Paper ID: 258

Title: STEP: Extending Relational Query Engines for Efficient XML Query Processing

Authors: Feng Tian, David J. DeWitt

Topic Area: Core Database Technology

Category: Research

Subject Area:
Semi-Structured data, XML (Primary)
Optimization and performance

Contact Author:
Feng Tian
Department of Computer Science
1210 W. Dayton
Madison, WI, 53706
USA
ftian@cs.wisc.edu

STEP[1]: Extending Relational Query Engines for Efficient XML Query Processing

Feng Tian David J. DeWitt

Department of Computer Science
University of Wisconsin, Madison
Madison, WI, 53706
USA
{ftian, dewitt}@cs.wisc.edu

Abstract

One approach to building an efficient XML query processor is to use a relational database system to store and query XML documents. However, XQuery contains a number of features that are either hard to translate into SQL or for which the resulting SQL is complex and inefficient. In this paper, we propose an extension to the relational database systems that facilitates efficient XML query processing within the existing relational database execution framework. Our extension can evaluate complex queries against XML documents stored in a relational database system by processing a sequence of XML events by extending the relational query engine with an event sequence-processing engine. Results from each engine can be streamed into the other without incurring the overhead of crossing a middleware-database boundary.

1. Introduction

XML is the W3C standard for data representation and data exchange over the Internet. Together with XML Schema (or DTDs), XML can be used as a mark-up language for semi-structured data. The W3C has established an XML Query working group to define standards whose goal is to permit "collections of XML files to be accessed like databases". The XML Query working group has already released a query language (XQuery [26]), data model (XQuery 1.0 and XPath 2.0 Data Model [27]) and query algebra (XQuery Formal Semantics [28]). How to build an efficient XML query processor is still, however, largely an open problem.

One approach is to use a relational database system to store and query XML documents. This approach is very attractive because relational system is a mature technology with proven reliability and scalability after decades of research and development in storage management, query processing, and query optimization. Re-engineering similar technology for XML query processing would be extremely costly both in terms of time and money (as the object-oriented database system vendors discovered). Another advantage of using relational database systems is that large amounts of existing data are already managed by relational database systems. Thus, using a relational DBMS offers the possibility of integrating XML files with relational data in one database system.

There are many active projects exploring the use of relational database systems to support XML query processing. A number of these are implemented as middleware. A query written in XQuery is first translated into SQL and then sent to a relational DBMS for execution. The result table is restructured and tagged according to the original XQuery. In order to exploit the query processing power of the relational DBMS, the middleware attempts to push most of the query processing into the relational DBMS and makes the restructuring and tagging part only a thin wrapper over result table from the SQL query. This architecture is explored by, for example, XPERANTO [4][18]. Figure 1.1 shows an overview of middleware-based approach.

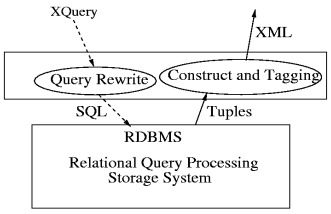


Figure 1.1: Middleware-based architecture

Certain XQuery features are difficult to support using this approach. First, XQuery has features that are usually only available to a general-purpose language (e.g. local variables, instance of operator, non-linear recursion) which are not available in SQL. Second, some XQuery

[1] STEP stands for STreaming Event Processing.

queries are hard to translate into SQL and the resulting SQL queries are often complex and inefficient. As a motivating example, consider the following XQuery.

```
For $v1 in document("a.xml")/a/b/c/d/e/f/g,
Let $v2 = Filter(document("b.xml") // (x | y | text()) ) /*/*
Where $v1 == $v2
Return $v1, $v2
```

Several problems are encountered when a middleware-based architecture is used to execute this query. First, a long path expression is used to select $v1. When XML documents are stored in a relational database, elements are stored as tuples in multiple relational tables [7][19] and steps in the path expression are translated into joins. A long XPath expression is likely to be translated into many joins. Second, if *x* and *y* elements can be nested arbitrarily in document "b.xml", the relational DBMS will need to employ recursive query processing in order to compute the filter operator of $v2. SQL queries on XML document structures such as those used for computing $v1 or $v2 are complex, hard to optimize, and inefficient. Figure 1.2 illustrates how this query would likely be executed in a middleware-based architecture.

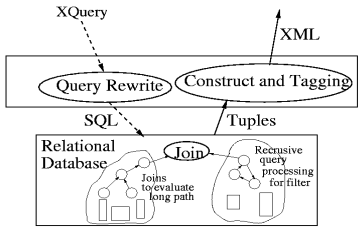


Figure 1.2: Query execution of the motivating example in middleware-based approach

It is well known that regular (XPath) expressions like $v1 or $v2 can be evaluated using a finite state machine with only *one* sequential scan of the original document. Therefore, it is reasonable to implement some sequence-processing capability in the middleware layer instead of using relational operators to evaluate complex path expressions. However, evaluating $v1 and $v2 in middleware leads to another problem -- $v1 and $v2 need to be joined. While one could add a join operator to the middleware layer, this is not a trivial task and ends up duplicating functionality already provided by the relational DBMS. Alternatively, the results of evaluating $v1 and $v2 can be sent *back* to the relational DBMS for further processing.

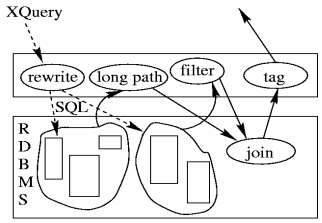


Figure 1.3: Execution path of the motivating query when complex XPath evaluated in middleware

Figure 1.3 shows the execution flow for our example query if the path expression and filter are evaluated in a middleware layer but the relational DBMS is used to execute the join of $v1 and $v2. This strategy incurs a number of inefficiencies. First, query execution crosses the middleware/relational boundary multiple times, incurring significant overhead for binding out tuples and returning the results of the filter and path expression operations. Second, user-defined functions that return tables are often second-class citizens in relational database systems. If temporary tables are used, there will be unnecessary I/O or blocking. Third, the middleware layer needs to make decisions on what parts of a query are to be issued to the relational DBMS. This decision is usually heuristic based, often without help from the relational optimizer. Another problem is there is no mechanism for the middleware layer to provide hints to the relational optimizer to help it select a good join algorithm.

In this paper we describe STEP which addresses these inefficiencies by adding a new class of operators termed **S-Operators** to the relational DBMS engine. S-Operators process streams of XML events using an event sequence data model. The power of an S-Operator is that it can evaluate a complex regular path expression with one sequential scan of the input event stream, avoiding the use of multiple join operators. A second new type of operator, termed a **Converter**, is used to convert an XML event stream into streams of relational tuples and vice versa. Thus, S-operators and relational operators can be composed with one another in a seamless fashion.

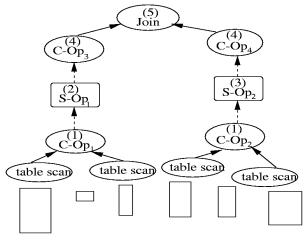


Figure 1.4: STEP plan for motivating XQuery

Figure 1.4 shows how our example would be executed by STEP. As the input tables are scanned, Converters (C-Op1 and C-Op2) are used to convert tuple streams into event streams that are consumed by two S-operators (S-Op1 and S-Op2) to evaluate the path expression and the filter. The outputs of the two S-operators are then converted back into tuple streams using two more Converters (C-Op3 and C-Op4). The outputs of the Converters are then joined with one another using a standard relational join operator. Our S-Operator implementation reads each input sequence once and requires only a small amount of memory for its operation. The use of S-Operators eliminates the need for complex joins to evaluate path expression and the use of recursive query processing for processing the filter. Compared to the execution plan in Figure 1.2 and 1.3, the STEP plan is

both more efficient and more scalable. In addition, STEP avoids the overhead of repeatedly transferring data across middleware/database system boundary.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the event sequence data model and Section 4 describes the implementation of our extension in the Predator [14] relational database system. Experiments and performance results are shown in Section 5. Section 6 concludes and describes future work.

2. Related Work

For a number of years database researchers have realized that the relational model is not ideal for handling sequence data [20][21]. [9] uses a finite state machine to process history events in order to check trigger conditions. [22] considers sequence query processing in the temporal database environment. The event-sequence model used by our S-Operator was inspired by these earlier efforts.

Recently a large number of papers have been written on processing XML queries using relational database systems. [2][5][7][19][23] explore ways of storing XML data in a relational database system. Our implementation uses the approach described in [19]. [4] and [8] discuss how to publish relational data efficiently. [12] describes how to translate queries in XQuery into SQL, presenting several examples of queries for which no translation is possible.

Our work differs from these early efforts in that we propose an entirely new paradigm for processing XML queries that combines traditional relational operators operating on streams of tuples with a new class of operators operating on sequences of XML events. This approach is simple to implement, and as we will demonstrate below, provides far superior performance to a pure relational approach. Our extensions are capable of evaluating complex path expression and complex operators like filter, relieving the relational engine from having to execute complex, inefficient plans.

[11] considers evaluating regular XPath using finite state machine over streaming XML data. Our implementation of S-Operator is very similar to their implementation of the xscan operator. Their technique is standalone and the input stream of xscan is generated by parsing XML files. STEP extends the standard relational database system functionality with the ability to query XML document structures. S-Operators operate on event streams constructed from the results of SQL queries as well as event streams generated by an XML parser.

3. The XML Event Sequence Model

In this section, we describe the XML event sequence model used in STEP. The following XML file illustrates how XML documents are stored in a relational database and how XML events are processed by STEP. Figure 3.1 and Figure 3.2 show the example DTD and XML file. Each tag, text data, or end tag in the XML document is assigned a sequence number, as shown in Figure 3.2.

```
<!ELEMENT items (item*)>
<!ELEMENT item (name, color, description)>
        <!ATTLIST item id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT color (#PCDATA)>
<!ELEMENT description
        (#PCDATA|bold|emph)*>
<!ELEMENT bold (#PCDATA|bold|emph)*>
<!ELEMENT emph (#PCDATA|bold|emph)*>
```

Figure 3.1 Example DTD

```
<?xml version="1.0"?>
<items(1)>
<item id="i1"(2)><name(3)>Item1(4)</name(5)>
<color(6)>red(7)</color(8)>
<description(9)><emph(10)>
<bold(11)>bold emph(12)</bold(13)>
</emph(14)> is strong(15) </description(16)>
</item(17)>
<item id="i2"(18)>
<name(19)>Item2(20)</name(21)>
<color(22)>red(23)</color(24)><description(25)>
<emph(26)>
<emph(27)>double emph(28)</emph(29)>
</emph(30)> is stronger(31)</description(32)>
</item(33)>
</items(34)>
```

Figure 3.2 Example XML file

STEP stores XML data in a relational database using the techniques described in [19]. The main idea in [19] is that whenever an element can appear only once in its parent, the element is stored in the same table with its parent (inlining). Elements or TEXT data that can appear multiple times are stored in separate tables. The sequence numbers in Figure 3.1 are stored as *xmlid* and *endid* in order to remember an element's position in the original document. Another column *xmlpid* is used to link an element to its parent. The relational schema and tables for

xmlid	xmlpid	endid
1	null	34

Table items

xmlid	xmlpid	endid
11	10	13

Table bold

xmlid	xmlpid	endid
10	9	14
26	25	30
27	26	29

Table emph

xmlid	xmlpid	Data
12	11	"bold emph"
15	9	"is strong"
28	27	"double emph"
31	25	"is stronger"

Table PCDATA

xmlid	xmlpid	endid	id	name	name_xmlid	color	color_xmlid	description_xmlid	description_endid
2	1	17	"i1"	"Item1"	3	"red"	6	9	16
18	1	33	"i2"	"Item2"	19	"red"	22	25	32

Table item

Figure 3.3 Sample XML file stored in relational tables

the example in Figure 3.1 are shown in Figure 3.3

3.1 The Event Sequence Model

An alternative view of XML data is to treat the XML as a sequence of events. This viewpoint is taken for example, by the XML SAX [16] parser. Instead of treating an XML file as a tree, a SAX parser reports a sequence of events to the invoking application. From the application's point of view, the sample XML document is, in fact, a sequence of SAX events: "*start_element* items", "*start_element* item, with *attribute* id='i1'", "*start_element* name", "*text* item1", "*end_element* name" etc. Our definition of events is slightly different from that of the SAX parser which was not designed or optimized for database processing. For example, a SAX start element event may contain an arbitrary number of attributes, which is hard to fit into a pure relational model. Our definition borrows ideas from the SAX API and the sequence data type in [20]. Formally,

Definition: An **event** is a triple (type, name, value), written as *ev = (t:T, n:S, v:D)*. The domain of event type *T = {EV_STARTELE, EV_ENDELE, EV_ATTR, EV_TEXT, EV_ELETEXT, EV_WELLFORMED, EV_NULL}*. The event name, *ev.n*, is a string, which can be null. The event value, *ev.v*, can be any data type defined in the database.

Definition: An **event sequence** $S=\{ev_1, ev_2, \ldots\}$ is a mapping from positive integers to events where $S(i)=ev_i$. A sequence is **finite** if there exists integer *K*, such that *S(n).t = EV_NULL* if $n > K$. The minimum of such *K* is called the **length** of a sequence.

We consider only finite event sequences in this paper. Our definition of an event defers from that of a SAX event in the following ways:

1. Attribute events are distinguished from start element events, so that all events have a uniform structure.
2. An *EV_ELETEXT* type is used to represent a simple XML element like <A>a</A>. Since this case is likely to occur frequently, it is worthwhile compressing these three events into one.
3. We have an *EV_WELLFORMED* type which facilitates the efficient handling of large chunks of XML data (probably stored as LOBs in the relational database). The XML data LOB could be parsed during sequence processing when necessary.

Definition: An event sequence *S* is **well-formed** if

1. *S(1).t != EV_ATTR*
2. If *S(i).t = EV_ATTR*, then *S(i-1).t = EV_ATTR* or *S(i-1).t = EV_STARTELE*
3. The *EV_STARTELE* and *EV_ENDELE* pairs are perfectly matched, without interleaving.

The maximum nesting depth of *EV_STARTELE* and *EV_ENDELE* pairs is called the **depth** of the sequence.

Conditions 1 and 2 state that attribute events should immediately follow their start element events. Condition 3 states the usual requirement of matching start and end tags. We relaxed other XML well-formed requirements such as the existence of an unique root element.

With these notions of event and event sequence, we can now define operators on event sequences.

Definition: An **event sequence operator** takes one event sequence as input and generates another event sequence as output.

Note that this definition of an event sequence operator is very general. In the extreme case, any XQuery that takes an XML file as input and produces XML as output can be regarded as an event sequence operator. We need to restrict our focus to operators that are easy to implement and efficient to execute.

Definition: An event sequence operator is **streaming** if:

1. The operator uses a bounded amount of buffer space.
2. The operator scans the input sequence only once, and for each event in the input sequence, the operator requires a bounded amount of computation.

Streaming operators have the following composition property.

Proposition 1: Let Op_1, Op_2 be two streaming operators, then composition of these two operators $Op=Op_2 \circ Op_1$ is streaming.

Definition: An event sequence operator on a well-formed sequence is **D-streaming** if

1. The operator buffer size is *O(d)*, where *d* is the depth of the well-formed input sequence.
2. The operator scans the input sequence only once and, for each event in the input sequence, the operator requires a bounded amount of computation.

If the depth of the input sequence is not very large, a D-streaming operator consumes little memory and its computation cost is linear to the input sequence length. Similar to the streaming property which is preserved by composition, D-streaming operators have the following property,

Proposition 2: Let Op_1, Op_2 be two D-streaming operators. If the depth of $S_1=Op_1(S_0)$ is $O(d_0)$, where d_0 is the depth of S_0, then the composition operator $Op=Op_2 \circ Op_1$ is D-streaming.

Next, we introduce the S-Operator, which is implemented by STEP.

Definition: An **S-Operator** is a finite state machine, driven by events in its input sequence. Each S-Operator has a **context**, which contains one *state stack*, and some *local variables*. Upon each event, the S-Operator may execute a bounded number of following instructions,

1. State transition; optionally push last state onto *state stack*.
2. State transition to the state at the top of *state stack*, and pop the *state stack*.

3. Simple operation on local variables, such as assigning a value to a local variable, arithmetic operation, string concatenation, equality test.
4. Output an event to the output sequence.

An S-Operator is D-streaming if the state stack is pushed only on *EV_STARTELE* events, and popped the state stack is popped upon the corresponding *EV_ENDELE* events[2]. The S-Operators are very powerful in evaluating queries against the structure of an XML document. In particular, we have the following propositions,

Proposition 3: A regular XPath expression, consisting of child (including wildcard) and descendent, in which each step may contain predicates on the node's value or position (for example, a range operator), can be evaluated by a D-streaming S-Operator.

Proposition 4: If each filter expression of a filter operator, *filter(X // expressions)*, is a regular XPath expression as described in Proposition 3, the filter operator can be evaluated by a D-streaming S-Operator.

The S-Operator for evaluating a regular expression XPath is essentially a finite state machine that recognizes the regular expression. Well-known algorithms exist to construct such a finite state machine. For example, an S-Operator that evaluates "a/b/c" is shown in Figure 3.4[3]. Arrows denote state transition. Labels on each arrow denote events that trigger the transition. <*> is the "default" transition on any *START_ELE* event. '+' indicates that the destination state of the transition is pushed onto the state stack and '-' means when the state finishes processing the element (implemented by using a local variable to keep count on depth of nesting), it transits back to the state on the top of state stack and then the state stack is popped.

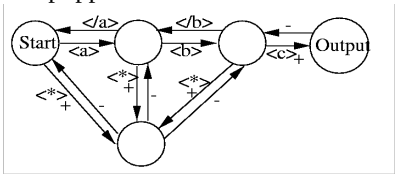


Figure 3.4 S-Operator for 'a/b/c'

The S-Operator for evaluating the filter operator is a finite state machine that recognizes the filter expressions. An S-Operator used to evaluate filter expressions outputs events in the same order as the input sequence. Because the structure information of XML data is represented by the nesting of <start-element, end-element> pairs, preserving the order of events in the input sequence ensures that the structural information is preserved in the output sequence.

3.2 Integrating Event Sequence Model with Relational Model

The event sequence model and the S-Operator framework provide a powerful processing mechanism for queries against XML document structures. However, there is mismatch between the event sequence model and the relational model. While ordinary relational operators process tuple streams, S-Operators process XML event streams. We introduce a new class of operators, called **Converters**, to bridge this mismatch. An **S-to-R Converter** takes one stream of events as its input and produces one or more tuple streams as its output. An **R-to-S Converter** consumes one or more tuple streams as input and produces one event stream as output. Converters are used to connect relational operators and S-Operators with one another. Details of Converters and how they are implemented in STEP are described in the next section.

4. STEP Implementation

We have implemented a prototype of the STEP architecture as an extension to the Predator object-relational database system [14]. The overall architecture of STEP is shown in Figure 4.1. Modules outside of the dashed polygon are off-the-shelf components from Predator.

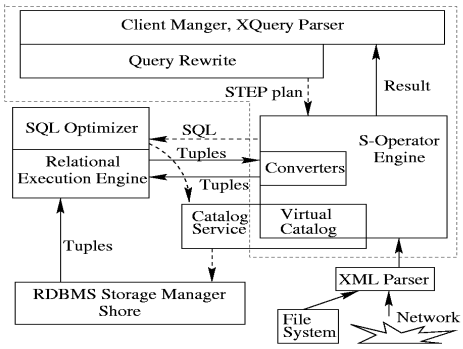


Figure 4.1 STEP Architecture

We first give an overview of each module, following the execution path of a query. XML data can be stored either in relational tables using techniques like those described in [19], as LOBs in a relational database, or as files. If stored in relational tables, the tables are clustered using *xmlid*, which is the order of the elements in the original XML document. After an XQuery is parsed, a global optimizer rewrites the query into a STEP query

[2] The context may include other stacks. An S-Operator will be D-streaming as long as each stack follows same the push-pop protocol as the state stack. Currently, STEP only uses the state stack.

[3] In many cases, the state stack is not necessary (thus making the S-Operator streaming), as the S-Operator is essentially a finite state machine that recognizes the regular expression. For example, the S-Operator in Figure 3.3 can be implemented using local variables to remember states that initiated the transition. However, in more complex situations, using the state stack could reduce the number of states required by the S-Operator. We use the state stack because we believe that the size of the stack will be small, and the S-Operator using the stack is much simpler.

plan. The most important function of the global optimizer is to divide the workload between the relational execution engine and the S-Operator engine. If the query involves XML data that is not stored in database tables, for example, XML files on the Internet, STEP uses an ordinary SAX parser to parse the XML file to produce an event stream. If the query is posed against data stored in relational tables, the S-Operator engine issues SQL queries to the relational engine. The results of SQL queries are converted to event streams using R-to-S Converters. The S-Operator engine augments the Predator catalog service with a virtual catalog module. When the SQL optimizer asks for catalog information about a table, the catalog service extracts information from the virtual catalog as well as from the ordinary relational catalog. The virtual catalog is used by an S-to-R Converter when forming a tuple stream as input to ordinary relational operators. Like the relational operators, the S-Operators and Converters provide a Volcano-like [10] iterator interface (GetNext tuple or GetNext event) to the operators above them.

The finite state machine implementation of an S-Operator is straightforward. Each state contains a list of actions (including state transitions) that are performed when a certain XML event is received in that state. The S-Operator keeps a context of information like current state, state stack etc. In the remainder of this section, the integration of the S-Operator engine and the relational engine is described in more detail.

4.1 R-to-S Converter

When the STEP engine needs to retrieve data from the database it issues one or more SQL queries. The resulting tuple streams are then converted to event streams using R-to-S Converters. STEP uses two classes of R-to-S Converters. The first class, trivial R-to-S Converter, converts each row of the SQL result *R(col1, col2, …)* to a sequence of events equivalent to one XML element <R col1="value1" col2="value2" …/>. Trivial R-to-S Converters convert query results from the relational database to event streams that are ready for grouping (for example, S-Operators to performing range queries) or tagging.

The second class of R-to-S Converters reconstructs the event sequence corresponding to the original (or part of the original) XML document. This class of operators uses a priority queue, implemented by a heap H. The heap H is initialized when the R-to-S Converter is initialized (algorithm Init in Figure 4.2). The GetNext cursor interface is implemented as algorithm GetNext in Figure 4.2.

The algorithm in Figure 4.2 has the following property that is similar to D-streaming:

Proposition 5: If the SQL queries used as inputs to an R-to-S converter return tuples sorted by *xmlid*, then the heap size of H is *O(d)*, where d is depth of nesting of original XML data.

In fact, the size of H is bounded by $d+t+e$, where t is the number of scanned relational tables and e is number of possible elements defined in the DTD.

```
Implementation of R-to-S Converter
H: Heap, implementing a priority queue
H.add(k, V): Add an entry to H, with k as heap value
H.getFrist(): Remove top of heap, and return V
Algorithm Init(List of SQLs):
    H = {}
    For each SQL statement:
        Issue SQL to RDBMS, Open Cursor
        Fetch one row r
        If fetch succeeded, H.add(r.xmlid, r)
Algorithm GetNext():
    V = H.getFirst()
    If V is NULL, return End of Stream
        If V is an Event, return V
        If V is a row from table PCDATA
            Fetch a row r from table PCDATA
            If fetch succeeded, H.add(r.xmlid, r)
            Return Event (EV_TEXT, null, V.data)
        If V is a row from other tables
            New an EV_STARTELE event E1 for the row
            H.add(r.xmlid, E1)
            New an EV_ENDELE event E2 for the row
            H.add(r.endid, E2)
            For all attributes, inlined elements in the row
                Generate corresponding events
                Add these events to H, according to
                        (xmlid, endid)
                Fetch a row r from the SQL cursor
                        where V comes from
                If fetch succeeded, H.add(r.xmlid, r)
                Return GetNext()
```

Figure 4.2 Algorithm of R-to-S Converter

By clustering relational tables using *xmlid* column, we can support complex XPath expressions by fetching tuples from the relational database, converting the resulting tuple streams into event streams, evaluating the XPath expression using an S-Operator, all in a streaming fashion.

The R-to-S Converter can also be used to retrieve only part of the original XML document. For example, in order to evaluate XPath "item//bold/text()", we only need to issue SQL queries that scan table ***Bold***, ***PCDATA***, and *xmlid, endid* columns of table ***Item***. The algorithm produces an event stream equivalent to the original document with irrelevant elements such as "*name*" and "*color*" filtered out.

4.2 S-to-R Converter

One important feature of STEP is that STEP can seamlessly connect S-Operators and relational operators with each other. Unlike the middleware-based approach, STEP can redirect results of an S-Operator into a relational operator by inserting S-to-R Converter before the relational operator. The implementation of S-to-R Converter is very similar to an S-Operator, replacing output events by output tuples. For example, Figure 4.3

shows an S-to-R Converter converting an Item element in our example document to four relational streams.

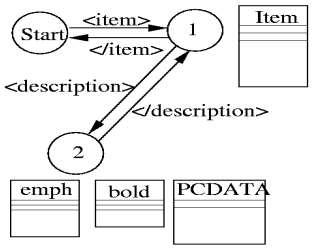


Figure 4.3 An S-to-R Converter

One important difference between an S-to-R Converter and an S-Operator is that the S-to-R Converter may have more than one output stream. This makes an S-to-R Converter non-streaming. For example, when a relational operator calls GetNext on one tuple stream, let's assume table ***emph*** in Figure 4.3, the S-to-R Converter may need to buffer many tuples for table ***bold*** or ***PCDATA*** using temporary files. An S-to-R Converter has the D-streaming property when it has only one output stream, which is often true when the S-to-R Converter is used to supply data stream for further relational processing. S-to-R Converters with multi output streams are primarily used when loading the database, which is discussed later in this section.

The S-Operator engine in STEP calls the relational engine through its SQL interface. Ordinary relational operators can also consume streams generated from S-to-R Converters, by using the virtual catalog. When a SQL query is optimized, the relational optimizer contacts the catalog service module for table schema and statistics. The Catalog service module tries to find this information from the standard relational catalog and from the virtual catalog information supplied by the S-Operator engine. The tuple streams generated from S-to-R Converters act the same as tables on disk – with table scan as the only supported access method.

4.3 XQuery rewriter

We briefly describe the query rewrite/optimization module of STEP in this subsection. Currently, this module is still under development.

We first give a brief description of the algebra used by STEP. We refer to [24] for a full description of the algebra, including the equivalent rules and rules for translating XQuery into the algebra. Besides ordinary relational operators such as selection, projection and join, the new algebra introduces several new operators. The unnest operator expands element nodes reachable via an XPath. The construct operator can create new nodes that form the query result. The group operator is used to form a collection (list or bag, depending on whether the collection is ordered) of nodes. Although filter is defined as a function in XQuery and can be written as combinations of other operators, it is treated as an operator in order to simplify the logical plan. As an example, the logical plan for the motivating query in the introduction is shown in figure 4.4

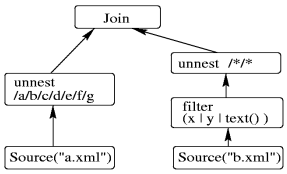


Figure 4.4 STEP logical plan for motivating query

After logical plan is rewritten using rules stated in [24] (for example, selection is pushed down), it is optimized into physical plans consisting of relational physical operators and S-Operators. While operators like selection, join are naturally executed by the relational engine, the primary targets for the S-Operator engine are,

1. Unnest operators with long, complex path expression.
2. Filter operators.
3. Operations on node list, like tagging or the RANGE operator.

In cases 1 and 2, a finite state machine constructed from the XPath expressions is used to evaluate the unnest or filter operator. In case 3, a trivial R-to-S Converter can convert the relational tuple stream to an event stream to form the input to an S-Operator for counting or tagging.

If the optimizer decides to use the S-Operator engine to evaluate both the filter operator and the unnest operator in the plan in Figure 4.4, it will produce the physical plan shown in Figure 1.4 (in the Introduction).

Some logical operators can be evaluated by both engines. In such case, a cost model will be used to select the better plan. As an example, consider the operator unnesting 'a/b/c/d/e/f/g', with elements for each step stored in separate relational tables. There are many possible plans, for example,

1. Join all tables using the relational engine.
2. Produce an event stream using a Converter, then evaluate the whole XPath using one S-Operator.
3. Mixed approach. We can evaluate part of the path expression using an S-Operator and other parts of the expression using relational join operators. For example, if we know 'b/c/d/e/f' is selective, a good plan probably would use an S-Operator to evaluate 'b/c/d/e/f' followed by two index nested loop joins to look up 'a' and 'g'.

Techniques for estimating the cost of the relational plan has been well studied since [15]. A cost model for S-Operator will be used to estimate execution cost of plan 2 and part of plan 3. The I/O cost for producing the event stream is a sequential scan of several relational tables. The CPU cost is the cost of maintaining a priority queue for the Converter (constant queue size) and state transitions of the finite state machine used by S-Operators. Another important parameter is the output size of an S-Operator or S-to-R Converter. This parameter is important to the relational engine when optimizing

relational operators that follow an S-to-R Converter. It is also useful to divide the workload between the relational and S-Operator engines. In our example, a good estimation of output size of 'b/c/d/e/f' is crucial to selecting between plan 2 and 3. We plan to exploit estimation techniques like those developed in [1].

4.4 Loading Database and Handling Remote URL

Loading a database from an XML file is handled by an ordinary S-to-R Converter that can be automatically generated from the corresponding DTD. A SAX Parser is used to produce an event stream that is converted into several tuple streams. As discussed in Section 4.2, such an S-to-R Converter is usually non-streaming. When used to load the database, there is no need for buffering because tuples are immediately inserted into corresponding tables. Tuples in each stream are ordered by the order they appear in original XML data, i.e., ordered by *xmlid* column. This is the preferred clustering order of STEP.

An XML database system should be able of running queries against both XML files from a remote site and data stored in a local database. One approach to querying XML files from a remote site is to first fetch and load the remote files into the relational database [6][12][13]. In this case, the system can be viewed as a cache for the remote files. However, if a file is used only once or infrequently, it is preferable not to load the file into the database system. The S-Operator can handle remote XML files gracefully without first loading the file into the database. The step of parsing and storing the file can be replaced by an S-to-R Converter that converts streaming XML information to a relational tuple stream. The two different approaches are shown in Figure 4.5.

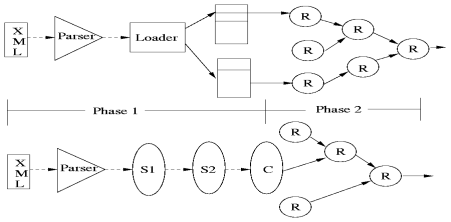


Figure 4.5 Querying remote XML file

In the new framework, we can eliminate the overhead of storing the remote XML files in relational tables. In addition, the stream of events from XML parser can first be pipelined through some S-Operators. The S-Operators could carry out restructuring or apply selection predicates to simplify or reduce subsequent work by the relational engine.

5. Experimental Evaluation

In this section, we evaluate STEP empirically and compare its performance with a middleware-based approach. Since the purpose of STEP is to facilitate existing relational database to efficiently execute XML queries, our focus is on complex XQuery queries that are hard to translate into SQL and queries that the resulting SQL queries are inefficient.

5.1 Experiment Setup

We used the XMark [25] benchmark in our experiments. XMark models data gathered from an Internet auction site. Figure 5.1 shows the graph representation of the DTD. Each arrow denotes a parent-children relationship. Dashed arrows indicate a child with the specific tag that can appear at most once in the parent node; solid arrows indicate those child elements that can appear multiple times. While most parts of the DTD represent data that has a well-defined structure (for example, each item has an id, a name, etc.), the elements in the dotted box are irregular in the sense that they contain TEXT data and elements that can be nested arbitrarily. The XMark DTD was mapped to a relational schema using techniques described in [19]. One should notice that each element in the dotted box is mapped into a separate table. The underlying relational table schema is shown in Figure 5.2.

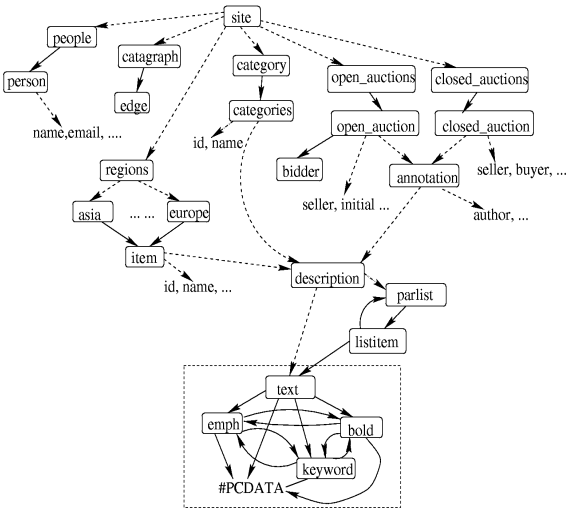


Figure 5.1 Graph representation of XMark DTD

site (xmlid, xmlpid, endid, people_id, ..., rgeions_asia_id, ...
regions_europe_id, open_auctions_id, closed_auctions_id);
... ...
item (xmlid, xmlpid, endid, id, name, ..., description_id);
open_auction (xmlid, xmlpid, endid, seller, initial ...,
annotation_description_id, annotation_author);
closed_auction (xmlid, xmlpid, endid, seller, buyer, ...,
annotation_description_id, annotation_author);
parlist(xmlid, xmlpid, endid);
listitem(xmlid, xmlpid, endid);
text(xmlid, xmlpid, endid);
emph(xmlid, xmlpid, endid);
bold(xmlid, xmlpid, endid);
keyword(xmlid, xmlpid, endid);
PCDATA(xmlid, xmlpid, data);

Figure 5.2 Relational Schema mapped from XMark DTD

In the original XMark data, the depth of nesting for elements in the dotted box is shallow (less than three). In order to experiment with queries over deeply nested irregular structure, we added some deeply nested structure (up to 12 levels of nesting) to one percent of the *text* elements. We will evaluate three queries defined in XMark plus several additional queries[4]. These additional queries focus on complex queries over the structure of XML documents, such as the filter operators and regular path expressions. We experiment with the original XMark database (scale factor 1, about 110MB) up to XMark scale factor 5 (560MB) to study the scalability of STEP with respect to database size.

Our experiments were conducted on an 800MHz Pentium III with 256M of memory, running Linux 2.2. Our implementation is based on Predator[14], which uses Shore[3] as the underlying storage management system. Shore was configured to use 30M buffer pool. Some SQL queries issued by middleware-based approach are very complex and the optimizer of Predator is not able to select a reasonable execution plan. In these cases, we manually forced a reasonable plan.

5.2 Loading

As described in Section 4.3, loading the database from an XML file is an ordinary STEP query (except logging is turned off). Parsing the XML file using a SAX parser produces a valid event stream. The event stream is sent into an S-to-R Converter, which converts the event stream into several tuple streams. Tuples in each tuple stream are then inserted into the corresponding tables. We built B-Tree indices on *xmlid*, *xmlpid*, *endid* columns of each table. Table 5.1 shows the sizes of databases and indices.

XMark Scale	**1**	**2**	**3**	**4**	**5**
Raw XML	111	223	335	447	560
STEP	168	341	513	680	856
STEP-index	96	205	310	400	516

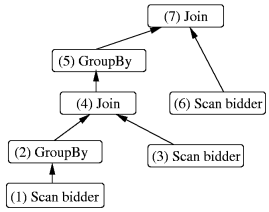
Table 5.1 Database sizes (in MB)

5.3 Complex Group By

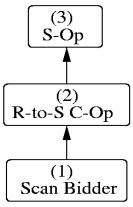
XMark Q2 tests the database's ability to perform complex group by queries.

Query Q2: Return the initial increases of all open auctions.
FOR $b IN document()/site/open_auctions/open_auction RETURN <increase> $b/bidder[2]/increase/text() </increase>

The node position test and range operator are common operators of XQuery. However, if naively translated into SQL using aggregate functions and SQL group-by clauses, the resulting SQL queries take a very long time to finish. We forced Predator to use the plan shown in Figure 5.3 to get the second *bidder* element of an *open_auciton*. The ability to generate such a "smart" SQL query is almost certainly beyond what middleware can do. The STEP plan scans the *bidder* table and uses a local variable as a counter to select the second *bidder* of an *open_auction* element. Figure 5.4 shows the performance results of XMark Q2.



Relational Plan:
(1,2): Groupby *xmlid* to find *bidder* in *open_auction*
(3,4): Join to find *bidder*s that are not first in *open_auction*
(5) GroupBy to find second *bidder*s
(6,7) Retrieve query result



STEP Plan:
(1) Table Scan *bidder*
(2) Convert to event stream
(3) An S-Operator to groupby *xmlpid* and count the second *bidder*

Figure 5.3 Plans for XMark Q2

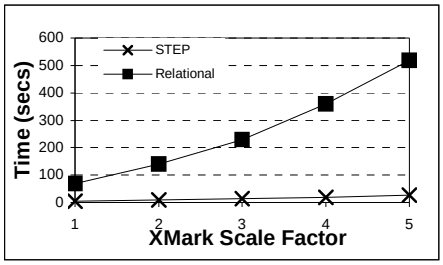
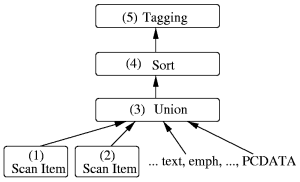


Figure 5.4 Performance results for XMark Q2

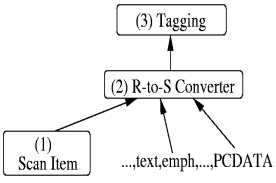
5.4 Reconstruction

XMark Q13 tests the database's ability to reconstruct large, complex elements of the original document.

Query 13: List the names of items registered in Australia along with their descriptions.
FOR $i IN document()/site/regions/australia/item RETURN <item name=$i/name/text()> $i/description </item>



Relational Plan:
(1) select *xmlid* as position, *START_ELE* as type from item
(2) select endid as position, *END_ELE* as type from item
Scan text, emph, etc twice
PCDATA is scanned only once because it only contains *TEXT*
(3, 4) Union and sort by position
(5) Add tag according to type



STEP Plan:
(1) Table scan Item, text, etc only once
(2) Convert to event stream
(3) Tagging

Figure 5.5 Plans for XMark Q13

[4] We only present results from only three XMark queries because most of XMark queries can be translated into simple SQL queries. Results of the whole XMark benchmark and the SQL queries used are available from http://www.cs.wisc.edu/~ftian/paper/step

A similar problem, publishing relational data as one or more XML documents, has also been studied recently [8][18]. Their approach, for example, the Sorted-Outer-Union, cannot be directly applied to this case because the element being constructed is involved in a cycle in the DTD. Unnesting a cycle in the DTD leads to the use of recursive queries. One cannot employ an outer union to combine the results of the recursive SQL queries because the upper bound of the depth of unnesting is unknown. However, the idea of using sort to produce result that is ready for tagging can be adapted to handle cycles in the DTD because the relational schema that we used contains position information (xmlid, endid) of each XML element. The STEP plan for reconstruction is simply an R-to-S Converter. Both plans are shown in Figure 5.5.

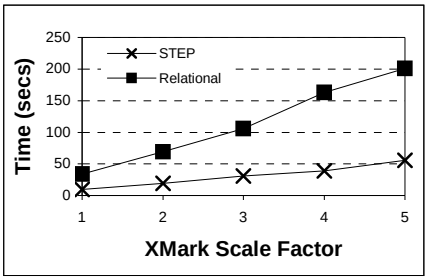


Figure 5.6 Performance results for XMark Q13

Figure 5.6 shows the performance results of XMark Q13 for the two approaches. The STEP plan is more efficient because:

1. STEP scans each table only once.

2. The R-to-S Converter in the STEP plan and the sort operator in the relational plan produce the same output sequence. The R-to-S Converter algorithm is D-Streaming and thus is more efficient than the external sort. In fact, we can consider the R-to-S Converter as a specialized sort operator that takes advantage of the nesting structure of XML documents.

5.5 Long Path Expression

XMark Q15 and Q16 test the database system's ability to evaluate long path expressions. Since the long paths in Q15 and Q16 are actually the same, we only show results of Q16 in this section. We added another query L2, which contains wildcards in the path expression.

Query 16: Return the IDs of those auctions that have one or more keywords in emphasis.
FOR $a IN document()/site/closed_auctions/closed_auction WHERE NOT EMPTY ($a/annotation/description/parlist/listitem/parlist/\ listitem/text/emph/keyword/text()) RETURN <person id=$a/seller/@person />
Query L2: Return the number of nodes at the second level below text elements
FOR $a IN document("auction.xml")//text/*/* RETURN COUNT($a)

In the relational plans, some steps in the path expressions are mapped to two columns of the same table; others are translated into joins between tables containing parent and child elements. A wildcard in the path expression can be evaluated using a union operator to combine the results from all possible instantiations of the wildcard. The STEP plan scans all the relevant tables and merges the tuples from these scans using an R-to-S Converter. Then the path expression is evaluated by an S-Operator, which contains a finite state machine that recognizes the regular expression.

There are situations when plans with multiple joins are better than the STEP plans. For example, if the intermediate result of one step of the long path expression contains only a few tuples, the relational optimizer may choose to evaluate this step first and then use an index nested loop join to avoid scanning the other tables (which could be very large). In other situations, if such a small intermediate result does not exist, the relational plan that joins many tables is usually more expensive than the STEP plan. The cost of the STEP plan, which is mainly the cost of table scans, can be easily estimated. An optimizer should be able to choose the better of the two plans. Performance results for Q16 and L2 are shown in Figure 5.7 and 5.8.

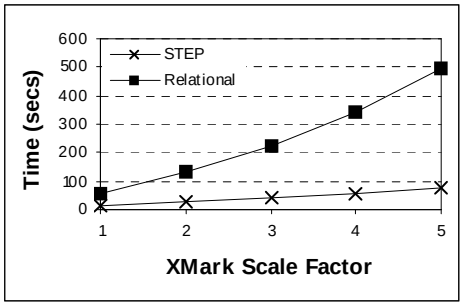


Figure 5.7 Performance results for XMark Q16

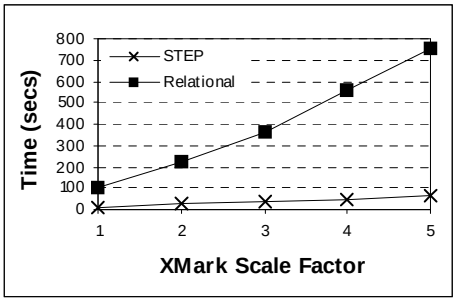


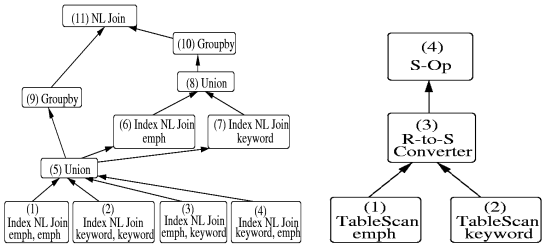
Figure 5.8 Performance results for L2

5.6 Filter Operator

XQuery provides features that let users query the structure of an XML document. One important operator is filter. Filter operates on an XML element, retaining some of the sub-elements while preserving the structure of these sub-elements.

Query F2: Return the number of nodes at the second level after filtering "emph" and "keyword" elements.
LET $a = filter(document("auction.xml"), //(emph \| keyword)) RETURN <Result>COUNT($a/*/*)</Result>

A filter operator on irregular elements like "text" in XMark may be translated to complex, probably recursive, SQL queries if the middleware layer prefers to execute the query inside the relational DBMS. Since query F2 only counts the number of elements in certain places of the filter result, we can execute the query without actually explicitly computing the filter result. The relational plan and the STEP plan for query F2 are shown in Figure 5.9.



Relational Plan:
(1-5): Produce *emph*s and *keyword*s that are nested in some other *emph* or *keyword*s. Index nested loop join is used because the join predicates are "*left.xmlid<right.xmlid*" and "*right.xmlid< left.endid*". Sort-Merge on "*left.xmlid=right.xmlpid*" cannot be used because the parent/child in the filter result may be separated by other elements in the original document
(6-8) Compute *emph* and *keyword* at or deeper that third level
(9) Count result of (5)
(10) Count result of (8)
(11) (9) minus (10) (Join two tables, each constains only one row)

STEP Plan:
(1-3) Table scan *emph* and *keyword*. Convert tuple streams into event stream.
(4) A finite state machine for regular expression "*/*". Count the number for results.

Figure 5.9 Plans for query F2

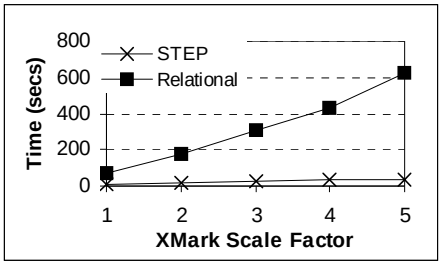


Figure 5.10 Performance results for query F2

As in queries with long path expression, the STEP plan scans the *emph* table once, which is more efficient and more scalable than executing a very complex SQL query.

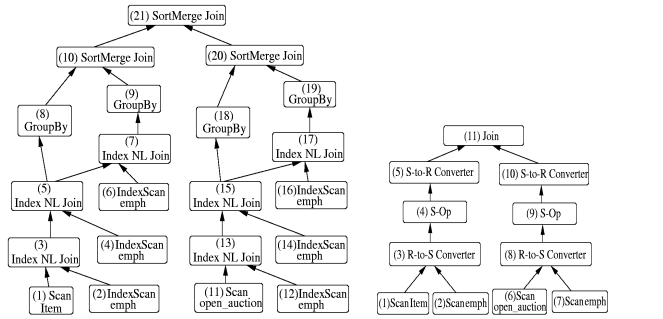
5.7 S-to-R Converter

One advantage of the STEP approach over a middleware-based approach is that STEP can seamlessly connect S-Operators and relational operators with one another. We use F2Join as an illustrating example. In query F2Join, we need to compute a join after evaluating the filter operators.

The STEP plan uses S-to-R Converters to connect the outputs of S-Operators to ordinary relational operators. Plans for F2Join are shown in Figure 5.11.

Query F2Join: Return item and open_auction that have same number of double emphasized elements.

FOR $o IN document()//open_auction,
$i IN document()//item
LET $oc = count(filter($o, //emph)/*/*),
$ic = count(filter($i, //emph)/*/*)
WHERE $oc = $ic
RETURN <Result>$o/name/text(), $i/name/text()</Result>



Relational Plan:
(1-5,8) Count *emph* elements that are at second level or deeper in *item* elements
(6-7, 9) Count *emph* elements at the third level or deeper
(10) Compute difference of (8) and (9)
(11-20) Count *emph* elements at the second level in *open_auction*
(21) join

STEP Plan:
(1-3) Table scan *item* and *emph*. Convert to event stream
(4) A finite state machine count *emph* element at second level
(5) Convert to tuple stream for join
(6-10) Generate tuple stream for *open_auction*
(11) Join

Figure 5.11 Plans for query F2Join

The same sort merge join operators are used as the final join operators of the relational plan and the STEP plan. STEP plan is more efficient in computing the input streams to the sort merge join operator. The results are contained in Figure 5.12.

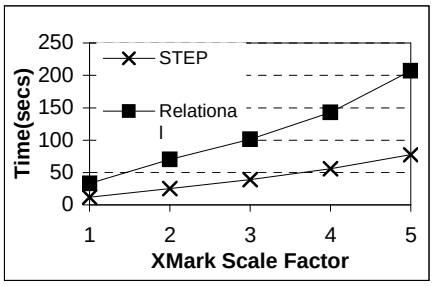


Figure 5.12 Performance results for query F2Join

5.8 Summary

Our experiments clearly demonstrate that STEP is a very effective mechanism for extending a relational database system to process complex queries on XML documents that are stored in relational tables. For queries over complex irregular data of XML files or queries containing complex path expressions, STEP scans tables once instead

of using very complex relational plans. Though our implementation stores XML data in relational database using strategy described in [19], we believe the same technique can be applied to other strategies like the Attribute strategy of [7]. In fact, part of XMark DTD (dashed box of Figure 5.1) is mapped to the same relational schema regardless weather strategy in [19] or [7] is used and our experiments will still be valid if the Attribute strategy of [7] is used. Our results demonstrate that STEP plans scale linearly with respect to the database size. The STEP plan is also much simpler than the corresponding relational plan, which typically contains a complex join or sub-queries. The cost of a simpler plan is easier to estimate, thus provides the optimizer better opportunities to choose good execution plan while relieving middleware programmers from having to write "smart" SQL queries.

6. Conclusion and Future Directions

This paper describes STEP, an XML event sequence model for processing complex queries against XML documents that we have implemented as extension module inside the Predator object-relational database system. STEP combines the sequence processing ability of S-Operators with the tuple processing ability of a traditional relational engine. Our experiments demonstrate that STEP can evaluate complex XPath expressions using only table scans, which are more efficient and more scalable than queries requiring joins of multiple tables by the relational execution engine. STEP connects the S-Operator engine and the relational engine with Converter operators, enabling results from one engine to be streamed into the other.

We are currently developing a cost model for STEP as the first step in developing a cost-based global query rewriter/optimizer to divide the workload between the S-Operator engine and the relational engine. Another important problem is to supply accurate statistics to the relational optimizer so that relational engine can choose good plans for queries whose inputs correspond to the outputs of S-Operators. In additional to being able to query XML documents stored in relational tables, STEP can also handle XML data stored as LOBs in the database, or XML documents stored in the file system. Our cost model will consider important factors like the parsing cost and network latency.

References:

[1] A. Aboulnaga, A.R. Alameldeen, J.F. Naughton: Estimating the selectivity of xml path expressions for internet scale applications. VLDB 2001

[2] P. Bohannon, J. Freire, P. Roy, et al. From XML Schema to Relations: A Cost-based Approach to XML Storage. ICDE 2002

[3] M.J. Carey, D.J. DeWitt, M.J. Franklin, et al. Shoring up Persistent Applications. SIGMOD 1994

[4] M.J. Carey, J. Keirnan, J. Shanmugasundaram, et al.: XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. VLDB 2000

[5] A. Deutsch, M.F. Fernandez, D. Suciu: Storing Semistructured Data with STORED. SIGMOD 1999

[6] L. Fegaras, R. Elmasri. Query engines for Web-accessible XML data. VLDB 2001

[7] D. Florescu, D. Kossman: Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin 22(3), 1999

[8] M. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries. SIGMOD 2001

[9] N.H. Gehani, H.V. Jagadish, O. Shmueli. Composite event specification in active databases: Model and implementation. VLDB 1992

[10] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. TKDE 6(1), 120-135, 1994.

[11] Z.G. Ives, A.Y. Levy, D.S. Weld, Efficient Evaluation of Regular Path Expression on Streaming XML Data. Technical Report UW-CSE-2000-05-02, University of Washington

[12] I. Manolescu, D. Florescu, D. Kossman: Answering XML queries on heterogeneous data sources, VLDB 2001.

[13] J.F. Naughton, D.J. DeWitt, D. Maier, et al. The Niagara Internet Query System. IEEE Data Engineering Bulletin 24(2), 27-33, 2001.

[14] Predator. http://www.cs.cornell.edu/predator/.

[15] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, et al. Access Path Selection in a Relational Database Management System. SIGMOD 1979

[16] The SAX API. http://sax.sourceforge.net

[17] J. Shanmugasundaram, E.J. Shekita, S. N. Subramanian, et al. Querying XML views of relational data. VLDB 2001

[18] J. Shanmugasundaram, E.J. Shekita, R. Barr, et al. Efficiently Publishing Relational Data as XML Documents. VLDB 2000

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. VLDB 1999

[20] P. Seshadri, M. Livny, R. Ramakrishnan, Sequence Query Processing, SIGMOD 1994

[21] P. Seshadri, M Livny, R. Ramakrishnan, SEQ: A Model for Sequence Databases, ICDE 1995.

[22] M.D. Soo, Bibliography on Temporal Databases, ACM SIGMOD Record 20(1) 14—23, 1991

[23] I. Tatarinov, S. Viglas, J. Shanmugasundaram, Storing and Querying Ordered XML Using a Relational Database System, SIGMOD 2002.

[24] S. Viglas, L. Galanis, D.J. DeWitt, et al. Putting XML Query Algebra into Context. Submitted for publication

[25] http://www.xml-benchmark.com/

[26] http://www.w3c.org/TR/xquery.

[27] http://www.w3c.org/TR/query-datamodel

[28] http://www.w3c.org/TR/query-semantics