XPath on Steroids: Exploiting Relational Engines for XPath Performance †

Haris Georgiadis
Athens University of Economics and Business
76 Patission Str, GR 10434 Athens, Greece
harisgeo@aueb.gr

Vasilis Vassalos
Athens University of Economics and Business
76 Patission Str, GR 10434 Athens, Greece
vassalos@aueb.gr

ABSTRACT

A lot of research has been conducted by the database community on methods and techniques for efficient XPath processing, with great success. Despite the progress made, significant opportunities for optimization of XPath still exist. One key to further improvements is to utilize more effectively existing facilities of relational RDBSes for the processing of XPath queries. After taking a comprehensive look at such facilities, we present techniques for XPath processing that work by identifying the best relational join algorithms, indices and file organization strategies for XPath queries. Our techniques both reduce the latency of the resulting SQL translations and guarantee their pipelined execution. We also propose a new technique for XML reconstruction from relations-mapped XML that "splits the difference" between schema-aware and schema-oblivious XML-to-relational mapping for a significant performance improvement. An extensive experimental study confirms the performance benefits of our optimization techniques and shows that a system implementing these techniques on top of a commercial RDBMS is competitive with respect to query performance with other native and relational-based state-of-the-art XPath processing systems, commercial as well as research prototypes.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Systems – *Query processing, Relational databases.*

General Terms

Algorithms, Measurement, Performance, Experimentation.

Keywords: XML, XPath, Relational databases, XML Reconstruction, Schema Mapping, Structural Joins, Indices, Dewey encoding

1. INTRODUCTION

There is today wide and increasing use of XML for a variety of data exchange, data processing, and data integration tasks. XML is used in many application domains where there are large quantities of semistructured data to be exchanged or processed, from bioinformatics [5] to astronomy [6] to large corporate data processing.

The wide applicability of XML and the difficulties in processing XML data, arising from its semistructured and hierarchical nature, the rich schema query language facilities, and the diverse application requirements, have led the data management community to devote a lot of effort to the XML processing challenge [21].

Even though these efforts have yielded very efficient systems and techniques, there is still significant room for improvement in a variety of areas related to XML processing, and especially in using relational infrastructure for XML processing [4][9]. We have identified two such distinct areas of possible improvement:

- Reconstructing XML results from relational tables resulting from XML shredding [13][12].
- Exploiting existing relational facilities for XML processing. To the extent that relational back-ends are used to store and process decomposed (i.e., *shredded*) XML data, it seems natural to try to use the existing relational facilities in as efficient a manner as possible. Not much effort has been devoted to "fitting" SQL translations of XML queries to the abilities of their relational "hosts".

With regards to the second point, we take a comprehensive look at the choices open to an XML-to-relational mapping system, and the corresponding choices for XML-to-SQL translation. We identify four key issues and devise techniques to improve performance in each case:

a) Choosing the right relational join technique. We compare relational join techniques and conclude that index nested loop joins are superior to sort merge for queries processing shredded XML data, for a variety of reasons (Section 5.2.1).
b) Eliminating redundant ordering and duplicate elimination operations, which affect both latency and time to first result. The properties of order encoding schemes used in XML-to-relational mapping allow us to prove that often structural joins produce correctly sorted results. In the other cases, performance is improved by pushing ordering down the plan tree. Both result in generation of SQL statements with fewer (or none) ORDER BY and DISTINCT clauses (Sec. 5.2.2).
c) Generating pipeline-able plans, to improve "first result" and total query performance. Using index nested loop joins, making the effort to create pipeline-able plan trees, improves performance further, as we discuss in Section 5.2.3.

† Research supported by the PENED Programme of the Greek Secretariat of Research and Technology.

SIGMOD'07, June 12–14, 2007, Beijing, China.

d) Picking the right physical organization. Among the many available options, indexed file organization is shown to be the best choice (Section 5.3)

All of the above choices are validated with extensive experiments over synthetic and real data presented in Section 6. We also note that the relational optimizer, when left alone, makes the wrong choices, so we create a relational-based system for XPath processing that automatically inserts the appropriate optimizer instructions, i.e., hints, in SQL translations.

Concerning the first point, we propose a new XML reconstruction technique that decomposes XML in relations in two different ways to combine the benefits of "schema-aware" and "schema-oblivious" XML-to-relational mapping. This *hybrid* XML reconstruction technique has significant performance benefit, as shown in our experimental study in Section 6.4.

We implement a system on top of a commercial relational engine that includes all the techniques we propose to exploit the observed inefficiencies. In Section 6.5, we compare this system with a variety of other state-of-the-art XPath processing systems, commercial as well as research prototypes.

2. RELATED WORK

A lot of research work has been done for both relational [11][23] and native [2][22] XML processing. A number of systems have been developed, such as Natix[2], MonetDB/XQuery [8] and Timber [22] that are able to store and query large collections of XML documents. Commercial RDBMSs have also enhanced their functionalities with XML management capabilities [4][24][3].

Many techniques have been developed to improve XML query performance. Special index structures and joins algorithms have been developed [25][26][27] and existing relational facilities for indexing, such as B-trees, have been used [11]. In the context of XML-to-relational mapping, several techniques [28][29][30] have been proposed to explore among mapping alternatives and pick the best for certain application characteristics. Not much attention has been paid to better exploiting the full set of relational facilities, e.g., join algorithms, indices, file organization methods, to speed up XML querying.

A key issue for XPath and XQuery processing is XML (re)construction, especially for relational-based systems. In *schema-oblivious* XML-to-relational mapping, XML data is shredded into a single element relation, and an order encoding scheme (e.g. dewey or pre/post encoding [11]) is used for structural relationships. These can be used to easily retrieve all descendants of a given element [11]. In schema-aware mapping, XML data is mapped to a relational schema derived from the XML Schema of the data. Little work has been done on XML reconstruction in this case. In [18], an XML Schema-to-Relational mapping that uses inlining is enhanced with range encoding *only* for handling XML reconstruction. For a given element, its XML descendants are retrieved by issuing as many queries as the number of relations that potentially store descendant elements. The results of these queries are unioned and sorted in document order outside the relational engine. The XML subtree is created from the sorted relation as in [11]. Hence, several queries must be executed for every result element of an XPath expression. In [30], a technique for XML reconstruction based on storing preparsed XML fragments as BLOBs is proposed.

There is also relevant work in XML publishing, i.e., exporting of existing relational data into XML, where nesting is implied by foreign-key references. Such techniques can be applied for retrieving the subtree of each element returned by the SQL translation of an XPath expression. In [12] and [13] two methods are described: path outer unions and sorted outer joins. Both evaluate all possible root-to-node paths as sequential foreign-key joins and try to exploit common joins by using temporary views. All queries corresponding to the root-to-node paths are unioned, with the latter technique sorting the results on all the primary key columns. Common joins for parent-child and sibling elements are exploited in [14].

XML reconstruction from schema-aware mapped relations has similarities to the problem of evaluating recursive queries: in both cases SQL translations must evaluate multiple join paths. Recent research [16],[17] handles such queries, even in the presence of recursive schemas, using the SQL'99 WITH clause and a special operator for linear recursion, respectively. The produced SQL statements are very complicated and therefore hard to optimize.

Finally, in [9], we present a set of high performance techniques for XPath processing on top of a relational engine. The main novel techniques are one-step processing of complex path fragments using regular expression filtering, and turning structural joins into dewey-encoding-based theta joins. The techniques are shown to offer significant performance benefits while being implemented purely on top of existing relational technology. Nevertheless, the system has significant limitations: it does not handle XML reconstruction, i.e., XPath queries return just a list of selected nodes. Also, there is no mention of physical data organization and no effort is made to better exploit existing relational facilities. Given the lightweight integration required with the underlying relational engine, and the performance characteristics of the techniques it introduces, we use [9] as the basis for our investigation and the implementation of the novel techniques and algorithms presented in this paper. In the following section, we provide a short background description of PPF-based XPath processing.

3. USING PRIMARY PATH FRAGMENTS FOR XPATH

The PPF-based XPath processing system we described in [9] is an RDBMS front-end for answering XPath queries. It creates a dedicated relational schema for the XML schema of an XML document (hence using a *schema-aware* XML-to-relational mapping). Based on the schema mapping, it processes XPath queries by translating them into SQL statements, which are then processed by the relational engine. A brief description of those two functions, necessary for the development of our work, follows in Sections 3.1 and 3.2, respectively. We will use, by abuse of abbreviation, PPFS to mean "the PPF-based XPath processing system described in [9]". We will refer to the novel system and techniques we develop in this paper as PPFS+.

3.1 Schema Mapping

There are three main issues that must be considered in XML-to-relational mapping. The first is how the main structures of XML schema are mapped into relational ones. PPFS uses the simplest alternative for that: each element definition is mapped into a separate relation, attributes as well as text nodes are mapped into columns of the appropriate type into the respective relations.

The second important aspect is how to store structural information for the XML documents, like element nesting and ordering. PPFS uses dewey encoding [19] for mapping both element nesting and ordering.[1] Each mapping relation has a `dewey_pos` primary key column for storing the structural position of each element. In dewey encoding, each element is assigned a unique vector of numbers. Each number of the vector represents the local order of the respective ancestor element among its siblings. Figure 1 shows an example: the node with dewey position 1-3-2 is second among its siblings and its parent is the third child of the root. PPFS uses a parent column for handling the parent-child axis. In PPFS+ we changed that: we don't maintain parent columns, since we noticed that all axes can be handled with dewey-based structural joins

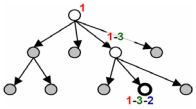
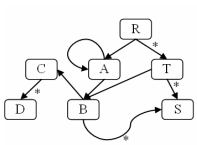


Figure 1. Dewey order of XML nodes

The third key issue is what additional information is stored to allow more efficient querying. PPFS stores for each element node its root-to-node path and uses it as an index. Paths are stored in a separate relation, named *Paths*, which is usually very small (some KB for XML databases of size in the hundreds of MB). All mapping relations maintain a foreign key reference to this relation, in a column named `path_id`.

Example 1: Consider a simple XML Schema H which will be used as a running example for the following sections. The schema graph I of the schema is illustrated in the left part of Figure 2. We suppose that among the element definitions, only S and D have a text node and that B and D have two attributes each, namely j and k. The corresponding relational schema is shown in the right part of Figure 2. □



R_(dewey_pos, tag_name, path_id)
T_(dewey_pos, tag_name, path_id)
S_(dewey_pos, tag_name, path_id, text_)
A_(dewey_pos, tag_name, path_id)
B_(dewey_pos, tag_name, path_id, j_, k_)
C_(dewey_pos, tag_name, path_id)
D_(dewey_pos, tag_name, path_id, text_, j_, k_)

Figure 2. Sample XML schema and corresponding relations

3.2 XPath-To-SQL Translation

Based on the XML mapping just described, PPFS uses an XPath-to-SQL translation algorithm to create an SQL statement that produces the tuples representing the *target* elements defined by the XPath expression (but not their subtrees). The SQL statement returns the tuples in document order [1]. To produce SQL statements that are efficiently executed, two techniques are employed: a) XPath expressions are divided into parts called Primitive Path Fragments (PPFs), where each PPF can be processed in a single step using regular expression filtering on the root-to-node paths and b) using a clever encoding of dewey order, any kind of structural join between PPFs (i.e., any XPath axis) is translated into theta joins on simple arithmetic conditions. Briefly, a PPF is:

- a forward path with no predicates except for the last step
- a backward path with no predicates except for the last step or
- a single step of one of the other axes: following, following-sibling, preceding or preceding-sibling.

PPFS identifies the relation, called the *prominent relation*, that corresponds to the final step of each PPF path. Only prominent relations participate in SQL translations. Each PPF is handled by filtering the path ids of the tuples of the prominent relation, so that their corresponding root-to-node paths match a regular expression derived directly by the PPF path. For the regular expression matching, PPFS uses the built-in `REGEXP` function of Oracle 10.2g (other RDBMS use similar functions).

Example 2: Consider the query `/R//B/*/D` over an XML document conforming to schema H of Example 1. The query consists of a single PPF whose prominent relation, according to our mapping, is *D_*. The regular expression equivalent of that PPF is '`/R/(.+/ )? B/[^/]+/D`$' (using the POSIX syntax). The final SQL translation will be:

```
SELECT DISTINCT D_.dewey_pos, D_.tag_name, D_.text_, D_.j_, D_.k
FROM D_, Paths
WHERE   D_.path_id = Paths.id
        AND REGEXP_LIKE(Paths.path,'^/R/(.+/ )? B/[^/]+/D$')
ORDER BY dewey_pos
```

Although the XPath expression contains a descendant axis (//) and a wildcard (*), only one mapping relation is involved in the SQL statement and no structural joins are needed. □

When more than one PPF exist, the prominent relations of consecutive PPFs are structurally joined on their dewey position columns. Lexicographical comparisons over dewey positions are used to handle all XPath axes.

Example 3: The XPath expression `/R/A/A[B/*/D/@k="2"]` consists of two PPFs, '`/R/A/A`' and '`/B/*/D`' with *A_*, *D_* being their prominent relations, respectively. The SQL translation is:

```
SELECT DISTINCT A_.dewey_pos, A_.tag_name
FROM A_,  Paths A_paths
WHERE A_.path_id =A_paths.id and A_paths.path = '/R/A/A'
AND EXISTS (
        SELECT NULL FROM D_, Paths D_paths
        WHERE D_.path_id = D_paths.id
        AND REGEXP_LIKE(D_Paths.path, '/R/A/A/B/[^/]+/D$')
        AND D_.dewey_pos BETWEEN A_.dewey_pos
                        AND A_.dewey_pos ||CHR(255)
        AND D_.k_ = 2 )
ORDER BY A_.dewey_pos
```

Note that chr(255) returns the character with ASCII code 255, thus the byte corresponding to FF in hexadecimal notation. □

In what follows, we use the term *backbone* to refer to the path of an XPath expression that remains if we omit all predicates. We also define the *selective prominent* relation of an XPath to be the prominent relation of the last PPF of the backbone path.

4. XML RECONSTRUCTION

PPFS only returns the sequence of XML nodes identified by an XPath expression. Each node may have arbitrarily complex content structure. For an XML management system to be practical, it must be able to return results in XML form. We call *target elements* those located by the XPath expression to distinguish them from the elements of the subtrees rooted at them. In this section, we study the problem of efficient XML reconstruction and describe the "first cut" PPFS+ solution. For efficient XML reconstruction, the following questions arise:

1. Which relations potentially store descendant elements?

[1] We have experimented with an alternative encoding scheme, pre/post encoding, and have identified performance benefits from using dewey encoding. The experiments are not reported due to lack of space.

2. Given an SQL translation of an XPath, how can we retrieve, for each target tuple (that corresponds to a target element), all its descendants in an efficient and *pipelined* manner?
3. For each target tuple, its descendants will be retrieved as a sequence of flat tuples. How can we construct an XML subtree from that sequence of tuples?

The first question is answered easily by resorting to the XML Schema graph. Regarding the second question, since dewey encoding is used for selecting descendant nodes, we retrieve the descendants of each target node using range conditions on `dewey_pos` columns. In short, we translate the XPath query into an SQL select statement as described in Section 3.2 and define with this statement a temporary view, called *target_elements*. Another temporary view, called *descendants*, unions the relations that possibly store descendant elements. Those two views are defined using the '`WITH` clause' of SQL'99 which allows the definition, within a single statement, of several temporary views along with a main query that refers to them. The main query joins the two views using the ancestor-descendant join condition. We present here a sketch of the technique via an example and elaborate on it in Section 5.4.

```
WITH target_elements AS (
        SELECT T_.text, T_.dewey_pos, T_.tag_name,
        FROM T_, PATHS WHERE T_.path_id = PATHS.id AND PATHS.path= 'R/T'),
descendants AS (
        SELECT dewey_pos, tag_name, text_, NULL AS attributes_
        FROM S_ UNION ALL
        SELECT dewey_pos, tag_name, NULL AS text_, 'j="'||j_||'" k='||k_||'"' AS attributes_
        FROM B_ UNION ALL
        SELECT dewey_pos, tag_name, NULL AS text_, NULL AS attributes_
        FROM C_ UNION ALL
        SELECT dewey_pos, tag_name, text_, 'j="'||j_||'" k='||k_||'"' AS attributes_
        FROM D_ )
SELECT target_elements.*, descendants.*
FROM target_elements LEFT OUTER JOIN descendants
WHERE descendants.dewey_pos BETWEEN target_elements.dewey_pos
                    AND target_elements.dewey_pos||CHR(255)
ORDER BY descendants.dewey_pos;
```

Figure 3. XML-reconstruction enabled SQL translation

Example 4: Consider the XPath query `/R/T` over an XML document that conforms to schema H. It consists of one PPF, with selective prominent relation *T_*. We define view *target_elements* using the algorithm discussed in Section 3.2. The relations corresponding to descendants of T are *S_*, *B_*, *C_* and *D_*. View *descendants* takes a union of these relations. Figure 3 shows the SQL statement that returns a denormalized relation produced by the outer join of these two view relations. The join matches each tuple of *target_elements* with its descendants from the *descendants* view. Therefore, each target element is repeated in the result relation as many times as the number of its descendants. To make the conversion of this result into XML form more efficient, a pseudo-column called `target_element_num` is added to the *target_elements* view that stores the row number of each tuple in the view. PPFS+ used the Oracle built-in pseudo-column `ROWNUM`. Target elements are distinguished by comparing the values of the `target_element_num` fields (which is much cheaper than comparing their `dewey_pos` fields).

Execution of the resulting SQL statement will vary among different RDBMSs. We discuss the generation of "better" SQL statements and the generation of optimal execution plans for such queries in the next section. The `ORDER BY` clause in the final query guarantees that the results are sorted in document order. We will see in Section 5.2.2 that there is a much cheaper way to guarantee correct ordering. Moreover, note that the view descendants should not be precomputed, resulting in a possibly huge intermediate result. Instead, the final query that joins *target_elements* and *descendants* should repeatedly compute parts of *descendants* that correspond to the descendants of each tuple of *target_elements*, exploiting existing indexes on relations *E_*, *F_*, *G_* and *H_* to speed up the structural joins. □

Finally, note that the relations that are unioned in *descendants* may have different schemas, since some of them may have columns corresponding to text and attribute nodes and some may not. To eliminate these differences, we group for each relation all attribute columns under a single pseudo-column called `attributes_`, as shown in Figure 3. If a relation doesn't have attribute columns, `attributes_` equals NULL. Similarly, for those relations that do not have a `text_` column, we add a pseudo-column named `text_` equaling NULL. In Figure 3, *B_* is enhanced with column `attributes_` which is defined as the concatenation of: the string 'j="', the value of column `j_`, '" k="', the value of column `k_`, and '"'. Therefore, the value of the `attributes_` field of a *B_* tuple with values 'a' and 'b' for the fields `j_` and `k_` will be 'j="a" j="b"'. The column is formatted as attributes are in XML elements, so it can be included in XML elements reconstructed in the final result with no extra effort.

The last question is how to reconstruct an XML document from the result relation. For this, PPFS+ uses a technique similar to that described in the appendix of [11], where the dewey position of the element being fetched is compared with that of the previously fetched element, to decide whether it should be nested in the previous element or not. In PPFS+, a relation tuple includes information for both a target element and a descendant. To produce results correctly, we must identify when the target element changes, which is done, as mentioned above, by comparing the values of the field `target_element_num`.

5. TUNING THE RELATIONAL IMPLEMENTATION

As discussed in previous sections, many different shredding schemes and XPath-To-SQL translation techniques have been proposed. A key requirement for all is that the SQL translations of XPath queries must be as efficient as possible. The relational back-end executes the plan that the query optimizer estimates as the optimal; however, in most cases SQL translations are complicated and, moreover, shredded XML data have some non-obvious characteristics that can be exploited for better perfor-mance – which relational optimizers always miss. In this section we take a closer look at XML-to-Relational mapping, XPath-to-SQL translation and the XML reconstruction technique presented in the previous sections, and focus on the implementation strategies available with a relational storage and execution engine. We conclude on certain strategies and significantly modify the XPath processing techniques to improve query performance.

In Section 5.1 we discuss some distinctive features of our SQL translation and mapping techniques. Based on these features, we first focus on the relational join algorithm and indices that fit better with dewey-based structural joins and discuss how they affect the XPath-to-SQL translation algorithm (Section 5.2). We then compare file organization methods and identify the most appropriate one for XPath processing (Section 5.3). Based on these findings, we revisit XML reconstruction and provide new algorithms that significantly improve efficiency (Section 5.4). Detailed experiments (Section 6) validate the proposed techniques.

5.1 Key observations and requirements

An important requirement of XPath semantics [1] is that the sequence of target elements returned by the evaluation of an XPath expression must be sorted by document order and have no duplicates. This implies that the result relation of the target query must be ordered by the `dewey_pos` column. Likewise, the elements of the sub-trees rooted at each target element must preserve the ordering of the original XML document.

One interesting observation for dewey order that cannot be explicitly declared in the database engine is the following: if the dewey position of element A is greater than the dewey position of element B, and A is not a descendant of B, then the dewey position of every descendant of A will be greater than the dewey position of every descendant of B. This is important because it implies that, under certain conditions, the result of ancestor-descendant structural joins is naturally ordered by the `dewey_pos` column of the *descendants* relation without extra effort.

When deciding on indices and file organization, it is important to know what columns of the relational encoding are accessed to answer an XPath query. The `dewey_pos` and `path_id` columns of all prominent relations participate in selection conditions. The existence of filtering predicates may also cause columns corresponding to text or attribute nodes to participate in selection conditions. The selective prominent relation returns all its columns. Concerning the *descendants* view, almost all columns of the relations are projected since they are needed for the construction of the resulting XML document.

Finally, note that, given the wide use of XML in web-based applications and their interactive nature, it is usually vital for XML processing to provide "first answers" quickly. To achieve this goal, execution plans must be pipelined and blocking operations must be avoided, which means that explicitly declared sorting should be avoided. When XML reconstruction is required, the number of tuples processed and returned by the SQL translation can be very large, since every single element is mapped into a separate tuple (no inlining is used), which makes sorting them prohibitively expensive. We exploit the above observations in the following sections.

5.2 Index Nested Loops For Structural Joins

This section answers three interrelated questions: what relational join algorithms map better to dewey-based structural joins, what indexes help the most, and under which circumstances `DISTINCT` or `ORDER BY` clauses are redundant.

5.2.1 Nested Loops and b-tree index on dewey_pos

Since structural joins involve range comparisons, a standard B-tree index for the *dewey_pos* column is optimal since it can efficiently handle range conditions. A simple example follows.

Example 5: Consider an ancestor-descendant join between element types C, D defined in the XML schema H of Example 1. C_ and D_ are the respective relations. The SQL statement that performs this join is the following:

```
SELECT D_.dewey_pos, D_.text_
FROM C_, D_ /* index nested loops for joining C_ and D_
WHERE       /*with D_ being the inner table*/
D_.dewey_pos BETWEEN C_.dewey_pos AND C_.dewey_pos||CHR(255)
```

The query enforces the condition that elements corresponding to *D_* tuples are descendants of those corresponding to *C_* tuples. Dewey encoding allows the discovery of all descendant D elements of a given C element with a simple range index lookup. Particularly, if the dewey position of the C element is x then we need all D elements with dewey positions ranging from x up to x||CHR(FF). These can be easily retrieved from the linked list of the block leaves of B-tree indices. □

Given the above, index nested loops (index-NL) are ideal for performing structural joins based on dewey encoding: Scan the ancestor relation and, for each dewey position key, perform a range index lookup in the dewey position index of the descendant relation. This way, only B-tree leaf blocks containing descendant tuples are visited. A clustered index on the dewey position column is present since this column is the primary key.

We can force the optimizer to use index-NL by incorporating special implementation instructions, i.e., *hints*, as comments into the SQL translations before we issue them to an RDBMS. The RDBMS reads the hints and tries, if possible, to produce physical plans that conform to these instructions. Hints can demand specific join algorithms, access path method, use of indices, join ordering, etc. In the statement of Example 5 as well as in following SQL statements, we use natural language to describe hints, for ease of understanding. In the PPFS+ implementation, the necessary Oracle hints are generated automatically.

NL can also provide the first matching row quickly. Sequences of NL joins can be executed in a pipelined fashion, which as discussed is desirable for XPath-to-SQL translations. The execution plan of the SQL translation of an XPath query with predicates will have a tree structure of nested loops joins, which still preserves pipelining. However `DISTINCT` and `ORDER BY` clauses will spoil pipelining.

In Section 6.1, index-NL join is compared with the other well-known relational join algorithm for dewey-based structural joins, Sort Merge join. As shown by the experiments NL join performs much better than Sort Merge.

5.2.2 Eliminating redundant sorting

In the previous section we highlighted that the result relation of an SQL translation must be ordered by the `dewey_pos` column of the selective prominent relation. For Example 5, this means that we must add an `ORDER BY` clause on the `D_.dewey_pos` column. Since relation *D_* is the inner argument of a nested loop join, the index on the dewey position column cannot help for sorting; the engine has to sort the join results. However, we can exploit the property of `dewey_pos` mentioned in the previous section in order to omit the sorting overhead. First, a necessary definition: We call *recursive* a relation whose corresponding node in the schema graph is part of a cycle. In schema graph I from Example 1, relation *A_* is the only recursive relation.

Theorem 1: Consider an ancestor-descendant join between relations *R_* and *S_* corresponding to element types R and S respectively and assume that ancestor element type R is not recursive. If relation *R_* is accessed in `dewey_pos` order, relation *S_* has a B-tree index on the `dewey_pos` column and index-NL join is used, then the projection of the join tuples on the columns of *S_* will be duplicate-free and sorted by document order. □

The theorem can be generalized to sequences of structural ancestor-descendant joins: When the backbone of an XPath expression consists of forward PPFs whose prominent relations are

not recursive, `ORDER BY` and `DISTINCT` clauses can be omitted from the target query.

Example 6: Let's see why we can omit the `ORDER BY` clause in the structural join of Example 5. Element type C is not recursive, so for two C elements that are accessed in document order, all the descendant elements of the first will open and close before the second C element opens. Therefore, if relation *C_* is accessed in `dewey_pos` order and index-NL is used, the result will have descendant *D_* tuples in `dewey_pos` order without duplicates. □

If the ancestor relation is recursive, result ordering and duplicate elimination are necessary.

Example 7: For the structural join between relations *A_* and *B_* from schema H, the ancestor relation is recursive. Therefore, it is possible that *A_* contains two elements a_1 and a_2, the first being an ancestor of the second, or, equivalently, `a1.dewey_pos < a2.dewey_pos < a1.dewey_pos||CHR(255)`. Let's call $B(a_1)$ and $B(a_2)$ the descendant sequences of a_1 and a_2. It holds that $B(a_2) \subset B(a_1)$. During index-NL join, for element a_1 , the inner iteration retrieves $B(a_1)$. When a_2 is accessed, the inner iteration retrieves $B(a_2)$. The result is not ordered by `dewey_pos` and contains duplicates. □

When intermediate structural joins produce duplicates, they must be removed as early as possible: duplicated tuples imply redundant processing effort, since they can cause useless iterations over identical tuples in the following NL joins. Moreover, because of the hierarchical structure of XML, duplicates are likely to be multiplied in ancestor-descendant structural joins. Because of this, once a recursive prominent relation *A_* is identified along a backbone path, if the following prominent relation *B_* in the backbone path is not recursive, we put the SQL statement generated so far, augmented with the `A_-B_` join, in a view (using the `WITH` clause). We apply on that view the `DISTINCT` and `ORDER BY` clauses. Note that the use of `ORDER BY` inside view definitions is not supported by all RDBMS. However, Oracle permits this as long as the view is not updatable, which is the case for temporary views created within a "`WITH` clause" statement. The *target* view uses the above view as its first prominent relation. The algorithm is illustrated below.

`parsePPF` is a recursive function that is called for each PPF of the backbone of an XPath. It defines and adds temporary views in the global `tempViewList` list and defines also the global `targetSelect` which represents the *target_elements* view of the final SQL statement. The `curSelect` is the sub-query currently constructed, the `prevPromRelation` is the prominent relation of the previously parsed PPF and the `curPPF` is the PPF that is being parsed. The function `translateNewPPF` enhances the subquery under construction by adding the prominent relation of the current PPF into the `FROM` clause and the appropriate structural join and `path_id` filtering predicate in the `WHERE` clause of `curSelect`. As mentioned earlier in the section, if the `prevPromRelation` is recursive and the prominent relation of the `curPPF` is not (lines 3-12), then `DISTINCT` and `ORDER BY` clauses are added to the `curSelect` (lines 4-5). Then, unless the `curPPF` is the last one (lines 6-7), a new temporary view called `tmpView` is added to the `tempViewList`. The `parsePPF` function is called for the following PPF (lines 9-11). The cases where the `curPromRelation` is also recursive and where `prevPromRelation` is not recursive are dealt appropriately also (lines 14-23 and 23-28 respectively). The function returns in lines 7, 18 or 25, after setting the target subquery to the currently constructed one.

Algorithm 1 Elimination of redundant sorting

```
parsePPF(String curSelect, String prevPromRelation, PPF *curPPF){
 1  translateNewPPF(curSelect, prevPromRelation, curPPF);
 2  String curPromRelation = curPPF->getProminentRelation();
 3  If (IsRecursive(prevPromRelation) and not IsRecursive(curPromRelation))
 4    Project distinct dewey_pos of curPromRelation in curSelect;
 5    Add order by on dewey_pos of curPromRelation in curSelect;
 6    if (curPPF->getNextPPF()==NULL) // selective PPF
 7      targetSelect = curSelect;
 8    else {
 9      Add temporary view tmpView into tempViewList defined as curSelect;
10      curSelect = "";
11      parsePPF("", tmpView, curPPF->getNextPPF());
12    }
13 }
14 If (IsRecursive(prevPromRelation) and IsRecursive(curPromRelation)){
15   if (curPPF->getNextPPF()==NULL) { // selective PPF
16     Project distinct dewey_pos of curPrRelation in curSelect;
17     Add order by on dewey_pos of curPrRelation in curSelect;
18     targetSelect = curSelect;
19   }
20   Else
21     parsePPF(curSelect, curPromRelation, curPPF->getNextPPF());
22 }
23 Else { // prevPrRelation is not recursive
24   if (curPPF->getNextPPF()==NULL) // selective PPF
25     targetSelect = curSelect;
26   else
27     parsePPF(curSelect, curPromRelation, curPPF->getNextPPF());
28 }
29 }
```

Example 8: Consider the XPath fragment `//A[…]/B[…]/S` over XML data conforming to the XML Schema H of Example 1. The only recursive prominent relation of the query is *A_*. The main query of the SQL equivalent (omitting path filtering and XML reconstruction) doesn't need `DISTINCT` or `ORDER BY`:

```
WITH intermediate_view AS (
      SELECT DISTINCT B_.dewey_pos
      FROM A_,    B_    /* index nested loops for joining A and B
                          with B being the inner table*/
      WHERE  B_.dewey_pos  BETWEEN  A_.dewey_pos
          AND A_.dewey_pos||CHR(255)
      AND EXISTS (…)
      ORDER BY B_.dewey_pos
)
SELECT S_.dewey_pos, S_.tag_name, S_.text_
FROM intermediate_view, S_ /* index nested loops for joining
intermediate_view and S  with  S being the inner  table*/
WHERE    S_.dewey_pos BETWEEN intermediate_view.dewey_pos
          AND intermediate_view.dewey_pos||CHR(255)
AND EXISTS (…)                                              □
```

5.2.3 *Path filtering and pipelining*

In the previous sections we discussed the benefits of using index-NL for structural ancestor-descendant joins, where a b-tree index on the inner relation is required for such a join to be efficient. Moreover, to eliminate redundant sorting, the outer relation must be accessed in `dewey_pos` order. It is nontrivial to satisfy these two requirements while also using root-to-node path filtering. Recall that each prominent relation must be also joined with the *Paths* relation, on which a regular expression filtering predicate is added. The following example illustrates the problem.

Example 9: The SQL translation for query `//A/B//D[F/*/H]` is shown in Figure 4(a). The prominent relation *H_* corresponds to a PPF that is included in a predicate, which is translated into a correlated subquery. The result must be sorted by `dewey_pos`. Our aim is to do this without an `ORDER BY` clause, taking into account that NL joins preserve the ordering of the outer relation, which is *D_* in our example.

```
SELECT …FROM D_, paths p1
WHERE  D_.path_id = p1.id AND REGEXP(p1.path, '/(.+/)?A/B/(.+/)?D$')
AND  EXISTS (
       SELECT NULL FROM H_, paths p2
       WHERE H_.dewey_pos BETWEEN D_.dewey_pos
                    AND D_.dewey_pos||CHR(255)
       AND H_.path_id = p2.id
       AND REGEXP(p2.path, '/(.+/)?A/B/(.+/)?D/F/[^/]+/H$'))
```

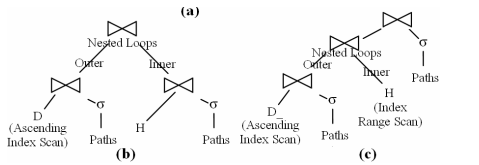


Figure 4. Translation of //A/B//D[F/*/H] and alternative plans

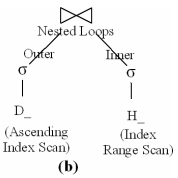
Figure 4 (b) and Figure 4(c) show two plans that both use NL join. In the first plan, even if *D_* is accessed ordered by `dewey_pos`, there is no guarantee that *D_* ⊳⊲ *σ(Paths)* will preserve the ordering unless a NL join is used with the *Paths* being the inner relation. The most important thing is that *H_* ⊳⊲ *σ(Paths)*, which is the inner relation of the top level NL join, is not indexed, making the performance of NL join extremely poor. In the second plan, the result of *D_* ⊳⊲ *σ(Paths)* is sorted only if a NL join is used. Similarly, the final join with the *Paths* relation would preserve the ordering if a NL join is used. □

An alternative option is to avoid joining prominent relations directly with the *Paths* relation. In particular, whenever a path filtering is needed, we can filter the *Paths* relation with a separate SQL statement, create a list of ids of the returned root-to-node paths and store the comma-separated ids as a string. This string is used in the main query filtering the `path_id` of the respective prominent relation using an `IN`-list selection clause, as shown in the following example. This way, the SQL translations produce simpler plans and, moreover, the *Paths* relation is accessed fewer times. We made experiments, not presented in this paper due to lack of space, which show performance benefits when this alternative is applied.

Example 10: Figure 5(a) shows the SQL translation of query `//A/B//D[F/*/H]` using `IN`-list selection clauses for filtering the path ids. The execution plan, shown in Figure 5(b), uses simple selection operations for filtering the path ids of relations *D_* and *H_*, without affecting ordering: *D_* is accessed in `dewey_pos` and the selection on the `path_id` column will preserve this ordering. For each outer iteration, an index range lookup occurs on the index of *F_*, producing a sequence of tuples sorted by `dewey_pos`. This sequence is filtered by `path_id` before it is joined with the current *D_* tuple. □

```
SELECT …FROM D_
WHERE  D_.path_id IN (4, 62, 120)
AND  EXISTS (
       SELECT NULL FROM H_
       WHERE H_.dewey_pos
       BETWEEN D_.dewey_pos
       AND D_.dewey_pos||CHR(255)
       AND  H_.path_id IN (17, 75, 137))
```

(a)



(b)

Figure 5. New translation of //A/B//D[F/*/H] and its plan

5.3 Organizing relations as indices

As discussed in previous sections, NL joins in combination with dewey position indices are optimal for structural joins as well as for preserving the document order of the results without the need for keeping intermediate relations. However, index range scans introduce an inefficiency: When the query involves columns other than `dewey_pos`, additional disk reads are needed to access those columns from the actual relation. In particular, the `path_id` column of every prominent relation is accessed. Also, almost all columns of the selective prominent relation as well as of the descendant relations that are unioned in the *descendants* view must be projected. To improve efficiency we can use a concatenated (composite) index on the `dewey_pos` and `path_id` columns. The problem remains for the projected columns of the selective prominent relation and for the relations storing descendants of the target elements (in case of XML reconstruction).

A better alternative is to use indexed file organization for the mapping relation. In this file organization method, the data entries of the index, stored in the leaf block of the b-tree, include the actual data records. This way one level of indirection is omitted since both relation and index coexist in a single structure. Modern RDBMS support such file organization, e.g. IOT in Oracle, tables with clustered index in SQL Server, etc. With this organization, presence of non-key columns in the projection or in the selection part of an SQL query will not cause additional block accesses. Similarly, because rows are physically stored in key order, range access by the primary key involves the minimum number of block accesses. Experiments shown in Section 6.3 justify the organization of relations as indices, especially when XML reconstruction is required; the performance benefit is remarkable.

5.4 Accelerating XML reconstruction

The above observations and developed techniques can also help in developing new efficient XML reconstruction methods. We first improve on the basic SQL translation for XML reconstruction. We then propose an alternative solution based on maintaining redundant data, which "splits the difference" between schema-oblivious and schema-aware XML-to-relational mapping and significantly accelerates XML reconstruction without sacrificing the efficiency of XPath filtering.

5.4.1 Using Index NL join and reducing sorting cost

The "direct" SQL translation for XML reconstruction presented in Section 4 suffers from a severe disadvantage. In order to return the result relation sorted by `dewey_pos` of the descendant elements, the basic query (that joins the *target_elements* and *descendants* views) needs an `ORDER BY` clause, as shown in Figure 3. There is a better alternative: we can push the `ORDER BY` clause inside the *descendants* view, as shown in Figure 6. SQL of Example 4 using basic XML reconstruction. We call this XML reconstruction method *basic XML reconstruction*.

Using hints, we force the use of index-NL for the outer structural join of *target_elements* and *descendants*. The *target_elements* subquery is executed as discussed in Section 5.2 and its result tuples are pipelined in document order. Whenever a target element tuple is pipelined, it is joined with its descendant tuples from the *descendants* view. These descendants are evaluated as the union of the results of many index range scans: as many as the number of the descendant relations that participate in the *descendants* view. Descendant tuples from each such relation are produced in document order, due to index range scanning, but overall, the result of the union will not be sorted in document order. Hence, an `ORDER BY` clause inside the *descendants* view is needed, but not one in the main query. Since complexity of sorting is *O(nlgn)*, the more tuples are produced by the *target_elements* view, the bigger the benefit of pushing the `ORDER BY` clause into the *descendants*

view. Moreover, since smaller sets of tuples are sorted, it is more likely that the sorting will occur in main memory. We call this technique *basic XML reconstruction*. Note that it has similarities with [18] but uses a single SQL statement. In [18] for each target element multiple queries are issued to the RDBMS for retrieving descendants from all possible relations.

```
WITH target_elements AS (
        SELECT T_.text, T_.dewey_pos, T_.tag_name,
        FROM T_ WHERE T_.path_id IN (27) ),
descendants AS ( /* without pre-calculation, keeping indexes of E_, F_, G_ and H_ exploitable*/
            SELECT dewey_pos, tag_name, text_, NULL AS attributes_
            FROM S_ UNION ALL
            SELECT dewey_pos, tag_name, NULL AS text_,'j='''||j_||''' k='''||k_||''' ' AS attributes_
            FROM B_ UNION ALL
            SELECT dewey_pos, tag_name, NULL AS text_, NULL AS attributes_
            FROM C_ UNION ALL
            SELECT dewey_pos, tag_name, text_,'j='''||j_||''' k='''||k_||''' ' AS attributes_
            FROM D_
            ORDER BY 1 )
SELECT target_elements.*, descendants.*
FROM target_elements LEFT OUTER JOIN descendants  /* index nested loops for joining
target_elements and descendants with the later being the inner relation*/
WHERE descendants.dewey_pos BETWEEN target_elements.dewey_pos
            AND target_elements.dewey_pos||CHR(255);
```

Figure 6. SQL of Example 4 using basic XML reconstruction

5.4.2 A global relation (for XML reconstruction)

The *basic XML reconstruction* method described in the previous section has some shortcomings. For each target element, there are as many b-tree index lookups as the number of descendant relations, and sorting is still required. The cost for a b-tree index range scan is the height of the tree plus the number of leaf blocks covering the range condition, if we assume for simplicity that only I/O accesses count. Consider an XPath query where 5 possible descendant relations are unioned in the *descendants* view. For the 116 MB XMark dataset, the average height of each b-tree index is 2. Therefore, for each target element, 10 blocks are accessed on average. If the number of target elements is n, the upper bound total cost for b-tree I/O is $10n$. Many of these index blocks are gradually cached as the cache warms up. Nevertheless, many key comparisons still take place.

We propose a completely different technique, *hybrid XML reconstruction*, that improves significantly the performance of XPath-to-SQL translations that also perform XML reconstruction. Sorting is completely avoided and index accesses to retrieve descendants are eliminated. A global materialized view, called *global_table*, is maintained, that unions all the mapping relations, with the `dewey_pos` column as primary key. We eliminate the schema differences among the mapping relations using the same method we used for defining the *descendants* union view (see Section 4): for each mapping relation we accumulate all attribute columns under a single one called `attributes_` and introduce a NULL-valued column named `text_` for those relations that don't have a column for storing text nodes.

To reconstruct the XML result of an XPath query, the descendents of the target elements are retrieved exclusively from the *global_table* relation. This way we preserve optimal performance for retrieving target elements, provided by PPF-based processing, while accelerating the XML reconstruction process. The first reason why XML reconstruction occurs much faster is because, for each target element, a single lookup takes place on the b-tree of the *global_table*, instead of having to lookup on multiple b-tree indices. Moreover, there is no need for sorting, since for each target element its descendants are retrieved naturally sorted due to index range scanning. Using *hybrid XML reconstruction*, the SQL translation of Example 4 is shown in Figure 7

Experiments presented in Section 6.4 confirm that *hybrid* XML reconstruction is more efficient than *basic* XML reconstruction, and that both methods give better overall performance than using exclusively the global table for evaluating and constructing the XML result of an XPath query (which is what happens with schema-oblivious XML-to-relational mapping).

```
WITH target_elements AS (
            SELECT T_.text, T_.dewey_pos, T_.tag_name,
            FROM T_ WHERE T_.path_id IN (27) )
SELECT target_elements.*, global_table.*
FROM target_elements LEFT OUTER JOIN global_table
 /* index nested loops for joining target_elements and global_table
with the global_table view being the inner relation*/
WHERE global_table.dewey_pos BETWEEN target_elements.dewey_pos
            AND target_elements.dewey_pos||CHR(255);
```

Figure 7. SQL of Example 4 using hybrid XML reconstruction

6. EXPERIMENTAL EVALUATION

The goals of the experimental evaluation are twofold:

- to evaluate the performance of each proposed optimization and measure their impact on XPath processing, and
- to compare a system using our complete set of techniques, implemented on top of a commercial RDBMS, against a wide variety of existing systems, both relational-based and native, commercial and research prototypes.

Table 1. Queries for the XMark datasets

Q1 :/site/regions
Q2: /site/closed_auctions
Q3: /site/regions/europe/item/mailbox/mail/text/keyword
Q4: /site/closed_auctions/closed_auction/annotation/description /parlist/listitem
Q5: /site/closed_auctions/closed_auction/annotation/description /parlist/listitem/parlist/listitem/*//keyword
Q6: /site/regions/*/item
Q7: /descendant-or-self::listitem/descendant-or-self::keyword
Q8: /site/regions/*/item//keyword
Q9: /site/people/person[address and (phone or homepage)]
Q10: /site/regions/*/item[(@id='item0') and ((mailbox/mail/from) or (mailbox/mail/to))]//keyword
Q11: /site/regions/*/item[mailbox/mail/from]/mailbox/mail
Q12. /site/regions/*/item[mailbox/mail[to='Marin Samtaney mailto:Samtaney@utexas.edu']/from]/mailbox/mail
Q13: /site/people/person[profile/education='Graduate School' and address/country='United States']
Q14: /site/people/person[profile/education='Graduate School' and address/country='United States']/name
Q15: /site/closed_auctions/closed_auction/annotation[happiness='1']//parlist
Q16: /site/regions/*/item[mailbox/mail/from]//keyword
Q17: //keyword/ancestor::listitem/parent::parlist
Q18: //keyword/ancestor::listitem
Q19: //keyword/ancestor-or-self::mail
Q20: /site/regions/*/item[@id='item0']/following::item
Q21: /site/regions/*/item[parent::namerica or parent::samerica]
Q22: /site/regions/*/item[@id='item0']/following-sibling::item

We used three XML datasets, the 116 MB and 580 MB documents from the XMark [20] benchmark and the 700MB Protein Sequence DB [10].The 116 MB XMark document has 1,6 million elements and 514 different root-to-node paths whereas the 580 MB XMark document has 8 million elements and 514 different root-to-node paths. The Protein DB has 21.3 million elements and 85 different root-to-node paths. The XMark schema is recursive while the Protein schema is not.

The query sets for the XMark datasets and for the Protein Sequence DB are shown in Table 1 and Table 2. These queries, used in all experiments (unless otherwise mentioned), cover a wide range of XPath, including all XPath axes, wildcards, and nested predicates. The reported values are the average of four repeats. All experiments were performed on an Intel Pentium IV machine at 3.2 GHz with 2GB of RAM running Windows XP, with the

exception of the experiments on the Natix system, which were done on the same PC running SUSE 10. PPFS+ is implemented in C++ as a COM object on top of Oracle 10.2g.

Table 2. Queries for the Protein System DB

```
Q1: /ProteinDatabase
        /ProteinEntry[reference/accinfo/accession='AE0077']
Q2: /ProteinDatabase/ProteinEntry//accinfo/xrefs
Q3: /ProteinDatabase/ProteinEntry[reference/refinfo/authors/author
        ='Massung, R.F.']
Q4: /ProteinDatabase/ProteinEntry[organism/variety=
        'strain Marburg']/reference/accinfo/xrefs
Q5: /ProteinDatabase/ProteinEntry[reference//year='1988']/organism
Q6: /ProteinDatabase/ProteinEntry[organism/source=
        'Anabaena sp.']/reference/accinfo/xrefs
Q7: /ProteinDatabase/ProteinEntry[reference/accinfo/note
        or organism/note]
Q8: /ProteinDatabase/ProteinEntry[reference/accinfo/note]
Q9: /ProteinDatabase/ProteinEntry[reference/refinfo/year='1988']
        /reference/accinfo[status='preliminary']/xrefs
```

6.1 Picking the right join method

We tested six simple ancestor-descendant joins on the 116 MB XMark document, first forcing the optimizer to use Index-NL join and then Sort Merge. The experiments were performed with cold cache. Note that PPFS+ can handle these queries simply using root-to-node path filtering, as described in Section 3.2. We force PPFS+ to translate the queries using structural join for the purposes of the experiment. The choice of the optimizer when no hints are used is always Sort Merge. Execution times as well as the cardinalities of the join relations and the results of the joins are shown in Figure 8. The experiments confirm the dominance of Index NL for dewey-based structural joins over Sort Merge.

structural joins	cardinality			Index NL	Sort Merge
	ancestor	descendant	result		
item//mail	21750	20946	20946	0.23	95.41
annotation// keyword	21750	69969	27481	0.18	84.73
mail//from	20946	20946	20946	0.2	95.8
person//education	25500	6538	6538	0.27	23.64
item//text	21750	105114	61823	0.24	372.45

Figure 8. Execution times (sec) of different join algorithms for structural join

6.2 Reducing Sorting Impact

To verify the benefit from eliminating redundant sorting as well as duplicate removal, we experimented with five XPath queries. Queries, Q14, Q18, Q19 from Table 1, and two additional queries:

`QA:/site/people/person[address]//interest`
`QB:/site//parlist[listitem]//text[text()]//keyword`

The queries returned only target elements without XML reconstruction. We tested two versions of SQL translations of these queries: the original translation used in the PPFS system as outlined in Section 3.2, and the translation used in PPFS+, which is enhanced by Algorithm 1 discussed in Section 5.2.2 that eliminates redundant `ORDER BY` and `DISTINCT`. The queries are run on the 116 MB XMark dataset with cold cache.

The results, shown in Figure 9, confirm the benefits of the PPFS+ technique for eliminating sorting. The effects are more pronounced for the larger data sets.

Note that Q14, Q18, Q19 and QA satisfy the condition of Theorem 1. QB was constructed to not satisfy the condition, to evaluate the benefit of moving the `ORDER BY` to an intermediate view as in Example 8.

In particular, element `parlist` is recursive. Note that, even though the XMark schema shows element type `text` as recursive, it is not really: text elements cannot be nested in other text elements, and are not nested in either the 116 MB or the 580 MB versions of XMark. Since `parlist` is recursive, and considering `text` as non-recursive, the query fragment `/site//parlist[listitem]//text[text()]` is evaluated in a separate temporary view, as in Example 8, and the main query does not have any sorting or duplicate elimination.

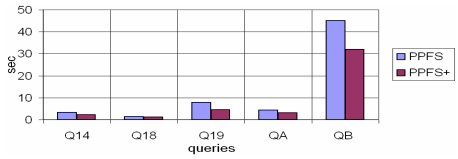
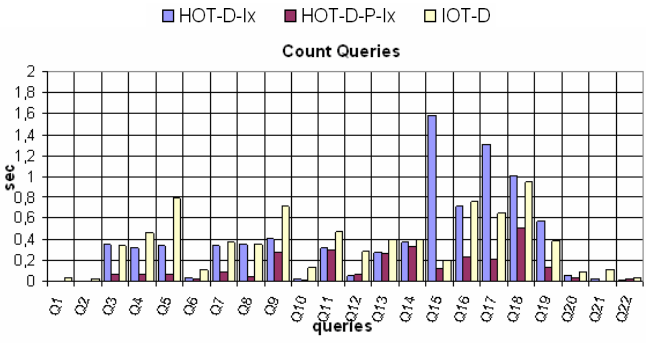


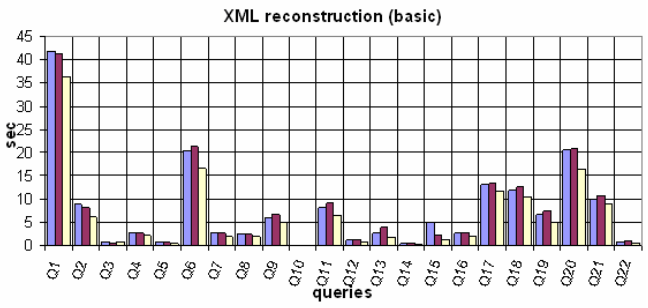
Figure 9. Removing sorting and duplicate elimination

6.3 Picking the right physical organization

We compare the execution times of the 22 XPath queries against the 116 MB XMark document for three different physical organizations of the relational tables: standard heap organized tables with indices only on the `dewey_pos` columns (HOT-D-Ix), heap organized tables with concatenated indices on the `dewey_pos` and `path_id` columns (HOT-D-P-Ix) and index-organized tables with `dewey_pos` being the index key (IOT-D). For each case, we measure the execution times for the `COUNT` version of the queries, which only return the number of target elements, and for queries with full XML reconstruction. The basic XML reconstruction method is used.



(a) COUNT Queries



(b) Queries with XML reconstruction

Figure 10. Comparing different physical organizations

The results are shown in Figure 10. For COUNT queries, the HOT-D-P-Ix implementation gives the best performance, as expected. When XML reconstruction is required, the IOT-D implementation dominates, as a much larger number of tuples are retrieved from the database -- for our query set, in most queries each target element has thousands of descendants. Therefore, if the index is

kept separately from the heap organized relation, even if the index is clustered, an extra block is accessed for each result tuple.

6.4 Comparing XML reconstruction techniques

We compare three XML-reconstruction enabled XPath-to-SQL translation techniques: the *basic* and the *hybrid* methods, presented in Sections 5.4.1 and 5.4.2, and a third method we call *single-relation* that uses exclusively the global table both for evaluating the target elements and for retrieving their descendants. The *single-relation* method resembles [11] (schema-oblivious). We test all queries in the 116 MB XMark dataset in warm and cold cache.

As shown in [9], the *single-relation* method is not efficient for identifying the target elements. XML reconstruction is expected to be more efficient when the global table is used, as in the *hybrid* and *single-relation* methods. Since the *basic* and the *hybrid* methods share the same technique for retrieving target elements, their only difference is the way descendants are retrieved .

Based on a simple cost model for the two methods, we expect the hybrid method to outperform the other two on all queries. For the basic method, the cost for retrieving the descendants of a single target element is approximately given by the following formula:

$$C_{basic} = \sum_{i=1}^{k}(h_i + n_i) + C_{sorting} = \sum_{i=1}^{k} h_i + n + C_{sorting} \qquad (1)$$

where k is the number of descendant relations, h_i is the height of the b-tree of index-organized relation i, n_i is the number of leaf blocks accessed during the range scanning on relation i, n is the total number of leaf blocks accessed from all descendant relations for retrieving the descendants of a target element and $C_{sorting}$ is the cost of sorting the descendants of a target element.

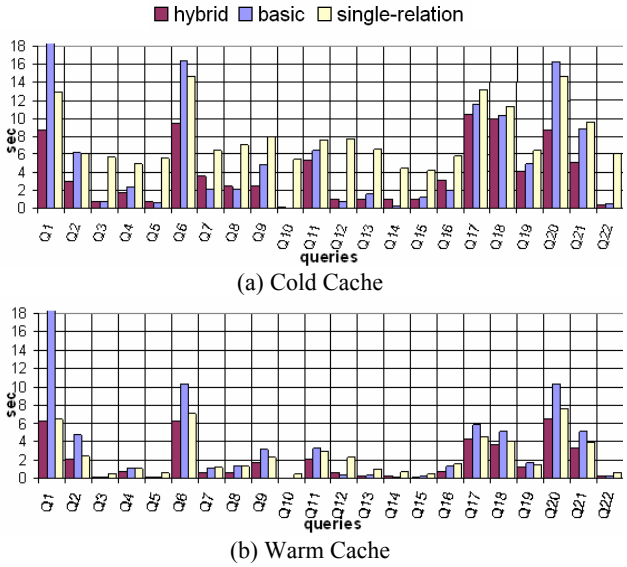
On the other hand, the cost for retrieving the descendants of the same target element from the global materialized view is given by:

$$C_{hybrid} = h_{gl} + n' \qquad (2)$$

where h_{gl} is the height of the b-tree of the index organized global view and n′ is the number of leaf blocks accessed during the index range scan of the global view. Given that $n \geq n'$ [2], C_{basic} is greater than C_{hybrid}, for all queries.

The experimental results when queries are executed in warm cache, shown in Figure 11(b), validate this conclusion: the *hybrid* method outperforms both other methods in all queries by a median of 63%.

Experimental results in cold cache, shown in Figure 11(a), show that the *basic* method is faster than the *hybrid* in a few (8 out of 22) queries. This is due to the different caching behavior of the two methods. During the execution of the NL join of the main query, the descendant relations (for the *basic* method), or the global view (for the *hybrid* method), are gradually cached, i.e., internal and leaf blocks of the indices are moved into memory. The blocks cached from descendant relations are more relevant (i.e., visited more often) than the blocks of the much larger global table: if an XPath query selects 10% of an XML document that are all B elements, up to 10% of the leaves of the global table index need to be visited, but a much larger percentage of the leaves of the much smaller *B_* table. This reduces I/O cost on queries using the *basic* reconstruction method on cold cache and explains why it even outperforms *hybrid* on a few queries. For cold cache, over all queries the median performance difference between *hybrid* and *basic* is 17%, and between *hybrid* and *single relation* it is 189%.



(a) Cold Cache

(b) Warm Cache

Figure 11. Comparing XML reconstruction techniques

6.5 Overall performance comparisons

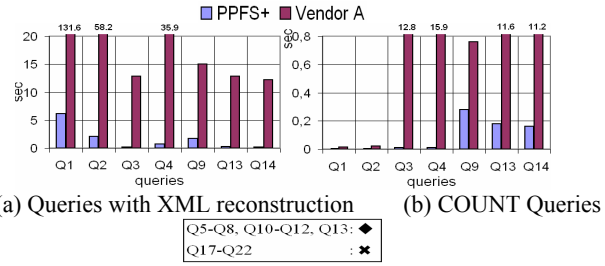
We conclude our experimental analysis with a comparison between PPFS+ and other systems for XPath processing. As described already, PPFS+ is based on PPFS [9], uses IOT tables as physical organization for the shredded relations and implements structural joins using Index NL joins (and outer joins). It produces optimized SQL translations that omit redundant sorting and duplicate elimination operations, filter root-to-node paths using `IN`-list selection predicates and implements *hybrid* XML reconstruction. PPFS+ uses Oracle 10.2g as the relational backend with 1.2 GB for buffer cache. We compare PPFS+ with:

- The native XML Management System Natix 2.1.0 [2]. We used the default configuration parameters except for the size of the buffer cache which we increased to 1.2 GB.
- The native XML engine of major relational vendor A. The engine supports schema-aware XML shredding as well as storage of XML documents as CLOBs. The second option has very poor query performance, so the first option was used. Experiments were run with 1.2 GB of buffer cache. The XML schema-to-relational mapping was performed by the engine.
- The native XML engine of major relational vendor B. The engine stores XML documents shredded into a table. Order is encoded using a variant of dewey order. Each tuple corresponding to an element stores the element's position, its root-to-node path, its tag name and its text value (if any). Buffer cache size is set automatically by the engine. For our experiments we additionally defined a path index, supported by the engine, to improve performance.
- The MonetDB/XQuery (Version 0.12). This is an XQuery implementation built on the foundation of the main memory DBMS MonetDB [8].

The experiments don't permit direct comparisons between the relational-based systems, as they use vastly different relational

[2] In an index range scan one more tree leaf block than necessary may be read. In the *basic* method we read from multiple tree indices, and hence read more such blocks.

engines. The efficiency of the underlying engine affects significantly the performance of XML processing, if it is built directly on top of the existing relational infrastructure (as with PPFS+ and, to a lesser extent, MonetDB/XQuery): a faster relational engine directly improves XML processing performance. Nevertheless, a number of useful conclusions can be drawn.



(a) Queries with XML reconstruction (b) COUNT Queries

(c) Queries Vendor A engine cannot handle

Figure 12. Comparison of PPFS+ and Vendor A engine

The Vendor A engine only succeeded in running a small number of queries of the XMark 116MB dataset. We were not able to load the other datasets. For the queries that ran successfully, the comparison with PPFS+ is shown in Figure 12(a) and (b). For the queries that didn't run, Figure 12(c) explains: we use '◆' if the execution didn't finish within a reasonable time, and '▲' if the query is not supported. Note that queries with '//' or '*' (such as Q6, Q7, Q8, etc) are evaluated by loading the entire document in memory and evaluating the queries on the DOM tree directly.

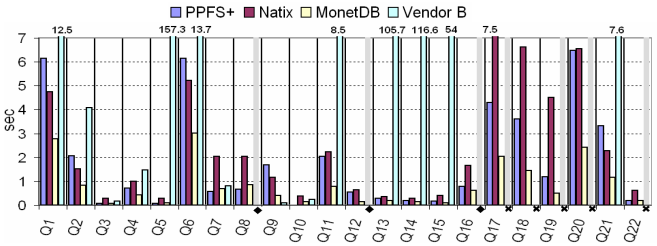


Figure 13. Queries with XML reconstruction-XMark 116MB

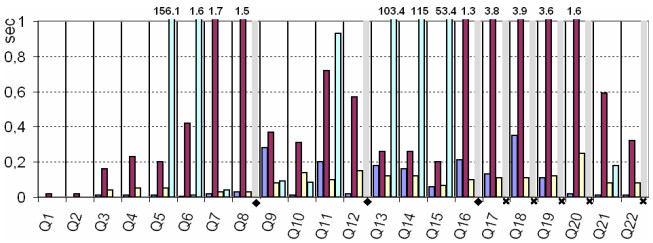


Figure 14. COUNT queries-XMark 116 MB

For the other systems (PPFS+, MonetDB/XQuery, Natix, and Vendor B engine), Figure 13 and Figure 14 show the execution times in warm cache of two versions of the 22 queries of Table 1 for the 116 MB XMark dataset. The same experiments are repeated for the 580 MB XMark dataset, as shown in Figure 15 and Figure 16. Some queries could not be run by the Vendor B engine, and are marked in the graphs below the query axis, similarly to Figure 12(c). Figure 17 shows the results on the 700 MB XML Protein Sequence Database. The dataset could not be loaded on the Vendor B system and on MonetDB/XQuery.

As shown in the graphs, PPFS+ outperforms other systems in the vast majority of queries, with COUNT or with full XML reconstruction, with the exception of MonetDB/XQuery.

In comparison to Natix, when XML reconstruction is required, PPFS+ is faster than Natix in 36 queries from the three datasets. The median performance difference is 130%. Natix is faster in 17 queries, the median difference being 36%. All 53 COUNT queries run faster in PPFS+, the median performance difference being 2000%. We conclude that XPath filtering techniques over relational engines can significantly outperform native XML processing techniques. XML reconstruction imposes a hardship on relational systems, but with our hybrid reconstruction technique overall performance is still superior in most cases.

We can also conclude that the XML engines of the commercial relational RDBMSs used in the comparison have a long way to go to offer satisfactory performance on relatively complex XPath queries. PPFS+ outperformed Vendor A and Vendor B engines on all queries (by a median of 1200%). In particular, note that, even though Vendor B engine stores the root-to-node paths for all elements in order to reduce the number of structural joins needed, it seemingly cannot avoid structural joins for handling wildcards. PPFS+ (following PPFS [9]) handles both '//' and '*' using regular expression filtering.

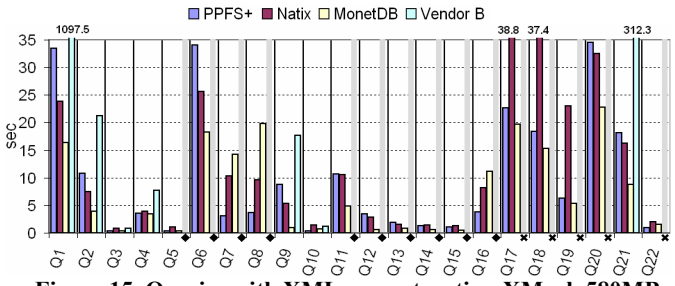


Figure 15. Queries with XML reconstruction-XMark 580MB

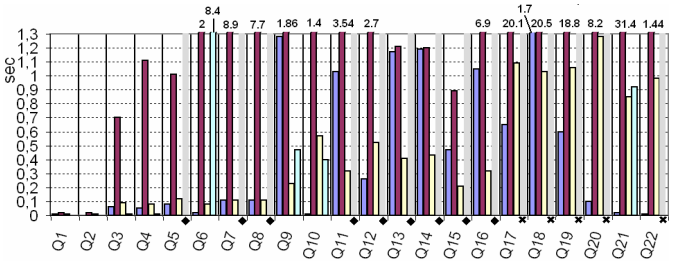
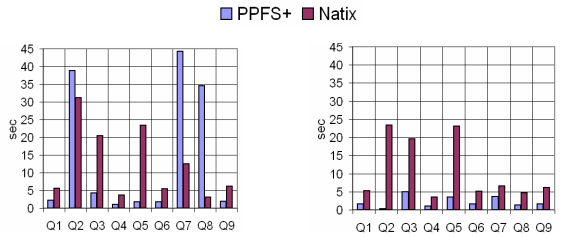


Figure 16. COUNT queries- XMark 580 MB

Regarding MonetDB/XQuery, when XML reconstruction is required, it outperforms PPFS+ in 31 out of 44 queries – the median performance difference is 117%. PPFS+ outperforms MonetDB/XQuery in 13 queries, showing a median improvement of 66%. In COUNT queries, PPFS+ is faster than MonetDB/XQuery in 23 queries (300% median difference) and slower in 15 (120% median difference). The underlying relational engine used by MonetDB/XQuery, MonetDB, is an in-memory database system that uses a special storage model and a CPU-tuned vectorized query execution architecture [7]. Previous studies [7] have shown a significant performance advantage of MonetDB over Oracle, which may explain the differences we observe. On the other hand, the existence of many queries where PPFS+ outperforms MonetDB/XQuery shows that easy to implement

techniques, even implemented at the application level, can significantly improve XPath performance.

Implementation of the novel techniques of MonetDB/XQuery directly *on top of* a traditional relational engine is not possible, as some techniques, e.g., staircase joins, need to be implemented inside the engine. We are planning to implement PPFS+ on top of MonetDB, in order to allow for a more direct comparison.



(a) Queries with XML reconstruction (b) COUNT Queries

Figure 17. Protein Sequence Database

7. CONCLUSIONS

Despite significant progress, processing of XPath queries still has room for improvement. A strategy that has yielded impressive dividends for relational data processing is devising storage and execution strategies appropriate for the data model characteristics. For XML processing using relational engines, using the right join algorithms, index structures and storage options hasn't received enough attention. As our results show, large performance improvements can be the result of more careful consideration of the impact of such choices. Moreover, we propose a novel and faster XML reconstruction technique.

Our techniques are implemented on top of a relational engine without any need for engine modification. We plan to investigate how to make optimizers natively aware of the issues we identified. We are also working on similar techniques for XQuery processing

8. REFERENCES

[1] J.Clark, S.DeRose: "XML Path Language (XPath) Version 1.0". *W3C Recommendation 16 November 1999.*

[2] T. Fiebig et al.: "Anatomy of a native XML base management system". *VLDB J. 11(4) .2002*

[3] M. Krishnaprasad, et al.: "Query Rewrite for XML in Oracle XML DB". *VLDB 2004: 1122-1133*

[4] S. Pal, I.Cseri, et al.: "XQuery Implementation in a Relational Database System". *VLDB 2005:1175-1186*.

[5] E. Cerami**:** *XML for Bioinformatics*, Springer-Verlag, 2005

[6] A.C. Davenhall, et al.: "VOTable: an XML data format for virtual astronomy". *XML Europe 2003*

[7] P. A. Boncz, M.Zukowski, et al.: "MonetDB/X100: Hyper-Pipelining Query Execution". *CIDR 2005: 225-237*

[8] P. A. Boncz, T. Grust, et al.: "MonetDB/XQuery: a fast XQuery processor powered by a relational engine". *SIGMOD Conference 2006: 479-490*

[9] H. Georgiadis, V. Vassalos: "Improving the Efficiency of XPath Execution on Relational Systems". *EDBT 2006.*

[10] XML Data Repository, at www.cs.washington.edu/research/

[11] T. Grust, M. Keulen et al.: "Accelerating XPath evaluation in any RDBMS". *ACM Trans. Database Syst. Vol 29*, *2004*

[12] J, Shanmugasundaram, et al.: Efficiently Publishing Relational Data as XML Documents. *VLDB 2000: 65-76*

[13] M. F. Fernandez, et al: Efficient Evaluation of XML Middle-ware Queries. *SIGMOD Conference 2001*

[14] S. Amer-Yahia, Y. Kotidis, et al: Teaching Relational Optimizers About XML Processing. *XSym 2004: 158-172*

[15] S. Chaudhuri, et al.: On Relational Support for XML Publishing: Beyond Sorting and Tagging. *SIGMOD 2003*

[16] R. Krishnamurthy, et al.: "Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation". *Proc. of the 20th ICDE, 2004*

[17] W. Fan, J. Xu Yu, et al.: "Query Translation from XPath to SQL in the Presence of Recursive DTDs". *VLDB 2005*

[18] A. Chebotko, D. Liu, et al: "Reconstructing XML Subtrees from Relational Storage of XML Documents". *XSDM'05*

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

[20] A. Schmidt, F. Waas, M. Kersten, et al.: "XMark: A Benchmark for XML Data Management". *VLDB 2002*

[21] I. Manolescu, Y. Papakonstantinou: "XQuery Midflight: Emerging Database-Oriented Paradigms and a Classification of Research Advances", tutorial, *ICDE 2005*.

[22] S. Paparizos, et al.: "TIMBER: A Native System for Querying XML". *SIGMOD Conference 2003*.

[23] S.Amer Yahia, F.Du, J.Freire: "A Comprehensive Solution to the XML-to-Relational Mapping Problem." *WIDM'04.*

[24] K. Beyer, et al.: "System RX: One Part Relational, One Part XML". *SIGMOD Conference 2005: 347-358*

[25] W. Wang et al.: "Efficient Processing of XML Path Queries Using the Disk-Based F&B Index." *VLDB 2005.*

[26] S.-Y. Chien, Z. Vagena, et al.: "Efficient Structural Joins on Indexed XML Documents." *VLDB 2002: 263-274*

[27] T. Grust, M. van Keulen, J. Teubner: "Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps", *VLDB 2003*

[28] Surajit Chaudhuri, Zhiyuan Chen, Kyuseok Shim, Yuqing Wu: "Storing XML (with XSD) in SQL Databases: Interplay of Logical and Physical Designs." *IEEE TKDE 17(12)*

[29] P. Bohannon, J. Freire, et al.: Bridging the XML Relational Divide with LegoDB. *ICDE 2003: 759-760*

[30] Yi Chen, S. B. Davidson, Y. Zheng: "BLAS: An Efficient XPath Processing System". *SIGMOD Conference 2004*

[31] A. Balmin, Y. Papakonstantinou: "Storing and Querying XML Data using Denormalized Relational Databases", *VLDB Journal 14(1), 2001.*