ROX: Relational Over XML

Alan Halverson[†], Vanja Josifovski[*], Guy Lohman[*], Hamid Pirahesh[*], Mathias Mörschel[+]

alanh@cs.wisc.edu, vanja@us.ibm.com, lohman@almaden.ibm.com, pirahesh@almaden.ibm.com, M.Moerschel@web.de

[†]University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706
[*]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
[+]Am Rederwald 9, D-66954 Pirmasens, Germany

Abstract

An increasing percentage of the data needed by business applications is being generated in XML format. Storing the XML in its native format will facilitate new applications that exchange business objects in XML format and query portions of XML documents using XQuery. This paper explores the feasibility of accessing natively-stored XML data through traditional SQL interfaces, called Relational Over XML (ROX), in order to avoid the costly conversion of legacy applications to XQuery. It describes the forces that are driving the industry to evolve toward the ROX scenario as well as some of the issues raised by ROX. The impact of denormalization of data in XML documents is discussed both from a semantic and performance perspective. We also weigh the implications of ROX for manageability and query optimization. We experimentally compared the performance of a prototype of the ROX scenario to today's SQL engines, and found that good performance can be achieved through a combination of utilizing XML's hierarchical storage to store relations "pre-joined" as well as creating indices over the remaining join columns. We have developed an experimental framework using DB2 8.1 for Linux, Unix and Windows, and have gathered initial performance results that validate this approach.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1. Introduction

After two decades of commercially-available products, relational database systems (RDBMSs) supporting the SQL query language standard are an unqualified commercial success, with a huge industry-wide investment in applications such as Enterprise Resource Planning (ERP) [SAP04, PSW04, ORC04] and Customer Relationship Management [SIB04, ORC04] that query an RDBMS with SQL. As the acceptance and sources of XML documents have proliferated, many commercial relational database systems have adapted by developing techniques for storing XML documents in relational systems by shredding documents into relations [EDO01, SB00, SQX04] and/or by storing each document as an unstructured, large object (LOB) [EDO01]. However, shredding and recomposing all documents, many of which will never be retrieved, is unduly expensive. Alternatively, searching XML documents stored as LOBs is prohibitively slow. As more enterprises exchange business objects, such as purchase orders, in XML format, applications will increasingly need to efficiently query portions of XML documents via the emerging XQuery standard [BCF03]. This will lead to storing the data in some native XML format that efficiently supports XQuery.

Legacy relational interfaces and native XML storage appear to be on a collision course that raises many interesting questions. Can the relational and XML data be treated separately, storing each in the appropriate type of repository? In other words, will data from relational sources be queried exclusively by SQL, and XML data exclusively by XQuery? Or will databases of the future have to be hybrids, storing both relational and XML? Or will we just convert relations into XML objects and store everything in XML format? Regardless, what is to become of the "legacy" applications written in "good old" SQL that need access to data that increasingly originates as XML data? Do they need to be re-written, or can XML

repositories support both XQuery and SQL? Will there be evolution, or a revolution?

We are convinced that XML adoption must necessarily be an evolution – that existing relational applications are too big and complicated to convert them all rapidly or inexpensively from SQL to XQuery. We also project that the data accessed by these SQL applications will increasingly come from XML sources and need to also be accessible via XQuery, and hence will be stored in native XML format.

This paper therefore explores how to efficiently support Relational Over XML (ROX), i.e. the existing SQL interface to a native XML store. We postulate a database containing a blend of both tables and XML documents, with an increasing percentage of XML documents over time. The ROX scenario limits our consideration to SQL queries as input that return rows as output, in order to support legacy applications, even though the system is very likely to also support XQuery interfaces to the same database.

The ROX scenario alone raises many important issues. Perhaps the most important is whether ROX can perform as well as today's SQL engines. What is the impact of the obvious expansion of data caused by tags and other structuring information? How much should XML documents be normalized, and does the denormalization supported by XML help or hinder performance? Or is normalization of data obsolete with the advent of XML?

The remainder of this paper is organized as follows. The next section summarizes the evolution of XML data management. Section 3 discusses issues involving query semantics of SQL and XQuery, tradeoffs for selecting an

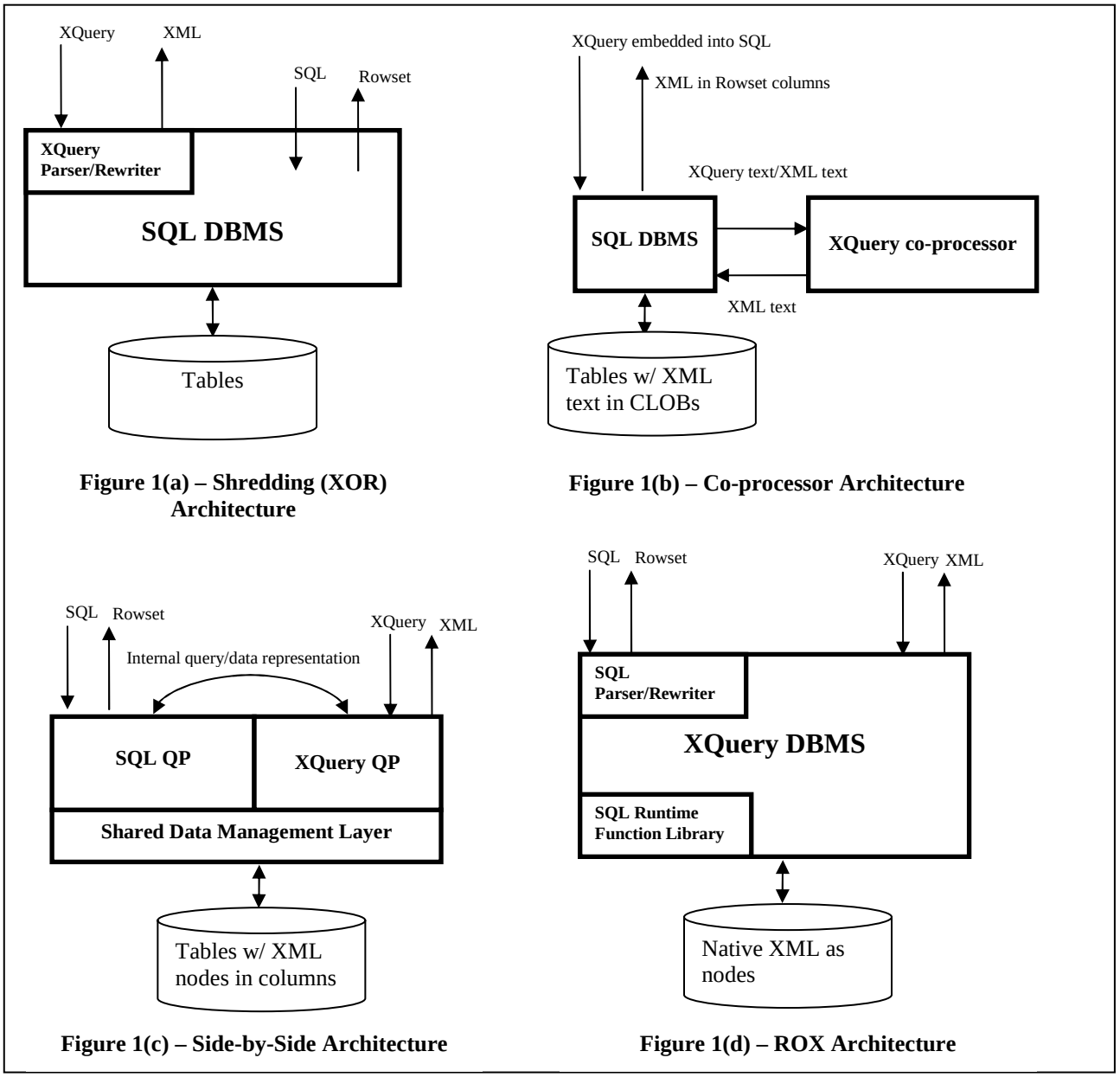

Figure 1(a) – Shredding (XOR) Architecture

Figure 1(b) – Co-processor Architecture

Figure 1(c) – Side-by-Side Architecture

Figure 1(d) – ROX Architecture

Figure 1 - Architecture choices for mixed SQL and XQuery systems

appropriate XML schema, and performance concerns. We present our ROX experimental design in Section 4, and the results of those experiments in Section 5. Our conclusions and directions for future research comprise the last section.

2. The Evolution of XML Data Management Systems

Storing and processing XML data have been a focus of the database research community for much of the last decade. Several XML data management systems have been proposed, most based on various degrees of adaptation and reuse of relational technology. There are two main reasons for reuse of relational technology. First, adaptation is presumably less expensive and allows faster time to market than development from scratch. The other reason is that such hybrid systems are capable of storing both relational (structured) and XML (semi-structured) data. As most applications are likely to operate over both types of data, the new generation of databases will need to support both allowing the application to access a single data repository.

Several different architectures have been proposed for building a hybrid XML-relational database, as illustrated in Figure 1. Chronologically, the first attempts were based on reusing the whole RDBMS stack when processing XQuery queries: from the SQL query language to the relational data storage. In this XML-Over-Relational (XOR) approach, the XML documents are shredded into atomic values that are then stored in relational tables. XQuery queries are translated into SQL queries to be evaluated by the existing query processor. Several research prototypes have explored this scheme, such as LegoDB [BFH00], and XPeranto [SKS01], and several products offer different shredding and XPath querying capabilities based on this approach [EDO01, MSF00]. The advantage of this architecture is that it requires almost no modification of existing database engines. As such, an XQuery implementation can easily be adapted to several different DBMS systems. However, as the XQuery language has evolved into an elaborate and complex standard, it has become clear that translating XQuery queries into SQL queries is a daunting task. While shred-and-query systems claim compatibility with a subset of the language, none has managed to produce a fully compliant XQuery implementation.

Next in the timeline were systems that are on the other side of the architectural spectrum, named Co-processor Architecture in Figure 1(b). Here, XML is stored as unparsed text in VARCHAR or LOB columns of relational tables. The XML data is opaque to the RDBMS and only the storage layer is re-used. The XML data is queried using an XQuery processor that is external to the database and invoked much like a user-defined function. The communication between the two processors is using textual or equivalent format. The SQL and the XQuery processors can be developed separately and interchanged. This solution is attractive for its relative simplicity and modularity.

Most of today's commercial systems support this type of XML manipulation using stored procedures to invoke an external XQuery processor [EDO01, MSF00], in conjunction with XOR support. However, 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

Several systems have been reported that support only XQuery. Systems such as Niagara [NDM01] and Timber [JAK02] 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 do not support SQL or relational storage.

More recently more native storage of XML documents has been proposed in [KM99] and [ZKO04]. 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-index portions of the query. Parent-child traversal does not require a join between different tables. Since most XPath expressions require parent-child traversal, this scheme allows for efficient access to the data. The details of the storage are beyond the scope of this paper. We use this model as an example to explore the consequences of representing relational data with hierarchical trees.

At this point we consider where will the system architecture move beyond today's state of the art? Can we project the direction of the path based on the evolution so far? Is the current situation similar to the introduction of the relational database systems compared to IMS? We try to analyze the issues from several angles and answer to these questions.

One probable direction in the short term is the side-by-side architecture, as shown in Figure 1(c). In this architecture there is a tighter coupling between the query processors than in the architecture based on shredding. Query fragments can be translated from one language to the other and exchanged using internal data structures that may not adhere to the language semantics. Such a mechanism improves the efficiency of the translation and allows more degrees of freedom in the evaluation. For example, when returning values from XQuery to SQL, as required when evaluating SQL/XML, queries might require that an element is constructed by an embedded XQuery query and then shredded by an SQL table function. Instead, the optimizer can re-write such queries

so that rows of values are returned from the XQuery processor directly into the SQL processor, although rows are not part of the Query Data Model.

While more efficient than the first two architectures, the side-by-side architecture introduces many complexities. It requires that various system components have compatible definitions on both sides of the system. For example, the catalogue description of internal objects such as indexes and materialized view definitions need to be matched to both the SQL and XQuery queries. While these issues pose interesting research challenges, we view this architecture only as a partial solution that will be simplified and eventually morph into the Relational over XML architecture shown in Figure 1(d) where the primary processing is performed by an XQuery engine with native XML storage model. In its engine all transformations are governed by the XQuery language specification and the Query Data Model. The SQL support is divided between a thin parse-and-rewrite layer and a library for support of the SQL functions and operators that cannot be mapped to the XQuery functions and operators.

The ROX architecture is at the opposite extreme of the solution space when compared to the first XQuery query processor designs, in which the XQuery processor was a thin layer over SQL database systems. The obvious question is what makes this architecture viable if the opposite solution has not been implemented in any of the major database products? Furthermore, in terms of development cost, this architecture requires a complete XQuery engine that is adapted to run SQL queries, seemingly a much more demanding path than the opposite route.

As XQuery and the QDM data model conceptually subsume the SQL language and the relational model, implementing SQL on top of an XQuery engine poses significantly lesser challenge than the opposite. We also believe that this architecture will not be achieved by developing an XQuery engine from scratch. Existing relational engines will be morphed into this architecture possibly through the intermediate stages represented by the other architectures depicted in Figure 1. It seems to us that, beyond the initial releases of the commercial database products for XML data management, the main forces in the database engine evolution will be to increase the performance and reduce the complexity of the relational-XML engines. These two forces will be the major factors in the appearance of the ROX architecture, shown in Figure 1(d).

3. ROX Model Issues

While unable to implement a complete ROX architecture, we single out three issues that are crucial to demonstrate the viability of the infrastructure and explore in more details each. We first overview the language semantics issues and how the semantics differences between SQL and XQuery impact the ROX architecture. Then we turn our attention to the data layout and normalization related issues as posed by the nested XML data model. Finally we consider the performance impact of an XML native format, query optimization issues, and XML data manageability.

3.1. Language Semantics

The main difficulty in running SQL queries over an XQuery implementation is providing semantically correct answers to the queries. Although SQL and XQuery have similarities, there is an abundance of differences. First of all, the languages are defined over different data models. The SQL language is defined over the relational model [SQL98], while the XQuery language is based on the Query Data Model [FMM03] that represents XML data as typed trees. SQL queries operate over column values, while XQuery manipulates ordered, heterogeneous sequences of values and node references. While a detailed description of the differences is beyond the scope of this discussion, in general, QDM is much more elaborate than the relational data model. This is the core reason why XQuery-to-SQL translation is unsuitable as a basis for a fully functional XQuery system.

The languages also differ in their operational semantics. The most quoted difference is the document order preservation of XQuery vs. unordered semantics of SQL. Furthermore, each language standard contains precise descriptions of the language operators. These specifications seldom match. For example, the comparison operators in SQL use 3-value logic, operating over Boolean operands and returning true, false or NULL. The same XQuery operators (general comparison) operate over sequences of nodes or values and return true and false. There is no NULL value defined in the XQuery data model. Another discrepancy stems from the different definitions of the basic data types as decimals and date-time. As many of the built-in functions operate over such values, they might potentially return different results.

Despite these differences, XQuery is designed to be able to manipulate structured data along with unstructured [CFF03]. Therefore, there is an overlap in the functionality of SQL and XQuery. While different in data model and semantics, when constrained over structured data, many XQuery operations have semantics close to that of SQL, and under certain cardinality constraints match the SQL semantics. For example both XQuery and SQL numeric operations are based on the IEEE standard and seem to be reconcilable. Furthermore the XQuery arithmetic operators treat empty sequences in the same manner as SQL operators treat the NULL values. This is also true for the XQuery value comparison operators (eq, gt, neq, etc.) which have 3-value logic (returning empty sequence if any of the operators is an empty sequence) as the comparison operators of SQL. Translating the SQL comparison operators into XQuery value comparison

operators, we can achieve the same semantics as in SQL. This allows pushdown of simple arithmetic and comparison predicates from SQL to XQuery. While the XQuery Boolean operators operate using 2-value logic, it is simple to implement 3-valued Boolean operators in XQuery that have same semantics as the SQL operators. With such a small implementation effort, a large class of SQL predicates over numeric types and strings can be translated into equivalent XQuery predicates. However, based on the current standards proposal for XQuery, it seems that it will not be possible to translate all SQL functions to existing XQuery functions. We envision this necessary for SQL datatypes that are not subsumed into XQuery datatypes, such as types representing date-time and timestamps [SQX04].

3.2. Normalization

One of the key benefits of a native XML store is not having to normalize the elements that make up a business object by shredding them into tables. For example, consider a common business object -- a purchase order, which might contain some customer elements and one or more line items describing each object being purchased:

```
<order>
  <date>12 July 2003</date>
  <customer>
   <ID>43839</ID>
   <name>Slaghorn Bolts</name>
   <contact>Joyce Smith</contact>
   <address>
    53495 N. First St.
    Cleveland, OH 45678
   </address>
   <order_discount>
    0.10
   </order_discount>
  </customer>
  <line_item>
   <part_ID>RYZ04856-8945</part_ID>
   <quantity>33</quantity>
   <discount>0.12</discount>
  </line_item>
  <line_item>
   <part_ID>KFE389745-2248</part_ID>
   <quantity>15</quantity>
   <discount>0.05</discount>
  </line_item>
  <line_item>
   <part_ID>OI230988-2833</part_ID>
   <quantity>100</quantity>
   <discount>0.21</discount>
  </line_item>
</order>
```

Figure 2 – Purchase order in XML format

Since the purchase order arrives in XML format, it is tempting to store the entire document as it comes into the system, to minimize any processing. But is that the right thing to do? Does the nesting of XML documents make normalization of objects in databases obsolete? And if not, what elements should be normalized and which should not?

The answer is that normalization is still needed in XML databases, for the same reason it was needed in relational databases: redundancy of data and update anomalies [DAT03]. In the above example, the line items nested within an order are wholly owned by that order, so they cannot suffer the update anomalies of shared sub-objects. However, the customer information is a bit more subtle. It is likely to be shared by many other orders, so keeping it with each purchase order would both be unnecessarily redundant and risk update anomalies. For example, the customer address is on the purchase order, but it's probably the same address as on hundreds of other orders from the same customer. But if it suddenly changed, this order might be sent to the address that was in effect at the time the order was made, rather than the address in effect when the order is shipped. Similarly for other attributes of the customer. So pretty clearly the customer information should be normalized. However, some elements of the customer are really elements of the interaction of the customer and this order. For example, the order_discount element might depend upon the size of the overall order and how valued this customer is. Hence it cannot be normalized out of the purchase order.

The good news is that native storage of XML documents permits denormalization when it makes sense semantically (sub-objects are not shared), while still retaining the option to normalize data, i.e. when sub-objects may be shared and hence risk update anomalies. The database designer is thus free to do what best models the data, rather than forcing the design into a large number of overly-normalized, homogeneous tables. And, as we shall see later, denormalization can also have performance benefits by obviating the need for some joins when the data is queried.

Today's relational engines use data redundancy in form of pre-joined relations to speed up evaluation of queries [HAL01]. Materialized view techniques will also be important in XML databases in general and with the ROX model in particular. In section 5.3, we show that data nesting that matches the query structure allow for much better evaluation times. As opposed to relational systems in which the materialized, pre-joined views are flat tables, an XML engine allows these to be in non-first normal form, similar to the proposal of [SPS87].

3.3. Performance

To achieve good query performance for the ROX model, we must consider several issues. The storage of the XML

tree could be sorted in either depth-first (document) or breadth-first order. The depth-first order is advantageous when the goal is efficient reconstruction of the XML. However, if our XML documents have several levels of hierarchy, queries referencing only data at the top levels of the document will suffer.

Another concern is the overhead of storing the structure of the document inline with the data being represented. Although it is possible that the stored XML document conformed to a stated XML schema, in general our storage format must allow for XML documents which lack a predefined schema. This storage overhead forces a native XML store to consume more space for representing a certain dataset than the relational storage. The absence of an XML schema also forces data in the document to be stored as text, which also adds storage overhead.

To facilitate efficient selection and value-based joins between XML documents, an XML index is required. As with relational systems, such an index will allow us to find documents which contain a certain value in a certain location. Many indexing strategies for XML have been proposed, such as inverted lists of elements for structural joins and path indexes.

3.4 Optimization

Compiling SQL queries on XML documents presents new challenges for query optimization. Although denormalized data in the form of materialized views and join indexes is already widely exploited by relational query optimizers, both the query and the denormalized data are defined in relational terms, usually SQL. In ROX, the optimizer must now match joins and predicates in the SQL query to XPath expressions that define the schema of XML documents (presumably the XML documents manipulated by ROX will have a schema with sufficient homogeneity to permit a tabular view of them). Join predicates between documents must also be folded into predicates at various points of an XPath expression, depending upon the join order. In our experiments, discussed below, we performed this mapping manually to avoid the challenge of automating it. Having documents with various schemas – or even no schema at all! – mixed together in the same repository, called "schema chaos", negates the homogeneity that simplified the cost model and the database statistics on which relational optimization depended. And though the denormalization of XML documents reveals correlations among objects, it is not at all clear what database statistics are needed to summarize those correlations and how those statistics can be exploited to accurately estimate the number of documents satisfying a particular SQL query. And this doesn't even consider the considerable challenges of optimizing XQuery queries!

3.5 Manageability

Will a database of XML documents be easier or more difficult to manage than relational tables?

Some would argue that management of XML repositories should be child's play. Since real-world objects no longer need to be normalized into homogeneous collections of rows (tables), the XML repository can be reduced to a single, virtualized heap of heterogeneous objects (documents), creating the relational equivalent of the Universal Relation [MUV84]. In lieu of perhaps tens of thousands of normalized tables, there would be only one collection of documents to configure, backup, recover, reorganize, collect statistics on, etc. Database design would be trivial, normalization would be unnecessary, and one index over this entire collection would suffice to find any object in the database -- the "Google model" applied to databases!

On the other hand, management of modern databases entails far more than just deciding what tables and indexes to create. As argued in Section 3.2 above, some normalization will still be required to avoid update anomalies, so logical database design may be less constrained but certainly not obviated. Eventually, XML systems will permit the definition of the XML equivalent of materialized views, and deciding which to create will surely be no easier than it is now for relational systems. Even if all documents are in one monolithic collection, administrators will probably have to define arbitrary boundaries within that collection for administration purposes, so that pieces can be maintained while the rest of the database is available for querying and updating, much as the rows of large tables are usually divided into ranges for administrative purposes [IBM01]. And given the increased challenges posed by optimizing queries against these heterogeneous collections (see previous section), it is likely that the database statistics required for optimization will be far more extensive than for relational systems. For performance reasons, we might still want to cluster related documents together to exploit the larger pre-fetching chunks that relatively slower disk arms necessitate, or possibly de-cluster them to spread access among multiple arms for greater I/O parallelism, rather than simply append each new document to the end of the heap.

4. Design for Experimentation

In Section 2, we described a number of architectural alternatives for a mixed SQL and XQuery system, including the ROX model. We now provide a description of the implementation we chose for our experimental framework.

A full implementation of the ROX architecture would require a fully-functional XQuery DBMS, upon which a thin SQL-to-XQuery translation layer would sit. However, building a system like this would take a

significant number of person-years to implement. Instead, we took advantage of a prototype XML store available to us and implemented a much simpler mapping layer. This experimental architecture is described in detail in the following sections.

4.1 SQL to XQuery translation using the XML Wrapper

For our experiments, we modified an existing product called the XML Wrapper, which is part of the IBM DB2 Information Integrator, version 8.1. The unmodified XML Wrapper provides a mechanism for presenting relational views of XML data stored as text files on disk. Each relational view defined over an XML document is called a nickname, and utilizes syntax similar to the CREATE TABLE statement.

```
CREATE NICKNAME REGION(
  R_REGIONKEY int
    OPTIONS(XPATH 'R_REGIONKEY/text()'),
  R_NAME char(25)
    OPTIONS(XPATH 'R_NAME/text()'),
  R_COMMENT varchar(152)
    OPTIONS(XPATH 'R_COMMENT/text()'))
FOR SERVER xml_server
OPTIONS(XPATH '/REGION');
```

Figure 3 - Nickname definition

```
<REGION>
   <R_REGIONKEY>2</R_REGIONKEY>
   <R_NAME>ASIA</R_NAME>
   <R_COMMENT>sladfkj weoiu sdflkj
   </R_COMMENT>
</REGION>
```

Figure 4 - Sample XML document

In Figure 3 we show a possible CREATE NICKNAME statement that DB2 would use in conjunction with the XML Wrapper to query the XML document shown in Figure 4. The product version of the XML Wrapper uses the Xerces [XER03] XML parser and the Xalan [XAL03] XPath evaluator to find data in the XML document(s). Both Xerces and Xalan are subprojects of the Apache XML project [APX04]. The wrapper queries the XML and creates relational rows conforming to the CREATE NICKNAME statement to hand back to the database engine.

Because XML allows hierarchical nesting of elements, entities may be stored physically together in the same document. For example, you might store a Customer with all of the Orders he has placed as child elements of the Customer. To exploit this, the XML wrapper allows special columns to be specified as the PRIMARY_KEY or FOREIGN_KEY for a nickname. For example, a column 'fk' in a nickname for ORDERS may be defined as the FOREIGN_KEY of the PRIMARY_KEY column 'pk' in a nickname for CUSTOMER. When a SQL query references these two nicknames with an equality join predicate between fk and pk, the wrapper knows that any Order information returned will be found as sub-elements of the Customer in the XML document. Any paths specified by the ORDERS nickname must be relative to the XPATH specified for the CUSTOMER nickname. The value for the pk column is simply a serialization of a Xerces element reference.

For our experiments, we modified the existing XML Wrapper to be an interface to a prototype XML store. Since our data had been previously parsed and stored in this native XML store, we removed the Xerces code. Also, the Xalan XPath evaluator could not be used, since it operates over in-memory DOM trees only. In its place, we used a custom XPath evaluation engine that evaluates paths over the prototype XML store. To implement the PRIMARY_KEY column option, we used an internally generated XML node identifier. We present an overview of our experimental architecture in Figure 5.

One of the primary advantages of a native XML store is that we have an opportunity to create one or more indices over the loaded data. The prototype XML store contains a path-based XML indexing module, but lacks automatic XML index selection in the query optimizer. To enable using each XML index created, we wrote custom parameterized table functions that take a key value as input and return relational rows that are the inner join result for that key value. This works because the XML index stores the same XML node reference value that the XPath evaluator uses. This idea also allows us to hand-optimize the join order for queries that refer to more than two nicknames by using a column from the result of one table function call as the input for another. For example, if we want to force a scan of the CUSTOMER nickname to be the outer entity in an index nested-loops join with ORDERS, we would write the following SQL query:

```
SELECT O.O_ORDERDATE
FROM CUSTOMER C, tfORDERS(C.C_CUSTKEY) O;
```

In this example, `tfORDERS()` is a user-defined table function that takes as input a customer key and returns columns from the ORDERS nickname from rows that contain a matching `O_CUSTKEY` value. The XML documents that match are found by performing a lookup in the XML index to find all documents which contain the path /ORDERS/O_CUSTKEY/text() = [C_CUSTKEY], where C_CUSTKEY is the value passed to the table function.

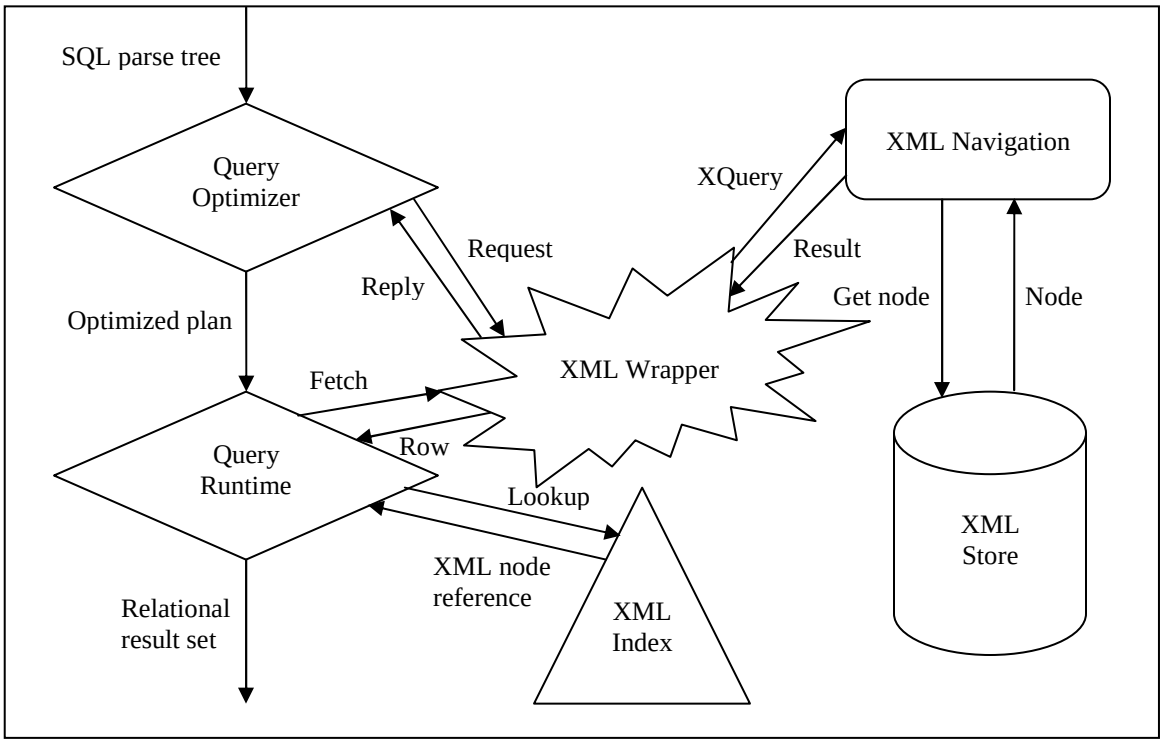

Figure 5 - Experimental Architecture

4.2. Prototype Walkthrough

To illustrate the prototype ROX architecture, we will describe how the following SQL query is executed:

```
SELECT r_name,
  COUNT(n_nationkey) AS n_count
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_name;
```

Logically, the input to the query optimizer is a SQL parse tree. For our example query, it will contain references to our nickname definitions for REGION and NATION, as well as the R_REGIONKEY = N_REGIONKEY predicate. Since DB2 only knows about the definition, it must consult the server specified by the CREATE NICKNAME statement, and therefore our modified XML wrapper, to create alternate execution plans and cost estimates. Plans enumerated include plans for: REGION only, NATION only, a plan that pushes the equality predicate into the wrapper and returns rows containing both REGION and NATION columns, and a plan for NATION which takes as input a context R_REGIONKEY and returns rows with an equal N_REGIONKEY column. We would accept this last plan only if R_REGIONKEY was defined with the PRIMARY_KEY option in the REGION nickname, and N_REGIONKEY defined with the FOREIGN_KEY option and referencing the REGION nickname. With a full cost model in the wrapper, we could tell the optimizer that one or more of these plans would provide the best performance. For each plan the wrapper can accept, we create a structure containing everything necessary to execute the plan later at runtime, and return control back to the optimizer. In the prototype, the structure would contain an XQuery to be executed at runtime. For example, we would create the following XQuery when asked to scan the REGION nickname:

```
for $a in /REGION, $b in $a/R_NAME
return $a, $b;
```

Once the optimizer chooses a final query plan, the query runtime takes control and begins to execute the plan. Any operator in the plan containing a packed structure created by the wrapper during optimization now calls back into the wrapper requesting to open a cursor based on the information contained in that structure. For our example query, the first request might be to do a table scan on the REGION nickname.

For each row returned from the wrapper for that scan, a second cursor would be opened over the NATION nickname, with an additional parameter containing the value of the R_REGIONKEY column for the current REGION row. Recall that the value of the R_REGIONKEY column would be the internal XML node identifier for the REGION element parenting the NATION information to return. The XML navigation would begin with the node identifier passed in, rather than from the document root. If the XML nodes are stored on

disk in document order, we likely have the relevant NATION elements already in memory.

The prototype expects DB2 to perform the calculation of the N_COUNT output column and to handle the GROUP BY R_NAME clause. Note that better performance could be achieved for this query by pushing the aggregate down into the XML Wrapper, but the prototype did not do so.

4.3. Experimental Dataset

We chose the TPC-H [TPC02] dataset for our experiments. This dataset is well known throughout both the industrial and academic research communities, and is representative of a normalized relational schema that can be adapted to the ROX model. The schema consists of eight entities, namely REGION, NATION, SUPPLIER, PART, PARTSUPP, CUSTOMER, ORDERS, and LINEITEM. The PARTSUPP entity exists to allow a many-to-many relationship between PART and SUPPLIER.

For the corresponding XML schema of this dataset, we have quite a few choices. As with the relational schema, we discard any choice which results in data duplication. Please refer to section 3.2 for our discussion of data normalization. We compare three XML schemas for our experiments, named Unnest, Nest2, and Nest3.

Our Unnest schema consists of one XML document per relational row per relational table. The root element of each document is the name of the relation from which it came, each sub-element the name of a column from that relation, and the text contained in each sub-element is a value from the row that we used to generate the document. Figure 4 shows an example XML document created from one row of the REGION table. Our Nest2 schema stores LINEITEM elements nested within the correct ORDERS element, and PARTSUPP within PART, but leaves the remaining data as in the Unnest schema. Finally, the Nest3 schema stores LINEITEM elements within ORDERS elements, which in turn are nested within the correct CUSTOMER element, with all other data as in the Unnest schema. With the TPC-H schema, it is not possible to create a semantically meaningful, properly normalized document with four levels of nesting.

5. Experiments and Results

This section presents the experimental results we gathered to validate the feasibility and performance of the ROX model.

All experiments were executed on a quad processor PowerPC-based machine running AIX 5.1, equipped with 16GB of main memory and SCSI disks. Data and indices were loaded into separate DB2-managed tablespaces striped across 22 5GB SCSI disks. All timings reported in this section are an average of 5 runs. We calculated that all timings for each average are within 1% of the average value with 95% confidence.

All experiments are run using data generated at TPC-H Scale Factor 0.1. This means our largest entity, LINEITEM, has approximately 600,000 rows. The raw data is nearly 100 MB on disk.

5.1. Storage Comparison

In this section, we examine the storage requirements of both the relational and XML versions of the TPC-H data set. The number of disk pages required to store the data has a direct impact on the cost of any sequential scan. For this experiment, we loaded several of the TPC-H relations into both standard DB2 tables and our native XML storage engine, and present the disk storage requirements in Table 1.

Table 1 - Relational and XML Storage Requirements for selected TPC-H relations, in KB

Relation(s)	Relational	XML
CUSTOMER	2656	13312
ORDERS->LINEITEM	100960	888832
PART->PARTSUPP	15904	66560

It is clear that a generic XML store generates significant storage overhead when compared to the same data stored relationally. These overheads are due mostly to three factors. All text data is stored in Unicode format in the prototype XML store. Although DB2 allows tables to store Unicode data, it does not do so unless explicitly asked to by the user. This is a factor of two size increase for any text data in the TPC-H tables. Secondly, a generic XML store must duplicate the document structure for every relational record converted to XML format. For XML documents with a high structure-to-data ratio, this overhead is high. Finally, our XML storage engine currently stores all XML data in text format. For any numeric data, this adds significant storage overhead. Storage for the element tags does not require significant overhead, however. Each unique element and attribute name is entered into a mapping table, which allows us to store an integer tag ID for each document node.

5.2. Bufferpool Effects

As discussed in section 5.1, the storage required for the XML schemas under test is significantly more than for the relational load of the data into DB2. It therefore makes sense to consider the effects of varying the size of DB2's bufferpool on query performance. We chose to run each series of queries using four different bufferpool sizes. The sizes were chosen to be 10%, 25%, 50%, and 100% of the total storage (data and indices) required for the schema. Using this definition implies that the 10% case for the relational schema is a much smaller number of pages than for the 10% XML schema case. Please refer to Table 2 for the specific bufferpool sizes tested.

It may seem unfair to use different bufferpool sizes for the tests. After all, when 100% of the relational data and indices fit in memory, only 11% of the XML data and

indices fit. Further, given a specific memory budget, the relational data has a size advantage and should benefit from it. However, if the scale factor of the data increased, we would not have a choice but to choose a bufferpool size < 10% in both cases.

Table 2 - Tested bufferpool sizes (in number of 32K pages)

Size	Relational	XML
10%	450	4000
25%	1125	10000
50%	2250	20000
100%	4500	40000

We present a graph in Figure 6 which illustrates the effects that the bufferpool size has on the performance of TPC-H query 10. The results presented are normalized to the execution time when the 100% bufferpool size is used. For the relational schema, additional memory directly contributes to decreased query execution time. However, additional memory does not give an advantage for queries executed using the XML schemas. The additional CPU cost of navigating through the XML schema and converting the retrieved text to the correct column datatype may be to blame, but additional experiments are necessary. Similar results were obtained for the other queries we tested.

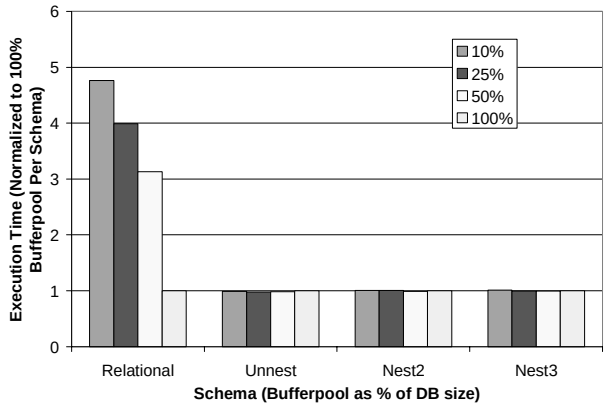

Figure 6 - TPC-H Q10 bufferpool effects. For each schema, the execution times are normalized to the 100% bufferpool execution time. The percentages listed are relative to the total size of the data and indicies being tested.

When the number of bufferpool pages are roughly the same for the relational and XML schemas, the relational schema appears to be the clear winner. In Figure 7, we present the results of two queries executed over all schemas. The relational schema was tested at the 100% bufferpool level of 4500 pages, while the XML schemas used their 10% level of 4000 pages. For Q10, the best XML schema is still a factor of about 19x slower than the relational schema. For Q22, we see that the Unnest and Nest2 schemas are about a factor of 5 slower.

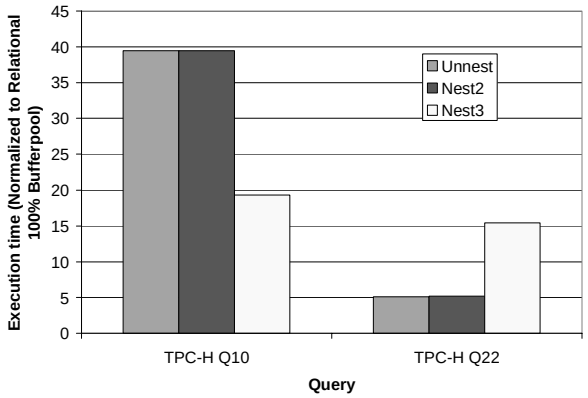

Figure 7 - Bufferpool effects when all queries are executed with approximately the same number of bufferpool pages

5.3. Schema Variations

In this section, we discuss the measured effects of varying the nesting of the XML schema. All experiments discussed in this section assume a 10% bufferpool size.

As the level of nesting in each XML document is increased, we encounter mixed performance results. Consider the graph in Figure 8, which shows the normalized execution times for two TPC-H queries for our three XML schemas. TPC-H Q10 is called the Returned Item Reporting Query. This query is basically a four-way join between NATION, CUSTOMER, ORDERS, and LINEITEM. This query fits our Nest3 schema extremely well, and the results show that this query is about twice as fast with Nest3 as with either the Nest2 or Unnest schemas. One obvious question, though, is why we don't see any benefit from the Nest2 schema, which has LINEITEM nested in ORDERS. The answer lies in the fact that we are utilizing the XML index to join CUSTOMER to ORDERS in both cases, and also for ORDERS to LINEITEM in the Unnest case. As we will see in the next section, the XML index performs very well and brings Unnest's performance in line with Nest2.

Although Nest3 was the clear favorite for Q10, it suffers for other queries such as Q22. This query scans CUSTOMER looking for customers in specific countries who have never placed an order but have a good account balance. The country selection predicate is reasonably selective, and so the join to ORDERS can be avoided for most customers. The Nest3 schema performs very poorly for this query due to the storage of each CUSTOMER's ORDERS and LINEITEM information – the very attribute that made it much better for Q10. Since this query does not use the ORDERS information very often and never uses the LINEITEM information, we needlessly pay to load them from disk. CUSTOMER information packs much better in the Unnest and Nest2 schemas, as both utilize the CUSTOMER-only XML document format.

These results suggest that the XML schema chosen should factor in the expected query workload, if known.

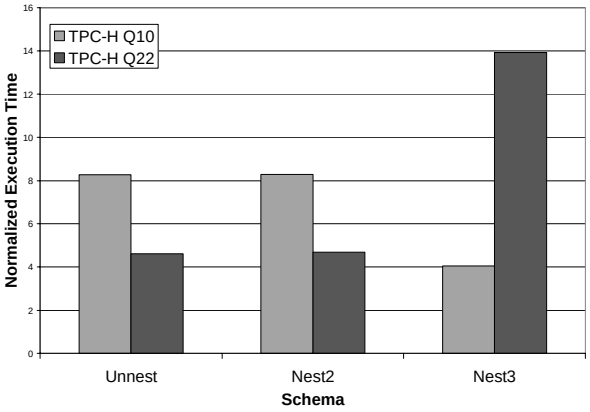

Figure 8 - Schema Effects for two TPC-H queries, normalized to the Relational time for each query

5.4. XML Index Exploitation

We now consider the benefits of utilizing the XML index to aid in joining across documents. In all of our chosen XML schemas, we maintain some normalization. For example, all of our schemas keep PART and LINEITEM unnested. To find the name of a part given a specific lineitem, we must do a standard join. In this section, we compare two types of joins – index nested loops join using the XML index, and DB2's hash join. In Figure 9, we show results for TPC-H query 5 – the Local Supplier Volume Query. This query joins six of the eight tables using a total of six equijoin predicates. The extra join ensures that the supplier and customer are from the same nation. Here, we have normalized the results to the execution time for the Relational schema. For this query, utilizing the XML index provides a very tangible benefit. The Unnest schema shows a much larger improvement over the hash join plan than the Nest2 schema, with both schemas at about 3 times the relational query time when the XML index is used. In the Hash Join case, the Unnest2 performs better because the plan executed still takes advantage of the nesting of LINEITEM in ORDERS, thereby obviating a very expensive join.

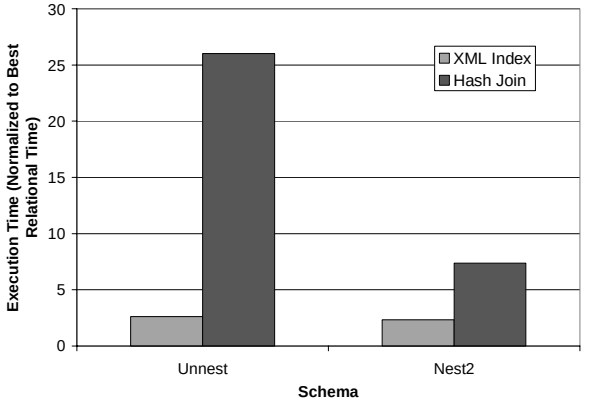

Figure 9 - TPC-H Q5: Local Supplier Volume Query

These results show that our immature ROX prototype can achieve performance within a factor of three of a mature relational database system for an important category of queries. With proper tuning and optimization of the storage format, XML navigation, and XML index utilization, even better results could be obtained.

6. Conclusions/Future Work

In this paper, we have discussed an alternate solution to the problem of integrating relational and XML data sources to support both new XQuery and legacy SQL queries. We called this solution the ROX model, and described the architectural tradeoffs involved. Our solution allows existing SQL applications to continue to run unmodified, and allows a gradual transition of some or all data to the XML storage format. We created a prototype to compare the performance of the ROX model to the standard relational model, and found that it can compete for an important class of queries. We found that the choice of the XML schema to represent the relational data can have a profound impact on performance. Also, by utilizing an index over the XML storage, we could achieve performance within a factor of three of a mature relational DBMS for queries with many joins.

Many research questions remain open for future work. Updates in a native XML storage system can pose problems such as document order anomalies and subtree locking issues. Further, an XML update standard (or even a candidate specification) does not yet exist. It would be interesting to consider the ROX model as the XML update standard is created.

Storage overheads associated with general native XML stores are a significant source of performance problems when using the ROX model to perform sequential scans. Identifying ways to store XML compactly and exploring tree storage alternatives based on document access patterns are interesting areas for future research.

For a given query workload and XML schema, we could utilize the query optimizer to create alternate plans that would be possible if a different XML schema was available, and use this information to automatically suggest a better XML schema for that query workload, much as was done in DB2's Index Advisor [VZZ00] and Design Advisor [ZIL04].

Resolving these questions will bring us closer to the time when XQuery and SQL queries can both be processed efficiently against both structured and semi-structured databases.

Acknowledgement: In order to assemble the prototype, we relied on components built by many contributors, including Bert van der Linden, Brian Vickery, Tuong Truong, Bob Lyle, George Lapis, Bobbie Cochrane, Chun Zhang and others. We would also like to thank Kevin Beyer and Matthias Nicola for useful discussions and

support provided while performing the experimental evaluation.

7. References

[APX04] The Apache XML Project: http://xml.apache.org/

[BCF03] Boag, Scott D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon: *XQuery 1.0: An XML Query Language (Working Draft).* November 2003. http://www.w3.org/TR/xquery

[BFH00] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, J. Simeon, *LegoDB: Customizing Relational Storage for XML Documents*, In Proc of VLDB 2000, Septermber 2000

[CFF03] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, J. Robie: *XML Query Use Cases*, W3C Working Draft, November 2003, http://www.w3.org/TR/xquery-use-cases/

[DAT03] C. Date: *An Introduction to Database Systems, Eighth Edition*, Pearson Addison Wesley, 2003.

[EDO01] L. Ennser, C. Delporte, M. Oba, 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

[FMM03] M. Fernandez, A. Malhorta, J. Marsh, M. Nagy: *XQuery and XPath 2.0 Data Model, W3C Working Draft*, November 2003, http://www.w3.org/TR/xpath-datamodel

[HAL01] A. Y. Halevy: *Answering queries using views: A survey*. The VLDB Journal 10(4), 2001.

[IBM01] *DB2 for z/OS and OS/390 Version 7 Using the Utilities Suite*, IBM Red Book, http://publib-b.boulder.ibm.com/Redbooks.nsf/0/03b3f70ce5666bec85256a5300663f26?OpenDocument

[JAC02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattanan, Y. Wu, C. Yu: *Timber: A Native XML Database*, In Proc of VLDB 2002, September 2002

[JS03] V Josifovski, P. Schwarz: Querying *XML data sources in DB2: the XML Wrapper*. In Proc of ICDE 2003, Banglore India, 2003.

[JFB04] V. Josifovski, M. Fontoura, A. Barta: Querying *XML Streams*, to appear in the VLDB Journal, 2004

[KM99] C. Kanne, G. Moerkotte: *Efficient Storage of XML Data*, In proc of ICDE 1999, 1999

[MUV84] David Maier, Jeffrey D. Ullman, Moshe Y. Vardi: *On the Foundations of the Universal Relation Model*. ACM Trans. Database Syst. 9(2): 283-308 (1984)

[MSF00] *Microsoft SQL Server 2000 SDK Documentation*, Microsoft 2000, http://www.microsoft.com

[NDM01] J. Naughton, D. DeWitt, D, Meier et al.: *The Niagara Internet Query System*, IEEE Data Engineering Bulletin, volume 24, number 2, June 2001, pp. 27-33.

[ORC04] Oracle: http://www.oracle.com/solutions/

[PSW04] Peoplesoft: http://www.peoplesoft.com/corp/en/products/ent/index.jsp

[SAP04] SAP: http://www.sap.com/solutions/erp/

[SB00] M. Scardinia, S. Banerjee: *XML Support in Oracle 9i*, Oracle Corporation, December 2000.

[SIB04] Siebel: http://siebel.com/products/index.shtm

[SKS01] J. Shanmugasundraram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk: *Querying XML Views of Relational Data.* In proc of VLDB 2001, September 2001.

[SPS87] M. Scholl, H.-B. Paul, H.-J. Schek: *Supporting Flat Relations by a Nested Relational Kernel.* In proc of VLDB 1987, September 1987.

[SQL98] *Database Language SQL – Part 2: Foundations (SQL/Foundations)*, ISO Final Draft International Standard, ISO 1998.

[SQX04] A. Eisenberg, J. Melton: *SQL/XML is Making Good Progress*. SIGMOD Record 31(2), 2002.

[TPC02] *TPC Benchmark H*, Transaction Processing Performance Council, San Jose, CA 2002. http://www.tpc.org/tpch/spec/tpch2.1.0.pdf

[VZZ00] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, A. Skelley: *DB2 Advisor: An Optimzer Smart Enough to Recommend its Own Indexes*, In proc of ICDE 2000, San Diego, CA, 2000.

[XAL03] *Xalan an XSL Processor*,The Apache XML project,http://xml.apache.org/xalan-c/index.html

[XER03] *Xerces: a validating XML Parser*, The Apache XML project, http://xml.apache.org/xerces-c/index.html

[ZIL04] D. Zilio et al.: *Recommending Materialized Views and Indexes with IBM's DB2 Design Advisor*, To appear in proc of ICAC 2004.

[ZKO04] N. Zhang, V. Kacholia,, M. T. Özsu: *A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML*, In proc of ICDE 2004, Boston, MA, March 2004.