Classification of Index Partitions to Boost XML Query Performance*

Gerard Marks, Mark Roantree, and John Murphy

Interoperable Systems Group,
Dublin City University, Dublin 9, Ireland
{gmarks,mark.roantree,jmurphy}@computing.dcu.ie

Abstract. XML query optimization continues to occupy considerable research effort due to the increasing usage of XML data. Despite many innovations over recent years, XML databases struggle to compete with more traditional database systems. Rather than using node indexes, some efforts have begun to focus on creating partitions of nodes within indexes. The motivation is to quickly eliminate large sections of the XML tree based on the partition they occupy. In this research, we present one such partition index that is unlike current approaches in how it determines size and number of these partitions. Furthermore, we provide a process for compacting the index and reducing the number of node access operations in order to optimize XML queries.

1 Introduction

Despite the continued growth of XML data and applications that rely on XML for the purpose of communication, there remains a problem in terms of query performance. XML databases cannot perform at the same level as their relational counterparts, and as a result, many of those who rely on XML for reasons of interoperability are choosing to store XML data in relational databases rather than its native format. For this reason, the advantages of *semi-structured* data (i.e. schema-less data storage) are lost in the *structured* world of relational databases, where schema design is required before data storage is permitted. The result of this is that many domains such as sensor networks are using rigid data models where more flexible and dynamic solutions are required. Over the last decade, many research groups have developed new levels of optimization. However, there remains significant scope and opportunity for further improvements.

In this paper, we adopt some of the methodology that has been applied in the past but introduce a new approach where we dynamically partition the XML document, together with a metadata structure, to improve the performance of the index. In doing so, we can demonstrate new levels of optimization across XPath expressions.

The paper is organized as follows: in the remainder of this section, we provide further background and motivation and state our contribution to this area; in

* Funded by Enterprise Ireland Grant No. CFTD/07/201.

J. Parsons et al. (Eds.): ER 2010, LNCS 6412, pp. 405–418, 2010.
© Springer-Verlag Berlin Heidelberg 2010

§2, we examine similar research approaches in XML query optimization; in §3, we provide a detailed description of our partitioned index; in §4, we describe how query processing can take advantage of our approach; in §5, we present our experiments and discuss the findings, before concluding in §6.

1.1 Background and Motivation

Current XML query optimization solutions can be placed in two broad categories. On one hand, *index based* approaches build indexes on XML documents to provide efficient access to nodes, e.g. XRel [1], XPath Accelerator [5], Xeek [10]. On the other hand, *algorithmic based* approaches are focused on designing new *join* algorithms, e.g. TJFast [9], StaircaseJoin [7]. The former approach can use standard relational databases to deploy the index structure and thus, benefit from mature relational technology. The latter depends on a modification to the underlying RDBMS kernel [2], or a *native* XML database may be built from scratch.

The *XPath Accelerator* [5] demonstrated that an optimized XPath index that lives entirely within a relational database can be used to evaluate all of the XPath axes. However, the XPath Accelerator, and similar approaches [6], suffer from scalability issues, as this type of node evaluation (even across relatively small XML documents) is inefficient [10].

A more recent solution is to partition nodes in an XML tree into *disjoint* subsets, which can be identified more efficiently as there will always be less partitions than there are nodes. After the relevant partitions are identified, only the nodes that comprise these partitions are evaluated using the inefficient node comparison step. Based on *pre/post* encoding, [11] is an index based approach that requires a user defined *partitioning factor* to divide the *pre/post* plane into disjoint sub-partitions. However, an optimal partitioning factor cannot be known in advance and as a result, rigorous experimentation is needed to identify this parameter (as is discussed in our related research).

1.2 Contribution

The main contributions in our work can be described as follows:

- We provide a novel partitioning method for XML document indexes that offers new levels of optimization for XML queries.
- We have developed efficient algorithms that automatically identify and resize these document partitions in a *single pass* of the XML dataset; user defined partitioning factors are not used.
- Using structural information we can allow identical node partitions to be merged and thus reduce the size of the index and avoid processing large numbers of equivalent nodes.
- Finally, for the purpose of comparing our approach to similar works we use *pre/post* encoding. However, to the best of our knowledge, in this paper, we present the first index based partitioning approach that is independent of the specific properties of the XML node labeling scheme used. Therefore,

our approach can be more easily integrated with other XML node labeling schemes such as ORDPATH [12].

We also provide a longer version of this paper, which includes the concepts and terminology that underpins our work, and extended related research, optimization and experiments sections [3].

2 Related Research

The *XPath Accelerator* [5] exemplifies an XML database built on top of a relational database. In this work, *pre/post* region information, i.e. *region encoding*, is used as their XML node encoding scheme. However, querying large XML datasets using *pre/post* labels is inefficient [10].

In [11], the *pre/post* plane is partitioned based on a user defined *partitioning factor*. Fig 1 illustrates the *pre/post* plane partitioned *parts* using a partitioning factor of 4. For each node, the *pre/post* identifier of its *part* is the *lower bound* of its x and y values respectively. For example, in Fig 1 the *part* P associated with node $x(6, 5)$ is: $P(4, 4)$. The ancestors of node x can only exist in the *parts* that have a lower bound x value ≤ 4 and a lower bound $y \geq 4$, i.e. the shaded parts (Fig 1). Similar is true for the other major XPath axes, i.e. descendant, following and preceding.

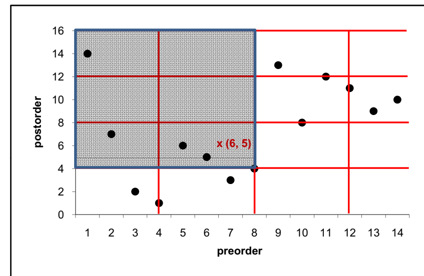


Fig. 1. Partitioning factor $N=4$

The problem with this approach is that an ideal partitioning factor is not known in advance and requires rigorous experimentation to identify. For example, in reported experiments each XML document was evaluated for the partitioning factors 1, 2, 4,.., 256 [11]. We believe this type of experimentation is infeasible even for relatively small XML documents. Additionally, as XML data is irregular by nature, a single partitioning factor per dataset is less than ideal. Finally, although it is suggested in [11] that the partitioning approach may be tailored to other encoding schemes such as *order/size*, it relies heavily on the *lower bound* of each x and y value in the partitioned *pre/post* (or order/size) plane. Therefore, this approach does not lend itself naturally to *prefix* based encoding schemes such as ORDPATH [12], which have become very popular in recent years for reasons of updatability.

The work presented in this paper overcomes these issues by automatically partitioning nodes based on their individual layout and structural properties within each XML dataset. We do not rely on user defined partitioning factors. Also, our approach is not dependent on the specific properties of XML node labels, and thus can be used in conjunction with any XML encoding scheme.

3 Optimization Constructs

In this section, we present the optimization process together with the various constructs and indexing layers it comprises. We start by introducing a small number of new constructs that form part of the optimization process. Following this, we provide a step-by-step description of how we create the dynamic partition index that is influenced by the structure and data content of each XML document.

Definition 1. *A branch is a* set *of connected node identifiers within an XML document.*
A branch (sometimes referred to as a sub-tree) is the abstract data type used to describe a partition of nodes. In our work, we will deal with the local-branch and path-branch sub-types of a branch.

Definition 2. *A local-branch is a branch, such that its members represent a single branching node and the nodes in its subtree. A local-branch cannot contain a member that represents a descendant of another branching node.*

In a tree data structure, a *branching node* will have a minimum of two child nodes, whereas a *non-branching node* has at most one child node [8]. The local-branch uses the branching node to form each partition. Our process uses the rule that each local-branch must not contain nodes that are descendants of another branching node, to create primary partitions.

Definition 3. *A path-branch is a branch with a single path.*

The path-branch is an abstract type with no branching node. Each member is a child member of the preceding node. Its three sub-types (*orphan-path*, *branchlink-path* and *leaf-path*) are used to partition the document.

Definition 4. *An orphan-path is a path-branch such that its members cannot belong to a local-branch.*

The orphan-path definition implies that members of the orphan-path cannot have an ancestor that is a branching node. The motivation is to ensure that each node in the XML document is now a member of some partition.

Definition 5. *A branchlink-path is a path-branch that contains a link to a single descendant partition of its local-branch.*

In any local-branch, there is always a single branching node and a set of non-branching nodes. With the non-branching nodes, we must identify those that share descendant relationships with other partitions. These are referred to as branchlink-path partitions and each member occupies the path linking two branching nodes (i.e. two partitions).

Definition 6. *A leaf-path is a path-branch that contains a leaf node inside its local-branch.*

A leaf-path differs from a branchlink-path in that it does not contain a link to descendants partitions. In other words, it contains a single leaf node and its ancestors.

3.1 Creating the Primary Partitions

When creating the first set of partitions, the goal is to include all nodes in local-branch or path-branch partitions. As explained earlier, path-branches are abstract types and at this point, all path-branch instances will be orphan-paths.

The algorithms for encoding an XML document using a *pre/post* encoding scheme were provided by *Grust* in [5]. In brief, each time a *starting tag* is encountered a new element object is created, which is assigned the attributes: *name, type, level,* and *preorder*. After which, the new element is pushed onto an *element stack*. Each time an *end tag* is encountered an element is popped from the element stack and is assigned a postorder identifier.

Once an element has been popped from the stack, we call it the *current node*, and the *waiting list* is a *set* in which elements reside temporarily prior to being indexed. The first step in the process is to determine if the current node is a branching node by checking if it has more than one child node. The next steps are as follows:

1. If the current node is non-branching and does not reside at *level 1*, it is placed on the *waiting list*[1].
2. If the current node is branching, it is assigned to the next local-branch in sequence. Also, the nodes on the waiting list will be its descendants. Therefore, the nodes on the waiting list are output to the same local-branch as the current node.
3. If the current node is non-branching, but a node at *level 1* is encountered, the current node does not have a branching node ancestor. Therefore, the current node is assigned to an *orphan-path* (Definition 4). For the same reason, any node currently on the waiting list is assigned to the same orphan-path.

3.2 Partition Refinement

Although the local-branches are rooted subtrees they may contain nodes that do not have an ancestor/descendant relationship. As we will discuss in §4, the separation of nodes that do not have a hierarchical association leads to an optimized pruning effort.

Each local-branch instance has a single branching node *root* which may have many (non-branching node) descendants. It is the non-branching descendants of the root that are checked to see if they share a hierarchical association. For this reason, we partition the non-branching nodes (in each local-branch) into disjoint path-branches (Definition 3). As orphan-paths and local-branches are *disjoint*, each of these path-branch instances will be a *branchlink-path* (Definition 5) or a *leaf-path* (Definition 6).

The `RefinePartitions` (algorithm 1) replaces all steps outlined for creating the primary index (above). The new branch partitions are created by processing

[1] Examples and illustrations of the primary partitions are provided in the long version of this paper [3].

two local-branches simultaneously. All *current nodes* (see creating primary partitions above), up to and including the first branching node, are placed in the first *waiting list* (*wList1*) where they *wait* to be indexed. Subsequently, the next set of current nodes, up to and including the second branching node, are placed on the second waiting list (*wList2*). At this point, *wList1* and *wList2* contain the nodes that comprise the first and second local-branches respectively.

Algorithm 1. RefinePartitions

1: **if** node at *level 1* encountered **then**
2: move nodes that comprise *wList2* to *orphan-path*;
3: **end if**
4: move non-branching nodes from *wList1* to *leaf-path*;
5: **for** each node n in *wList2* **do**
6: **if** n = ancestor of $wList1.ROOT \wedge n \neq$ branching node **then**
7: move n to *branchlink-path*;
8: **else if** $n \neq$ ancestor $wList1.ROOT$ **then**
9: move n to *leaf-path*;
10: **end if**
11: **end for**
12: move *local-branch* from *wList1* to local-branch;
13: move *local-branch* from *wList2* to *wList1*;

If a node at *level 1* is encountered, the nodes that comprise *wList2* are an orphan-path (*line 2*). If a *branchlink-path* (Definition 5) exists, `RefinePartitions` identifies it as the non-branching nodes in *wList2* that are *ancestors* of the root node in *wList1* (*lines 6-7*). If one or more *leaf-paths* (Definition 6) exist, they will be the nodes in *wList2* that are not ancestors of root node in *wList1* (*lines 8-9*). The remaining nodes that comprise the first local-branch (*wList1*) are then moved to the index (*line 12*); this will be the *single* branching node root of the first local-branch only. At this point, the only node that remains in *wList2* is the root node of the second local-branch. This local-branch is then *moved* to *wList1* (*line 13*); *wList2* will not contain any nodes at this point. The next local-branch is placed in *wList2* and the process is repeated until no more branches exist. When this process has completed, the result will be a lot more partitions, with the benefit of increased pruning. This is illustrated in Fig 2.

The process will also track the *ancestor-descendant* relationships between branch partitions. This is achieved by maintaining the *parent-child* mappings between branches. Given two branches: `B1` and `B2`, `B2` is a child of `B1` *if and only if* the *parent* node of a node that comprises `B2` belongs to `B1`. When the `RefinePartitions` process is complete, the *ancestor-descendant* relationships between branches are determined using a recursive function across these parent-child relationships, i.e. select the branch's children, then its children's children recursively.

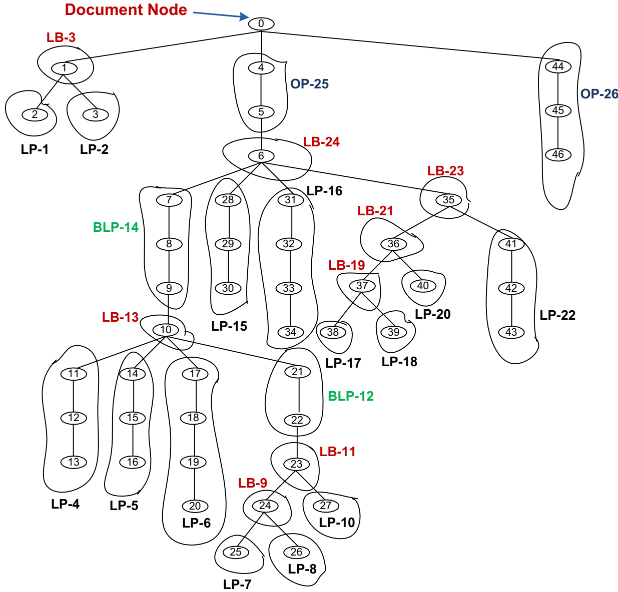


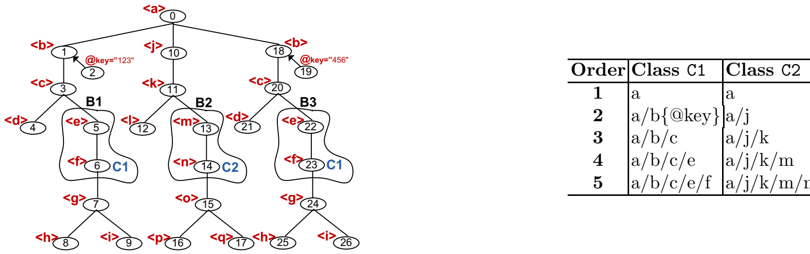
Fig. 2. After Partition Refinement

3.3 Branch Class Index

The indexing process results in the creation of a large number of branch partitions. This benefits the optimization process as it facilitates a highly aggressive pruning process and thus, reduces the inefficient stage of node comparisons. The downside of aggressive pruning is the large index size it requires. Our final step is to reduce the size of our index while maintaining the same degree of pruning. To achieve this, we use a *classification* process for all branches based on root to leaf structure of the partition.

Definition 7. *A branch class describes the structure of a branch, from the document node to its leaf node, and includes both elements and attributes.*

Every branch instance can belong to a *single* branch class. A process of classifying each branch will use the structure of the branch instance and its relationship to other branch instances as the matching criteria. Earlier work on DataGuides [4] adopted a similar approach, although here the branch class includes the DataGuide and set of attribute names associated with each element node on the path from the document node to the *leaf* node within each branch instance. Additionally, in order to belong to the same class, each branch instance must have an identical *set* of descendant branches. The latter is required to ensure that there is no overlap between branch classes, which we will discuss in §4.



Order	Class C1	Class C2
1	a	a
2	a/b{@key}	a/j
3	a/b/c	a/j/k
4	a/b/c/e	a/j/k/m
5	a/b/c/e/f	a/j/k/m/n

Fig. 3. Branch Classifications

Fig 3 depicts a sample XML document showing three branch instances, `B1-B3` (left) and the *extended* DataGuides associated with two branch classes, `C1` and `C2` (right). Note that the *order* of the extended DataGuides associated with each branch class is important. After classification, if `B1` and `B3` have an identical set of descendant branch instances, they will be instances of the `C1` class, while branch `B2` is an instance of the `C2` class.

Finally, the process that maintains parent-child relationships between branch instances (discussed earlier), must be replaced with one that maintains parent-child relationships between branch classes. The ancestor-descendant relationships are then generated for branch classes in the same manner as they were for branch instances.

4 Index Deployment and Query Processing

In this section, we describe the indexing constructs resulting from the indexing process in §3. Following this, we give an overview of our query processing approach and continue with a worked example to illustrate how query optimization is achieved.

Using the sample XML document in Fig. 4, Tables 4-4 illustrate the `NODE`, `NCL` (Name/Class/Level), and `CLASS` index respectively. The `NODE` index contains an entry for each node in the XML document. The `NCL` is generated by selecting each distinct *name*, *class*, *level* and *type* from the `NODE` index. The `CLASS` index contains ancestor-descendant mappings between branch classes, where the attributes *ac* and *dc* are the ancestor-or-self classes and descendant-or-self classes respectively. The `NCL` index allows us to bypass, i.e. avoid processing, large numbers of nodes (discussed shortly).

In the traditional approach to XPath query processing, there is a two step process: (1) retrieve nodes (based on the XPath axis and NodeTest), (2) input these nodes to the subsequent step (i.e. context nodes), or return them as the *result set* (if the current step is the *rightmost* step in the path expression). In partitioning approaches, a third step is added. Thus, the query process is performed in the following steps:

1. Identify the relevant partitions, i.e. prune the search space.
2. Retrieve the target nodes from these partitions, i.e. by checking their labels (e.g. *pre/post*, *dewey*).
3. Input these nodes to the subsequent step, or return them as the result set.

The `NODE` and `CLASS` indexes are sufficient to satisfy all three steps, where the `CLASS` index prunes the search space (*step 1*), thus optimizing *step 2*. However, ultimately we are only concerned with the nodes that are output from the rightmost step in an XPath expression, as these will form the *result set* for the query. Nodes that are processed as part of the preceding steps are only used to navigate to these result nodes. Using the `NCL` index instead of the `NODE` index (where possible), enables us to *bypass* (or avoid processing) many of these nodes that are only used to navigate to the result set, thus *step 2* is optimized further.

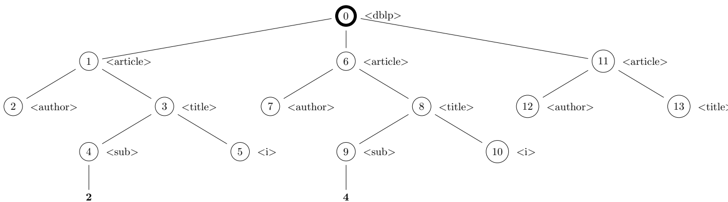


Fig. 4. XML Snippet taken from the DBLP Dataset

Table 1. Node Index

<table>
<tr><th>label</th><th>name</th><th>type</th><th>level</th><th>class</th><th>value</th></tr>
<tr><td>(0,13)| 0</td><td>dblp</td><td>1</td><td>0</td><td>n/a</td><td>-</td></tr>
<tr><td>(1,4)| 0.0</td><td>article</td><td>1</td><td>1</td><td>5</td><td>-</td></tr>
<tr><td>(2,0)| 0.0.0</td><td>author</td><td>1</td><td>2</td><td>1</td><td>-</td></tr>
<tr><td>(3,3)| 0.0.1</td><td>title</td><td>1</td><td>2</td><td>4</td><td>-</td></tr>
<tr><td>(4,1)| 0.0.1.0</td><td>sub</td><td>1</td><td>3</td><td>2</td><td>2</td></tr>
<tr><td>(5,2)| 0.0.1.1</td><td>i</td><td>1</td><td>3</td><td>3</td><td>-</td></tr>
<tr><td>(6,9)| 0.1</td><td>article</td><td>1</td><td>1</td><td>5</td><td>-</td></tr>
<tr><td>(7,5)| 0.1.1</td><td>author</td><td>1</td><td>2</td><td>1</td><td>-</td></tr>
<tr><td>(8,8)| 0.1.2</td><td>title</td><td>1</td><td>2</td><td>4</td><td>-</td></tr>
<tr><td>(9,6)| 0.1.2.1</td><td>sub</td><td>1</td><td>3</td><td>2</td><td>4</td></tr>
<tr><td>(10,7)| 0.1.2.2</td><td>i</td><td>1</td><td>3</td><td>3</td><td>-</td></tr>
<tr><td>(11,12)| 0.2</td><td>article</td><td>1</td><td>1</td><td>7</td><td>-</td></tr>
<tr><td>(12,10)| 0.2.1</td><td>author</td><td>1</td><td>2</td><td>8</td><td>-</td></tr>
<tr><td>(13,11)| 0.2.2</td><td>title</td><td>1</td><td>2</td><td>6</td><td>-</td></tr>
</table>

Table 2. NCL Index

NAME	CLASS	LEVEL	TYPE
author	1	2	1
sub	2	3	1
i	3	3	1
title	4	2	1
article	5	1	1
title	6	2	1
article	7	1	1
author	8	2	1

Table 3. CLASS Index

ac	dc
1	1
2	2
3	3
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
6	6
7	6
7	7
7	8
8	8

Bypassing is not possible across all steps in an XPath expression. Therefore, a selection process is required to choose which steps must access the `NODE` index, and which steps can access the (much smaller) `NCL` index instead. We are currently in the process of formally defining this process across all steps. Thus, in this paper we present the rules for the selection process that we have currently defined:

1. The `NODE` index must be used at the *rightmost* step in the path expression, i.e. to retrieve the actual result nodes. For example, see the rightmost step (`/education`) in `Q1` (Fig 5).

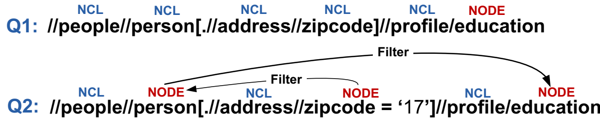


Fig. 5. Index Selection

2. If the query does not evaluate a text node, the `NCL` index can be used in *all* but the rightmost step. For example, `Q1` does not evaluate a text node, thus only the rightmost step accesses the node index as required by *rule 1*.
3. All steps that evaluate a text node must use the `NODE` index, e.g. `//zipcode = '17'` (`Q2`).
4. A step that contains a predicate filter that subsequently accesses a text node must use the `NODE` index, e.g. *step two* in `Q2`.

`NODE` index accesses are required to filter nodes based on the *character content* of text nodes, i.e. the `VALUE` attribute (Table 4), or to retrieve the *result set* for the rightmost step. The character content of text nodes was not considered during the branch classification process (§3) in order to keep the number of branch classes, and therefore, the size of the `CLASS` index small. However, `NODE` index accesses (based on the character content of text nodes) are efficient as they usually have *high selectivity.* In fact, where the character content of text nodes that do not have high selectivity can be identified, e.g. *gender* has only 2 possible values, they can be included as part of the classification process ensuring high selectivity for all remaining `NODE` accesses. However, we are currently examining the cost/benefit aspects of including text nodes in our classification process.

Example 1. `//people//person`

```
1. SELECT * FROM NODE S_RM WHERE S_RM.TYPE = 1 AND S_RM.NAME = 'person'
2.   AND S_RM.BRANCH IN (
3.     SELECT C_1.DC FROM NCL N_1, CLASS C_1
4.       WHERE N_1.NAME = 'people'
5.       AND N_1.CLASS = C_1.AC
6.       AND S_RM.LEVEL > N_1.LEVEL
7.     )
8. ORDER BY S_RM.PRE
```

In Example 1, notice that the `NODE` index is only accessed in the rightmost step (*line 1*). The layout of the final branch partitions (see Fig 2) enables us to evaluate the ancestor (or self), descendant (or self), parent or child axis by checking the `LEVEL` attribute (*line 6*). Note, this would not be possible if we allowed overlap between branches (discussed in §3). Similar approaches must evaluate unique node labels, e.g. *pre/post* or *dewey.* An additional benefit of the fact that we do not allow overlap between branch classes is that the inefficient `DISTINCT` clause that is required by related approaches [11, 5] to remove duplicates from the result set can be omitted. Also, as large numbers of nodes are bypassed, the `IN` clause is efficient as the sub-query usually returns a small number of branch classes.

5 Experiments

In this section, we compare our branch based approach to similar (lab-based) approaches. Following this, we evaluate how our approach performs against vendor systems.

Experiments were run on identical servers with a 2.66GHz Intel(R) Core(TM)2 Duo CPU and 4GB of RAM. For each query, the time shown includes the time taken for: (1) the XPath-to-SQL transformation, (2) the SQL query execution, and (3) the execution of the SQL **count()** function on the `PRE` column of the *result set*. The latter is necessary as some SQL queries took longer to return *all* rows than others. Each query was executed 11 times ignoring the first execution to ensure *hot cache* result times across all queries. The 10 remaining response times were then averaged to produce the final time in *milliseconds*. Finally, we placed a 10 minute timeout on querys.

Table 4. XPath Queries

	XMark
Q01	/site/regions/africa
Q02	/site/people/person[@id = 'person0']
Q03	//regions/africa//item/name
Q04	//person[profile/@income]/name
Q05	//people/person[profile/gender][profile/age]/name
Q06	/site/keyword/ancestor::listitem/text/keyword
Q07	/site/closed_auctions/closed_auction//keyword
Q08	/site/closed_auctions/closed_auction[./descendant::keyword]/date
Q09	/site/closed_auctions/closed_auction/annotation/description/text/keyword
Q10	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date

5.1 Comparison Tests with Lab-Based Systems

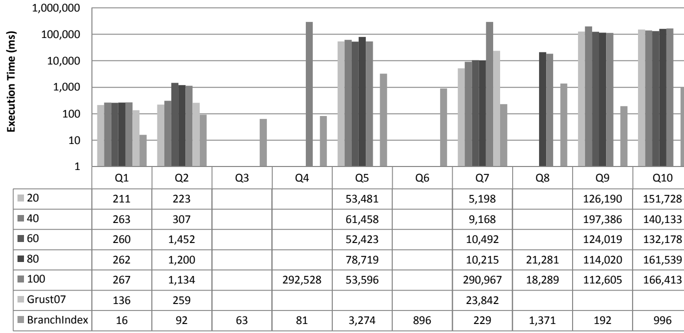
In this section, we will evaluate the performance of a traditional *node based* approach to XPath: *Grust07* [6], and the *partitioning approach* most similar to ours: Luoma07 [11].

For *Grust07*, we built the suggested *partitioned B-trees*: *Node(level,pre)*, *Node(type,name,pre)* and *Node(type,name,level,pre)*. Additionally we built indexes on *size*, *name*, *level*, *value* and *type*. For *Luoma07*, we used partitioning factors 20, 40, 60, and 100. As suggested in this work, *Node(pre)* is a primary key. *Node(part)* is a foreign key reference to the primary key *Part(part)* and indexes were built on *Node(post)*, *Node(name)*, *Node(part)*, *Part(pre)*, and *Part(post)*.

Our overall findings for both approaches are that they do not scale well even for relatively small XML documents. As such, we had to evaluate these approaches using a relatively small dataset. Later in this section, we evaluate large XML datasets across vendor systems.

For the following experiments, we generated an XMark dataset of just 115 MB in size and tested both approaches against queries from the *XPathMark* [13] benchmark and Grust07 (Table 4).

In Fig 6, the query response time for each of these queries is shown. These results show the following:



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
20	211	223			53,481		5,198		126,190	151,728
40	263	307			61,458		9,168		197,386	140,133
60	260	1,452			52,423		10,492		124,019	132,178
80	262	1,200			78,719		10,215	21,281	114,020	161,539
100	267	1,134		292,528	53,596		290,967	18,289	112,605	166,413
Grust07	136	259					23,842			
BranchIndex	16	92	63	81	3,274	896	229	1,371	192	996

Fig. 6. Query Response Times for Luoma07, Grust07 and BranchIndex

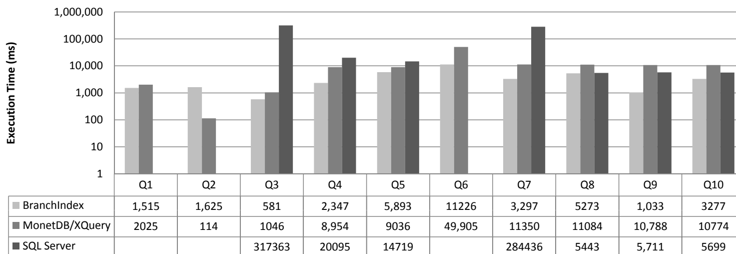
- Grust07 timed out on all but Q01, Q02, Q07.
- In Luoma07, a partitioning factor of 100 returned results for the greatest number of queries: Q01, Q02, Q04, Q05, Q07, Q08, Q09, Q10. Q07 shows an increase in processing times as the partitioning factor increased, whereas Q09 showed a decrease. The remaining queries do not provide such a pattern.
- Luoma07 returned results for a greater number of queries then Grust07 across all partitioning factors.
- The BranchIndex is orders of magnitude faster across all queries.

Queries Q01 and Q02 have *high selectivity* as they return a single result node. Also, the first two steps in Q7, i.e. `/site` and `/closed_auctions`, both access a single node. We attribute the fact that Grust07 returned results for queries Q01, Q02 and Q03 to the high selectivity of these queries. As the second *bullet point* indicates that there is no consistent pattern between the incrementing partitioning factors, we suggest that a single partitioning factor per dataset is not ideal. Luoma07 provides superior results than *Grust07*, both in terms of the query response times, and number of queries that returned a result within 10 minutes. However, the exhaustive experimentation required to identify suitable partition factors is infeasible. Both approaches do not scale well for queries that have low selectivity, even for relatively small XML datasets, e.g. *115 MB*, the query response times are relatively large.

5.2 Comparison Tests with Vendor Systems

In this section, we will evaluate the branch index against a leading commercial XML database solution (*Microsoft SQL Server 2008*) and a leading open source XML database (*MonetDB/XQuery*) [2] using the XPathMark [13] benchmark.

The XPathMark Benchmark. The standard XPath benchmark (XPath-Mark [13]) consists of a number of categories of queries across the *synthetic*



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
BranchIndex	1,515	1,625	581	2,347	5,893	11226	3,297	5273	1,033	3277
MonetDB/XQuery	2025	114	1046	8,954	9036	49,905	11350	11084	10,788	10774
SQL Server			317363	20095	14719		284436	5443	5,711	5699

Fig. 7. XMark Query Response Times

XMark dataset. In this paper, we are examining the performance of the *ancestor*, *ancestor-or-self*, *descendant*, *descendant-or-self*, *parent* and *child* axes. The queries in Table 4 where chosen for this purpose.

Fig 7 shows the following:

- SQL Server threw an exception on `Q6` as it contains the ancestor axes.
- `Q1` and `Q2` have high selectivity (discussed earlier), thus all three systems took a small amount of time to return the result.
- In queries `Q3`, `Q4`, `Q6`, `Q7`, and `Q9` the BranchIndex shows orders of magnitude improvements over the times returned by SQL Server and MonetDB/XQuery.
- In queries `Q5`, the branch index is almost twice as efficient as MonetDB/XQuery and three times as efficient as SQL Server.
- In `Q8` and `Q10`, the BranchIndex and SQL Server returned similar times, and MonetDB/XQuery took twice as long.

The branch index is the preferred option across all queries except `Q2`, in which case the time difference is negligible. SQL Server performs well across queries that have multiple parent-child edges, e.g. `Q8` `Q9` and `Q10`, which we attribute to the *secondary* `PATH` index we built. For instance, SQL Server performs very poorly in `Q3`, which has an *ancestor-descendant* join on the third step. MonetDB/XQuery is quite consistent across all queries, i.e. taking around 10/11 seconds across all low selectivity queries. However, it performs particularly poorly in `Q6`, which could indicate that it does not evaluate the ancestor axis efficiently.

6 Conclusions

In this paper, we presented a partitioning approach for XML documents. These partitions are used to create an index that optimizes XPath's hierarchical axes. Our approach differs from the only major effort in this area in that we do not need to analyze the document in advance to determine efficient partition sizes. Instead, our algorithms are dynamic, thus they create partitions based on document characteristics, e.g. structure and node layout. This provides for a fully automated process for creating the partition index. We obtain further optimization by compacting the partition index using a classification process. As each

identical partition will generate identical results in query processing, we need only a representative partition (a branch class) for all partitions of equivalent structure. We then demonstrated the overall optimization gains through experimentation. Our current work focuses on evaluating non-hierarchical XPath axes, e.g. following, preceding, and on using real-world datasets (sensor-based XML output) to test different XML document formats and to utilize real world queries to understand the broader impact of our work.

References

1. Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. ACM Trans. Internet Technol. 1(1), 110–141 (2001)
2. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: SIGMOD 2006: Proceedings of the, ACM SIGMOD international conference on Management of data, pp. 479–490. ACM Press, New York (2006)
3. Marks, G., Roantree, M., Murphy, J.: Classification of Index Partitions. Technical report, Dublin City University (2010), http://www.computing.dcu.ie/~isg/publications/ISG-10-03.pdf
4. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., Loucopoulos, P., Jeusfeld, M.A. (eds.) Proceedings of 23rd International Conference on Very Large Data Bases, VLDB 1997, Athens, Greece, August 25-29, pp. 436–445. Morgan Kaufmann, San Francisco (1997)
5. Grust, T.: Accelerating XPath Location Steps. In: SIGMOD 2002: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 109–120. ACM, New York (2002)
6. Grust, T., Rittinger, J., Teubner, J.: Why off-the-shelf RDBMSs are better at XPath than you might expect. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 949–958. ACM Press, New York (2007)
7. Grust, T., van Keulen, M., Teubner, J.: Staircase Join: Teach a Relational DBMS to Watch Its (axis) Steps. In: VLDB 2003: Proceedings of the 29th international conference on Very large data bases, pp. 524–535. VLDB Endowment (2003)
8. Lu, J., Chen, T., Ling, T.W.: Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In: CIKM 2004: Proceedings of the thirteenth ACM international conference on Information and knowledge management, pp. 533–542. ACM, New York (2004)
9. Lu, J., Chen, T., Ling, T.W.: TJFast: Effective Processing of XML Twig Pattern Matching. In: WWW 2005: Special interest tracks and posters of the 14th international conference on World Wide Web, pp. 1118–1119. ACM, New York (2005)
10. Luoma, O.: Xeek: An efficient method for supporting xpath evaluation with relational databases. In: ADBIS Research Communications (2006)
11. Luoma, O.: Efficient Queries on XML Data through Partitioning. In: WEBIST (Selected Papers), pp. 98–108 (2007)
12. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. In: SIGMOD 2004: Proceedings of the, ACM SIGMOD international conference on Management of data, pp. 903–908. ACM, New York (2004)
13. XPathMark Benchmark. Online Resource, http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/