance of an expression has to be limited using path templates. Only those queries that match such templates can be evaluated using the associated index. The knowledge of which kinds of expression have to be supported may either be derived from the structure of the underlying documents (using heuristics) or has to be provided from outside (e. g., through the database administrator). Another possible application area could be the use of ad-hoc indices.

- **Further algebraic techniques** [9]. This approach is listed for completeness only, because it actually belongs to the logical access model. As part of the mapping of a query expression to an algebraic equivalence, all path expressions are also decomposed into operators which can be considered under non-algebraic optimization.

Creating indices for a class of path expressions seems to be too restrictive. Therefore, we believe that a combination of all three strategies results in a conceivable solution. XPath expressions should be included in the algebraic optimization and further processed keeping the findings of Gottlob et. al. in mind. To speed up certain common evaluation tasks, indices may be used.

In relational query processing, there are three major physical operator classes for the join operation: nested-loop-, sort-merge-, and hash-based algorithms. When considering a join in XQuery, order plays an important role, because XML documents are inherently ordered by their textual representation. For XQuery [2], the semantics of the join operator is defined by a nested-loop join, where the outer sequence defines the order of the result. Therefore, when a sort-merge join or hash join is used—both do not obey document order in general—or when the join order is altered to minimize the intermediary result size, a sort operator has to process the generated result sequence. We will explore whether there are further circumventions of this non-commutativity and whether their cost is lower than the quadratic bound of the nested-loop join.

2.4 Storage Model

A critical input for the QEP optimizer is the cost model, because it builds the foundation of the QEP rating. It has to include information about the following four issues: I/O costs where the (physical) page references should be counted rather than the (logical) node references; CPU costs reflecting the processor usage during query execution; storage costs for intermediary results and, finally, communication costs, which are especially relevant in a distributed system environment. Of course, the overall costs of a QEP can only be estimated, and because many different system properties as well as document-related statistics (meta-data) have to be taken into account, there will be a trade-off between cost model accuracy and meta-data maintenance overhead. If, e. g., the system gathers data about the names and occurrences of child nodes for each node in the document, rather than only the average number of children for that type of node, the selectivity estimation of a child axis in a path expression may be more accurate. However, when the document is modified, the more accurate information becomes obsolete much earlier, requiring a recalculation of the meta-data. For a reasonable cost model, relevant and “stable” meta-data items have

[3] XPath was released as a standard by the W3C in 1999 whereas the first working draft of XQuery was proposed in 2001.
[4] The parent axis is an example for this problem.

to be identified. Additionally, when no meta-data about a document is available, certain default values (based on experience) have to be derived from common cases and taken instead.

So far, several alternatives for DBS-supported storing of XML documents have been explored, from simple LOBs (*Large Objects*), over certain XML-to-relational mappings (*shredding*), as well as the use of object-oriented DBS, to specifically tailored (native) storage formats. Certainly, the costs for I/O and CPU usage heavily depend on the underlying storage model. The evaluation of the path expression `//departement[location="KL"]` from Figure 3 requires a possibly large number of physical page lookups, depending on how many pages have to be fetched to evaluate a child axis. In turn, this number depends on information contained in the specific storage model like, for example, the node numbering scheme and the storage layout of the document. Therefore, if the QEP evaluator is parameterized by the different cost models resulting from the specific storage models of heterogenous XDBS, we are able to optimize a query on behalf of each of those systems.

3 Conclusion

In this article, we introduced an approach to XML query evaluation, inspecting queries on four different layers of abstraction. For each layer, we highlighted several tasks:

- **The Language Model**. Among the proposed languages, we have chosen XQuery as the query language to support. Because of its complexity, a first step is the search for an “optimizable” sublanguage. Furthermore, a suitable internal representation has to be found.
- **The Logical Access Model**. Various XML query algebras have been proposed so far, but none has lead to a standard. A comparative survey of these algebras with respect to expressive power, underlying data model, and the set of equivalence rules has still to be done. For our own algebra, we have shown a brief example together with its algebraic optimization.
- **The Physical Access Model**. For each operator of the logical access model, an implementation has to be found that adheres to information about existing index structures, object orders, document structure, etc. Interchangeable physical operators lead to different QEPs that have to be rated using a cost model. Because path expressions and joins are frequent operations, their effective implementation is crucial.
- **The Storage Model**. Cost models are heavily influenced by the assumed storage model. Parameterizing the rating of QEPs by cost models enables query optimization on behalf of different XML database systems.

In the future, we will focus our work on the two lowermost layers. Therefore we will assume a limited algebra that handles only central XQuery constructs like path expressions, joins, etc. Regarding the physical access model, we will elaborate on effective operator implementation, index support, and optimal operator order. Furthermore, the creation of a QEP enumeration and rating framework, customizable by different cost models, is our aim.

References

[1] Michael P. Haustein. Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery. In *11. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW)*, 2005. `http://wwwdvs.informatik.uni-kl.de/pubs/papers/Hau04.BTW.html`.

[2] Denise Draper, Peter Frankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium (W3C), 2004. `http://www.w3.org/TR/xquery-semantics/`.

[3] Christian Mathis. Anwendungsprogrammierschnittstellen für XML-Datenbanksysteme. Master’s thesis, Kaiserslautern University of Technology, 2004. (german only).

[4] David Maier. Database Desiderata for an XML Query Language, 1998. `http://www.w3.org/TandS/QL/QL98/pp/maier.html`.

[5] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages 2004, Montreal Quebec*, 2004. `http://tcl.sfs.uni-tuebingen.de/~kepser/papers/EML2004Kepser01.pdf`.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, 2002.

[7] Bernhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen, Entwurfs- und Implementierungskonzepte*. Vieweg Verlag, 1995. (german only).

[8] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT ’99: Proceeding of the 7th International Conference on Database Theory*, pages 277–295. Springer-Verlag, 1999.

[9] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged Algebraic XPath Processing in Natix. Technical report, University of Mannheim, 2005.