Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect

Torsten Grust Jan Rittinger Jens Teubner
Technische Universität München, Munich, Germany
{torsten.grust,jan.rittinger,jens.teubner}@in.tum.de

ABSTRACT

To compensate for the inherent impedance mismatch between the relational data model (tables of tuples) and XML (ordered, unranked trees), *tree join* algorithms have become the prevalent means to process XML data in relational databases, most notably the TwigStack [6], structural join [1], and staircase join [13] algorithms. However, the addition of these algorithms to existing systems depends on a significant invasion of the underlying database kernel, an option intolerable for most database vendors.

Here, we demonstrate that we can achieve comparable XPath performance without touching the heart of the system. We carefully exploit existing database functionality and accelerate XPath navigation by purely relational means: *partitioned B-trees* bring access costs to secondary storage to a minimum, while *aggregation functions* avoid an expensive computation and removal of duplicate result nodes to comply with the XPath semantics. Experiments carried out on IBM DB2 confirm that our approach can turn *off-the-shelf* database systems into efficient XPath processors.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*; H.3.4 [**Information Systems**]: Information Storage and Retrieval—*Systems and Software*

General Terms

Performance, Experimentation, Languages

Keywords

XPath, SQL, Partitioned B-Tree, Relational Databases

1. INTRODUCTION

The use of suitable *tree encodings* can turn relational database back-ends into highly efficient XPath processors, an XML storage approach that has since become widely accepted in research and industry. With efficient RDBMS implementations in the back, relational XQuery implementations excel with a scalability to multi-gigabyte XML instances (*e.g.*, [5, 17]).

To compensate for the lack of tree-awareness in relational systems, various algorithms have been established to process encoded XML tree instances, most notably the *TwigStack* [6], *structural join* [1], *multi-predicate merge join* [24], and *staircase join* [13] algorithms. But although care has been taken to limit the required change impact to a single database operator, all of these proposals depend on modifications to the internals of the underlying RDBMS kernel, a requirement that may be unacceptable in actual systems (*e.g.*, if the RDBMS does not allow kernel modifications).

In this paper, we demonstrate that *existing* and commonly available database techniques can make up for a large part of this limitation. We deliberately use an off-the-shelf RDBMS implementation for the task of evaluating XPath location steps (IBM DB2 for that matter). A closer look at the indexing and execution techniques that this system uses reveals interesting insights into why relational systems perform so well at XPath. Three common techniques from the relational domain proved particularly effective for XML tree processing:

(i) The use of *partitioned B-trees*, a technique recently described by Graefe [9] significantly speeds up the evaluation of non-recursive XPath axes (`child`, `parent`) and XPath node tests (name and/or kind tests). The same idea can implement TwigStack's node streams [6] or simulate schema-based encoding techniques [20, 22] in a seamless fashion.

(ii) *Aggregation functions* provide a purely relational implementation of the *pruning* idea, a key aspect in the aforementioned tree join algorithms.

(iii) By purely relational means, *RDBMS query optimizers* embrace rewriting techniques for queries over tree-structured data that required tailor-made XML support in earlier work.

We will motivate all these techniques with experiments carried out in IBM DB2 (using its SQL functionalities only) and closely look at the query plans employed by this system. Among the tree encodings proposed in the literature, we chose the *pre/size/level range encoding* to perform these experiments. The effects we describe, however, apply equally well to other node-based numbering schemes, including those based on *pre/post* [10] or Dewey numbers [23].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

We will proceed as follows. Section 2 briefly reviews relational XML tree encodings, the *range encoding* (*pre*/*size*/*level*) in particular. Based on this encoding, we demonstrate the use of *partitioned B-trees* to accelerate the evaluation of XPath's `child` and `parent` axes as well as XPath node tests (Section 3). In Section 4, we realize staircase join's pruning idea [13] on an SQL-only system, before we exploit further RDBMS facilities for efficient XPath processing in Section 5. We compare our results with related work in Section 6 and wrap up in Section 7.

2. RELATIONAL TREE ENCODINGS

Any relational XQuery implementation that strives for compliance with the W3C specifications [4] is bound to represent XML document trees in a *schema-oblivious* fashion. Acceptable support for the recursion inherent to XML data is provided by Dewey-based encodings (*e.g.*, [18, 23]) and encodings that use pre- and postorder ranks to describe node relationships in terms of region conditions (*e.g.*, [5, 7, 10, 15, 24]).

2.1 Zooming in: Range Encodings

We will focus on a representative of the latter category, where the idea is to record for each node v a *range* that hosts all nodes v' in the subtree below v. More specifically, we enumerate all nodes according to the XML document order (v's preorder rank $pre(v)$). Further, for each node v, we maintain $size(v)$ as v's number of descendant nodes and $level(v)$, v's distance from the tree's root, which completes the structural component of our encoding. Two properties $kind(v) \in \{\mathsf{elem}, \mathsf{text}, \mathsf{comment}, \dots\}$ and $prop(v)$ (holding v's tag name or textual content for text/comment nodes) account for v's semantical content. Figure 1(b) illustrates this encoding for the XML fragment (see Figure 1(a) for the corresponding tree):

<a>c<d><e/><f/></d><g><h>i<j/></h></g> .

The use of this *range encoding* variant is a reasonable choice: it serves as the backbone of the open-source XQuery implementation MonetDB/XQuery[1] and, hence, has proven its applicability for large-scale XML processing. Note that the correlation

$$pre(v) - post(v) = level(v) - size(v)$$

for any tree node v (with $post(v)$ denoting v's postorder rank) makes the encoding equivalent to other region-based tree encodings described in the past.

2.1.1 Query Regions for XPath

Range encoding allows the characterization of XPath navigation axes in a way that perfectly suits the relational processing model. The recursive axis `descendant`, *e.g.*, turns into a simple range condition over preorder ranks:

$$v \in c/\texttt{descendant} \Leftrightarrow \\ pre(c) < pre(v) \leq pre(c) + size(c) \ . \qquad (\textsc{Desc})$$

This range query for the nodes v in the `descendant` region of context node c is efficiently supported in commodity systems, *e.g.*, in terms of a B-tree index on column *pre*. Similar characterizations arise for other XPath axes as well, which we illustrated in Figure 1(c), assuming a singleton context node sequence containing node d of the example document in Figure 1(a).

[1] `http://www.monetdb-xquery.org/`

3. PARTITIONED B-TREES FOR XPATH

While the range encoding efficiently describes the semantics of recursive XPath axes, ideally this should not negatively affect the efficient evaluation of steps along any of the non-recursive axes. The two axes `child` and `parent` seem particularly crucial here. On range-encoded data, we characterize axis `child` based on Condition Desc and an additional predicate on column *level*:

$$v \in c/\texttt{child} \Leftrightarrow \\ pre(c) < pre(v) \leq pre(c) + size(c) \\ \wedge\ level(v) = level(c) + 1 \ . \qquad (\textsc{Child})$$

Assuming a context node sequence *ctx* encoded in the database as table `ctx`, the path expression *ctx*`/child::a` then compiles into the SQL expression

```
SELECT DISTINCT d.*
  FROM ctx c, doc d
 WHERE c.pre < d.pre AND d.pre ≤ c.pre + c.size
   AND d.level = c.level + 1
   AND d.kind = elem AND d.prop = 'a'
 ORDER BY d.pre
```

The condition on *pre* values may efficiently be answered in terms of an index scan along a B-tree on column *pre*. The subsequent test on the *level* property (and the name test for elements labeled `a`), however, will render a large share of these tuples *false hits*.

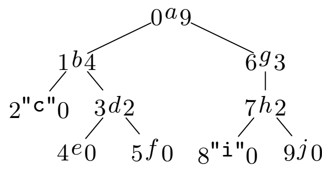
Earlier work [10] has suggested to overcome these false hits by explicitly modeling the parent/child relationship in the relational tree encoding. A foreign key reference to each node's parent turns the navigation along the `child` and `parent` axes into an operation particularly suited for relational systems: a join over key columns. The price we pay, however, is the additional storage and maintenance overhead involved in adding a reference *parent* to each node.

3.1 `child` Axis Evaluation on IBM DB2

To assess the apparent performance penalty due to false hits with respect to the *level* column, we encoded XML instances from the XMark benchmark [21] using the *pre*/*size*/*level* range encoding. In addition, we included a *parent* reference for each node in the document. We loaded generated documents of sizes up to 1.1 GB into an IBM DB2 UDB 9.1 ESE system, running on a SuSE Linux Enterprise Server 9 system equipped with 2×3.2 GHz Xeon processors and 8 GB RAM. Two 10,000 rpm SCSI drives hosted the tablespaces for DB2. We ran the DB2 index advisor utility `db2advis` and created indexes as suggested for the given workload.

Figure 2 documents the execution times we observed for the path `/descendant::open_auction/bidder/increase`[2] using both alternatives to express the navigation along the `child` axis (based on Condition Child and using a foreign key join on *parent*, respectively). The performance penalty for the *pre*/*size*/*level*-based evaluation, however, remained remarkably small over the entire range of document sizes:

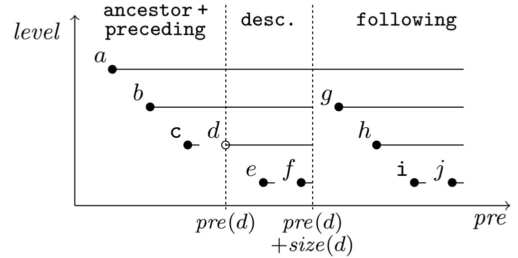
[2] The step `/descendant::open_auction` provides the context node sequence for the two `child` steps that are of actual interest to our discussion.



(a) Sample tree, annotated with properties *pre* (left) and *size* (right). Nodes `c` and `i` denote text nodes.

pre	size	level	kind	prop
0	9	0	elem	*a*
1	4	1	elem	*b*
2	0	2	text	c
3	2	2	elem	*d*
4	0	3	elem	*e*
5	0	3	elem	*f*
⋮	⋮	⋮	⋮	⋮

(b) Relational tree encoding.



(c) Associated *pre*/*level* plane. Lengths of the lines correspond to the *size* property. Properties *pre* and *size* determine the `descendant` and `following` regions of context node *d*.

Figure 1: Sample tree and associated relational encoding. An illustrative representation is the *pre*/*level* plane.

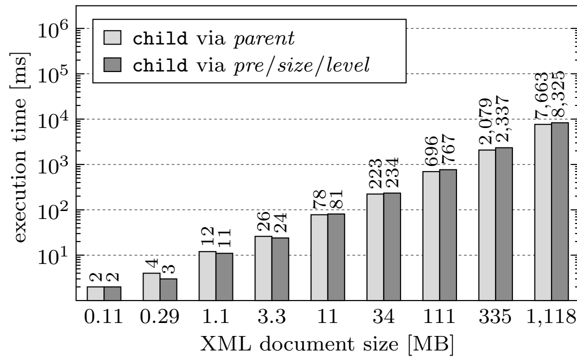


Figure 2: XPath performance for edge mapping and range encoding as observed for the path expression `/descendant::open_auction/bidder/increase`.

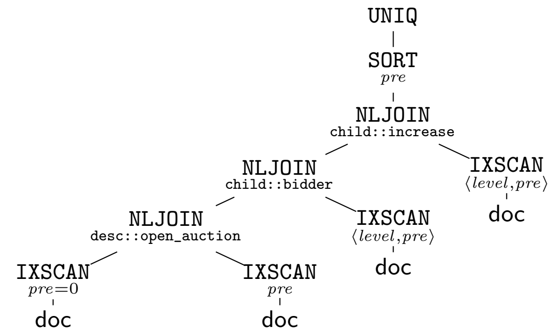


Figure 3: DB2 execution plan corresponding to the path `/descendant::open_auction/bidder/increase`.

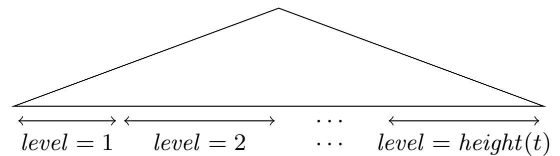


Figure 4: B-tree partitioning.

both query variants returned their results in roughly 8 seconds on, *e.g.*, the 1.1 GB XML instance.

The reason for this unexpectedly efficient execution on the *pre*/*size*/*level* encoding becomes apparent when we look into the query plan that IBM DB2 employs to evaluate our query. A (simplified) sketch of this plan is shown in Figure 3. IBM DB2 implements the two `child` steps with the help of a concatenated $\langle level, pre\rangle$ B-tree, with primary ordering on column *level*. We will now see why this index is particularly efficient in accelerating queries along the `child` axis.

3.2 Partitioned B-Trees

B-trees of this kind have also been referred to as *partitioned B-trees* [9]. With typical XML tree heights $height(t)$ of only 10–20, the selectivity of column *level* is very low, particularly for large document instances. Effectively, the prepending of *level* to a concatenated B-tree thus leads to a *partitioning* of the resulting index tree into $height(t)$ partitions (see Figure 4).

Fortunately, a low-selectivity prefix will only have a marginal impact on the B-tree's storage consumption in commodity RDBMS implementations. Prefix compression [2] avoids the repeated storage of the *level* information and minimizes the CPU overhead for search key comparisons during B-tree lookups. As discussed in [9], this economical dealing with system resources may even allow the use of a $\langle level, pre\rangle$ as a replacement for a *pre*-only key, *e.g.*, to defeat the cost of index maintenance incurring with updates.

3.2.1 Partitioned B-Trees for the `child` Axis

Most importantly, however, the partitioning leads to an evaluation strategy that will never encounter false hits with respect to the *level* property (see Figure 5 for an illustration). For each context node c, the system

1. *initiates* a scan of the partitioned B-tree using $level(v) = level(c) + 1$ and $pre(v) > pre(c)$ and

2. *scans* the index as long as $pre(v) \le pre(c) + size(c)$.

All nodes encountered during this index scan will satisfy Condition Child and, hence, qualify as children of c.

3.2.2 Reduced Complexity

Contrasted with a scan of an index with primary ordering on the key column *pre*, the work required to evaluate `child` on a partitioned $\langle level, pre\rangle$ B-tree no longer depends on the size of the *subtree* below the context node c. To demonstrate this effect, we chose `/site/regions/africa` as a path that

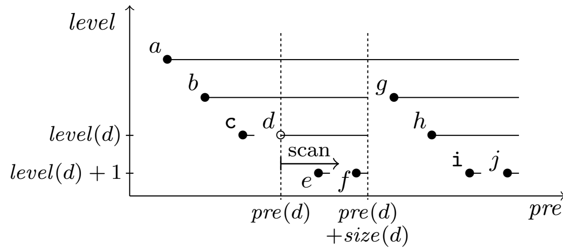


Figure 5: The sequential scan of a concatenated $\langle level, pre\rangle$ index will answer an XPath `child` navigation and not encounter any false hits.

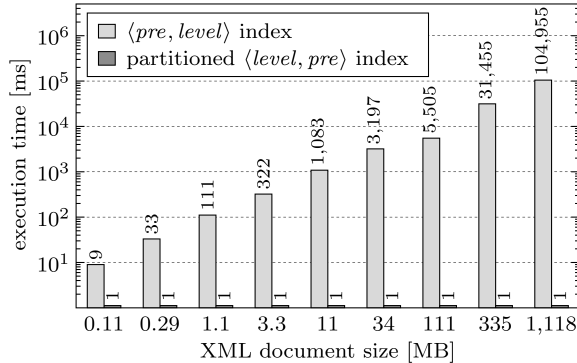


Figure 6: XPath `child` navigation using a partitioned B-tree. (path: `/site/regions/africa`).

returns exactly one node for all document sizes, while the subtree below the single context node grows proportionally to the XML instance size. We then removed all indexes from the *pre*/*size*/*level* table and left the system with only

(a) a $\langle pre, level\rangle$ B-tree or

(b) a partitioned $\langle level, pre\rangle$ B-tree

to evaluate the three child steps.[3]

Figure 6 documents how the partitioned B-tree decouples the query complexity from the total document size. While the use of the $\langle pre, level\rangle$ B-tree requires an execution time linear in the size of the XML document, the partitioned B-tree leads to sub-millisecond response times independent of the XML instance size.

3.2.3 Partitioned B-Trees and ORDPATH

ORDPATH labels [18] encode the XML tree structure in terms of a sequence of ordinals. To access a node's subtree, the encoding depends on the lexicographic order among labels, which coincides with the preorder rank $pre(v)$. Thus, a navigation along the `descendant` axis amounts to a range scan equivalent to Condition Desc. Likewise, the characterization of the XPath `child` axis on ORDPATH labels is equivalent to Condition Child, where a regular expression match assumes the place of the test on property *level*.

We therefore expect similar performance advantages from a partitioned $\langle level, ordpath\rangle$ B-tree in ORDPATH-based systems. Microsoft SQL Server does not currently maintain *level* information in its XML storage. *Functional indexes*, however, provided by most of the commercial systems (including SQL Server), allow tuple indexing based on *computed* values and could implement a $\langle level, ordpath\rangle$ index without the need for an explicit storage of *level*.

3.3 More Partitioned B-Trees

So far we have only exploited the presence of low-selectivity columns for the *structural* component of the XML encoding (column *level*). The same idea, however, applies to the *data* component as well. The *kind* column is an enumeration of only six XML node kinds, and even large XML instances are build from only few tag names.[4]

Both columns are suitable candidates to prefix a partitioned B-tree. A leading *kind* column (such as $\langle kind, pre\rangle$ or $\langle kind, level, pre\rangle$) effectively pushes XPath *kind* tests (`*`, `text()`, `comment()`, ...) into the index scan, while a prefix $\langle kind, prop\rangle$ (*e.g.*, $\langle kind, prop, pre\rangle$ or $\langle kind, prop, level, pre\rangle$) will similarly speed up the evaluation of *name* tests.

We have reported on the effectiveness of predicate pushdowns in XPath location steps in earlier work [13]. Partitioned B-trees provide an efficient implementation on commodity systems. In fact, we found the DB2 index advisor `db2advis` to seize this chance and suggest appropriate indexes for workloads that made use of XPath name and/or kind tests. Note also how an index scan along a $\langle kind, prop, pre\rangle$ index readily provides an implementation of the *element streams* required by the TwigStack algorithm in [6] using established indexing techniques only.

3.4 Benefit from Early-Out: `parent` Axis

Unfortunately, the idea of scanning a concatenated $\langle level, pre\rangle$ index to evaluate the `child` axis on range-encoded data cannot directly be transferred to an evaluation of XPath's `parent` axis. This axis is described by a constraint on *two* independent columns in the *pre*/*size* encoding:

$$\begin{aligned} v \in c/\texttt{parent} \Leftrightarrow \\ pre(v) < pre(c) \leq pre(v) + size(v) \\ \wedge\ level(v) = level(c) - 1\ . \end{aligned} \qquad \text{(Parent)}$$

B-tree indexes, however, can only be used to answer queries for ranges in a single dimension and, hence, cannot be used to find tuples qualifying for Condition Parent.

Yet, if we consider tree-specific properties inherent to the *pre*/*size* encoding, we can still remedy this restriction and evaluate a `parent` step in terms of a single index lookup. The idea is illustrated in Figure 7 (assuming node d as the context). Given the properties $level(d)$ and $pre(d)$ of the context node d, we can trigger a *reverse* index scan[5] on a concatenated $\langle level, pre\rangle$ index, starting at the index position $\langle level(d)-1, pre(d)\rangle$. As shown in Figure 7, such a scan will always encounter d's parent node as its *first* hit (if d has a parent at all).

This approach to the evaluation of the `parent` axis blends perfectly with the execution model of existing database systems. IBM DB2, *e.g.*, allows index scans to be executed as *single record* scans, a feature that precisely matches the evaluation strategy we are after here. If evaluated as a *single*

[3]Both combinations allow the evaluation of Condition Child based on the index data only.

[4]The XMark [21] DTD, *e.g.*, lists 77 tag names.

[5]The functionality to scan indexes in a reverse fashion may presuppose an explicit declaration during index creation, *e.g.*, in terms of DB2's `CREATE INDEX ... ALLOW REVERSE SCANS` instruction.

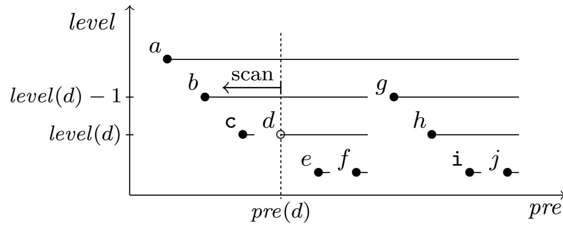


Figure 7: XPath `parent` navigation by means of a reverse $\langle level, pre\rangle$ index scan (starting from context node d).

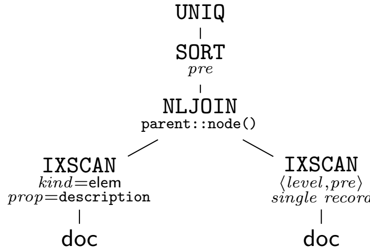


Figure 8: (Desired) DB2 execution plan to evaluate $Q_{\mathbf{Parent}}$ (`/descendant::description/parent::node()`).

record scan, index scans will only return their *first* matching tuple for each index re-scan, which exactly matches our needs. In Figure 8, we illustrated the corresponding execution plan for the path

`/descendant::description/parent::node()` . (Q_{PARENT})

Unfortunately, the *decision* to use a *single record* execution strategy requires an explicit knowledge about the data's tree origin, knowledge that is hardly expressible on the SQL level. Therefore, we cannot evoke our intended execution strategy using an SQL-only interface to the database system. An XQuery front-end with direct access to the planner component of the underlying system, however, could easily generate the desired execution plan as an implementation of the `parent` step. Note that an XQuery extension of this kind would not require any modifications to the system's execution engine but merely recycle existing functionality.

To evaluate the potential of a $\langle level, pre\rangle$-based `parent` evaluation, we *simulated* the respective query plan on our DB2 instance in terms of a modified version of Query Q_{PARENT}:

`/descendant::description [parent::node() ]` . (Q'_{PARENT})

This path essentially computes the result for Query Q_{PARENT}, but returns `description` nodes instead of their parents. Expressed in SQL, Query Q'_{PARENT} reads:

```
SELECT DISTINCT d.*
  FROM doc d
 WHERE d.kind = elem AND d.prop = 'description'
   AND EXISTS ( SELECT *
                  FROM doc p
                 WHERE p.level = d.level − 1
                   AND p.pre < d.pre
                   AND d.pre ≤ p.pre + p.size )
 ORDER BY d.pre .
```

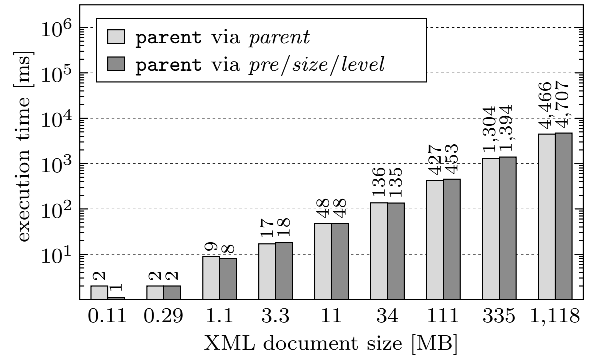


Figure 9: The evaluation of `parent` using a partitioned $\langle level, pre\rangle$ index is almost on par with the foreign key-based alternative (path `/descendant::description[parent::node()]`).

The most selective condition in this query is the restriction on `description` nodes, which DB2 will evaluate first. The `EXISTS` clause allows the system to choose an early-out strategy, a feature that DB2 implements in terms of a *single record* scan. The outcome is the plan in Figure 8, where the only difference to our initially intended plan lies in the set of attributes that are returned at the plan's root.

We benchmarked this evaluation strategy against an explicit foreign key join (property *parent*) as mentioned earlier. We created a partitioned $\langle level, pre\rangle$ B-tree to support the *pre*/*size*/*level*-based evaluation and a *pre* index to support lookups of the foreign key reference. The execution times for both evaluation strategies (illustrated in Figure 9) are indeed almost on par. The use of the partitioned B-tree, however, fully eliminates the need to maintain an explicit *parent* column in the relational encoding.

4. CONTEXT PRUNING IN A CONVENTIONAL RDBMS

Generally, an XPath location step originates in an entire *sequence* of context nodes. If multiple of these lead to the same result node, an evaluation on a node-by-node basis will consequently face a large number of duplicates in the intermediate step result. An expensive duplicate removal is then required to comply with the XPath semantics of a duplicate-free location step result. Assuming the document in Figure 1(a), *e.g.*, nodes g, h, `i`, and j are reachable via the `following` axis from both nodes of the context sequence (`c`, d) (context node `c` will additionally contribute d, e, and f).

4.1 Pruning in the *pre*/*level* Plane

In the *pre*/*level* plane, this situation surfaces as an *overlap* of the corresponding query regions, indicated in Figure 10 with different shadings for the `following` regions of nodes `c` and d. To avoid the overlap in the *pre*/*level* plane (and thus the generation of intermediate duplicates), we could *remove* node d from the context set before processing the step, retaining only the node with the minimum `following` (*i.e.*, $pre + size$) boundary.

This idea of *context sequence pruning* has already been suggested as a part of staircase join [13], an algorithm that invariably requires a modification of the underlying data-

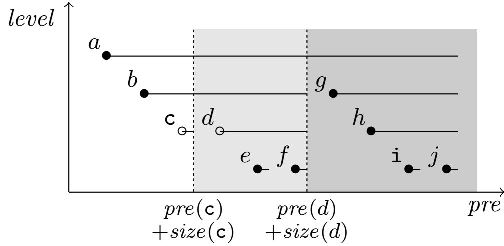


Figure 10: Nodes g, h, `i`, and j are located in the `following` regions of both context nodes `c` and d.

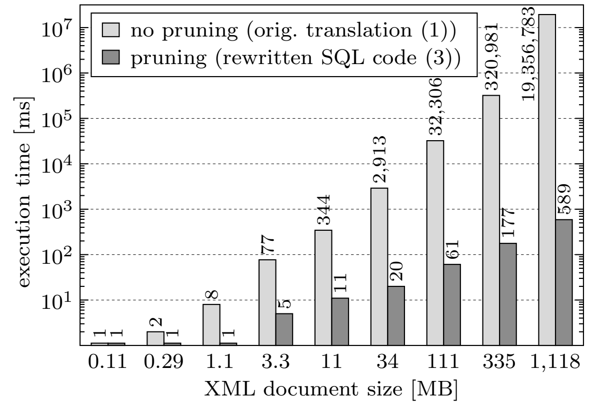


Figure 11: Context set pruning speeds up the evaluation of `/descendant::city/following::zipcode` by orders of magnitude.

base kernel. We can achieve the same effect, though, with purely relational plan rewrites in an off-the-shelf system. To illustrate, we use the SQL representation of the step *ctx*`/following`:

```
SELECT DISTINCT d.*
  FROM doc d, ctx c
 WHERE d.pre > c.pre + c.size                    (1)
 ORDER BY d.pre ,
```

essentially a semi-join between the persistent document container `doc` and a set of context nodes `ctx`. Due to the `DISTINCT` clause, this semi-join may be rewritten into

```
SELECT d.*
  FROM doc d
 WHERE d.pre > ANY ( SELECT c.pre + c.size       (2)
                       FROM ctx c )
 ORDER BY d.pre ,
```

which is most efficiently computed by aggregating over the context relation `ctx` first:

```
SELECT d.*
  FROM doc d
 WHERE d.pre > ( SELECT MIN (c.pre + c.size)     (3)
                   FROM ctx c )
 ORDER BY d.pre .
```

The rewritten query now captures the idea of pruning by purely relational means. In contrast to the technique suggested in [13], however, the rewrites do *not* depend on an explicit tree-awareness in the underlying kernel. Moreover, both rewrites are universally applicable to relational plans and could speed up query execution even on data that does not originate from an XML document encoding.

4.2 Context Pruning on IBM DB2

Apparently, both rewrites are not among the rule set of the DB2 query optimizer. To assess their potential, we used the two SQL equivalents (1) and (3) to translate the XPath expression `/descendant::city/following::zipcode`.

Figure 11 illustrates the resulting query execution times, where our pruning-enabled SQL code clearly outperforms the original SQL query. The same experiment also demonstrates how the demand for duplicate removal in the XPath semantics puts a serious strain on the relational system. On XMark-generated documents, the number of result nodes retrieved from the base tables scales quadratically with the XML document size. On large instances, we thus have to pay the toll for a growing *sort overhead* (8.1×10^9 nodes to sort for the 1.1 GB instance) and DB2 significantly falls back behind quadratic scaling. The rewritten query, in contrast, reaches sub-linear scaling across the entire size range.

5. XPATH OPTIMIZATION ON RDBMSS

Their effective measures to rewrite query execution plans for most efficient evaluation certainly are key aspects that made relational database systems so successful. It is interesting to see, since, how the same techniques will naturally lead to efficient XPath evaluation plans on off-the-shelf RDBMS implementations.

5.1 Existential Quantification and Early-Out

The XPath language specification [3] makes extensive use of *existential quantification*, *e.g.*, to define the semantics of XPath predicate expressions. It has been demonstrated in [8] that the lack of an *early-out* evaluation strategy to implement predicates can seriously hamper the scalability of any XPath implementation.

Such a strategy perfectly blends with the aforementioned *single record* execution strategy. A *single record* scan in the relational plan tree will immediately abort the processing of its subplan as soon as the first matching tuple is found. In Section 3.4 we took advantage of this feature to demonstrate an enhanced evaluation strategy for the XPath `parent` axis.

The Pathfinder XQuery compiler[6] implements a compilation procedure that translates arbitrary XQuery expressions into SQL:1999 queries [12, 11]. In the execution plans resulting from this compilation, we found DB2 to make frequent use of its *single record* functionality: the execution plan that evaluates the SQL counterpart of XMark Query Q1, *e.g.*, contains seven instances of a *single record* scan. On our test platform, this allows the evaluation of the query in less than one second over a 1.1 GB XMark instance.

5.2 Bottom-Up Location Step Processing

The same query plan also demonstrates how the application of join reordering mechanisms—a well-studied problem in the relational domain—enables the system to perform effective rewrites that proved to be quite a challenge to native XPath processors in the past. As discussed by [16], a step-by-step evaluation of XPath location paths usually leads to

[6]`http://www.pathfinder-xquery.org/`

Strategy	exec. time [s]
(a) arbitrary join order	129.1
(b) step-by-step evaluation	1.037
(c) simulation of "binary associations"	0.001

Table 1: DB2 execution times for XMark Q15 using (a) a system-determined join order, (b) step-by-step join order, and (c) a ⟨*path*, *pre*⟩ B-tree that simulates a binary associations encoding (111 MB document).

a top-down navigation in the XML document tree. Depending on the selectivity of the individual steps, however, it may be advantageous to process a path in a *backward* fashion and navigate the tree bottom-up or use a hybrid combination of both.

On the relational back-end, the same alternatives surface as different *join orders* in the query execution plan, a situation that modern RDBMSs know well how to deal with. Commodity systems typically rely on statistical information about the underlying tables to decide for a specific join order. These statistics turn out to be a suitable measure also to find appropriate evaluation strategies for XPath, even if the system is completely unaware of the tree structure that defines the relational table content.

XMark Query Q1 is essentially a measure for the system's XPath performance for the path

```
/site/people/person[@id="person0"] .
```

The query plan that corresponds to the full XMark query amounts to 45 operators (as obtained with DB2's plan analyzer `db2expln`). Despite this complexity, DB2 detected the chance to evaluate this path in a backward fashion. Starting from the highly selective predicate on the `@id` attribute (accessed via an index on attribute values), the system processes the above path from the leaf to the root. For other XMark queries, we found DB2's optimizer to opt for a top-down strategy or even hybrid combinations of both. For an equivalent decision, earlier work [16] had depended on specialized tree statistics (*e.g.*, data guides).

Taking a good decision on one of the possible join orders can be a serious challenge to the underlying RDBMS optimizer. Paths that involve a large number of location steps can quickly exceed the limits of eager join re-ordering algorithms. XMark Q15 essentially consists of a sequence of 13 child steps—apparently too much for the query optimizer of DB2. Assuming an XMark instance of 111 MB, this lead to an execution time of 129 seconds on our test platform. If we trick the system into using a step-by-step path evaluation by rewriting the SQL code, we can reduce the execution time to now only one second (see Table 1(b)).

5.3 Schema-Awareness with Partitioned Trees

The evaluation techniques we described assume a schema-oblivious storage, where each tree node maps to one tuple of a single database table in absence of any XML Schema information. Others have suggested to incorporate such information and collect nodes with a common root-to-node path into a unique database table [20, 22]. The resulting *semantical grouping* may—depending on the specific query workload—improve data access patterns to the underlying tables at the cost of an increased assembly overhead to combine (intermediate) results from multiple base tables [22].

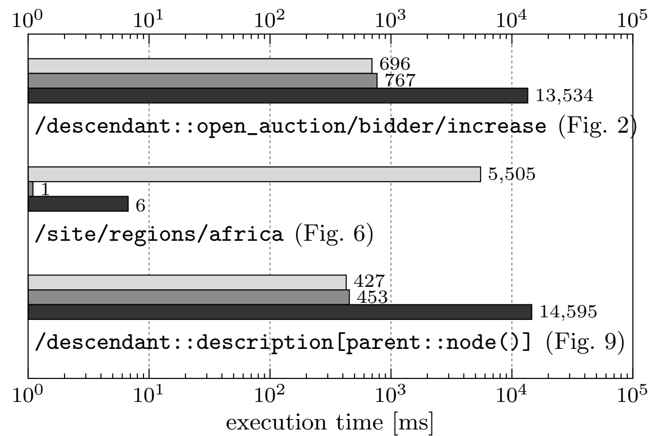


Figure 12: Relational XPath performance (light and medium gray) vs. DB2's built-in XQuery support (dark) as observed for a 111 MB XML instance.

The use of partitioned B-trees covers both situations in a surprisingly simple manner. To this end, we enrich our encoding with a column *path* that records the root-to-node path for each XML tree node. The XML storage of Microsoft's SQL Server, in fact, comes with an implementation of a *path* column (dubbed *PATH_ID*) [19]. Used as the prefix of a partitioned B-tree, this new column will partition the single document table `doc` into separate ranges for *each path* in the tree. For example, a concatenated ⟨*path*, *pre*⟩ B-tree readily realizes the binary association encoding of [20] over range-encoded data.

We implemented this idea on DB2. For queries that contain long sequences of `child` steps, the impact can be significant: the execution time for Query Q15 from the XMark benchmark, *e.g.*, drops to one milli-second if a ⟨*path*, *pre*⟩ B-tree is used to simulate the binary association encoding (see Table 1(c)).

6. RELATED TECHNIQUES

Much like other commercial database systems, the latest version of IBM DB2 ships with the integrated "pureXML" XQuery execution engine. Obviously, this functionality provides an interesting baseline for the relational XPath evaluation strategies that we investigate here.

We have run all aforementioned queries on the same system over an 111 MB XMark instance using the pureXML query processor.[7] Since the figures of our relational XPath evaluation do not include serialization time, we wrapped the respective path expressions into a call of `fn:count()` to also eliminate this cost factor for DB2's native XPath processor.

Figure 12 contains the execution times shown in previous figures (Figures 2, 6, and 9) for the relational path evaluation, along with execution times achieved with the pureXML processor for the same queries.

The workload we are using here is quite different to the typical pureXML workload. The DB2 column type `XML` is designed to hold small but many XML instances, such as

[7] DB2's built-in XQuery processor does not support the XPath `following` axis, which is why we omitted a test for the query of Figure 11.

they arise, *e.g.*, in message processing systems. Such data can optionally be indexed with *XML pattern indexes* that index all *values* that are reachable via a given path expression. Since the focus of our experiments is raw XPath navigation performance, DB2 cannot benefit from any of its value-based indexes, which puts a serious strain on the built-in XPath processor. The resulting performance, shown in Figure 12, indicates that—at least for certain query workloads—a relational approach to XPath evaluation can significantly outperform an industrial-strength native XQuery processor.

6.1 More Related Work

To add efficient tree processing capabilities to relational systems, earlier work proposed to modify the existing database engine and add highly specialized XPath processing algorithms to the DBMS kernel. The *multi-predicate merge join* (MPMGJN) has been suggested in [24] to address *containment queries* by relational means, a generalization of XPath `descendant/child` navigation steps. A natural extension of MPMGJN are the *structural join* [1] and *staircase join* [13] algorithms. Considering data dependencies that originate from the encoded tree structure, they target the full set of axes in the XPath language. Similarly, the *PathStack* and *TwigStack* algorithms [6] are tailor-made to process tree navigation primitives in relational systems.

While these proposals make profitably use of relational processing paradigms, we feel that the inherent invasion of the database kernel to add the new functionality is an unacceptable option for most real-world systems. The techniques we described here solely depend on *existing* DBMS technology, available in *off-the-shelf* database implementations.

Crucial to these techniques is the use of concatenated B-tree indexes. This distinguishes our work from earlier approaches that depended on specialized index variants, such as XB- or XR-Trees [6, 14].

7. SUMMARY

In the interest of efficient XML processing on relational systems, earlier work has proposed a number of novel database operators that provide specialized tree processing support in the database kernel. As an invasion of the system's kernel seems hardly an option for any DBMS vendor, we evaluated how much *existing* and *widely available* database functionality is suited to efficiently process XML data.

Among this functionality, the use of *partitioned B-trees* proved particularly effective. Low-selectivity prefixes in concatenated B-trees implement a semantical grouping which allows for efficient XPath evaluation strategies. Further, well-known *rewrite techniques* from the relational domain turned out to readily implement tree-processing functionality that proved challenging in the past. Most notably, we realized staircase join's *pruning* idea and *bottom-up path processing* by purely relational means.

The techniques we saw are of universal nature and equally applicable to any of the established tree encodings, including *pre/post-* [10] and Dewey-based [23] numbering schemes. As such, they may prove fruitful for RDBMS vendors, as well as XQuery implementors.

Acknowledgements

We thank Rudolf Bayer who has contributed to this work with insightful feedback. Our research is supported by the German Research Foundation DFG (grant GR 2036/2-1).

8. REFERENCES

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the 18th Int'l Conference on Data Engineering (ICDE)*, San Jose, CA, USA, February 2002.

[2] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems (TODS)*, 2(1), March 1977.

[3] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. World Wide Web Consortium Candidate Recommendation, September 2005.

[4] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium Candidate Recommendation, September 2005.

[5] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 2006 ACM SIGMOD Int'l Conference on Management of Data*, Chicago, IL, USA, June 2006.

[6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, Madison, WI, USA, 2002.

[7] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. of the 2003 ACM SIGMOD Int'l Conference on Management of Data*, San Diego, CA, USA, June 2003.

[8] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems (TODS)*, 30(2), June 2005.

[9] Goetz Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. of the 1st Int'l Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2003.

[10] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, Madison, WI, USA, June 2002.

[11] Torsten Grust, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. of the 2007 ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[12] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[13] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, Berlin, Germany, September 2003.

[14] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. of the 19th Int'l Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[15] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, Rome, Italy, September 2001.

[16] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, UK, September 1999.

[17] Microsoft Corporation. SQL Server 2005, 2005.

[18] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of Data*, Paris, France, June 2004.

[19] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML Data Stored in a Relational Database. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[20] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In *The World Wide Web and Databases (WebDB), 3rd Int'l Workshop*, Dallas, TX, USA, May 2000.

[21] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, Hong Kong, China, August 2002.

[22] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[23] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, Madison, WI, USA, 2002.

[24] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the 2001 ACM SIGMOD Int'l Conference on Management of Data*, Santa Barbara, CA, USA, 2001.