Building a Scalable Native XML Database Engine on Infrastructure for a Relational Database

Guogen Zhang
IBM Silicon Valley Lab
555 Bailey Ave
San Jose, CA 95141, USA
gzhang@us.ibm.com

ABSTRACT

We describe the architecture and some aspects of System R/X, a native XML database engine that is built on the same mature infrastructure for a relational database and integrated with the relational engine. We describe what parts of the infrastructure can be reused, what need to be extended, and what are totally new to the XML database and their techniques. Our overall strategy is to base XML storage and search on the scalable relational technology with substantial extensions. Many techniques are novel to our knowledge. We also provide perspectives along the discussion and point out some open research issues.

1. INTRODUCTION

It took about one decade for relational database products to mature commercially and become the mainstream database management systems. Will XML database products take the same time to mature? The answer is maybe no, and maybe yes. The decades-long industrial experiences of developing database products and huge research and development efforts by both industry and academia will no doubt accelerate the maturity of XML databases. However, the XML data model and the XQuery language [29] are inherently more complex and more powerful than their relational counterparts. This makes solid delivery of XML databases ever-more challenging.

It is a critical part of engineering effort in implementation of XML databases to leverage existing database engine infrastructure. Furthermore, it is also a customer business necessity to integrate XML capability with an existing relational product for smooth and incremental adoption of XML database technology.

While there is a large volume of literature on XML support using relational databases, we believe that directly mapping XML into relational model and translating XQuery into SQL would not work well in terms of performance and integrity due largely to the hierarchical and flexible nature of the XML data model. In this paper, we describe our experiences of extending a relational database with native XML support in System R/X. System R/X is a sister project of System RX [23] on the IBM z/OS platform targeted at enterprise users with integrated relational and XML support. By native XML support, we mean that storage and processing of XML data are neither using relational or object-oriented data model directly, nor using large objects (LOBs), but using those specifically designed for XML, including storage format, indexing, query processing, and concurrency control, among others.

Indexing is critical to database scalability. System R/X supports XPath value indexes. An XPath value index maps the values of nodes identified by a path expression from documents to the logical positions of the nodes in the documents and physical record positions in the storage. XPath value indexes can be used to answer efficiently XPath queries with predicates on values.

One of the interesting features of System R/X is to leverage heavily on the data management infrastructure for a relational database, and to adapt and extend it for the native XML functionality.

The rest of the paper is organized as follows. First in Section 2, an architectural overview is presented, including what are reused, what are extended, and what are new. Then in the following sections, major new features and extensions are described. In Section 3, we describe the storage, data insertion and traversal. In Section 4, we describe query processing, including basic scan-based algorithm and index-based access methods. In Section 5, we describe concurrency control issues. In Section 6, we conclude the paper and point out the future work.

2. ARCHITECTURE

The high-level architecture of System R/X is shown in Figure 1. XML services are added in parallel to relational services (e.g. searching based on predicates, join algorithms) on the same data management infrastructure for relational data that has been tuned for decades. The data management features that need no enhancement or need minor enhancement include storage (data manager, buffer manager, and external storage management), catalog and directory, logging, backup and recovery and other utilities, instrumentation, etc. It is worth pointing out that to the lower level components of the infrastructure, our packed XML data looks like rows in relational tables.

New features in XML services include new data format for XML hierarchical data model, XML parsing and validation, XML construction and serialization, XML data traversal and XPath evaluation, XPath index key generation, and new index-based access methods.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
XIME-P 2005, June 16–17, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00

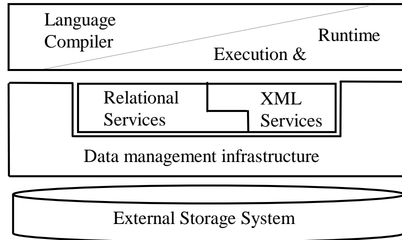


Figure 1. High-level architecture of System R/X

Language compiler and query execution are significantly enhanced to support compilation, optimization and execution of SQL/XML and XQuery languages, including client connectivity, distributed facilities, and stored procedures. Index manager and lock manager are enhanced to support XPath indexes and concurrency of XML operations. In the meantime, we isolate XML related processing from existing SQL processing as much as possible to avoid making the code too complex to cause functional regression and performance degradation of existing SQL features.

Currently, all the manipulation and querying of XML data are through SQL and SQL/XML with embedded XPath and XQuery. To SQL, XML is just a new data type with a more complex content, similar to large objects. Except for the streaming and deferred access purposes, all the standard processing flows remain largely unchanged for XML, and quite similar to LOBs.

3. STORAGE OF XML DATA

3.1 Persistent XML Data

In SQL/XML, a table can have XML columns. A value for an XML column does not have a length limit. In many aspects, XML data is similar to LOB data. However, the limited operations for LOBs impose significant restrictions on XML subdocument update if XML data were stored as LOB. On the other hand, XML data trees of the XQuery data model [29] are similar to objects in object-oriented databases. However, many object-oriented database storage schemes were not proven to be as scalable as relational databases.

Relational table spaces are well tuned for efficient space management, reliability and scalability. They are a natural choice to store persistent XML data also. But we are not using a decomposition or mapping approach [9]. Figure 2 shows how XML column data can be stored.

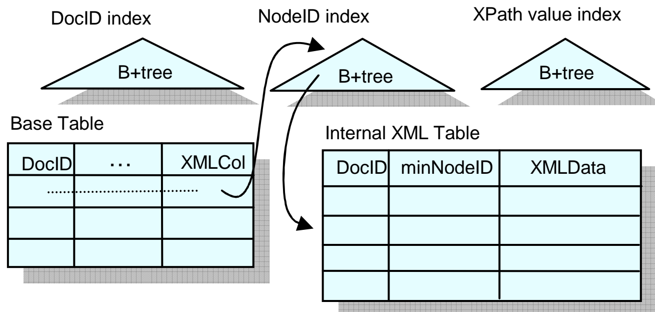
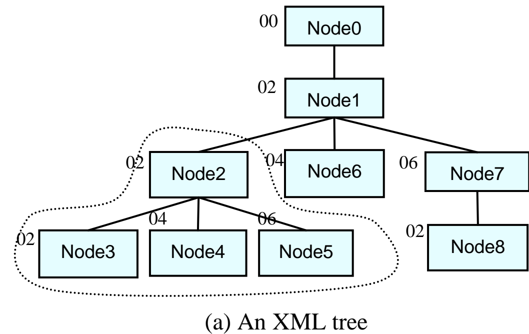


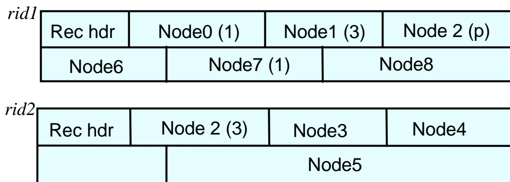
Figure 2. Storing XML data using a regular table space

In our scheme, an internal table space is created for each XML column in a base table. The internal XML table is a table that has three columns (*DocID*, *minNodeID*, *XMLData*), where the *XMLData* column is a SQL VARBINARY type that contains packed XML data to be explained shortly. The minimum node ID in *XMLData* is put in *minNodeID*, which, together with *DocID*, is used for clustering. A base table with an XML column will have an implicit DocID column, shared by all the XML columns in the table, and used to link from the base table to the XML table. In addition, a DocID index on the base table is used for getting to base table rows from XPath value indexes.

The way XML tree data is packed into records is illustrated in Figure 3. In the stored XML data, all the names for elements, attributes, and namespaces are encoded using integers across the entire database. There are seven kinds of nodes in the XQuery data model. Within each packed record, structure nesting is used to represent the parent-child relationship between nodes, similar to what Natix does [18]. Each non-leaf node contains the number of children (shown in parentheses in Figure 3(b)), followed by the child nodes, recursively. Subtree length is also contained in non-leaf nodes to support efficient tree traversal by using the *firstChild* and *nextSibling* operations. Assuming the tree is too big for one record, we pack a subtree or a sequence of subtrees into a separate record, in a bottom-up fashion. A packed subtree is represented using a proxy node in its containing record. No explicit physical link is used between records for maximum flexibility of record placement that is one of the salient features of relational data. Instead, logical node IDs are used to link between records through a NodeID index that will be explained shortly.



(a) An XML tree



(b) Two packed records

Figure 3. Packing an XML tree into two records

Each XMLData contains a single subtree or a sequence of subtrees that share a common parent node. The parent node is also known as the context node for the record. The packed record includes a record header that contains the context path information, including the absolute node ID, the path from the root (a list of name IDs), and in-scope namespaces for the context node. Each node contains a relative node ID, as shown next to a node in Figure 3(a). Node IDs are prefix encoded [7] as Dewey

IDs in such a way that they are stable upon update of the tree. Basically, a relative node ID ends with an even-numbered byte; and any odd-numbered byte means that the relative ID is extended to the next byte. The absolute node ID of a node is the concatenation of relative node IDs along the path from the root to the node. The root node ID is an exception, which is always 00, so it is implicit in the absolute node IDs. String comparison on the node IDs provides document order. And there is always space for insertion in the middle by extending the node ID length when necessary. The relative node ID of each level can be recovered from the absolute node ID.

A NodeID index is created on each XML table to map a logical node ID to its physical record ID (RID). For each contiguous interval of node IDs for nodes within a record in document order, only one entry is in the node ID index, which is the upper end point of the node ID interval. For illustration, there are three entries for the two records in Figure 3(b): (02, *rid1*), (020206, *rid2*), and (020602, *rid1*), where *rid1* and *rid2* are RIDs for the two records.

This tree packing scheme makes sense in terms of performance when compared with the relational representation of one row per node (or edge) [28]. Assume the storage overhead for each row is α on average, the packing factor is p, i.e. p nodes per record on average, and the average node body size is n. Under the one node per record scheme, the storage for a tree of k nodes is: $n\,k + \alpha\,k = k\,(n + \alpha)$, with an overhead of α/n. In comparison, using p packed nodes per record scheme, the required storage is about $n\,k + \alpha\,k/p = k\,(n + \alpha/p)$, with an overhead of $\alpha/(np)$. If we use a node ID index for both cases, the one node per record scheme requires k index entries, while the packed nodes scheme only requires about $2*k/p$ entries or less.

Now let us consider tree traversal time, assume one relational join for each node requires time of t, traversal of a k-node tree with one row per node requires time of $(k\text{-}1)*t$ (or $(k\text{-}1)(t + t_1)$, t_1 is the time of local access within a record, which is much smaller than t). For the packed tree scheme, it requires time of kt/p (or $kt/p + kt_1$ with local traversal time within a record). The ratio is approximately $1/p$ (assuming $t >> t_1$ and a large k).

The above analysis shows that the larger the packing factor p, the more efficient the storage and tree traversal operations. However, there is an upper limit on the record size by the page size, and also the counter factor for update overhead. To update one single node, under the one row per node scheme, we only need to touch storage of one record, with size of $n + \alpha$, while in the packed tree scheme, we will touch storage of $\alpha + pn$. However, except for larger log spaces required, touching a relatively large size may not be too bad, since the I/O unit is a page.

Some other factors considered in the design of this format are the efficiency of the node packing algorithm, easy maintenance of in-scope namespaces, compression of nodeIDs, skipping subtrees in XPath evaluations, simple move and copy operations of subtrees, being self-contained when accessed from an XPath value index, and subtree locking and versioning for concurrency control.

We use a simple size-based grouping method. In comparison, the Natix storage scheme uses a complex split matrix mechanism to control the grouping, which we do not know how to automatically apply in real systems. In addition, we are not clear how Natix handles in-scope namespaces and node IDs.

3.2 Insertion of XML Data

Insertion of relational data incurs some minor cost of type conversion or character encoding conversion. None of these is complex compared with insertion of XML data, which requires expensive parsing or validation. Application domain interfaces for XML, such as SAX or DOM interface, suffer from significant overhead of excessive procedure calls for event handling or in-memory construction of intermediate data structures. To reduce the overhead, we use a proprietary parsing and validation interface, which is the buffered token stream. The token stream is a binary stream of tokens with namespace prefixes resolved, namespace and attribute order adjusted, and optionally with type annotation if a document is Schema-validated. It is similar to the token stream in BEA/XQRL [10]. Buffering reduces per-token procedure call cost significantly.

In addition, high-performance validation with LALR parser generator technique is used for XML schema validation. As shown in Figure 4, an XML schema has to be registered before it can be used. During the registration, it is compiled into a binary format like a parsing table and stored in the catalog. At the execution time, the binary schema is loaded and executed by a validation runtime to generate a token stream. Both validating and non-validating parsers are custom-made for high-performance.

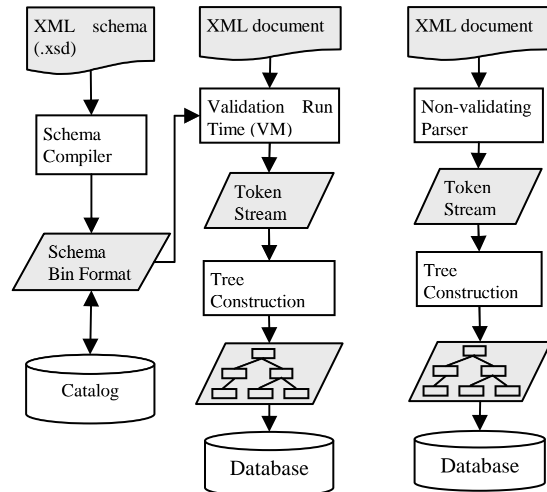


Figure 4. Schema registration, XML validation and parsing

During tree construction, no separate trees of in-memory format are built. Rather, tree-packed records are generated from the bottom up in a streaming fashion. Index keys for the node ID index and XPath value indexes are generated per record, which fits existing infrastructure very well.

3.3 XPath Value Indexes

Initial XPath index support in System R/X uses and extends the same B+tree infrastructure for relational indexes, and is relatively simple. It is our belief that index size should be kept much smaller than data size for efficiency, and maintenance of too complex indexes can become a bottleneck for high volume systems.

Users can create XPath value indexes on frequently searched elements or attributes by specifying a simple XPath expression without predicates, such as "`/catalog//productname`", and a data type for the key values, such as string (equivalent SQL

VARCHAR(n) is used). A few simple types supported, such as double, string, and date. Key values are converted from the string values of the nodes, identified by the XPath expression. A value index entry contains (*keyVal*, *DocID*, *NodeID*, *RID*), which can map a key value to a logical ID (*DocID*, *NodeID*) or physical ID (RID) in the XML table, or both. A simplified version of our streaming XPath algorithm (QuickXScan) is used to evaluate the XPath on each record. One major difference of an XPath value index to the index manager or utilities maintaining the index is that there may be zero, one or more index entries per record, as commonly seen in extended indexes, while for relational data, there is one-to-one correspondence between an index entry and a row. See Section 4 for the use of XPath value indexes in query processing.

3.4 Traversal of Stored XML Data

To traverse in document order a persistently stored XML document with a given *docID* value, first the NodeID index is searched with (*docID*, 00) as the key. The root record can be identified. The *XMLData* is then traversed. If a proxy node is encountered, its node ID *nodeID* is used to search the NodeID index with (*docID*, *nodeID*) as the key to find the RID for the corresponding record. Stacking has to be used during traversal. At a higher level, the records form a block-based tree, and traversal of this tree is also in a depth-first order, with fetch sequence matching the clustering order of (*DocID*, *minNodeID*).

When a (*docID*, *nodeID*) is given from an XPath value index, to find the record containing the corresponding node, use this pair as the key on the node ID index, the RID will be returned. Traversal inside the record by node ID can find the right node with the given *nodeID*. All the information required by the data model is available. The successful search on the NodeID index is attributed to the arrangement for the NodeID index keys by using the upper end points of NodeID intervals.

It is worth noting that skipping to the next sibling may result in skipping an entire subtree beneath a node, which may contain many records.

4. QUERY PROCESSING

An input query goes through the typical query processing steps: query parsing, semantics and transformation [25], access path selection [27], plan generation, and execution.

The XQuery/XPath parser is generated by a LALR(k) parser generator [1], separately from the SQL parser. It is worth noting that in our case LALR(1) is used with a much simpler lexical scanner than what is described in the W3C specification, achieved by rewriting the BNF production rules. Query semantics checking and transformation are performed to optimize the query by query rewrite. Access path selection is relatively simple at the moment. Below we cover some topics related to constructor functions, XPath evaluation, and run time organization.

4.1 SQL/XML and XQuery Constructors

The constructor functions are to generate XML data. Due to known structure of constructors, there is a great opportunity to optimize the constructor functions. We use an example to illustrate the technique. Assuming we have the following nested functions in SQL/XML:

```
XMLELEMENT(NAME "Emp",
    XMLATTRIBUTES(e.id as "id",
          e.fname ||' '|| e.lname AS "name"),
     XMLFOREST(e.hire, e.dept  AS "department"))
```

It is fairly common to have nested constructor functions due to the nature of XML. The standard function evaluation process is to evaluate the arguments first, then evaluate the function. If we follow the standard steps, it will either involve small data items linked by pointers or need multiple copies of the same data items.

We optimize constructor functions by flattening the nested functions into one function and represent the nesting structure with a tagging template, as illustrated in Figure 5. In the template, the number means which argument to fill in. The result of the constructor functions is an intermediate result representation that includes a pointer to the template with a data record as shown in the bottom of Figure 5. This intermediate result is optimized because no repetition of the tagging template occurs, which is very effective for generating XML for large number of repeated rows or the aggregate function XMLAGG.

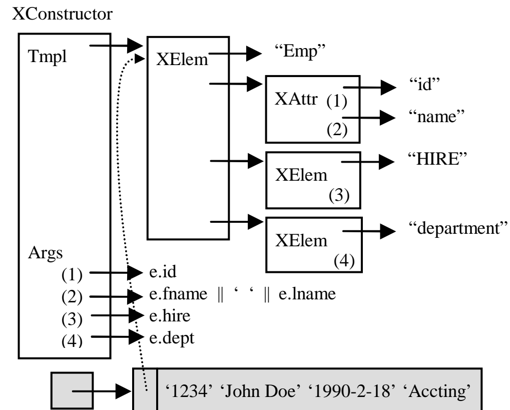


Figure 5. Constructor function optimization

In addition, for XMLAGG ORDER BY evaluation, typical external SORT will need to sort each group of rows, suffering from significant overhead. We apply in-memory quicksort to the linked list representation of rows in each group of XMLAGG, achieving high performance. For XML values referenced in the constructors (or other functions), a reference construct called an *XML handle* is used. The same techniques apply to XQuery constructors, since the variables referenced in XQuery constructors are logically components from tuples.

4.2 XPath Evaluation by QuickXScan

There are many XPath and XQuery evaluation algorithms [5][11][12][14][20][30]. Some are based on relational representation and structural joins [2], which in turn can exploit indexes [16]. However, a base algorithm should have the characteristics of a relational scan – it evaluates an XPath expression by one pass scan of a document without help from extra indexes, and also has similar performance characteristics, i.e. not much more expensive than the scan. We have invented an optimal streaming XPath algorithm, called QuickXScan [31] for a subset of XPath path expressions that consist of the following five forward axes: child, attribute, descendant, self, and descendant-or-self. The parent axis can also be supported based on query rewrite [24] or minor extension to the base algorithm.

Like many other XPath algorithms, such as TurboXPath [17], QuickXScan models a path expression with a query tree. For example, a path expression /b//s[.//t = "XML" and f/@w > 300] can be represented as a query tree shown in Figure 6(a), where r is the root step, each node is labeled by the name test or kind test, and the axis of each step is differentiated by a single-line edge for child axis or a double-line edge for descendant axis to its previous step (note that in some cases the descendant-or-self axis can be reduced to the descendant axis).

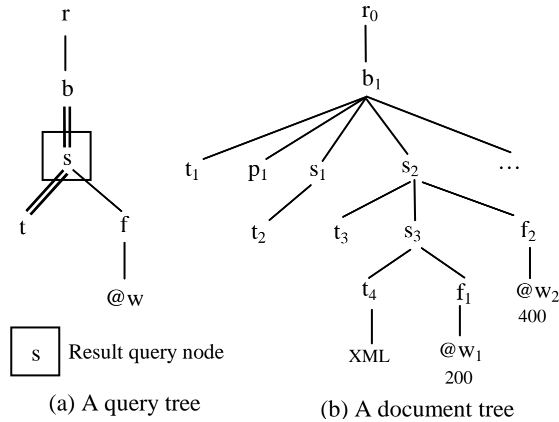


Figure 6. An example query tree and document tree

QuickXScan is based on the principles of attribute grammars [19] and syntax-directed evaluation [1]. An XPath expression is converted into a set of "attributes" in an attribute grammar necessary to evaluate the path expression, which is different from using explicit attribute grammars to query XML documents [21][22]. Both inherited attributes and synthesized attributes are used. A basic inherited attribute is to determine whether an XML tree node matches a query node or not, evaluated during the top-down traversal of an XML document tree. Non-matched XML tree nodes are discarded, while matched XML tree nodes, called matching instances (or just matchings), form a logical sub-graph (also a tree) of the original XML tree. A typical synthesized attribute for a matching instance is to compute the sequence of children or descendants under the node, and eventually used for predicate evaluation or for deriving the result sequence. Predicate pushdown can be achieved by using Boolean-valued attributes. Synthesized attributes are evaluated during the bottom-up traversal as usual. A set of attributes is associated with each query node, but is not shown in Figure 6(a). The XPath evaluation reduces to the evaluation of the attribute grammar constructed from the XPath expression.

QuickXScan uses two important transitivity properties among matching instances and their attributes. The first transitivity is among matching instances. For an XPath expression such as //*A*//*B*, if a_1 and a_2 are of ancestor-descendant relationship and both match with *A*, and if b_1 matches *B* and is a descendant of a_2, then b_1 is also a descendant of a_1. The second transitivity is for sequence-valued attributes. For the matching configuration just mentioned, if s_1 is a sequence of *B* descendants of a_2, all nodes in s_1 also belong to the sequence of *B* descendants of a_1. QuickXScan uses these two properties to avoid unnecessary matching tests and compute sequence-valued synthesized attributes incrementally using propagations.

At the execution time, a logical (horizontal) stack is associated with each query node to keep track of matching instances with transitivity, as in the Twig Stack algorithm [6]. During the top-down traversal of a document tree, each node is matched with one or more query nodes. Matched nodes are pushed onto a stack that is associated with each query node, and inherited attributes, such as *count*, can be evaluated. To implicitly record the matching paths and facilitate attribute value propagation, a matching instance has an upward link to the matching instance at the stack top of the previous step if it does not share the matching in the previous step with its ancestor in the same step.

Illustrated in Figure 7 is a comparison of matching state between QuickXScan and other streaming algorithms [17][26] at the time when t_4 of the document tree in Figure 6(b) is matched with t. Only the stack top needs to be checked for matching a node, which reduces the number of active states (in term of states of an automaton) from potentially exponential (when a path expression like //a//a//a matches with a document with recursively nested "a" elements) to the number of query nodes at maximum.

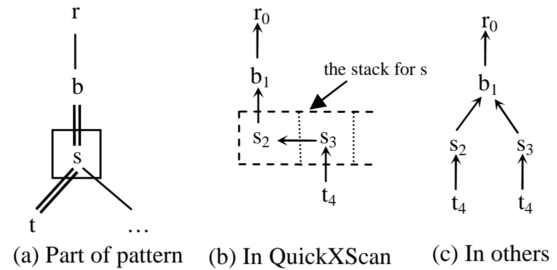


Figure 7. Part of the pattern and matching

During the bottom-up traversal, nodes are popped off from the stacks and synthesized attribute values for matching instances are evaluated, including candidate result sequences, which will go through filtering by predicates associated in the upper query nodes. QuickXScan propagates attribute values among matching instances when a matching instance is popped off from a stack. Table 1 shows propagation scenarios of basic sequence-valued attributes. Propagation can be upward or sideways, or both. Duplicate propagations can be avoided if we follow these rules:

- For *b*: propagate the value upward if there is an upward link or else propagate sideways, and
- For *a*: propagate the value sideways and accumulate for *b* descendants of *a*.

When there are predicates, the propagation rules become a little complex, but we can guarantee no duplicates for a sequence-valued attribute, see [31] for details.

QuickXScan is an optimal streaming XPath algorithm in terms of the number of active states and minimum buffer requirements [5]. It needs to maintain $O(|Q|*r)$ matching units at any time in the worst-case, where $|Q|$ is the number of query nodes and r is the recursion degree of a document, or how many nodes with the same name are nested within each other at maximum. The time complexity of QuickXScan is $O(|Q|*r*|D|)$ in the worst case, where $|D|$ is the document size. Experiments show that it outperforms the existing state-of-the-art streaming XPath algorithms in both elapsed time and memory consumption, and is orders of magnitude better than some DOM-based algorithm.

QuickXScan achieved our design goal of linear performance with regard to the document size for a subset of XPath expressions in practice (because of a small *r* value).

Table 1. Propagation of basic sequence-valued attributes

Path and matchings	Path, attributes, and propagations

a | b ; $a_1 \uparrow b_1$	Path: …a/b
	s: sequence of b children of a
	Init: $s_1 := \varepsilon$; // when a_1 is created
	At end of b_1: $s_1 := s_1 \cup \{b_1\}$; // upward

a | b ; $a_1 \leftarrow a_2$, $a_1 \uparrow b_1$, $a_2 \uparrow b_2$	Path: …a/b
	s: sequence of b children of a
	Init: $s_i := \varepsilon$; // when a_i is created
	At end of b_i: $s_i := s_i \cup \{b_i\}$; // upward
	// no sideways propagation for s

a ‖ b ; $a_1 \uparrow b_1 \leftarrow b_2$	Path: …a//b
	s: sequence of b descendants of a
	t: sequence of b descendant-or-self of b
	Init: $s_1 := \varepsilon$; // when a_1 is created
	$t_i := \{b_i\}$; // when b_i is created
	At end of b_2: $t_1 := t_1 \cup t_2$; // sideways
	At end of b_1: $s_1 := s_1 \cup t_1$; // upward

a ‖ b ; $a_1 \leftarrow a_2$, $b_1 \leftarrow b_2$, $a_1 \uparrow b_1$, $a_2 \uparrow b_2$	Path: …a//b
	s: sequence of *b* descendants of *a*
	t: sequence of *b* descendant-or-self of *b*
	Init: $s_i := \varepsilon$; // when a_i is created
	$t_i := \{b_i\}$; // when b_i is created
	At end of b_i: $s_i := s_i \cup t_i$; // upward
	At end of b_2: $t_1 := t_1 \cup t_2$; // sideways
	At end of a_2: $s_1 = s_1 \cup s_2$; // sideways

4.3 XPath Evaluation by Indexes

Although QuickXScan is critical to the system performance, a system based on QuickXScan alone will not possess desirable scalability, just as a relational database with relational scan only. The scalability of database systems largely relies on the efficient access methods based on indexes. While special indexes can be created to support evaluation of XPath solely based on the indexes, these indexes will have to be a complete copy of the base data and even larger than the base data, which in our opinion only works for read-only documents, due to high index maintenance cost. Our approach is to use indexes to quickly identify a small subset of candidates and then perform further processing on them.

For small documents, using indexes to identify qualifying documents would be efficient, which we call *DocID list access*. That is, a list of unique DocIDs is returned from an XPath value index, and documents are then fetched by using the DocIDs. For large documents, the DocID list access is no longer efficient. Instead, the *NodeID list access* applies. Since we do not keep complete path information in an XPath value index, when the XPath expression of the index contains a query XPath expression but is not equivalent to it, we use the index for *filtering*, and re-evaluation of the query XPath expression on the document data is necessary. When multiple indexes are used to evaluate a single XPath expression, we use *DocID anding/oring*, or *NodeID anding/oring* at document level or node level, respectively.

Table 2. Access method examples

Access method	Example
(1) DocID/NodeID list	Query: '/Catalog/Categories/Product[RegPrice > 100]' Index: '/Catalog/Categories/Product/RegPrice' as double
(2) DocID/NodeID filtering list	Query: '/Catalog/Categories/Product[Discount > 0.1]' Index: '//Discount' as double
(3) DocID/NodeID anding/oring	Query: '/Catalog/Categories/Product[RegPrice > 100 and Discount > 0.1]' Indexes: (1) '/Catalog/Categories/Product/RegPrice' as double (2) '//Discount' as double

Table 2 lists some query and index examples for the access methods. When the XPath predicate and value type of the query match with the XPath expression and value type of the index, DocID/NodeID list access applies, which is illustrated in the first case of the table. When the XPath expression of the query is contained in the index path expression, the filtering applies, as shown in the second case of the table. When two or more predicates match with multiple value indexes, anding/oring applies, as shown in the third case. If all the indexes match exactly with the predicates, the result DocID/NodeID list is exact. If one of them is exact match, while the others are containment, NodeID level anding will result in an exact list. Otherwise, the result list will not be exact but filtering.

It is worth noting that our value indexes can be viewed as a simple version of XPath views [3]. In implementation, we use decimal floating-point number based on the new IEEE 754r for numeric value indexing, which provides precise values within its range.

4.4 Virtual SAX and Runtime Architecture

Ideally, one single representation, such as the persistent store format, should be used for all processing needs. However, due to modularity or best-fit to a task, XML data can be in one of the many forms during the query processing: token stream, persistent store format, constructed format, or in-memory sequence, where an in-memory sequence is the result of XPath/XQuery, and constructed data can contain any of the other formats. To avoid data copying and format conversion cost, we do not construct a single unified in-memory tree representation for a task. There are three major tasks for XML data in addition to parsing: (1) serialization: to generate a serialized XML string for output to applications; (2) tree construction: to generate packed records for insertion into XML columns; or (3) XPath evaluation: to generate an in-memory sequence as result.

Figure 8 shows how we use virtual SAX to organize the runtime components to achieve shared code and pipelining. To perform one of the tasks, a proper iterator is attached to the data as the input interface according to the data format. As the iterator traverses through the data, each input data item is converted into a virtual SAX-like event, which is a set of parameters required by the routines performing the task. All the routines are shared, and

are inlined to avoid actual data copying and procedure call cost, thus achieving high performance.

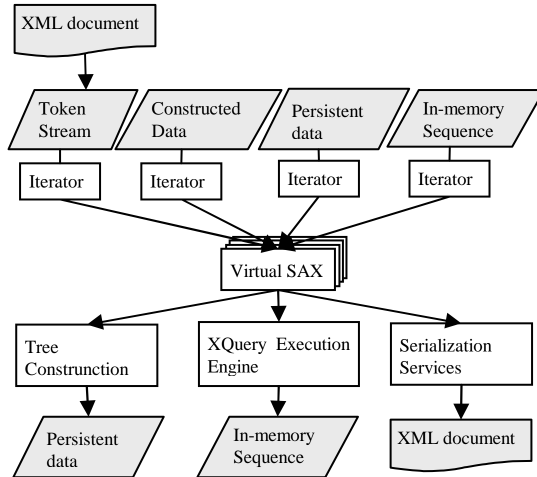


Figure 8. Runtime architecture

As in traditional relational processing [13], pipelining is exploited whenever possible. However, sometime it is necessary to materialize the result. Traditional temporary work files are used for relational data. More efficient alternative to work files, 64-bit virtual memory, is used for XML temporary data. XML handles are widely used to link between relational data and XML data. Fetch of persistent XML data is deferred until when it's necessary.

5. CONCURRENCY CONTROL

There has been some recently work on adapting concurrency control mechanisms to XML databases, such as path lock [8] and other lock-based protocols [15]. We can define two concurrency levels for XML data, that is, document level concurrency and subdocument concurrency. We discuss briefly and provide some perspectives here.

5.1 Document Level Concurrency

In SQL/XML or XQuery using a base table for a collection, an XML column can be viewed as an indivisible unit, the basic level of concurrency is at the document level. The isolation levels for transactions on relational data can be naturally extended to cover XML columns.

In lock-based document level concurrency, if we follow the access sequence from a base table row to the XML column data, the lock on the base table can cover the XML data. However, if we allow direct access to the XML data from value indexes or from an uncommitted reader that does not lock the base table rows, a DocID locking scheme is required. For deferred access to XML data, DocID locking is also needed. Care must be taken also to prevent reading a partially inserted document by using a lock.

Alternatively, multiversioning can be applied to avoid locking by readers, which is more efficient for mostly read workload. To support multiversioning at document level, one scheme is to keep most up-to-date data for XPath value indexes, but keep versions for XML data and the NodeID index required. Without versioning, the index entries for a NodeID index contain (*DocID*, *NodeID*, *RID*), while with versioning, the entries will also include a version number, i.e. (*ver#*, *DocID*, *NodeID*, *RID*) or (*DocID*, *ver#*, *NodeID*, *RID*), with *ver#* in descending order. This will guarantee a reader's deferred access to be successful.

5.2 Subdocument Concurrency

In subdocument concurrency, the isolation levels for XML data become not so well-defined. Certainly we are only interested in a consistent document at certain point of time.

In lock-based subdocument concurrency, we believe a multiple granularity locking [4] is needed given the hierarchical nature of XML data. Since we use prefix-encoded node IDs, locking using node IDs can support the protocol efficiently because ancestor-descendant relationship can be checked by testing if one is a prefix of the other. However, with our tree packing scheme for storage, a group of nodes form a record and the stored rows represents a tree of records. Our study point us to the direction of combining logical node ID-based multiple granularity locking with multiversioning at subdocument level to make record level consistency.

To support multiversioning at subdocument level, the NodeID index entries will have to be different from whole document versioning. One of the solutions is to let the index entries contain (*DocID*, *NodeID*, *ver#*, *RID*), where the *NodeID* may be a real interval end point, or a virtual index point without a real corresponding version. Details are omitted here.

Efficient subdocument concurrency control with meaningful isolation levels for weaker consistency remains a research area.

6. CONCLUSION AND FUTURE WORK

We have described the architecture and various aspects of a native XML database that is built on the same infrastructure for a relational database engine and integrated with the relational engine. This paper is a report of work in progress, with many unanswered questions remaining. However, we believe scalable native XML database engine can leverage the existing infrastructure tremendously, and only need to extend new storage scheme, XML-specific operations and query processing, and concurrency control when necessary. It is also our belief that at least in the near term, it will have a better chance of success to extend mature scalable relational storage and technology with techniques that follow the same principles that made relational databases scalable. These extended techniques include a value-based storage model, highly efficient QuickXScan XPath evaluation algorithm and XPath index-based access methods.

Our experience also confirms that XML processing is highly CPU-intensive, with major contributors being parsing and validation, traversal, and serialization, despite our efforts in reducing the CPU cost in these areas.

We also realize that the current implementation of strict XQuery data model for XML storage limits its applications to the data-centric domains. For example, it is not sufficient to store the data model alone to achieve byte-for-byte retrieval that is required for XML content applications with XML resources. Building full-text indexes on top of the XQuery data model, in addition to a LOB-based original content, would cost too much for such applications. An alternative approach would be to use specialized storage for efficient query processing [32], along with an intact copy. Also it remains to be seen if it requires new architectural extensions

beyond the current scope to efficiently model and implement collections and folders with XML.

Some future work includes tuning of the new extensions, new efficient methods (e.g. join order enumeration), and new capabilities, such as more complete XQuery and full-text search.

7. ACKNOWLEDGMENTS

I would like to thank many talented engineers of the System R/X project and acknowledge contributions from engineers and researchers of IBM Almaden, Toronto, Poughkeepsie, and Silicon Valley Lab. Anonymous reviewers provided valuable comments to improve the presentation.

8. REFERENCES

[1] Aho, A. V., Sethi, R. and Ullman, J. D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[2] Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M., Srivastava, D. and Wu, Y. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002

[3] Balmin, A., Özcan, F., Beyer, K. S., Cochrane, R. and Pirahesh, H. "A Framework for Using Materialized XPath Views in XML Query Processing", VLDB 2004, pages 60-71.

[4] Barghouti, N.S. and Kaiser, G.E. Concurrency control in advanced database applications, ACM Computing Surveys, 23:3, Sept. 1991.

[5] Z. Bar-Yossef, M. Fontoura and V. Josifovski. On the memory requirements of XPath evaluation over XML streams, In *Proceedings of PODS*, 2004.

[6] Bruno, N., Koudas, N., and Srivastava, D. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD 2002*, pages 310-321, 2002.

[7] E. Cohen, H. Kaplan and T.Milo, Labeling Dynamic XML Tree, In *PODS*, 2002.

[8] Dekeyser, S. and Hidders, J. Path locks for XML document collaboration. In *Proceedings of the 3rd Int. Conf. on Web Information Systems Engineering (WISE)*, pages 105–114, Singapore, 2002.

[9] Florescu, D. and Kossmann, D. Storing and Querying XML Data Using an RDBMS, Data Eng. Bulletin, 22(3), 1999.

[10] Florescu, D., Hillary, D., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M., Sundararajan, A. and Agrawal, G. The BEA/XQRL Streaming XQuery Processor. In Proceedings of VLDB, 2003.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of VLDB*, 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: improving time and space efficiency. In *Proceedings of ICDE'03*, Bangalore, India, Mar. 2003.

[13] Graefe, G. Query evaluation techniques for large databases. ACM Computing Surveys, 25:2, June, 1993.

[14] Gupta, A. and Suciu, D. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, 2003.

[15] Helmer, S., Kanne, C., and Moerkotte, G. Evaluating lock-based protocols for cooperation on XML documents, In *SIGMOD Record*, V.33, No.1, March 2004.

[16] H. Jiang, W. Huang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.

[17] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. The VLDB Journal, to appear.

[18] Kanne, C. and Moerkotte, G. Efficient storage of XML data, *ICDE* 2000.

[19] Knuth, D. 1968. Semantics of context-free languages. *Math. Syst. Theory 2*, 2, 127–145. See also *Math. Syst. Theory 5*, 2, 95–96, 1971.

[20] B. Ludascher, P. Mukhopadhysy, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. of VLDB*, 2002.

[21] Neven, F. Extensions of Attribute Grammars for Structured Document Queries. In *Proc. 8th International Workshop on Database Programming Languages (DBPL),* pages 99-116, 1999.

[22] Neven, F. and Van den Bussche, J. Expressiveness of structured document query languages based on attribute grammars. *Journal of the ACM*, 49(1):694-718, 2002.

[23] Özcan, F., Cochrane, R., Pirahesh, H., Kleewein, J., Beyer, K., Josifovski, V., and Zhang, C., et al. System RX: One part relational, one part XML. In *Proceedings of the SIGMOD 05*.

[24] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management*, 2002. Springer LNCS 2490.

[25] Pirahesh, H., Hellerstein, J. M., and Hasan, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. SIGMOD 1992, pages 39-48.

[26] Peng, F. and Chawathe, S. XPath Queries on Streaming Data. In *Proceedings of SIGMOD*, 2003.

[27] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. "Access Path Selection in a Relational Database Management System", SIGMOD 1979

[28] F. Tian, D. DeWitt, J. Chen and C. Zhang, The Design and Performance Evaluation of Alternative XML Storage Strategies, ACM SIGMOD Record, 31(1), 2002.

[29] W3C, XML Query related specifications, See http://www.w3.org/XML/Query

[30] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.

[31] Zhang, G. and Zou, Q. QuickXScan: an optimal streaming XPath algorithm, manuscript available from the author, 2005.

[32] Zhang, N., Kacholia, V. and Özsu, M. T., "A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML", I*CDE* 2004, March 2004.