XQuery Rewrite Optimization in IBM® DB2®*pureXML™

Fatma Özcan
IBM Almaden Research Center
650 Harry Road, San Jose

Normen Seemann
IBM Silicon Valley Lab
555 Bailey Road, San Jose

Ling Wang
IBM Silicon Valley Lab
555 Bailey Road, San Jose

Abstract

In this paper, we describe XQuery compilation and rewrite optimization in DB2 pureXML, a hybrid relational and XML database management system. DB2 pureXML has been designed to scale to large collections of XML data. In such a system, effective filtering of XML documents and efficient execution of XML navigation are vital for high throughput. Hence the focus of rewrite optimization is to consolidate navigation constructs as much as possible and to pushdown comparison predicates and navigation constructs into data access to enable index usage. In this paper, we describe the new rewrite transformations we have implemented specifically for XQuery and its navigational constructs. We also briefly discuss how some of the existing rewrite transformations developed for the SQL engine are extended and adapted for XQuery.

1 Introduction

XML 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. Key benefits of XML are its vendor and platform independence and its high flexibility. With the proliferation of XML data, several XML management systems [7, 10, 17, 5, 4, 6, 12, 11, 14] have been developed over the last couple of years. All major database vendors have released XML extensions to their relational engines, in addition to many native XML management systems. XQuery [18] and SQL/XML [9] are the two industry-standard languages that are supported by these systems to query XML. Most of the current research now focuses on optimization of XQuery and SQL/XML in these XML management systems.

In this paper, we describe XQuery rewrite optimization within the context of *DB2 pureXML* [4], which is a hybrid relational and XML database engine that provides native XML storage, indexing, navigation and query processing through both SQL/XML [9] and XQuery [18], using the XML data type introduced by SQL/XML. *DB2 pureXML* stores XML data in columns of relational tables, as instances of the XQuery data model [19] in a structured type-annotated tree. By storing binary representation of type-annotated trees, *DB2 pureXML* avoids repeated parsing and validation of documents. *DB2 pureXML* [4] query evaluation run-time contains three major components for XML query processing: (1) XML navigation engine, (2) XML index run-time and (3) the XQuery function library. Additionally, several relational runtime operators have been extended to deal

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*DB2 pureXML is a a trademark or registered trademark of International Business Machines Corporation.

with XML data. The XML navigation engine evaluates path expressions over the native store, by traversing the parent-child relationships in XML storage. It returns node references and atomic values to be further processed by the query run-time. Unlike other approaches in which every XPath step is modeled as separate operator [6, 16, 5], a single navigation operation in *DB2 pureXML* can evaluate multiple XPath expressions, consisting of multiple steps, as a whole. After parsing both SQL/XML and XQuery queries are mapped into a unified internal representation and optimized by the hybrid query compiler [4].

An important decision which impacted the whole XQuery compiler design is that *DB2 pureXML* does not require all XML documents in an XML column conform to a single schema, or to a collection of conforming schemas, and it does not implement static typing. Static typing is too restrictive for evolving schemas, as each document insertion or change in schema may result in recompilation of applications. As a result, XPath transformations that exploit schema information cannot be applied in *DB2 pureXML*. Instead, we focus on rewrites that optimize the general data flow in a complex XQuery or SQL/XML query. In this paper, we describe those rewrites that we developed for XQuery.

The rest of this paper is organized as follows: In Section 2, we provide an overview of how XQuery is modeled in *DB2 pureXML*, and then in Section 3 we describe rewrite transformation developed for XQuery. Finally, we conclude in Section 4.

2 XQuery Compilation in *DB2 pureXML*

DB2 pureXML provides a hybrid compiler, supporting both XQuery and SQL/XML queries. It contains several modules: two parsers, one for XQuery and one for SQL/XML, a global semantics module, a rewrite module, a cost-based optimizer module, and a code-generation module, executed in this order. XQuery and SQL/XML queries are first parsed using their respective parsers. The output of the parsers is a unified internal representation, i.e. the QGM (Query Graph Model) graph. The rest of the processing is common for both languages. The rewrite module contains a rule-based transformation engine [15], as well as several transformations that are applied before or after the rule-based engine. It applies algebraic transformation to the QGM graph. The cost-based optimizer translates the final QGM produced by the rewrite module into query execution plans and choses the optimal one. The focus of this paper is the rewrite module. But, in this section, we will start with an overview of basic QGM[15] and its extensions to XQuery, necessary to understand the rewrites.

In its simplest form, a QGM graph consists of operations (nodes) and quantifiers (arcs) which represent the data flow between operations. QGM supports arbitrary table operations, where the inputs and outputs are tables. Examples of operations include SELECT, GROUP BY, UNION, and etc. The SELECT operation node in QGM roughly represents a SPJ query block and handles restriction (selection), projection, as well as joins. Each operation consumes a set of input columns through its input quantifiers, and produces a set of output columns. Quantifiers range over operation nodes or base tables, and carry the input columns. There are two types of quantifiers: *ForEach* and *Any/All*. The expression within an operation node is applied to each tuple input by a *ForEach* quantifier. *Any/All* quantifiers are used to express universally (or existentially) qualified predicates.

XQuery [18] includes similar constructs to iterate over XML sequences, apply predicates and sort data. We exploit many existing features of QGM to model these XQuery features and introduce new entities to represent and manipulate XPath expressions and XML sequences. In general, the result of every XQuery expression is a sequence of items. Since XQuery sequences, i.e. XQDM (XQuery data model) [19] is represented as a column in *DB2 pureXML*, any sub QGM-graph that is created to represent a specific XQuery expression produces a table with a single row and a single column of type XML. FLWOR and quantified expressions define new variables that are in scope within their respective expressions. To keep track of these variable scopes, we model FLWOR and quantified expressions as scalar sub-queries, with explicit QGM operation nodes defining the query blocks. The rest of the XQuery expressions that we support are represented as scalar functions; they either have run-time counterparts that implement them, or they are expanded into detailed QGM operations later in the compiler.

Some XQuery expressions consume a sequence as a whole (such as functions), while others require iterating through the items in a sequence. We need to model these different ways of how XML data is flown into various XQuery expressions. For XQuery, we have introduced two new kinds of *ForEach* quantifiers, *FOR* and *LET*. A LET quantifier aggregates the output of an operation node into an XML sequence, whereas a FOR quantifier unnests XML sequences output by an operation node and iterates over every single item. For example, if an operation node produces a table with two rows containing $\{a, b, c\}$ and $\{d, e\}$, then the output of a LET quantifier is a single row that contains all items, i.e. $\{a, b, c, d, e\}$, whereas the output of a FOR quantifier is a table with five rows, each row containing a single item.

2.1 Representation of XPath Expressions

XPath [18] expressions consist of a series of steps, where each step either expresses navigation, or contains another XQuery expression, such as an XQuery built-in function, a FLWOR or a quantified expression, or a node constructor. The focus of earlier research has been on efficient representation and execution of XPath expressions, which contain only navigational steps. Most systems [16, 5, 6, 17, 14, 11, 7], represent and execute each step separately as selections. In other words, they normalize [20] XPath expressions into explicit FLWOR blocks, where iteration between steps and within predicates is expressed explicitly. Some provide indexes for efficient access to individual nodes. But, they all require structural joins [2] to establish parent-child (or ancestor-descendant) relationships.

In *DB2 pureXML*, we support XPath expressions, with its full generality and allow any XQuery expression in an XPath step or predicate. In general, we do not normalize XPath expressions, except in some certain cases. Instead, we represent XPath expressions, which may contain many steps and branches, as a pattern tree which computes a single variable binding. As the XML navigation engine of *DB2 pureXML* holistically computes an XPath expression, we do not need to model each step separately and we do not need structural joins to combine the results. Later, rewrites combine multiple XPath expressions into a single pattern tree, which computes multiple variable bindings.

We introduce a new operation, namely the *ExpBox*, to represent XML navigation. An ExpBox contains an annotated pattern tree, and produces tuples of XQDM bindings. A pattern tree is a tree representation of many co-mingled XPath expressions. A pattern tree node represents an XPath step and has three or more positional children. The first child of a pattern tree node represents the axis, the second one is either a name, a kind, or a wildcard test "*", and the third child represents the predicate. The rest of the children of a pattern tree node represents the next steps, and are other pattern tree nodes. Pattern tree nodes are annotated with flags to capture various properties. The *isExtraction* flag is set to *true*, if the pattern tree node computes a variable binding that needs to be extracted and returned to the run-time engine for further processing. The *isFor* flag is set to *true* if the pattern tree node represents the last step of a FOR binding. A pattern tree node can be marked as a FOR even if it does not represent an extracted variable binding. When XPath expressions are merged to eliminate unnecessary extractions, we need to remember the last step of a FOR binding so that navigation run-time can apply the correct duplicate elimination and document order rules. The *EmptyOnEmpty* flag signals when an empty sequence needs to be created if there is no qualifying node.

2.2 Representation of FLWOR Expressions

The FOR and LET bindings in a FLWOR expression produce a tuple stream, which is then filtered by the **where** clause, and the **return** clause is invoked for each surviving tuple. We model the FLWOR expression by using two SELECT operations. The lower one computes the FOR and LET bindings and applies the **where** clause predicates. We create a sub-graph for each binding and create either a FOR or a LET quantifier over it. These FOR and LET quantifiers, which provide the tuple stream as input to the lower SELECT node, reflect the join semantics of the FLWOR expression. Its output is fed to another SELECT operation, which is used to model the

return clause and the **order by** clause, if present. Later in query rewrites these two select boxes may be merged depending on the properties of the expressions in the **order-by** and **return** clauses.

3 XML Rewrites

The rule-based rewrite engine of DB2 provides several rewrite transformations for relational data [15]. Some of these rewrites are also applicable to XQuery, as they optimize the data flow in QGM by minimizing the number of operations and the length of the data flow, and both SQL and XQuery are modeled with QGM. For example, there is a rewrite which merges SELECT operation nodes. This rewrite is extended to deal with the new quantifier types, which are introduced for XQuery. This rewrite enables unfolding of nested FLWOR blocks, and minimizes the QGM graph significantly. There are other rewrites which would not be applicable and those are blocked for XQuery operations.

In this section, we focus on the new set of rewrite transformations introduced for XQuery, namely rewrites for optimizing XPath expressions and the new LET and FOR quantifiers. The main goal of these new rewrites is to consolidate XPath expressions into the least number of navigation operation nodes possible, as well as to bring comparisons into XPath expressions and close to the table access to enable XML index usage.

DB2 pureXML supports value indexes defined by XPath expressions. These indexes are used to answer XPath expressions which contain value or general comparisons. *DB2 pureXML* employs XML indexes to eliminate documents that do not satisfy XPath predicates, and uses XPath query containment algorithms of [3] to decide whether an index is eligible.

Most of the new rewrites work as part of the rule-based engine, but we also provide some transformations that are outside. If the transformation can fire multiple times and interacts with other rewrites to enable them or is enabled by them, we implement it as part of the rule-based engine. Otherwise, it is implemented as a one-time only transformation. The rewrites that are part of the rule-based engine work on one aspect, such as a quantifier or an operation node, of the QGM graph at a time and collectively simplify the QGM graph.

In addition to these rewrites, we also provide a separate rule-based transformation engine just for XPath expressions. The transformations in this set work on a single XPath expression, usually one XPath step at a time. These transformations include rules that normalize XPath expressions by eliminating parent axes, converting multiple predicates on a step into a conjunction when possible, among others.

Note that *DB2 pureXML* does not support static typing, but type information is important in query optimization. Type information can be derived from two places: from the XML schema against which the document has been validated, and from the signatures of the applied functions and operators. For example, fn:count() function always returns a single integer, and fn:data() function always generates an atomic type. We use the return data types of functions and operators, as well as literals, to infer the data type of an operation. We exploit type information both in index matching, as well as in some rewrites. For example, the FOR2REG rewrite, which is explained below, will fire if the data type of the XML column is a singleton.

In the following, we describe the general conditions under which the rewrites will fire. The actual rules contain more details, which we omit here due to space limitations.

3.1 LET and FOR Quantifier Rewrites

As discussed earlier, a LET quantifier requires aggregating the results of the operation node it ranges over, so it is translated into a group-by operation, and it is blocking. A FOR quantifier, on the other hand, needs to iterate over the results of the operation node it ranges over, and it is translated into an UNNEST operation. It desirable to eliminate both kinds of operations, if possible. We provide rewrites which tries to convert a LET quantifier into a FOR and a FOR quantifier into a regular (REG) quantifier. The first condition we check for both rewrite is that the operation node that the quantifier ranges over is not a common subexpression.

In its simplest form, we can convert a FOR quantifier into a REG quantifier if we can prove that the operation node it ranges over produces one singleton sequence. To prove this property, we may have to trace the computation back several operations. Converting a LET quantifier into a FOR is more involved and requires more properties to be proved. We check separate conditions depending on the operation node the LET quantifier ranges over. If it is a SELECT operation, then we check whether there is a subsequent FOR or a LET quantifier that obliterates this LET step, ensuring that there is no operation in between that requires to consume the output of the LET quantifier as a single sequence. If the operation node is an ExpBox, i.e. an XPath expression, then we need to prove that this XPath expression is *input independent*. We say that an XPath expression is *input independent* if its context sequence contains distinct nodes, and the subtrees pointed to by the nodes in the context sequence do not overlap. This will be true when the context column is a base table column, or the XPath expression consists of only navigational steps, and does not contain any descendant axis or positional predicates.

3.2 XPath Merging

There are two forms of XPath merging: one rewrite transformation which is part of the rule-based engine, and another one that is applied after all rewrites. The first one merges two XPath expressions, $xpath_1$ and $xpath_2$, if 1-) $xpath_1$ computes the context of $xpath_2$, 2-) there is no predicate on the first step, i.e. the context step, of $xpath_2$, 3-) the output of $xpath_1$ is only used in $xpath_2$ as the context, and 4-) $xpath_1$ and $xpath_2$ are compatible in their distinctness properties. When we merge these two XPath expressions, we create a new ExpBox containing the XPath expression that is the concatenation of $xpath_1$ and $xpath_2$, without its context step, and we mark the quantifier ranging over this new node same as the quantifier ranging over $xpath_2$. Note that if $xpath_1$ is a FOR binding, then we need to be careful to produce the correct set of results. For example, suppose $xpath_1$ is a FOR binding and produces $i as $\$doc//customer$ and $xpath_2$ is a LET binding given by $\$i/accountId$. If the document has multiple customers, the final output should be a set of account id's for each customer. When we merge the two XPath expressions into $\$doc//customer/accountId$ and mark the final output as a LET binding, we also mark the intermediate $customer$ step as a FOR step, so that our navigation run-time produces the correct output.

The second transformation takes as input the resulting QGM after all the rewrites have been applied. It first computes a dependency graph among the XPath expressions in a query block, i.e. a SELECT operation node. Next, the algorithm partitions the set of XPath expressions within the same query block that are over the same document into clusters, by taking into account the interactions with other operations in the query so as not to sacrifice an optimal execution plan. Finally, it merges the XPath expression within the same cluster, as long as the resulting dependency graph is acyclic. This transformation produces expressions which compute multiple bindings. The details of this rewrite can be found in [1].

3.3 Resetting EmptyOnEmpty Flag

A let-clause binds its variable to the result of the associated expression, even when the result of the expression is an empty sequence. As all values of the LET bindings need to be returned, we cannot use an XML index to compute the expression in a LET binding, unless we can prove certain properties. We introduce a new quantifier flag, called *EmptyOnEmpty*, which signals that the quantifier needs to produce an empty sequence, even if the operation node it ranges over produces no results. When we first parse an XQuery expression, we create a LET quantifier over all XQuery expressions, and over LET bindings, because all XQuery expressions have implied LET semantics [18]. Later, we provide a rewrite transformation which tries to reset this flag, enabling both index usage and several other rewrites, most notably the one that merges SELECT boxes.

In general, we can reset the *EmptyOnEmpty* flag when there is a **where** clause predicate which eliminates the empty sequence, and there are no other consumers of that LET binding. Moreover, there are two other XQuery operations which discard the empty sequences, iterators, such as FOR clauses, and sequence concatenation. If

we prove that the empty sequence is to be discarded later on due to one of these operations, we can reset the *EmptyOnEmpty* flag.

3.4 Local Predicate Pushdown into XPath Expressions

Similar to pushing down selections in a relational query, we provide a rewrite which tries to push down *local* predicates into base column accessing XPath expressions to filter out unqualified data as early as possible. We consider a predicate to be *local*, if it accesses only one document. Moreover, an XPath such as $\$doc/cid$ can also be considered as a local predicate by converting it into $\$doc[cid]$, and can be pushed down to its context XPath. We call this *XPath pushdown*.

3.4.1 XPath Pushdown

XPath itself can be considered as a local predicate, as navigation steps are also existential tests. A set of rewrite rules together implement XPath push down. This set mainly includes (1) rules to push down XPath through operations such as SELECT and UNION, base tables, and XML element construction, and (2) *XPIMPLY* rule, which converts XPath into a local predicate.

An XPath can be pushed down if the following conditions hold: 1-) The XPath expression consists of only navigational steps, and does not have any steps containing functions, such as $\$doc/a/fn:concat(b,c)/d$. Note that functions in predicates do not block this rewrite. 2-) There are no common subexpressions along the path where the XPath expression will be pushed down. 3-) The XPath expression is *input independent*. 4-) The target operation node does not have any sorting requirements.

During push down, each rule pushes down the XPath expression through one operation node at a time. The rule engine remembers the current pushable position and makes a new copy of the pushable XPath expression. It then recursively calls the next rewrite to further push down the XPath expression. This way we try to reach to the base table level, where we can enable index matching. Once the rule engine locates the operation node where the XPath expression cannot be pushed down any further, *XPIMPLY* rule fires and converts an XPath expression of the form $\$d/steps$ into $\$d[steps]$, provided that there is no other consumer for this XPath expression.

3.4.2 Local Predicate Pushdown

This rewrites pushes down a **where** clause predicate into an XPath expression. Consider the following query: *Query I:* $for\ \$c\ in\ db2-fn:xmlcolumn(\text{"}T2.DOC\text{"})/c,\ \$a\ in\ \$c/a\ where\ \$c/d=5\ return\ \$c$.

The predicate $\$c/d=5$ in this query can be pushed down into the first XPath expression, and rewritten as: *Query II:* $for\ \$c\ in\ db2-fn:xmlcolumn(\text{"}T2.DOC\text{"})/c[d=5],\ \$a\ in\ \$c/a\ return\ \$c$.

In general, a **where** clause predicate can be pushed down into the context XPath expression if: 1-) It is a *local* predicate, containing general and value comparisons, connected with conjunction and/or disjunction, 2-) It is not a predicate on an aggregation result, and 3-) The target XPath expression is a FOR binding. This rewrite does not work only in a single query block. Instead, when we locate such a candidate predicate, we disconnect it from its current SELECT operation node, and try to push it down as many query blocks as possible. This rewrite helps consolidate XPath expressions, and may enable merging of further XPath expressions. For example, for *Query II*, XPath merging rule will fire at some point, and merge the two XPath expressions, consolidating the whole query into a single XPath expression.

3.5 Join Pull up (Simple Decorrelation)

Consider *Query I* below, which contains an XPath expression with a correlated variable, expressing a join. There are several problems with this query: 1-) The join order is fixed due to the correlation, 2-) Only nested-loop join

method can be used, and 3-) Only an index on T1 can be used, and any index on T2 cannot be exploited, because the XPath expression on T2 needs to be executed first.

Query I:
```
for $i in db2-fn:xmlcolumn("T2.DOC")/c,
$j in db2-fn:xmlcolumn("T1.DOC")/a[b=$i/d]
return $j
```

Query II:
```
for $i in db2-fn:xmlcolumn("T2.DOC")/c,
$j in db2-fn:xmlcolumn("T1.DOC")/a
where $j/b=$i/d
return $j
```

To address these problems, we provide a rewrite, called *join pull up*, which pulls up join conditions embedded in XPath expressions into the **where** clause, decorrelating the query. For example, *Query I* will be converted into *Query II*. This enables the optimizer to consider using both join orders, all join methods, as well as both indexes on T1 and T2. In general, a join predicate can be pulled up when all of the following conditions hold: 1-) The quantifier ranging over the ExpBox containing the join predicate, is either a FOR quantifier, or a LET quantifier, which does not have the *EmptyOnEmpty* flag set and which is not consumed anywhere else. 2-) The join predicate is either a general or a value comparison. 3-) It is the last predicate of a predicate sequence. 4-) It is a predicate, which maybe connected by a conjunction. Given a predicate of form $xp[prd1AND(prd2ORprd3))$, only $prd1$ is considered for pull up.

3.6 Query Decorrelation Rewrites

A correlation is a reference to a variable that has been defined in a previous or enclosing query block. Correlated subqueries are quite common in XQuery. For example, most grouping queries in XQuery are expressed using correlation. Although this a natural way of writing queries, it provides several performance bottlenecks: It severely limits the optimizer choices, because the correlation imposes a partial join order, and only a nested-loop join method can be used. Moreover, in a parallel environment correlation creates a synchronization point, and becomes a bottleneck in the data flow.

As we discussed earlier, *DB2 pureXML* query compiler already employs a variety of simplifying rewrite transformations, which may decorrelate some of the simple cases, such as join-pull up rewrite. However, only the magic decorrelation rewrite [13] addresses the most general problem. The magic decorrelation algorithm is closely entwined with the magic sets rewrite [8]. For convenience, we highlight the aspects of these rewrites that need to be revisited for XML processing.

When magic processing a subquery that contains the correlation variable, we generate a magic operation node as a SELECT DISTINCT operation, joining all the eligible quantifiers. Eligible predicates are pushed to form a semi join under an adornment node, effectively filtering the data stream. The adornments consists of the set of conditioned, bound, and free variables which are determined by the pushed predicates. Simply put, the magic sets rewrite generalizes local predicate pushdown to join predicates. Enforcing distinctness in the magic node is important so that we do not increase the total cardinality. Magic decorrelation rewrite [13] extends the magic sets to correlations. In this case, the magic node flows all to-be-decorrelated columns.

The main challenge in decorrelating an XML-typed variable reference is enforcing distinctness in the magic node. There are different ways in which XML data can be compared. One natural way is to employ the $fn : data()$ function to retrieve a comparable value. However, this approach can be costly since we potentially deal with large XML-structures. Another way is to use node id's (which are comparable) to perform equality comparisons and GROUP BY operations. However, XML type can contain both nodes and atomic values, which do not have id's. If we can prove that the XML type only contains nodes, we can use the id-based approach. But, in the general case a better solution is to ensure that we do not have to enforce distinctness. We can achieve this by adding keys to the magic node and to the list of to-be-decorrelated columns during decorrelation. We can obtain keys from descendant nodes as follows: For base tables, we pull up any key defined on the table. If no such key exists, we can use the record identifiers of the base tables. For a node which enforces distinctness, we can pull up all of its output columns. For any join node, we can pull up keys from every join operand. If we can

determine and add such keys, then we do not have to enforce distinctness on the magic node, and we do not add any GROUP BY columns or equality predicates using any XML-typed columns. Naturally, we cannot always determine such keys. However, we observed that this approach is better-suited and more flexible for a majority of queries.

4 Conclusion

In this paper, we described XQuery compilation and algebraic rewrite optimization within the context of *DB2 pureXML*, a hybrid relational and XML database engine. We focused on rewrites whose main goal was to consolidate the XPath expressions in the query into the least number of possible navigation operations and enable index usage. We provide other rewrites, which are needed to simplify the QGM graphs generated for XQuery and SQL/XML, in addition to the rewrites we described here. We omit those due to space limitations.

References

[1] A. Balmin and F. Özcan and A. Singh and E. Ting. Grouping and optimization of XPath expressions in DB2 pureXML. In *Proc. of SIGMOD*, 2008.

[2] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of ICDE*, 2002.

[3] A. Balmin, F. Özcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. of VLDB*, Toronto, Canada, 2004.

[4] K. Beyer et al. System RX: One part relational, one part XML. In *Proc. of ACM SIGMOD*, pages 347–358, 2005.

[5] C. Re and J. Simeon and M.F. Fernandez. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of ICDE*, 2006.

[6] D. Florescu et al. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[7] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. of VLDB*, Toronto, Canada, 2004.

[8] I. S. Mumick and H. Pirahesh. Implementation of Magic-sets in a Relational Database System. In *Proc. of SIGMOD*, pages 103–114, 1994.

[9] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML), ANSI/ISO/IEC 9075-14:2006.

[10] H. V. Jagadish et al. TIMBER: A Native XML Database. *VLDB JOurnal*, 11(1):274–291, 2002.

[11] Z.H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery Processing in Oracle XMLDB. In *Proc. of SIGMOD*, pages 828–833, 2005.

[12] M. J. Carey. Data Delivery in a Service-oriented World: The BEA AquaLogic Data Services Platform. In *Proc. of SIGMOD*, pages 695–705, 2006.

[13] P. Seshadri, H. Pirahesh and T. Y. C. Leung. Complex Query Decorrelation. In *Proc. of ICDE*, 1996.

[14] S. Pal et al. XQuery Implementation in a Relational Database System. In *Proc. of VLDB*, 2005.

[15] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of SIGMOD*, pages 39–48, 1992.

[16] J. Shanmugasundaram et al. Querying XML Views of Relational Data. In *Proc.of VLDB*, pages 261–270, Roma, Italy, September 2001.

[17] T.Fiebig et al. Anatomy of a Native XML Base Management System. *VLDB JOurnal*, 11(4), 2002.

[18] *XQuery 1.0: An XML Query Language*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xquery`.

[19] *XQuery 1.0 and XPath 2.0 Data Model*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xpath-datamodel`.

[20] *XQuery 1.0 Formal Semantics*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/query-semantics`.