Software AG’s Tamino XQuery Processor

Thorsten Fiebig
Software AG
Uhlandstr. 12
64297 Darmstadt, Germany
Thorsten.Fiebig@softwareag.com

Harald Schöning
Software AG
Uhlandstr. 12
64297 Darmstadt, Germany
Harald.Schoening@softwareag.com

ABSTRACT
This paper is an experience report of implementing an XQuery processor for the Software AG’s Tamino XML server. The Tamino XQuery processor has already proven its industrial strength in several customer projects. Its purpose is to provide a World Wide Web Consortium (W3C) compliant query interface for high performance data and text retrieval on large collections of XML documents. Besides its support of the August 2002 working draft of the W3C XML query specification it offers extra functionality like text retrieval queries and data modification via update statements. Based on a short introduction to Tamino we describe the architecture of the Tamino XQuery processor and show which query processing techniques are used. We further sketch the implemented query optimizations.

1. INTRODUCTION
Supporting data retrieval via a declarative XML query language is a major requirement for managing XML data. In order to provide a standard XML query language the World Wide Web Consortium (W3C) is developing XQuery [2, 8, 9], a language for querying XML. Essential features of XQuery are extracting XML data from various data sources, transforming and combining the extracted XML data and the restructuring of the transformed XML data.

Software AG’s Tamino XML server includes a native XML database system, which offers full database functionality including transaction processing, recovery and data retrieval. Native XML database systems are tailored to meet the needs of managing XML data in contrast to database systems with an arbitrary data model and a mere XML layer on top [10, 4]. In order to provide a highly efficient W3C-compliant user interface for data and text retrieval, the Tamino XML server features its own XQuery processor.

In this paper we give an overview of the Tamino XQuery processor. Based on a short overview of the Tamino XML server, we describe its query capabilities and its design and sketch the available query optimizations.

The query optimization is performed by an algebraic rewriter-based query optimizer. Its focus is on exploiting indexes and other efficient access paths to get a query execution plan that can be evaluated efficiently on large data collections. Up to now the optimizations are not cost-based. The XQuery execution engine of the processor provides standard operations like joining, sorting and scanning of data as well as XQuery specific operations.

2. THE TAMINO XML SERVER
Tamino is available on Windows and several flavors of UNIX (including Solaris, AIX, HP-UX, Linux). Besides the database functionality it supports several W3C standards like W3C XML Schema, WebDAV, XPath and XQuery to foster the development of applications fully deploying XML technology [10].

2.1 Interfaces to Tamino
The primary access to the Tamino XML server takes place via the Hypertext Transfer Protocol (HTTP). The HTTP functionality is provided by Web servers that have a X-Port plug-in to communicate with Tamino. The usage of HTTP in combination with Web servers ensures that Tamino can be accessed from any client that talks HTTP. In order to avoid communication overhead Tamino also provides an Web server less API for more efficient data exchange.

By default, documents are completely stored in Tamino. However, it is possible to integrate other data sources into the Tamino XML view. Tamino X-Node provides access to external relational databases as well as to Software AG’s Adabas. Data residing in these systems can be included in the XML documents delivered by Tamino XML Server. Also, information included in XML documents stored in Tamino can be propagated to the external systems. The second option providing openness in Tamino XML Server is a feature called Tamino X-Tension: Rather than storing an element, attribute or sub-tree of an XML document, it can be passed to a user-defined mapping function responsible for storing the data. It can access an external system, or can decide to store information at some other place in Tamino's data store. At retrieval time the corresponding mapping function is used to get the same part of the XML document.

2.2 XML Storage
A Tamino database consists of XML documents which are grouped into multiple so-called collections. Each document stored in Tamino resides in exactly one collection. A collection has an associated set of W3C XML Schema descriptions. In each schema description doctypes can be
defined using a TaminO-specific notation in the extensible area of W3C XML Schema [1]. A doctype identifies one of the global elements declared in a W3C XML Schema as the root element. Within a collection each document is stored as a member of exactly one doctype.

The root element of an XML document identifies the doctype. A consequence, within a collection there is a 1:1 relationship between the doctype and the root element type. In cases where an associated user-defined schema exists, TaminO validates incoming documents against the schema.

Besides XML documents TaminO XML Server can also store arbitrary objects (non-XML objects) such as images, sound-files and MS Word documents. TaminO assigns an identifier, called a-id to each document or non-XML object. In addition, the user can specify a name for a document or object. This name must be unique within a doctype and can be used for directly addressing via a Universal Resource Locator (URL).

2.3 XML Indexing

For efficient data retrieval TaminO maintains three types of indexes that are updated whenever documents are stored, modified or deleted. The standard index is a value-based index. It serves for a fast document lookup when searching for elements or attributes having certain values. A standard index defined on an element or attribute indexes documents according to their values. Standard indexes are type-aware: indexes on numerical values support numerical order (i.e. 5 < 10), indexes on values of a textual data type are ordered lexicographically ("5" > "10") using customizable collations.

Text indexes are needed for efficient text retrieval. They provide a full-text index on elements or attributes and support text retrieval operations like searching for elements that contain particular words or phrases. Also, operations for searching adjacent or nearby text [11] can be evaluated via the text index. Like the standard index it supports document lookup. We are currently improving the standard and text index to support the lookup of elements and attributes.

The structure index keeps information about all paths occurring in the XML documents of a doctype. It comes in two flavors: "condensed" and "full". The condensed structure index only keeps track of all paths occurring in one doctype. This kind of structure index can be used for query rewriting. The full structure index stores the paths occurring in a document and allows the fast search of documents that contain a certain path.

3. THE TAMINO XQUERY LANGUAGE

The query language supported by the TaminO XQuery processor is a subset of the XQuery language specified in [2]. The subset is tailored to efficient data retrieval in the TaminO XML server. On the other hand the TaminO implementation offers some extra functionality in form of a library of text retrieval functions and update expressions for performing data manipulation.

3.1 Following the Working Draft

Because the XQuery working draft is still a moving target, Software AG has decided to follow the working draft carefully. Only those concepts are implemented that have been classified to be unlikely to change until the working draft has reached Recommendation status. This reduces the risk that customer applications use features that will be dropped from the XQuery draft. Another reason for focusing on a stable subset relevant for data and text retrieval is not to waste resources for implementing XQuery features which are removed from the next working draft.

The implemented subset includes path expressions, FLWR expressions, sort expressions, constructors and several built-in functions [8]. The supported functions constitute a comprehensive set for processing numeric and string values. The TaminO XQuery processor allows the arbitrary combination of the supported expressions. This makes it possible to state complex queries for joining data, determine aggregates and formulating complex XML constructions.

3.2 Text-Retrieval

TaminO's library of built-in functions contains text retrieval functions which support a wide range of basic full text search operations. The text retrieval functions also provide text searching based on phonetic similarities, stemming and facilitates highlighting of search results [11]. The text-search functions accepts a node and returns a boolean value that indicates whether the string-value of the node matches the given search string. The example query shown in figure 1 searches for books that have the word "XML" in their title.

```
for $b in input/()//@book
where tf:textContains($b/title,'XML')
return $b
```

Figure 1: XML query that finds books with the word "XML" in their title.

The text search predicate specified in the where clause calls the textContains function which searches the string value of title element for the string "XML". Since the text retrieval library has its own namespace, the call of textContains function refers to the namespace of the library which has to be declared in the query prolog.

3.3 Update

In order to state declarative data manipulation TaminO offers an extension of the XQuery language. The extension is based on the work described in [7, 12]. The possible manipulations include deleting, replacing and renaming of nodes in an XML tree. Also newly constructed nodes can be inserted via node-level updates. Figure 2 shows a For-Let-Where-Do (FLWD) expression that decreases the price of all books published before 1990.

```
update
for $b in input/()//@book
let $p := $b/price
where $b/Year < 1990
do replace $p with <price>{$p * 0.9}</price>
```

Figure 2: Update statement that decreases the prices of books published before 1990.

The for clause of the FLWD expression iterates over all book elements and the where clause filters the book elements by returning the "old" ones. The do clause contains the update operation that decreases the price for each old book via a replacement operation.
4. THE XQUERY PROCESSOR

The architecture of the Tamino XQuery processor which is shown in figure 3 reflects the different phases of query processing. A query is passed to the XML Tamino server. Here the query parser builds an abstract syntax tree which is the input to the preprocessor. The preprocessor performs the semantic checking and the query normalization. It also annotates the abstract syntax tree with type information needed by the query optimizer. The result is an annotated abstract syntax tree which is passed to the optimizer. The optimizer translates it into an algebraic expression, which is optimized by applying algebraic rewrite rules and by introducing index access operations. The optimized algebraic expression is translated by the code generator into the final query execution plan. Its execution yields the query result that is returned to the user.

These are bound by auxiliary let clauses to the replaced expressions. In a further step boolean expressions are transformed into conjunctive normal form. Let us illustrate this process by an example. Figure 4 shows a simple example query that retrieves the titles of books written in 1996. The extracted titles are wrapped by book elements and returned to the user.

```
for $b in input(/bib/book)
  where $b/genre = '1996'
return
  <book>
    {$b/title }
  </book>
```

Figure 4: XML query that finds the title of books written in 1996.

Figure 5 shows the result of applying normalization: Each path expression in the where and the return clause has been replaced by a variable defined in an auxiliary let clause.

```
for $b in input(/bib/book)
  let $t := $b/title
  let $y := $b/year
  where $y = 1996
return
  <book>
    {$t }
  </book>
```

Figure 5: Normalized query.

4.1 Query Normalization

The purpose of query normalization is to factorize common subexpression and to yield a normal form that simplifies the subsequent query processing steps. Static type analysis and query optimization require static namespace expansion. The preprocessor computes the QNames of element, attributes and function names used in the given query, by resolving namespace prefixes. The correlation between prefixes and the Universal Resource Identifier (URI) of a namespace is defined by a namespace declarations in the query prolog or in an element constructor. Namespaces can only be expanded if the prefixes can be resolved statically.

In the next step of normalization any given XQuery expression is transformed into a FLWR expression. A FLWR expression is normalized by replacing complex subexpressions in the where and the return clauses with variables.

4.1.1 Static Type Analysis

The type analysis performed by the XQuery processor is based on the fact that the XQuery language is strongly typed meaning that every expression has an associated type [2]. Static typing covers those areas of typing and type checking of expressions that can be performed during query compilation without analyzing any data stored in Tamino. Typing and type checking at query execution time is called dynamic typing. The purpose of the static typing is to detect typing problems as early as possible and to provide typing information that can be exploited by the optimizer. The type system of the Tamino XQuery processor contains the basic XQuery types and the types defined in the schemas of the queried doctypes. A type defined in a schema might be annotated with the Tamino specific schema annotations, which includes information about associated indexes.

During the static typing the preprocessor assigns a type to each subexpression of a given query and to the query itself. Expression types are derived from the types of literal constants, operator and function signatures. Path expressions that extract data from documents stored in Tamino are typed according to the schema of the document's doctype. The type of a complex expression, which is an expression consisting of subexpression, is derived from the type of these subexpressions.

Certain language constructs prohibit the specification of a unique type for every kind of expression. Examples are path expressions with descendant-or-self navigation axes or wildcard name tests. If a unique type cannot be determined the expression is annotated with a more general type and the specific type has to be determined and checked dynamically.

---

Figure 3: The Tamino XQuery processor architecture.
While combining the types of subexpressions the preprocessor checks for typing problems. In contrast to the static typing concept of the XQuery working draft, the preprocessor only issues a static type error if it detects a typing problem that will inevitably cause a type error during query execution. This means a typing problem that might not result in a dynamic type error will not be reported as a static type error.

4.2 The Optimizer Algebra

The optimizer of the Tamino XQuery processor uses an algebraic approach. From the observation that the clauses of an FLWR expression consume and produce ordered tuple lists containing variable bindings, an algebra seems to be appropriate for XQuery processing that acts on ordered tuple lists.

To deal with ordered tuple list the data model of the algebra provides table structures. A table structure is an ordered bag of tuples with an unordered set of typed and named variables. The bindings of the variables are taken from the domains of the data types defined by the XQuery data model [9]. This means a variable of a table structure can be bound to a node or a simple typed value or sequence of these. The variable set of the tuples is unique and defines the schema of the table structure.

The operators of the algebra maintain the order of the input table structures. The operator set contains basic operators for manipulating table structures. Some of the basic operators are already known from the processing of relational queries [3] like Select and Project. From the object-oriented context operators are used for evaluating functions (Apply operator) and dependent subexpressions (Disjoin operator) [3]. A Sort operator is available to create a new order on a table structure. Operators for nesting tables structures (Nested operator) and for unnesting sequence valued attributes (Unnest operator) complete the basic operator set.

XML-specific functionality is provided by a Path operator that navigates XML data and a Construct operator that constructs XML data from variable bindings stored in a table structure. For accessing data stored in Tamino the operator set contains Xscan operators that can access collections, doctypes, and indexes.

The path operator is provided with the script that parameterizes the function. The script parameter may refer to variables, contain path expressions, XML constructors or function calls. The set of available functions includes the supported XQuery and test retrieval functions. It further contains functions for type conversion and handling of XQuery data model instances. Other basic operators like atomization, determining the effective boolean value and comparisons are also realized by functions.

Let us illustrate the algebra by an example. Figure 6 gives the algebraic expression that results from translating the normalized query shown in figure 5.

```
Project[$S](
  Construct[$R:book[$S]]
)(
  Select[$R:1996](
    Path[$R:$/title](
      Path[$R:$/year](
        Path[$R:$/bib/book](
          DoctypeScan[3d:`bib`''])))
)
```

Figure 6: Algebraic expression.

The DoctypeScan operator scans the documents in the doctype bib and binds their document nodes to the variable $d$. The following Path operator extracts the book elements from the scanned documents and binds them to the variable $b$. The subscript of a Path operator consists of the output variable and a restricted XQuery path expression. The restriction disallows predicate qualifiers and general steps. The first step of the restricted path expressions refers to the input variable which holds a node sequence. A Path operator evaluates the path expression for each tuple of the input table structure and binds the output variable to the resulting node sequences.

The next two Path operators extract from each book element the year attribute and the title element. The result of the DoctypeScan and the Path operators is a table structure with the variables $d$, $b$, $y$ and $t$. It is filtered by a Select operator by evaluating the filter predicate of its subscript on each input tuple. Only those tuples are passed that have a $y$ variable that is bound to a year attribute with a value equal to 1996. The filter predicate of a Select operator can be an arbitrary boolean XQuery expression. Due to normalization a filter predicate does not contain neither path expressions nor FLWR expressions or constructors.

The filtered table structure is consumed by a Construct operator that creates a book element for each input tuple. Its content is taken from the binding of variable $t$. The constructed elements are bound to the variable $S$. The final Project operator restricts the variable set to variable $S$. In contrast to a relational algebra the Project operator does not perform any implicit duplicate elimination. It has to be stated explicitly via an duplicate eliminating Sort operator.

5. QUERY OPTIMIZATIONS

In order to get highly efficient query execution plans algebraic expressions are optimized by applying rule-based rewriting and by introducing index-accessing operators.

5.1 Algebraic Query Optimizations

The rule based rewriting simplifies algebraic expressions and strives for a cost reduction by applying heuristic rewrites. Rules and heuristics are inspired by the algebraic query optimization of relational and object-oriented queries [3]. In contrast to relational queries the order of tuples returned by an algebraic expression is significant, which complicates the design of rewrite rules.

The heuristics includes the pushing of filter (Select) operators to reduce the size of intermediate results as early as possible. Prerequisite for pushing filter operations is the transformation of filter predicates in conjunctive normal form which is also done by the optimizer. The XQuery not() function is difficult when normalizing filter predicates, since it has a different semantic than the boolean not operator. The pushing of Project operations also reduces the intermediate result size by reducing the number of variables in table structures. This reduction may also result in the elimination of unnecessary expressions, for example unnecessary function calls.

Further rewriting is targeted at the elimination of sequence-valued variables, which is done by pushing Unnest operators. The problem is here that due to the significant tuple order the Unnest operator is not commutative to itself. The set of rewrite rules also contains rules for basic unnesting of nested expressions.
Several customer scenarios have shown that accessing items at certain positions in a query result is an important requirement. An example of a so-called position range query is shown in figure 7 which retrieves the last 5 books in the bib doc-type.

\[ \text{input}() / \text{bib/book}[\text{position()} > \text{last()} - 5] \]

**Figure 7:** Position range query that finds the last 5 books in the bib doc-type.

For the optimization of position range queries the operator set of the algebra contains a Range operator. It retrieves a range of tuples from an input table structure depending on their position. The position range is specified by an upper and a lower bound position. The optimizer tries to push the Range operators down the algebraic expressions. Figure 8 shows the translated position range query.

\[ \text{Range}[\text{last} - 4, \text{last}]( \text{Project}[\text{Book}]( \text{Path}[\text{Book}/\text{bib/book}]( \text{DoctypeScan}[\text{Bib} / ' \text{bib'}]))) \]

**Figure 8:** Algebraic expression for evaluating a position range query.

Assuming that each document in the given doc-type contains exactly one book element, the Range operator can be pushed over the Path operator. Since the scan operators support position range queries Range operators can be merged into scan operators. The optimized query is shown in figure 9.

\[ \text{Project}[\text{Book}]( \text{Path}[\text{Book}/\text{bib/book}]( \text{DoctypeScan}[\text{Bib} / ' \text{bib'}'; (\text{last} - 4, \text{last}))) \]

**Figure 9:** Optimized algebraic expression for evaluating a position range query.

The Range operator cannot be pushed over a Path operator that holds a path expression selecting optional elements or attributes. Furthermore, operators that change the order or reduce the cardinality of their input table structure like the Sort, Select and Nest also avoid the pushing of Range operators.

The optimization of a Range followed by an Unnest operator is possible by cloning the Range operator and pushing one of the clones over the Unnest. Cloning is possible if the lower bound of the position range refers to the first or the upper bound refers to the last position.

### 5.2 Introducing Index-Access Operators

Accessing indexes is the prerequisite for efficient query execution on large data sets. Especially scanning large collections of XML documents is expensive. Therefore the optimizer’s focus is on selecting efficient access paths to avoid reading documents from disk. The scanning of a collection or a doc-type is replaced whenever possible by an index-accessing scan operator. The Tamino XQuery processor exploits the standard, text and index index.

The major problem the optimizer has to cope with, is a granularity mismatch. XQuery filter predicates affect nodes but the Tamino indexes only provide access to documents. This means the Tamino indexes can be used to determine documents containing nodes with certain properties but not for retrieving nodes. The advantage of the coarse index granularity is the limited size of the indexes.

For the index-based evaluation of filter predicates the granularity mismatch problem is resolved by introducing approximate index predicates. The approximate index predicate of a given filter predicate \( p \) checks documents for the existence of nodes satisfying \( p \). Therefore it holds the predicate \( p \) and the path expression specifying the nodes that are filtered by \( p \).

For each filter predicate in a given algebraic expression the optimizer tries to create an approximate index predicate. The created predicates are added to the expression via auxiliary Select operators. Figure 10 shows an algebraic expression with an approximate index predicate that accesses a standard index defined on the path /bib/book/\$year. The approximate index predicate looks for documents containing nodes reachable by the path /bib/book/\$year and holding the value 1996.

\[ \text{Project}[\text{Book}]( \text{Construct}[	ext{Book}/\text{bib/book}/\$year](( \text{Select}[(\text{Book}/\text{bib/book}/\$year >; 1996)]( \text{Path}[\text{Book}/\text{bib/book}/\$year]) \text{DoctypeScan}[	ext{Bib} / ' \text{bib'}']))) \]

**Figure 10:** Algebraic expression with approximate index access predicate.

The approximate index predicate refers to the scan variable \$d\$ that refers to the scan operator which reads the documents from which the bindings of \$by\$ are generated. The approximate index predicate can be pushed down the algebraic expression by applying rewrite rules for pushing selections. Since it does not refer to any variable that is bound to an element or attribute, it can be pushed until it reaches the DoctypeScan that creates the bindings for \$d\$.

If an approximate index predicate reaches a DoctypeScan, the operator is replaced by an index accessing operator. If it reaches an IdrsScan operator it is added to the predicate of the operator. Figure 11 shows the algebraic expression resulting from pushing the approximate index predicate and replacing the DoctypeScan in the algebraic expression depicted in figure 10.

\[ \text{Project}[\text{Book}]( \text{Construct}[	ext{Book}/\text{bib/book}/\$year](( \text{Select}[(\text{Book}/\text{bib/book}/\$year >; 1996)]( \text{Path}[\text{Book}/\text{bib/book}/\$year]) \text{IdrsScan}[	ext{Bib} / ' \text{bib'}'; \text{bib/book}/\$year >; 1996]))) \]

**Figure 11:** Algebraic expression with standard index accessing operator.

The IdrsScan delivers the document nodes of the documents satisfying the given predicates. Assuming each document contains exactly one book element with a year attribute the table structure does not need to be filtered by the Select operator. After removing the Select operator and pushing the Project operator the optimizer yields the final
6. THE QUERY EXECUTION ENGINE

The query execution engine (QEE) evaluates query execution plans (Qeps). It provides efficient implementations of the operators and functions of the optimizer algebra and of the instances of the XQuery data model. It also provides the infrastructure for accessing XML data and indexes.

A QEE generated by the Code Generator consists of an operator tree. The operators are implemented according to the iterator approach model [6]. That means the QEE operators produce and consume streams of QTE tuples. This helps to save resources for maintaining intermediate results and enables iterative evaluation of expressions.

The operator subscripts holding QTE functions and QTE path expressions are evaluated on single QTE tuples. The QTE functions can be combined to a tree of functions to build a complex expression. Besides functions for implementing the built-in functions, the QTE function set contains functions for performing dynamic type checking and type conversion.

The construction of XML instances is done via QTE constructor functions. The query result is passed via “side effects” [6] from the query execution engine to a response handler that returns the query result to the user. This means the query result is passed via QTE functions that call callback functions of the extended SAX interface of the response handler.

For certain functions the QEE provides different variants for processing a single item or sequence-valued input. This saves the computation overhead resulting from maintaining single items in a sequence. The decision when to use an item or sequence processing variant of QTE function is made by the code generator.

Despite the optimizers capabilities of avoiding sequence valued variables the QEE has to cope with them. In order to provide efficient processing sequences are lazily constructed. Furthermore, sequence-accessing functions and operators are optimized not to read more items from a sequence than necessary.

The data storage and the index management is done by the Tamino’s Data Store. A document manager maintains an XML tree for each XML document accessed during query execution. The XML tree of an XML document corresponds to the document’s XQuery data model representation. In order to provide efficient access to the XML trees and to limit the main memory resources needed for document management, the document manager provides full caching functionality.

7. CONCLUSION AND FUTURE WORK

This paper has given a brief overview of the Tamino XQuery processor. The main purpose of the Tamino XQuery processor is to provide a highly efficient W3C-compliant query interface on large collections of XML data. The next planned steps in the development of Tamino’s XQuery processor are to add the missing XQuery features and to improve the query optimization. Besides more pragmatic optimization issues like query pre-compilation, the most important issue is the introduction of cost-based optimization.

8. REFERENCES