System RX: One Part Relational, One Part XML

Kevin Beyer[1] Roberta J. Cochrane[1] Vanja Josifovski[1] Jim Kleewein[2] George Lapis[2] Guy Lohman[1] Bob Lyle[2]
Fatma Özcan[1] Hamid Pirahesh[1] Normen Seemann[2] Tuong Truong[2] Bert Van der Linden[2] Brian Vickery[2]
Chun Zhang[1]

[1]IBM Almaden Research Center
650 Harry Road
San Jose CA 95120

[2]IBM Silicon Valley Lab
555 Bailey Road
San Jose CA 95141

Abstract

This paper describes the overall architecture and design aspects of a hybrid relational and XML database system called System RX. We believe that such a system is fundamental in the evolution of enterprise data management solutions: XML and relational data will co-exist and complement each other in enterprise solutions. Furthermore, a successful XML repository requires much of the same infrastructure that already exists in a relational database management system. Finally, XML query languages have considerable conceptual and functional overlap with relational dataflow engines. System RX is the first truly hybrid system that co-mingles XML and relational data, giving them equal footing. The new support for XML includes native support for storage and indexing as well as query compilation and evaluation support for the latest industry-standard query languages, SQL/XML and XQuery. By building a hybrid system, we leverage more than 20 years of data management research to advance XML technology to the same standards expected from mature relational systems.

1. Introduction

XML first became a W3C recommendation in February 1998, as a standard way to delimit text data [42]. It has emerged in the industry as the predominant mechanism for representing and exchanging structured and semi-structured information across the Internet, between applications, and within an intranet. Virtually every industry is working to standardize XML representations for their common business objects. As one industry analyst put it, "Hundreds of vertical schemas, in fields as diverse as government, biology, finance, and travel, are publicly available and being actively used. Undoubtedly, there are thousands more in private hands" [5].

With the advent of Web services and services-oriented architectures, it is quite common for intra-company and inter-company interactions to be processed via XML messages. In such cases, the message is more than the transaction request; it is also a business artifact: a purchase order, an order inquiry, an invoice, etc. Such messages need to be retained for many reasons including auditing, regulatory compliance, and non-repudiation. For example, a large securities clearing house interacting with member brokers using Web services is legally obliged to store the XML messages for non-repudiation. Many of these uses also require extensive search capabilities, and the XML storage must have very high fidelity to preserve digital signatures as required for non-repudiation. So, although XML's original intent was data interchange, an increasing amount of XML is designed to be persistently stored, and enterprises are even persisting XML messages primarily used for data interchange for later analysis.

A large percentage of industries rely heavily on existing relational databases and applications to run their businesses, from which much of the information within the XML document is generated, or into which much of the information from the XML documents will be stored. We believe that the integration of this well-structured relational information with the self-describing XML data is an important evolutionary advance in the data industry.

This paper describes the overall architecture and design aspects of a hybrid relational and XML database system called System RX. The system understands both relational and XML data deeply, with new support for XML throughout the system, including native support for storage and indexing, as well as query compilation and evaluation support for the latest industry-standard query languages, SQL/XML and XQuery.

System RX is an experimental prototype that is currently being implemented as an extension to DB2 UDB. This paper describes the overall architecture and the design of the system. Later papers will describe major subsystems in more detail.

There are three driving factors that led us to build a hybrid relational and XML database system:

(1) XML and relational data will co-exist and complement each other in enterprise solutions. Some types of data are best modeled and stored in a relational format, but other types are best suited for XML. Although the data *could* be normalized into relational tables, it may not be appropriate to do so. There are many examples of this. (a) The data comes from a multiplicity of schemas and the aggregate size of the relational schema to model all the data is unacceptable given the usage. For example, an organization with 1,500 e-forms required over 30,000 relational tables to represent their data, despite the fact that most forms are seldom used. (b) The data has a highly variable schema with respect to time. We refer to such schemas as *dynamic schemas*. The impact of changing the corresponding relational schema frequently makes it impractical to model the data in relations. This is particularly pronounced when the corresponding schema change would require normalization, such as making a single-valued attribute into multi-valued. (c) The data contains many sparse attributes that are only accessed in the context of the parent object. Thus, the cost of normalization is prohibitive and de-normalization is impractical because of limits on the maximum width of a row or maximum number of columns in a table. Hence, there is a need to persist and search XML natively along-side relational.

SIGMOD 2005, June 14–16, 2005, Baltimore, Maryland, USA.

(2) A successful XML repository requires much of the same infrastructure that already exists in a relational database management system. The repository must support all the traditional database properties, such as transactional (ACID) properties, availability, scalability, reliability, usability, manageability and installability. The data must be quickly and efficiently updatable with existing, well-understood isolation levels and semantics. It must have performance characteristics close to a relational system. For high-performance bulk processing of XML data, it is important to have an underlying model that is based on a set-at-a-time processing, as also argued by [23]. Relational database engines are highly scalable as a result of many years of research and tuning. The XML data must be indexable for both parametric and full-text search predicates, and it must be stored in a way that avoids unnecessary joins. This is especially true for common operations like full document retrieval, which plagues the 'shred into relational' approach to XML storage. Furthermore, well-known query rewrite and optimization techniques can be applied to XQuery. The database community has years of innovation in this space and the database industry has a large investment in systems that solidly support these characteristics. Hence, much of the plumbing that supports these characteristics can be reused.

(3) XML query languages have considerable conceptual and-functional overlap with SQL. XQuery [3] and SQL/XML [21] are the two industry-standard languages that have emerged to query business artifacts encoded in XML. XQuery provides a rich query language that supports the hierarchical structure of XML. SQL/XML extends the relational model with an XML data type, constructs to query that type in conjunction with relational data, and functions for converting between relational and XML data. Despite the slightly different focus of each language, they both include many similar concepts including set-based and sequence-based processing, joins, selections, projections, and quantification. Regrettably, they are not directly convertible [31], but a significant portion of a relational data-flow engine is directly applicable to the processing of XML query operations. The principle difference is the introduction of intra-document structure-dependent operations due to the hierarchical nature of the XML data. The potential for such extensibility has already been proven through the integration of object capabilities in SQL. We are further demonstrating the extensibility of such systems with navigational support for XML.

In addition to these key reasons for building a hybrid system, the other consideration that shaped the overall design and architecture of the system was, as mentioned in 1(b) above, the need to support dynamic schemas. A repository must support schema evolution to minimize the impact to applications and existing XML data. Schemas for data represented as XML must be very flexible and can be highly volatile when viewed across a very large time horizon. XML is increasingly being used to represent actual business artifacts such as derivatives contracts, mortgages, and legislative and legal documents. Most of these artifacts have very long retention requirements: decades (derivatives contracts), centuries (mortgages, insurance forms), or indefinitely (legal and legislative documentation). Therefore, it is critical that an XML repository respond seamlessly to changes that occur to the schema. This evolution is either impractical or impossible in more rigidly structured relational systems because the XML documents must retain their original structure, schema, and content to preserve their digital signatures. So, while an individual document typically has an associated schema when it is inserted, a large collection of documents is unlikely to conform to a single schema.

2. System RX Architecture

System RX is a hybrid relational and XML system, as shown in Figure 1. System RX unifies new native XML storage, indexing and query processing technologies with our existing relational storage, indexing, and query processing. The system provides natural means for SQL applications to migrate to manipulating XML, as well as scalable and transactional support for XQuery applications.

Before the system architecture can be described, some background on the standards activities for XML data models and query languages is required, and is covered in the next subsection. In the remainder of this section, we provide a high-level description of the entire system, with detailed descriptions appearing in subsequent sections. We describe related work in Section 9, and conclusions and future directions in Section 10.

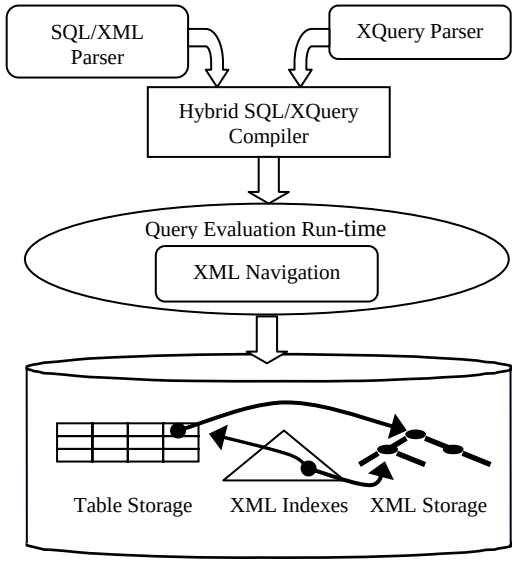

Figure 1. System Architecture

2.1 XML Data Model and Query Languages

System RX supports the two industry-standard languages to query XML data: XQuery [3] and SQL/XML [21]. Defined by the W3C, XQuery is a powerful functional language designed to query both structured and unstructured data. XQuery provides path-expressions [2] to navigate through XML trees and extract XML fragments, as well as expressions to create, sort, aggregate, combine and iterate over sequences, and construct new XML data. XQuery is a reference-based language, and hence subsequent expressions on the result of a path expression may traverse the document in both forward and reverse directions.

SQL/XML [21], which is standardized by ANSI and ISO, defines a new XML data type. SQL/XML defines second-order query functions such as XMLQuery, XMLTable, and XMLExists that take an XQuery statement as input and execute it over the XML values passed from SQL. SQL/XML also includes functions to construct new XML data and to convert between XML and relational data types.

The XQuery data model (QDM) [11] is based on the notion of sequences, which are ordered collections of zero or more items. SQL/XML [21] aligns the XML data type with the XQuery data model, which closes the algebra and allows XML values to be passed back and forth between SQL/XML and XQuery. By building a hybrid system, System RX enables seamless flow of XML data from SQL applications into an XQuery processor.

Relational tables can now define columns that use this new XML data type. This enables existing SQL applications to augment their current relational database designs with additional XML data, and provides an evolutionary path for XML support. Conceptually, in each row of the table, the column contains an XML document or, more generally, an instance of the QDM. This is depicted in the lower portion of Figure 1 by the arc from the table storage to the XML storage.

The XML type has an implementation-dependent internal format that is different from BLOB and CLOB types, and hence usual string operations (such as comparisons, substring, etc.) are not defined for the XML data type. Rather, SQL/XML defines a set of functions which consume or produce the XML data type. Frequently, implementations use BLOBs or user-defined types (UDTs) as the underlying storage for this type, to avoid the prohibitively expensive join processing and more complex insert/update/delete processing necessary to manipulate the normalized relational format. However, in System RX, we provide native storage for XML that we believe can be altered at considerably lower cost over time than a BLOB or UDT. Inserting XML
In System RX, a group of XML documents are defined as a column in a relational table. For example, a bibliography entry, such as a book or magazine, can be described by an XML document and an associated unique identifier that relates it to other tables in the database. It would be defined with the following create table statement:

create table bibliographies(id integer, ***bib xml****)*

where the *xml* descriptor identifies that the *bib* column has XML type. To insert an XML document into a table, it must be parsed, converted into the native XML storage, and indexed, as described in Sections 3 and 4 respectively. We use a new SQL/XML function, *XMLParse,* for this purpose:

```
insert into bibiliographies
  values(1492,
         XMLParse('<?xml encoding="UTF-8"?>
                     <book  price="23.98">
                       <lang>English</lang> ...</book>'))
```

If the XML document has an associated schema, it can be validated by specifying the schema during the insertion. System RX provides a function, *XMLValidate*, which validates an input document with a given XML schema:

```
insert into bibiliographies
  values(1492,
         XMLValidate(
            XMLParse('<?xml encoding="UTF-8"?>
                        <book  price="23.98">
                          <lang>English</lang>
                          ...
                        </book>')
           according to XMLSchema id bib.xsd))
```

The XML storage system remembers the type annotation derived during schema validation and the runtime engine uses this information during query processing. XML schemas are registered with the server, parsed and stored natively for stable access.

Since users can specify different schemas with each insert statement, an XML column may contain data that is validated according to many schemas, as well as those that have not been validated (i.e. untyped), which provides support for *dynamic schema*s.

2.2 Querying XML Data

System RX supports interfaces for both SQL/XML and XQuery in a unified query model, as shown in the top portion of Figure 1. XML-centric users can query XML data and XML views of relational data via XQuery. To provide access to XML data stored in relational tables, System RX provides an input function called *xmlcolumn,* which takes the name of an XML column in a relational table or view as an argument and creates a sequence of XML nodes stored in that column. For example, the bibliographies can be queried in XQuery as follows:

Example 1: Using XQuery to query the bibliographies

```
for $bib in xmlcolumn('BIBLIOGRAPHIES.BIB')/
                  bib[lang/text()='English'],
    $book in $bib//book
where $book/@price < 80
return <bookInfo>
          {$book/@price, $book/author/name}
       </bookInfo>
```

Given the nature of the hybrid system, relational-centric users using SQL will typically be aware of the XML data, but will only extend their relational queries as necessary to include navigation against the XML data using the extensions added in SQL/XML. However, the system can also provide a pure relational view of the database using SQL/XML views that convert the XML data to relational. To query the XML data using SQL, the query simply refers to the XML column and uses the new SQL/XML functions. A query similar to Example 1 can be written in SQL as follows:

Example 2: A similar query written in SQL

```
select T.price ,T. names
from bibliographies as B,
  XMLTable(
    'for $bib in $doc/ bib[lang/text()='English'],
        $book in $bib//book
     where $book/@price < 80
     return $book'
   passing B.bib as "doc"
   columns "price" double path './@price',
           "names" xml  path './author/name') as T
```

The *XMLTable* function receives one *bib* document at a time from the *bibliographies* table, and evaluates the FLWOR expression. For each matching book, a pair is returned: the price as a double, and the list of author names in an XML sequence.

Queries enter the system through either language and are then compiled into an execution plan, as described in Section 5. After parsing, the distinction between SQL/XML and XQuery is discarded in favor of a unified internal representation. The query is modeled as a query graph using an extended query-graph model

[34] to capture a superset of what is possible through both SQL and XQuery (Section 5.1). For example, the internal representation of the queries in Example 1 and Example 2 will be very similar. We can exploit the rich data-flow modeling to perform powerful cross-language optimizations. We extend traditional rewrite optimizations to work with the extended query model and introduce some rewrites specific to the XML query languages (Section 5.2). After the rewrite phase, the portions of the query that can be answered by an XML index are detected. This is significantly more challenging than for relational indexes (Section 5.3). The query then enters our enhanced cost-based optimizer to plan the new XML index and navigation operators (Section 5.4).

Once the evaluation plan is generated, the query can be evaluated by the combined relational-XML runtime engine, which is a relational run-time extended with XML navigation and indexing capabilities, as described in Section 8.

3. XML Storage

The amount and nature of the XML data can be quite variable, depending on the application. Small XML documents often do not exceed 3K bytes, while the largest XML documents can be multiple gigabytes in length. Small collections of documents have a few thousand of documents, while large collections have billions. Some applications retrieve the entire document, but others select only a small portion. Some of these documents, once created, are strictly read-only, while others are frequently updated.

Designing a storage system for such widely varying workloads is challenging. Documents must be able to span disk pages, and even a single text node could be larger than a page. We cannot afford to traverse every node of a gigabyte document to retrieve a small subtree, nor rewrite it whenever a single byte changes. Therefore, the system must support direct node access and sub-document updates. The store must support XQuery's node reference semantics in a concurrent system under all of SQL's isolation levels, and it must support rollback and recovery.

To meet these requirements, System RX introduces a new native XML storage format to store XML documents as instances of the XQuery Data Model (QDM) in a structured, type-annotated tree. By storing the binary representation of type-annotated XML trees, we avoid repeated parsing and validation of the document. However, the binary representation maintains the salient features of the document, so that any digital signatures on it are preserved. Furthermore, every node contains pointers to its parent and children to support efficient navigational queries. Path expressions are evaluated directly over the native format on buffered pages without copying or transforming the data. To achieve uniform performance for small and large XML documents, the store also supports direct access to a node, which avoids the top-down traversal through every node from the root to the target node.

During insertion into the store, the XML parser produces SAX (Simple API for XML) events representing nodes and content, which are collected into regions of related nodes and written to standard fixed size buffered pages with page slots and data area similar to [14]. Parent / child relationships between nodes that are within a page are stored as page slot entries for fast access. Parent / child relationships that span pages use a logical lookup to get to the parent, child or sibling node. Each node is given a unique identifier that gives nodes both a logical and physical addressability that can be used by indexing and query evaluation. Nodes with large content are 'chunked' across multiple pages, while nodes with a large fan-out of children are 'continued' across multiple pages. A persistent dictionary that maps strings to identifiers is used for compression of redundant namespace URIs and node names. This compression from strings to identifiers also improves the evaluation of path expressions by replacing string comparisons with integer comparisons. The same dictionary is used for all XML columns, achieving even greater compression, but more importantly, allows the system to easily perform navigation on a mixed set of nodes from several XML columns.

Each element node has a set of child slots for its associated attribute and ordered children. These child slots have hints within them to give an indication of what the child represents making it possible for fast navigation across a context node's set of children to find potentially qualifying children without actually visiting each child node. This is important because the node may be on another page and require additional I/Os. For example, when looking for a book with a specific 'language' child, all attributes of the book element can be skipped, as well as all children not having a 'hint' with the name 'language'. Small text and attribute nodes, and their respective content, are in-lined within their parent node when possible for compression purposes, and also for locality of reference when evaluating path expressions. This is similar to [25] but with a more logical addressability, since each child can be pointed to directly via a RID (row identifier) plus an index into the element node's child slot array.

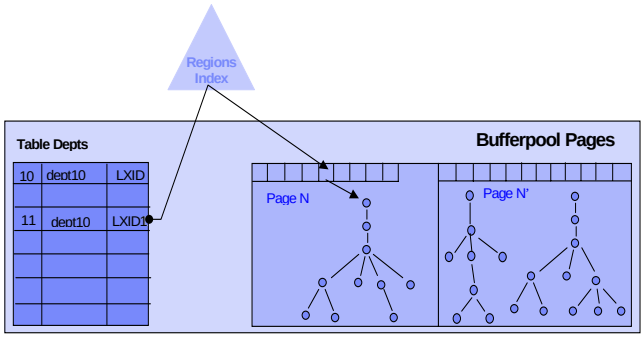

Figure 2. Native XML type Column Storage

Regions of nodes for a given XML document are grouped together on pages and linked together by a logical *regions index*. Inter-region access to a node outside the current region (i.e., the parent, the next sibling, or to a child node) requires a logical lookup through the regions index. This level of indirection enables several important features. Nodes can be accessed without traversing from the root by looking up the region that contains the node in the regions index using the node's identifier. Nodes can be inserted, updated, or deleted without affecting most of the other nodes in the document, and portions of the document can be reorganized.

The XQuery language uses node reference semantics in its results to allow additional navigation. This is achieved by versioning. As a document is updated, only those regions updated are versioned, leaving multiple versions of the node in the regions index. Only the current version is available in the XML value indexes; new readers will always find the latest version. Readers that have already qualified a node will continue to see a consistent version of the document for any traversals from that node. Old region

versions are removed when there are no references to it, which is detected based on the oldest reader of the table.

Regions are initially clustered, but can become fragmented during updates. A data reorganization utility can re-cluster the regions of a document when needed. The grouping of all regions for a given document in the regions index allows the prefetching of pages when appropriate, e.g., during serialization of the complete document.

The XML store uses much of the existing relational infrastructure. For example, the existing table spaces, buffer pools, lock manager, and log manager are used without modification. Reuse of existing relational engine services for transactions, concurrency, scalability, and recoverability simplified the implementation and is essential for coexistence with the relational data. Buffer pool services are used to keep active pages in memory. Record management services handle the placement of the nodes within a given page, while logging of nodes or sets of nodes enables failure recovery.

4. XML Indexing

In relational systems, indexing is the most important feature for query performance, and this remains true for XML data. However, the rich structure of XML introduces new challenges. The obvious interpretation of an index on a relational column is that the values of the column are organized so that the system can quickly locate the rows that satisfy range predicates on the column. But what does it mean to create an index on an XML column? We considered three classes of XML indexes:

- **Structural indexes** map distinct node names, paths, or tag-based path expressions to all matching node instances (e.g.,[26]), or they map node identifiers to nodes instances (e.g., the regions index).
- **Value indexes** allow quick retrieval of nodes based upon the node's data value.
- **Full-text indexes** map tokens (e.g., words) to the nodes that contain the token.

Each of these index classes is useful for some query, but we believe that value indexes are significantly more useful than structural indexes for our expected query workloads. Consider a query workload on employee records. Which is the more likely query: find employees with any recorded interests, or find employees interested in nanotechnology? The relational analogy is to find records with a particular value in some column versus looking for null values. That said, our value indexes do support some structural predicates.

Because our XML data is commingling with existing relational data and will be used by future versions of existing applications, we require our value indexes to support all the features of the existing relational system. This includes: transactions, concurrency, recovery, scalability, fast insertion, efficient update, reorganization, backup/restore, load, etc.

As a semi-structured data model, XML is a bridge between the rigid structural world of relational systems and the free-form world of text documents. Full-text indexing of XML data is required to complete that bridge. Therefore, we are extending our relational text indexing support [30] to include XML data.

As with relational systems, applications typically cannot afford to index every item. XML compounds the issue because of the sheer quantity of items that can be indexed. For example, not only can a range predicate be on any simple node in the document (the "leaf" elements and attributes), but also the processing instructions, the comments, the text nodes (which differ from their containing element), and the interior nodes (as the concatenation of all text nodes below it). If we only support indexing every item in the XML document, then the index storage would be several-fold larger than the original document. Moreover, the number of I/Os required to transactionally maintain the indexes would be staggering. Therefore, we support the indexing of nodes that are returned from a simple XQuery, as shown in the (simplified) CREATE INDEX DDL in Figure 3. However, we need to introduce a phase during query compilation to determine which indexes can be used, and this is described in Section 5.3.

ddl ::= CREATE INDEX *index-name* ON *table* '(' *xml-column* ')' USING '*pattern*' AS *type*

pattern ::= *namespace-decls?*
((/ | //) *axis?* (*name-test* | *kind-test*))+

axis ::= '@' | child:: | attribute:: | self::
| descendant:: | descendant-or-self::

name-test ::= *qname* | * | *nsprefix*:* | *:*ncname*

kind-test ::= node() | text() | comment()
| processing-instruction(*ncname*?)

Figure 3. Create Index DDL

The DDL also describes how the selected nodes are indexed. Because the XML column supports dynamic schemas, we cannot infer the data type associated with the indexed nodes. Therefore, the user must specify the data type used to index the nodes. The reuse of our existing relational index manager introduces a few minor restrictions on the supported XML data; for example, we cannot index arbitrarily large XML strings unless the strings are hashed. The way XQuery treats xdt:untypedAtomic data is challenging for indexing. The general comparisons (=, >, etc.) dynamically cast an untyped operand based upon the type of the other operand, which implies that untyped data must be indexed in every data type that it can be cast to. For example, is the untyped value '1234' a character string, a number, or a hexBinary string? The answer is any of these, depending upon how any particular *query* treats the value. To avoid casting untyped data to every possible data type, the index requires a target data type.

Ultimately, we are creating an index on the cast of the node to the indexed type, taking into consideration the node's type annotation derived during validation. This implies that some string-valued nodes appear in a numeric index, and that all nodes appear in a string index. These semantics are critical when we determine index eligibility. We carefully considered the proper action to take for nodes that match the pattern but cannot be cast to the index type. We decided to ignore the cast error and simply not create any index entry for it. The reason is that the node might never be cast to that type during an actual query and we did not want applications to deal with esoteric errors caused by the creation of an index. This implies that if a query does attempt to cast the node to the index type in a comparison and the index is used to answer the query, the error might not be detected.

Under the covers, an XML index is implemented with two B+Trees. The *path index* maps each distinct reverse path *(revPath)* to a generated path identifier *(pathId).* A reverse path (*revPath*) is a list of node labels from leaf to root – compressed into a vector of label identifiers. To make an analogy with relational systems, the path index is like a dynamic version of the COLUMNS catalog that slowly changes as documents are inserted. The paths are stored from leaf to root for efficient processing of descendant queries such as *//name* which only bind the tail of the path. Such queries bind a prefix of *revPath,* which can be found with a range scan on the path index.

The *value index* consists of the following key:

pathId, value, nodeId, rid

The *value* is the index's representation of the node's data value when cast to the index's data type. The *rid* identifies the row in the table and is used for locking. The *nodeId* identifies a node within the document using a Dewey node identifier [38] and can provide quick access to a node in the XML store through the regions index without accessing the table. The order of the keys in the value index is again a tradeoff. Placing the *pathId* first allows for quick retrieval of specific path queries. For example, if we create an index on *//name*, which might match many paths, then a query on */book/author/name* still has consecutive index entries. The path index plus placing the *pathId* first in the value index gives us some structural index support as well. But the tradeoff is that a query like *//name='Maggie'* will need to examine every location in the index per matching path.

5. XQuery Compilation

Figure 4 gives an overview of the hybrid query compiler. We have implemented a new component, the XQuery parser, and extended all other components in the compiler to process the XQuery data model and the XML query languages. First, an SQL statement or an XQuery expression is compiled into an internal data flow graph. Next, rewrite transformations are applied to normalize, simplify, and optimize the data flow. The optimizer uses this graph to generate a physical plan, which is translated into executable code by code generation. In this section, we describe each component and discuss tradeoffs that led to the current design.

Two major decisions impacted the whole compiler design. First, System RX does not implement static typing. XQuery [3] has both static and dynamic semantics, depending on when type-checking is enforced. Static typing is too restrictive for dynamic schemas, as each document insertion or change in schema may result in recompilation, even rewriting of applications. Recall that System RX does not require that all XML documents in an XML column conform to a single schema, or to a collection of conforming schemas. We expect non-conforming changes between schema versions. For example optional fields may become mandatory. Although System RX does not implement static typing, it still exploits any type information wherever possible.

Second, System RX does not normalize [9] the XPath expression into explicit FLWOR blocks, where iteration between steps and within predicates is expressed explicitly. Instead, path expressions that consist of solely navigational steps are expressed as a single expression. This impacts query modeling, rewrites and optimization, and will be discussed in the following sections.

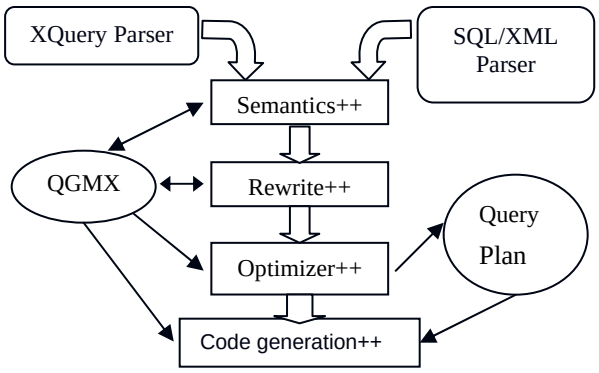

Figure 4. Hybrid SQL/XQuery Compiler

5.1 XML Query Modeling

In System RX, we represent XQuery via an internal query graph model (QGM) [34], which is a semantic network used to represent the data flow in a query. Although it is fine-tuned for efficient relational query processing, the data flow graph is more generic than relational algebra. As QGM is designed to be extensible [18], it is fairly easy to add new entities and capture multiple data models. Moreover, the data flow model proved to be very fruitful for other extensions in relational systems [19] [30]. It is important to emphasize that we are not translating XQuery into relational algebra or into SQL. On the contrary, we are augmenting our internal data flow model with native constructs that are specific to XML and that represent complex navigation of XPath and XQuery.

In its simplest form, a QGM graph consists of operations and arcs which represent the data flow between operations. XQuery provides similar constructs to iterate over XML data and apply predicates to join and sort data. We represent these operations of XQuery with the existing QGM entities, and introduce new entities to represent path expressions and sequences.

To coalesce the relational and XML data models, we first needed to decide how to represent XQuery sequences within the context of a relational engine. Recall that SQL/XML [21] introduces XML as a column in a relational table and is based on the XQuery data model. To accommodate this new SQL data type, we represent an XQuery sequence as a column value in QGM. Some XQuery expressions consume a sequence as a whole, while others iterate through the items in a sequence. We provide an operation that unnests the items in a sequence for set processing, and another operation that aggregates a stream of items into a sequence.

The focus of earlier research has been on efficient representation and execution of path expressions. On one side of the spectrum are fine-grained approaches. For example, XPeranto [37] represents each axis and name test on a single path step as a selection. As a result, a complex path expression requires a series of selections and a complex multi-way join operation. Although this approach was designed for a system in which the XML data is actually shredded into relational tables where the final goal is to compose navigation steps with XML construction, it turned out to be incapable of handling large queries and representing the full XQuery language. In particular, it implies an order of execution for navigation. On the other side of the spectrum is the coarse-

grained approach, where many binding path expressions are represented in a pattern tree, such as generalized tree patterns [7]. But such systems only deal with single FLWOR blocks. In System RX, we take a medium-grained approach, where we initially represent each path expression as a pattern tree in which there is only *one* bound variable. The main reason behind this decision is to represent each data flow (e.g. each variable) explicitly, so that semantics and rewrite analysis, which is built on explicit data flow representation, can reason about the query more efficiently. In addition, this approach allows us to represent not only path expressions as binding patterns, but also other FLWOR expressions, and support full compositionality of XQuery. However, as we explain in Section 5.2, after in query rewrite, we try to consolidate all navigation within a query block into a single pattern tree representation.

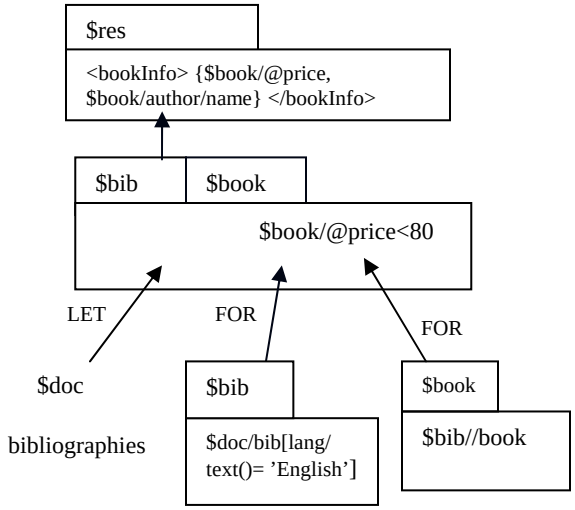

Figure 5. Graph Representation of a FLWOR Expression

The query graph model representation for the query in Example 1 is shown in Figure 5. Each rectangular box represents some computation. In this example, the topmost box is the FLWOR box, which computes the result of the FLWOR block. The middle box computes the tuple of bindings and applies the *where* clause predicates. Each lower box computes one binding, and represents each variable binding as a separate data flow. We omit the correlations between boxes for readability. The annotations on each arc represent whether or not the resulting XQuery sequence is to be iterated over (FOR), or aggregated into a sequence (LET).

5.2 Query Rewrite Transformations

The QGM graph output by the XQuery parser needs to capture the full compositionality of the XQuery expressions. As a result, it may not be the most compact or efficient representation of the query. The goal of the rewrite transformations is twofold: First, we try to optimize the data flow by consolidating some operations, eliminating redundant computation, and applying several logical transformations. Second, rewrites try to normalize the QGM representation, so that the query optimizer gets the same graph as input for semantically equivalent queries and has maximal flexibility.

To support the dynamic schema requirement, we decided not to apply any schema-based transformations [12][17]. With dynamic schemas, such transformations may require frequent recompilation and rewriting of queries and applications. For example, a schema change that converts a single-valued attribute/element into a multi-valued one will invalidate the query plan if such schema-based transformations are applied.

By building XML processing on top of a robust relational engine, we are able to exploit many sophisticated rewrite transformations. These transformations are designed to optimize the data flow, and hence some are also applicable to XQuery. For example, rewrites that merge nested query blocks, or eliminate unused variables are directly applicable. A hybrid system also enables query rewrite transformations across language boundaries by seamless compilation of XML querying functions of SQL/XML (i.e., XMLQuery, XMLExists and XMLTable) [21] into a single query graph.

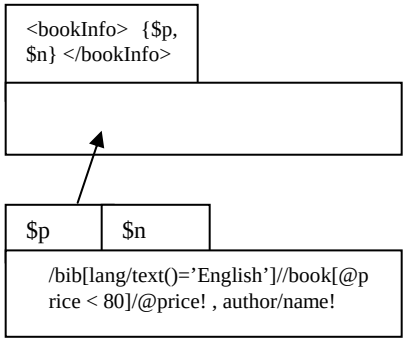

Figure 6. Graph Representation After Rewrite Transformations

In addition, we have developed several transformations specific to XQuery and XML navigation. Specifically, we try to push navigation down closer to the base data access to avoid navigating intermediate or constructed XML fragments and to exploit XML indexes. We also try to consolidate all path expressions in a single FLWOR block into one pattern tree that is annotated with several flags to represent FOR vs LET paths, whether an empty sequence needs to be created when there is no qualifying node, whether duplicates should be eliminated, etc. This pattern tree computes multiple bindings. Another important rewrite tries to push down predicates in the *where* clause into binding path expressions, enabling XML index matching for value and general comparisons. For example, the query graph shown in Figure 5 is transformed into the graph shown in Figure 6. Note that the lower box computes tuples of bindings, $p and $n in this case.

Fnially, we conclude this section by identifying two of several open problems. First, XQuery general comparisons are not transitive [3]. Hence, many rewrites that rely on transitivity are not applicable. Second, some of the XQuery expressions are sensitive to order, blocking other transformations.

5.3 Index Eligibility

An index is eligible for use during query evaluation when we can prove that the index contains a superset of the results required for the query. We adapted the XPath containment algorithm described in [1] to identify the indexes that are able to answer a part of a query. At a high level, this includes showing the following:

1. The query implies a predicate of the form: *$col/<path-expr> <cmp> <expr>*. In Example 1, *$doc/bib//lang/text() = 'English'* and *$ doc/bib//book/@price < 80* match this form.
2. The indexed column is used in a FOR binding. LET quantification is not particularly useful because the entire result is required when the predicate is satisfied; i.e., either all the results or none of the results are returned.
3. The path expression must produce the same or a subset of the indexed nodes.
4. The data type of the comparison must match the data type of the index. The data types are not required to be identical; for example, all numeric comparisons match the double index. However, the comparison performed by the index must imply the required comparison. We perform type-inferencing on the query graph to determine the type of the comparison based on the types of its arguments. Even with dynamic schemas, type inferences can be made. For example, literals, casts, arithmetic, and type-tests all establish data types of parts of the query.

If we create the following indexes:

create index I1 … '/bib/lang/text()' as string hashed
create index I2 … '//@price' as double
create index I3 … '/bib/book/@price' as double
create index I4 … '/bib//book/@price' as string(100)

The indexes I1 and I2 are eligible for the query in Example 1. Index I3 cannot be used because the data might include /bib/-collection/book/@price, which is not indexed, and I4 cannot be used because a numeric comparison is not compatible with a string index.

5.4 Physical Plan Generation

Physical plan generation is the phase in which the optimizer scans a QGM graph containing relational and XML entities and produces alternative execution plans. The optimizer utilizes data statistics to build a cardinality model, which is then used to estimate costs for the execution plans. Intermediate plans can be pruned based on costs and plan properties such as the order of input data. The final plan with the cheapest cost is chosen for execution.

A physical plan is a model of query evaluation at runtime; each physical operator models a runtime operation. Physical operators can be chosen at different granularities. They can be modeled at a primitive level, so that a complex run-time operation is composed using a tree of these primitive operators. Alternatively, a complex run-time operation can be modeled using one physical operator. The choice of physical operators affects cardinality and cost modeling. In System RX, new navigation and index runtime operations were introduced to support native XML processing (see Section 7); correspondingly, new physical operators were needed to model them. Our decision was to use one operator to model each. Part of the reason for this coarse-grained approach is that the complex new runtime operations cannot be broken down to trees of primitive operations. And part of the reason is that it is not necessary to model the runtime operations in fine detail, because there is no alternative in re-ordering the primitives at runtime. It is important to point out that modeling runtime operators in a coarse-grained manner also opens the opportunity for the runtime operations to be flexible and adaptive, based on information available during execution.

We modeled the XML navigation run-time operation using the physical operator XML Scan (XSCAN, analogous to the relational table scan), and modeled the index runtime operation using the physical operator XML Index Scan (XISCAN, analogous to relational index scan). A third new physical operator that we introduced is XANDOR, which models XML index ANDing and ORing. Figure 7 is an example that illustrates the execution plan generated for the query of Example 1, given in Section 2.2.

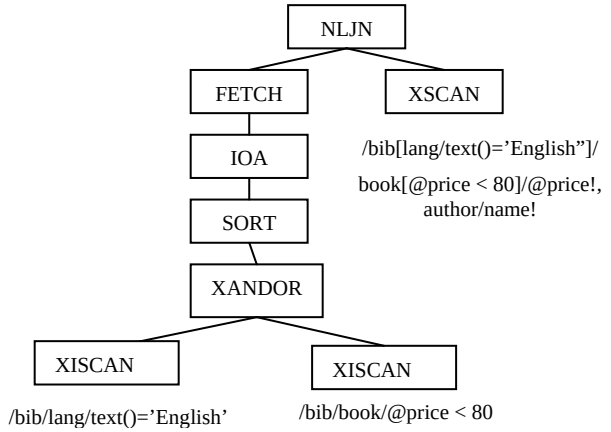

Figure 7. A physical plan for index ANDing

Much of the relational optimizer infrastructure is reused, including rule-based plan generation, join enumeration, join order and join method selection, computation and propagation of properties, and the cardinality and costing framework that we use to cost and prune plans. In particular, by utilizing the extensible rule-based plan generation mechanism [29], as well as extensible operator and plan data structures, plans with the above new XML operators are created by simply incorporating new rules. Since System RX allows seamless compilation of XQuery and SQL into a single query graph, the optimizer is able to generate plans with mixed relational and XML operators, and interchange them to produce alternative plans with different execution orders.

When extending the relational optimizer for hybrid relational and XML processing, new challenges arose. Some of the challenges were due to the fact that we were no longer dealing with a simpler relational model, but the more complex XQuery model and XML data. Other challenges came from adapting the existing architecture and engineering new functionality into the existing relational legacy system. Currently in System RX, we have made some simplifying design decisions, with the intention of developing more sophisticated solutions in the future and refining them over time once we have gained more experience with the system and its applications.

A case in point is our choice of plans. Because our statistics, cardinality, and cost models are still quite simple -- much in the spirit of early System R [35] – we currently we choose plans using a combination of costs and some heuristics. For example, we always choose indexing plans over non-indexing plans; this follows the mostly true assumption that we can use the index to efficiently narrow down data for further processing. Another example is that we order the legs in index ANDing and ORing by the resulting cardinality, without considering correlations. Furthermore, the

new XML runtime operations are able to adapt to the system environment, as well as the data and query at hand. This can be regarded as runtime heuristics that complement the compile-time decisions made by the optimizer.

System RX is in its early age, and there are still unsolved issues, which constitute interesting research problems. In the hybrid system, because we allow XML-typed columns, XML sequences can flow in place of the usual relational atomic values. Therefore in a single query, we can have the traditional tuple stream combined with sequences nested in the tuples. This extension takes the current system beyond the flat relational model. Aspects such as interesting order derivation, cardinality, and cost analyses must therefore be extended accordingly. Another interesting problem comes from the coarse modeling of physical operators. The XML navigation and the index runtime operations that we model with XSCAN and XISCAN operators can be very powerful and complex (see [24] for a description of the XML navigation operator, and [6] for the description of an operator similar to our index runtime). In addition, as we have mentioned, the operators can adapt to runtime information. Furthermore, the XML data that they operate on can be highly irregular. Storage characteristics such as clustering differ dramatically from those in the relational system or are simply absent. For these reasons, cardinality and cost modeling of these XML operations can be difficult, particularly with traditional analytical methods. We have on-going research efforts to address the above issues.

6. Query Run-Time Evaluation

To evaluate queries over XML data, System RX had to extend the relational query runtime to support the XQuery and SQL/XML operators. There were 3 major components added for processing queries over XML: (1) **XML Navigation**, (2) **XML Index Runtime,** and (3) the **XQuery Function Library**. In addition, several adaptations of existing relational runtime operators were required to support the new XML data type.

One issue that influences all aspects of the XML runtime is the dynamic nature of the XML data type. In the relational setting, all the types are known at compile time. The types of the columns are specified at DDL time and are unambiguous. XML data might have no schema associated with it. It might have a schema that has ambiguous type definitions (e.g. XML Schema [43] **union** construct), or in the extreme case, each XML element can be annotated with a basic type using the **xsi:type** attribute. To accommodate such uncertainty, the runtime support for XQuery relies on dynamic type dispatch.

6.1 Index Run-time

The XISCAN operator finds XML nodes that satisfy a predicate using an XML index. The general form of the predicate is *start-val* ≤ *path-expr* ≤ *stop-val*, which represents a range scan on the values of nodes with a path that matches the non-branching path expression *path-expr*. Internally, this results in two or three implicit nested-loop join operations.

First, the path index is used to find the set of paths that match *path-expr* by scanning the range of *revPath* values for the "known" tail of the *path-expr*. Subsequently, the *revPath* is matched against the full pattern. For the path expression *//bib/-/name*, all *revPath* values between *name* and *namf* are scanned and then checked for a *bib* element.

For each matching path, the value index is then probed with *value.pathId = path.pathId,* and the bounds specified in the *start-val* and *stop-val*. However, in our data model, the bounds themselves can be sequences, because cells in an SQL/XML table and LETs in XQuery represent sequences. For equality predicates, this results in one scan of the value index per matching *pathId* and per item in the *start-val* sequence. For range predicates, only the minimum/maximum value of the start/stop value is required.

The XANDOR operator combines the nodes that are output from multiple XISCAN operations, and implements branching path expressions using AND and OR using only the XML indexes. Because we use Dewey node identifiers, we only access the nodes with predicates ("leaf" steps) and avoid accessing the large number of branching nodes (which do not have a predicate). The details of XANDOR are beyond the scope of this paper, but it is similar in spirit to holistic twig joins [6].

6.2 XML navigation

The XML Navigation (XNAV) runtime module evaluates paths and predicate constraints over the native XML store, by traversing the XML store following the parent-child relationship between the nodes. It returns node references (logical node identifiers) and atomic values to be further manipulated by other runtime operators. XNAV is represented in the optimizer plan by XSCAN operators, as shown in the example in Figure 7 . Similar to the relational SCAN operator, XSCAN can also apply query predicates to reduce the size of the data returned by the operator. However, an XML document can correspond to one or several pre-joined relational records, or even a whole database. This makes the XSCAN operator more complex in terms of query features as well as robustness.

Design principles. XNAV has been designed to provide efficient processing of the pattern trees generated by the compiler. In XQuery language terms, it roughly corresponds to a FLWOR block, binding several variables. Unlike other approaches in which every XPath step is modeled as a separate operator [16][37], a single XNAV operator can evaluate multiple steps from multiple XPath expressions in the query. This reduces the number of operators in the query plan and eliminates the overlap in the evaluation of the individual steps. Internally, XNAV breaks the input query steps into *path groups*, each of which is evaluated using a one-pass algorithm similar to the one described in [24]. As with the relational SCAN operator, XNAV interfaces with the rest of the runtime using a tuple based interface and returns tuples of bindings. Each binding has the XML data type and can be a singleton or a sequence of items, each item being a reference or an atomic value. Finally, XNAV is designed to perform using limited memory and uses paged data structures for the intermediate results.

XNAV trees. The evaluation in XNAV is driven by a runtime query tree representation, the XNAV tree. XNAV trees are produced during the threaded code generation phase from the QGM representation of the query. As opposed to QGM, which is targeted for semantic analysis and query rewrite, the XNAV trees are aimed to make the runtime processing as streamlined and as efficient as possible. Figure 8 shows the XNAV tree for the run-

ning example. This tree represents all the 5 path expressions in the original example query.

XNAV trees have structural and predicate parts. The structural part contains the XPath steps of the connected paths and has a query root node. Each node can have attached predicates. Predicate operator nodes can point back into the structural part of the XNAV tree when a structural node is an argument of an operator. In the running example, XNAV re-evaluates the predicates evaluated by the index to guarantee correctness of the result. In Figure 8, predicate operators are shown using oval nodes.

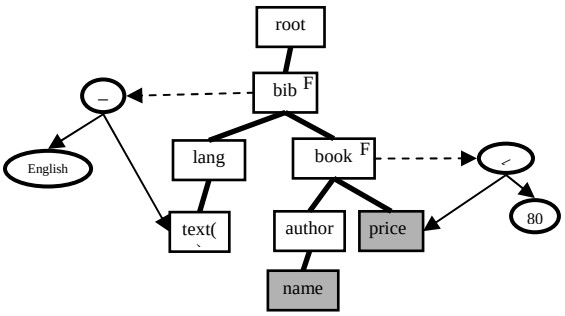

Figure 8. XNAV tree for Example 1

The query in the example above returns tuples of **price** and **name** bindings of the qualifying books. These two nodes are the extraction points indicated by gray fill color. XNAV trees fully capture the semantics of the query fragment using the data in the nodes. For example, in the example, the **for** nodes are marked with an 'F'.

Processing flow: We illustrate the processing of a single path group using the example in Figure 9. We assume that the document is inserted in the native XML store and the query is parsed and translated into the XNAV tree shown in the figure. The existential check for **c** nodes is translated into an invocation of the *effective Boolean value* (EBV) XQuery function [3].

To evaluate the query, XNAV traverses the document in depth-first order and alternates between two modes: **matching** and **tuple construction**. The document nodes are matched to the query using a data structure named **work array** (WA). Figure 9 shows the status of the WA during the document traversal. The WA keeps track of the query steps that have been matched, and the relevant tags to look for next in the document. Initially, the work array contains only an entry corresponding to the root that matches the input context node. The WA grows/shrinks when entering/exiting a document node that matches a query node. In the example, after matching the context node, an entry corresponding to **a** is added to the WA to indicate that next we are looking for **a** elements in the document. When an **a** element is found in the document, entries for **b** and **c** are added to signify that within the **a** element we are looking for **b** and **c** nodes. Each WA entry has several fields to aid the matching. Figure 9 shows two of these: an integer **level** is used to support child axis matches; a **status** flag is used to record if a document node has been found that satisfies all the predicates of the query. As the entry for **a** can stay on the WA across multiple **a** elements, the status flag is turned on if any **a** element satisfies the conditions, reflecting the existential semantics of the XPath language. More details of the matching algorithm can be found in [24].

During the traversal of the document, XNAV skips the nodes that will not affect the result of the query. XNAV relies on the WA to determine which nodes to skip. For example, if all entries in the WA correspond to child axis steps, all document nodes that do not match any WA entry can be skipped.

Tuple construction and buffer management. During the matching phase, nodes that could be part of the result or are needed for predicate evaluation are collected into node buffers. These buffers might contain a reference to the node or its atomic value, depending on the use of the node and the size of its atomic value. In the example above, **c** nodes and **b** nodes are saved for predicate evaluation and to be returned as results. XNAV applies several techniques to reduce the number of buffer entries required.

When the result of XNAV is a tuple with multiple bindings (as in the example in Figure 8), buffers for each of the extraction points are put together into tuples using an algorithm that performs a variant of a merge-join over the node identifiers of the ancestors. This **tuple construction** process is described in more detail in [24]

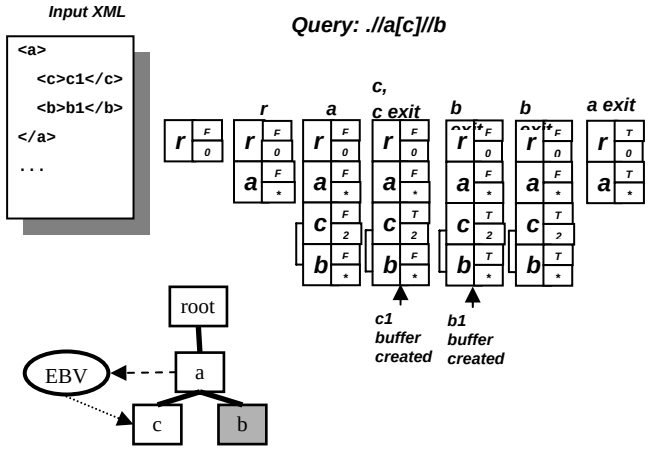

Figure 9. Example XNAV Evaluation

Multi-pass processing. The single-pass algorithm as outlined so far has several advantages: it is order preserving, has predictive traversal, and often minimizes the number of visited nodes. However, there are several cases where the one-pass algorithm as described above is not suitable. Although in general several branches can be evaluated in one pass, in some cases, the query might have branches that force navigation in different directions. For example, considering the query **//a[.//b > ../c],** from an **a** element node we need to navigate both through the descendants of **a** and upwards toward its parent. In other examples, the single-pass algorithm might generate too large intermediate results or traverse more nodes than necessary. For example, in the query **//a[@b >5]//d,** evaluating the predicate before examining the whole **a** sub-tree can eliminate **a** nodes that do not satisfy the predicate. In such cases, the XSCAN operator builds a set of correlated XML navigations, each evaluating a group of XPath steps using a one-pass algorithm. Packaging more than one XNAV into one operator avoids the expense of the operator invocation and allows for sharing latched storage pages across all the groups within a single operator.

7. Related Work

In the recent years, many different approaches have been proposed for XML data management both in academia and in industry. They can be classified mainly in two groups: native XML management systems and systems that reuse an underlying relational DBMS.

Relational based approaches either shred the XML documents into relational tables using some sort of encoding [4][8][13][15][28][36] or use a BLOB column to store the XML document [10][32][33]. The main advantage of this approach is that it requires no modification to existing relational engines, while exploiting their maturity, extensive tuning, proven scalability and sophisticated optimizers.

Shredding based approaches, on the other hand, need to translate an XQuery into SQL for evaluation. As [31] argues, due to the semantic mismatch between XQuery and SQL, not all XQuery expressions are translatable into SQL, or they translate into inefficient SQL statements. DeHaan et.al [8] articulates that a relational engine needs to be enhanced with XML specific primitives for efficient execution. More comprehensive review of methods for XML to SQL query translation and their limitations is beyond the scope of this paper, and can be found in [27]. In System RX, we reuse the data flow graphs of the relational system, and augment it with several XML specific operations to effectively capture the semantics of XQuery and efficiently execute XML specific operations and hence we are able to support the whole XQuery language, except recursive functions.

Another important disadvantage of shredding based methods is their inefficiency for retrieval of the whole or a subpart of the XML document. To reconstruct the document, they need to access relational tables and execute costly multi-way joins. Document insertion and deletion is also costly as it needs to touch many database records. Another important aspect of shredding approaches is whether or not they require a DTD or an XML schema for storage of XML documents. [39] argues that schema-based approaches, such as the inlining methods of [36] perform better than schema-less approaches such as the edge-table [15]. However, schema-based approaches cannot deal with the dynamic schema requirement, as each change in the schema requires database reorganization, which is very costly, and disruptive. As native XML storage of System RX does not require a specific schema, it is able to support dynamic schemas, and hence provides great schema flexibility. Schema-less approaches rely on encodings of XML trees to detect parent-child or ascendant-descendant relationships, resulting in a complex query with large number of so-called structural joins. Existing relational methods have been shown to be inadequate for efficient execution of these joins [40], without native support in the engine [6].

BLOB based approaches use stored procedures to invoke an external XPath/XQuery processor. The main advantage of this approach is fast retrieval of an entire XML document and full schema flexibility, as any XML document, irrespective of its schema, can be stored. Due to the loose coupling of the query processors, usually the entire XML document is brought into memory before processing, severely limiting the size of the data and optimization possibilities. As a result, search and retrieval of XML fragments from the document is relatively slower. Moreover, these approaches cannot support sub-document level updates; they need to replace the whole document. Compared to BLOB based approaches, System RX enjoys a similar fast retrieval of full documents as it stores the entire document together, but does not suffer from inefficient search, as it uses a directly traversable parsed structured tree format.

The second alternative to XML data management is to build a native XML database. Examples of native systems include TIMBER [22], Niagara [20] and Natix [14]. Systems such as Niagara [20] and TIMBER [22] break the XML document into nodes and store the node information in a B+-tree, with all document nodes stored in order at the leaf level. This allows for efficient document or sub-tree reconstruction by a simple scan of the leaf pages of the tree. In Niagara, additional inverted list indexes are created to enable efficient structural join algorithms for ancestor/descendant paths. However, these systems only deal with XML data and do not support SQL or relational storage.

More recently other native storage of XML documents has been proposed in [25] and [41]. In our work we take a similar approach where the XML data is stored as in a native tree format in which document nodes are in most cases clustered together on a page. Bulk processing is performed using indexes, while the storage is optimized for fast navigation to evaluate the non-indexed portions of the query. Parent-child and ancestor-descendant traversal does not require a join between different tables. Since most XPath expressions require these types of traversals, this scheme allows for efficient access to the data.

Finally, there are many XQuery implementations both in academia and industry. A comprehensive list of public XQuery implementations and links can be found on the home page of W3C XQuery working group (http://www.w3.org/XML/Query), and a detailed discussion is beyond the scope of this paper.

8. Conclusions & Future Directions

We have described the architecture and overall design of System RX, a hybrid relational and XML data management system. To the best of our knowledge, this is the first truly hybrid system, which natively supports both relational and XML data. We believe such a system is fundamental in the evolution of enterprise data management solutions, as XML and relational data will coexist and complement each other. System RX is designed to support efficient bulk processing of high volumes of XML data. By compiling both SQL/XML and XQuery statements into a uniform internal representation, System RX enables cross language optimizations.

Going forward, a hybrid system enables relatively easier incorporation of more traditional data management tools, such as triggers and materialized views into XML data management. A hybrid system allows us to leverage more than 20 years of data management research to advance XML technology to the same standards expected from the mature relational systems. To attain this goal, we identify several potential areas for future research. First, mixing traditional tuple streams with XQuery sequences introduces interesting challenges for rewrite transformations and query optimization. Dealing with dynamic schemas, which we think is a critical benefit that must be leveraged with XML data, requires more robust and adaptive run-time operators. Finally, certain properties of the XQuery language, such as node identities and reference based semantics of XPath, pose interesting challenges to parallelization. These areas are all key elements of enterprise

data management systems that are ripe for further exploration.

9. Acknowledgements:

We would like to thank many engineers and researchers at IBM Almaden, IBM Silicon Valley and IBM Toronto Labs for their contributions to the developments of System RX.

10. References

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

[2] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie and J. Simeon, "XML Path (XPath) Language", October 2004, http://www.w3.org/TR/xpath20

[3] S. Boag D. Chamberlin, M. Fernandez, D. Florescu, J. Robie and J. Simeon, "XQuery 1.0: An XML Query Language", October 2004, http://www.w3.org/TR/xquery

[4] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Simeon, "LegoDB: Customizing Relational Storage for XML Documents", VLDB 2000

[5] R. P. Bourret, Personal communication, http://www.rpbourret.com

[6] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching", SIGMOD 2002

[7] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan and S. Paparizos, "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery", VLDB 2003

[8] D. DeHaan, D. Toman, M.P. Consens and T. Özsu, "A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding", SIGMOD 2003

[9] D. Draper, P. Fankhauser, M. Fernandex, A. Malhotra, K. Rose, M. Rys, J. Simeon and P. Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics", October 2004, http://www.w3.org/TR/xquery-semantics/

[10] L. Ennser, C. Delporte, M. Oba and K. Sunil,"Integrating XML with DB2 XML Extender and DB2 Text Extender", IBM Redbooks, 2001, http://www.redbooks.ibm.com/pubs/-pdfs/redbooks/sg246130.pdf

[11] M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model ", October 2004, http://www.w3.org/TR/xpath-datamodel/

[12] M.F. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", ICDE 1998

[13] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, W. C. Tan, "SilkRoute: A framework for publishing relational data in XML", ACM Transactions On Database Systems, 27(4), pages 438-493, 2002

[14] T. Fiebig and et.al. , " Anatomy of a Native XML Base Management System", VLDB Journal 11(4), December 2002

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

[16] D. Florescu and et.al. "The BEA/XQRL Streaming XQuery Processor", VLDB 2003

[17] G. Grahne and A. Thomo, "Algebraic Rewritings for Optimizing Regular Path Queries", ICDT 2001

[18] L. M. Haas and et.al. "Starburst Mid-Flight: As the Dust Clears", IEEE Trans. On Knowledge Data Eng. 2(1), 1990

[19] L. M. Haas and et.al., "Optimizing Queries Across Diverse Data Sources", VLDB 1997

[20] A. Halverson and et.al. "Mixed Mode XML Query Processing", VLDB 2003, pages 225-236

[21] International Organization for Standardization (ISO). *Information Technology—Database Language SQL—Part 14: XML-Related Specifications (SQL/XML).*

[22] H.V. Jagadish and et.al. "TIMBER: A Native XML Database", VLDB Journal 11(1), 2002, pages 274—291

[23] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava and K. Thompson, "TAX: A Tree Algebra for XML", DBPL 2001

[24] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML Streams", The VLDB Journal, Accepted for publication

[25] C. Kanne and G. Moerkotte, "Efficient storage of xml data", Technical Report Nr. 8, Lehrstuhl fur praktische Informatik Ill, Universitat Mannheim, June 1999

[26] R. Kaushik and et.al. "Covering indexes for branching path queries" SIGMOD 2002.

[27] R. Krishnamurthy, R. Kaushik and J. F. Naughton, "XML-to-SQL Query Translation Literature: The State of the Art and Open Problems", XSym 2003, LNCS 2824, pages 1—18

[28] R. Krishnamurthy, V. T. Chakaravarthy, R. Kaushik and J. F. Naughton, "Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation", ICDE 2004, pages 42-53

[29] G. M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives", SIGMOD 1988

[30] A. Maier and D. E. Simmen, "DB2 Optimization in Support of Full Text Search", IEEE Data Eng. Bull. 24(4), 2001

[31] I. Manolescu, D. Florescu and D, Kossmann, "Answering XML Queries on Heterogeneous Data Sources", VLDB 2001

[32] Microsoft SQL Server 2000 SDK Documentation, Microsoft 2000, http://www.microsoft.com

[33] Oracle XML DB, http://www.oracle.com/technology/-tech/xml/xmldb/index.html

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

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

[36] J. Shanmugasundaram and et.al., "Relational Databases for Querying XML Documents: Limitations and Opportunities", VLDB 1999

[37] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, amd J. Funderburk, "Querying XML Views of Relational Data", VLDB 2001, pages 261-270

[38] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita and C. Zhang, "Storing and querying ordered XML using a relational database system", SIGMOD 2002

[39] 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

[40] C. Zhang, J. Naughton, D. DeWitt, Q. Luo and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems", SIGMOD 2001, pages 425—436

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

[42] Profressional XML, 2000 Wrox Press

[43] XML Schema, http://www.w3.org/XML/Schema, May 2000