Index Structures for Matching XML Twigs Using Relational Query Processors

Zhiyuan Chen
University of Maryland at Baltimore County
zhchen@umbc.com

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Flip Korn
AT&T Labs–Research
flip@research.att.com

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

Jayavel Shanmugasundaram
Cornell University
jai@cs.cornell.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

Various index structures have been proposed to speed up the evaluation of XML path expressions. However, existing XML path indices suffer from at least one of three limitations: they focus only on indexing the structure (relying on a separate index for node content), they are useful only for simple path expressions such as root-to-leaf paths, or they cannot be tightly integrated with a relational query processor. Moreover, there is no unified framework to compare these index structures. In this paper, we present a framework defining a family of index structures that includes most existing XML path indices. We also propose two novel index structures in this family, with different space-time tradeoffs, that are effective for the evaluation of XML branching path expressions (i.e., twigs) with value conditions. We also show how this family of index structures can be implemented using the access methods of the underlying database system. Finally, we present an experimental evaluation that shows the performance tradeoff between index space and matching time. The experimental results show that our novel indices achieve orders of magnitude improvement in performance for evaluating twig queries, albeit at a higher space cost, over the use of previously proposed XML path indices that can be tightly integrated with a relational query processor.

1. Introduction

XML employs a tree-structured model for representing data. Quite naturally, queries in XML query languages (see, e.g., [26]) typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. For example, the XQuery path expression:

```
/book[title='XML']//author[fn='jane' and ln='doe']
```

matches `author` elements that (i) have a child subelement `fn` with content `jane`, (ii) have a child subelement `ln` with content `doe`, and (iii) are descendants of (root) `book` elements that have a child `title` subelement with content `XML`. This expression can be represented naturally as a node-labeled twig pattern with elements and string values as node labels as shown in Figure 1(c).[1]

Finding all occurrences of a twig pattern in an XML database is a core operation in XML query processing, both in relational implementations of XML databases [7, 8, 23], and in native XML databases [9, 19, 20]. Prior solutions to this problem use a combination of indexing [4, 5, 10, 12, 18], link traversal [11, 17] and join techniques [1, 16, 30].

The focus of this paper is on developing index structures that can support the *efficient* evaluation of XML *ad hoc, recursive, twig* queries using a *relational database system*. By efficient, we mean that every fully specified, single-path XML query (without any branches and arbitrary recursion) should be answerable using a single index lookup; in particular, potentially expensive join operations should be avoided. By ad hoc queries, we mean that the index structures should be able to perform well even if the expected query workload is unknown; we believe that this feature is especially important for semi-structured databases, where user queries may be exploratory. Support for recursive queries means that the index structures should support queries having "//" (i.e., ancestor-descendant relationships) efficiently (though not necessarily in a single lookup). Support for twig queries means that the index structures should be able to process

1 IDREFs are encoded and queried as values in XML. Hence, we do not consider IDREFs as part of the twig pattern.

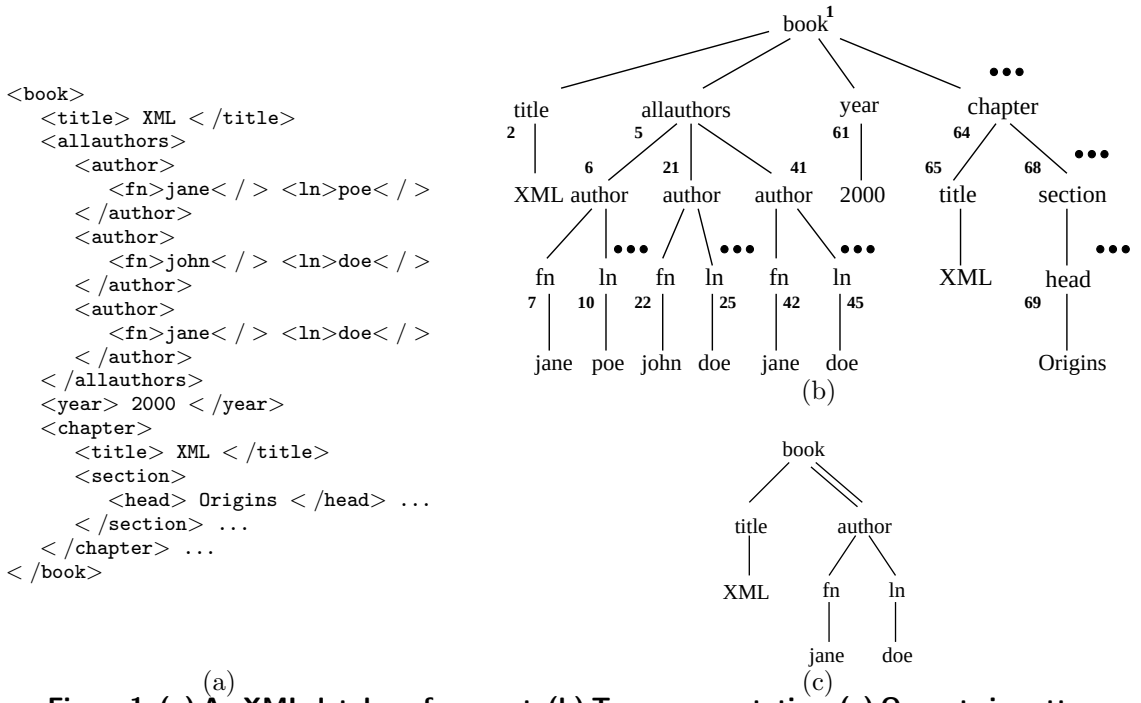


Figure 1. (a) An XML database fragment, (b) Tree representation, (c) Query twig pattern

branching path queries without significant additional overhead (compared to single-path queries). Finally, since XML data may often be stored in relational database systems in the future, we also require that the index structures be easily implemented in existing relational database systems, and tightly integrated with relational query processors.

While previously proposed XML path indices (see, e.g., [4, 5, 10, 12, 18, 21, 25]), relational join indices [24], and object-oriented path indices (see, e.g., [2, 14, 27]) do address some of these aspects in isolation, we are not aware of any index structure that handles all of these issues within a unified framework (see Section 2 for more details). Further, some existing index structures [5, 16] require either special index structures or join algorithms not available in today's systems, while others [21, 25] use existing relational access methods in unconventional ways that cannot be tightly integrated with relational query processors.

In this paper, we develop index structures that address the above requirements, and provide *orders of magnitude* improvement in performance over the use of existing indices for evaluating twig queries and recursive queries, while remaining competitive for fully specified, single-path queries. Specifically, the contributions of this paper are:

- A unified framework for XML path indices including most existing ones.
- Two novel index structures `ROOTPATHS` and `DATAPATHS` that are effective for the evaluation of ad hoc, recursive, twig queries.
- Techniques for implementing the family of index structures using the access methods of a relational database system, to support tight integration with relational query processors.
- An extensive experimental evaluation to compare our proposed indices with existing XML path indices, and to understand the performance tradeoff between index space and twig matching time.

The rest of this paper is organized as follows. We formally define the indexing problems we address in Section 3. We then review related work and demonstrate that it is not sufficient for solving these indexing problems in Section 2. In Section 4, we define the family of indices and propose two novel index structures. We present our experimental results in Section 5 and conclude in Section 6.

2. Related Work

The works in [4, 11, 10, 12, 13, 18] focus on indexing XML paths, *excluding the data values at the ends of the paths*. Thus they require a potentially expensive join operation or multiple index lookups because the data value is indexed separately from the path.The Index Fabric [5] indexes XML paths and data items together. However, if precise information about the query workload is not available, the Index Fabric cannot support branching queries efficiently. Moreover, the Index Fabric does not support recursive queries efficiently.

Recently, the ViST [25] and PRIX [21] techniques have been proposed which encode XML documents and queries as sequence patterns, and perform sub-sequence matching to answer twig queries. A consequence of the sub-sequence matching is that ViST and PRIX require multiple index lookups even for fully-specified single-path expressions. Further, since sub-sequence matching is not directly supported in a relational database system, the authors propose implementing these sophisticated strategies using special-purpose application logic that is opaque to the relational query engine and query optimizer. Thus, unlike our proposed approach, these techniques cannot be tightly integrated with a relational query processor.

XML path indexing is also related to the problem of join indexing in relational database systems [24] and path indexing in object-oriented database systems [2, 14, 27]. These index structures are targeted at workloads consisting of single path queries without recursion, and assume that the schema is fixed and known. These assumptions do not hold for XML queries, and we show the limitations of these previous approaches, especially for recursive queries, in the full version of our paper [3].

There are also recent approaches for indexing XML paths using a relational database [22, 29]. The ToXin approach [22] builds XML indices similar to Access Support Relations (ASR) [14] and Join Indices [24], thus have the same problem as ASR/Join Indices. The XRel approach [29] stores the actual paths in a different table, thus recursive queries require *multiple index lookups*: one to look up the path ids of the paths, and more to look up the results for each path id.

3. Preliminaries

3.1. Query Twig Patterns

An XML database is a forest of rooted, ordered, labeled trees, each node corresponding to an element, attribute, or a value, and the edges representing (direct) element-subelement, element-attribute, element-value, and attribute-value relationships. Non-leaf nodes correspond to elements and attributes, and are labeled by the tags or attribute names, while leaf nodes correspond to values. For the sample XML document of Figure 1(a), its tree representation is shown in Figure 1(b). Each non-leaf node is associated with a unique numeric identifier, shown beside the node.

Queries in XML query languages like XQuery [26] make fundamental use of (node-labeled) twig patterns for matching relevant portions of data in the XML database. The node labels include element tags, attribute names, and values; and the edges are either parent-child edges (depicted by a single line) or ancestor-descendant edges (depicted by a double line). For example, the XQuery path expression in the introduction can be represented as the twig pattern in Figure 1(c). In this paper, we assume all values are strings and only equality matches on the values are allowed in the query twig pattern.

In general, given a query twig pattern Q, and an XML database D, a *match* of Q in D is identified intuitively by a mapping from nodes in Q to nodes in D, such that: (i) query node tags/attribute-names/values are preserved under the mapping, and (ii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes.

3.2. Subpaths and PCsubpaths

A twig pattern consists of a collection of subpath patterns, where a *subpath pattern* is a subpath of any root-to-leaf path in the twig pattern. For example, the twig pattern "`/book[title = 'XML']//author[fn = 'jane' and ln = 'doe']`" consists of the paths "`/book[title = 'XML']`", "`/book//author[fn = 'jane']`", and "`/book//author[ln = 'doe']`". Each of these is a subpath pattern, as are "`/book/title`" and "`//author[fn = 'jane']`".

A subpath pattern is said to be a *parent-child subpath* (or *PCsubpath*) *pattern* if there

are no ancestor-descendant relationships between nodes in the subpath pattern (a "//" at the beginning of a subpath pattern is permitted). Thus, among the above subpath patterns, each of "/book[title = 'XML']", "/book/title", and "//author[fn = 'jane']" is a PCsubpath pattern. However, neither "/book//author[fn = 'jane']" nor "/book//author[ln = 'doe']" is a PCsubpath pattern. The importance of making this distinction will become clear when we formally define the indexing problems addressed in this paper.

3.3. Problem: PCsubpath Indexing

To answer a query twig pattern Q, it is essential to find matches to a set of subpath patterns that "cover" the query twig pattern. Once these matches have been found, join algorithms can be used to "stitch together" these matches. For example, one can answer the query twig pattern in Figure 1(c) by finding matches to each of the subpath patterns "/book[title = 'XML']", "//author[fn = 'jane']" and "//author[ln = 'doe']", and combining these results using containment joins [1, 30]. Alternatively, if there are few XML books, one could first find all book ids matching "/book[title = XML]", then use the book ids to selectively probe for authors that match the subpath patterns "//author[fn = 'jane']" and "//author[ln = 'doe']" rooted at each book id. Note that matches to the branching point book are needed, even though this node is not in the result of the query twig pattern. It is easy to see that any query twig pattern can always be covered by a set of PCsubpath patterns. This motivates the two indexing problems we address in this paper:

Problem FreeIndex: Given a PCsubpath pattern P with n node labels and an XML database D, return all n-tuples $(d_1, \ldots, d_n)$ of node ids that identify matches of P in D, in a *single* index lookup.

An index solving the FreeIndex problem can be used to retrieve ids of branch nodes or nodes in the result. For example, consider query "/book/allauthors/author[fn = 'jane' and ln = 'doe']". A lookup for the PCsubpath "/book/allauthors/author[fn = 'jane']" in the database in Figure 1 gives the id lists ([1,5,6,7], [1,5,41,42]), and author-id is the penultimate id in each of the lists. Similarly, a lookup on "/book/allauthors/author[ln = 'doe']" gives the id lists ([1,5,21,25], [1,5,41,45]). Since author id 41 is present in both cases, the selected author can be returned via merge or hash join, both of which are commonly supported by relational query processors.

Problem BoundIndex: Given a PCsubpath pattern P with n node labels, an XML database D, and a specific database node id d, return all n-tuples $(d_1, \ldots, d_n)$ that identify matches of P in D, rooted at node d, in a *single* index lookup.

BoundIndex problem is useful because it allows the index-nested-loop join processing strategy in relational systems to be used. For example, given query "/book[title='XML']//author[ln = 'doe']", and suppose we have evaluated PCsubpath "/book[title='XML']" and found the book id $d = 1$. Then an index that can solve the BoundIndex problem can be used in index-nested-loop join to return the "author" id under "book" id 1 and satisfying the PCsubpath pattern "//author[ln = 'doe']". The FreeIndex problem can be seen as a special case of the BoundIndex problem when the root node id d is not given.

4. A Family of Indices

In this section, we will present a unified framework defining the family of indices solving the FreeIndex and BoundIndex problems. This framework covers most existing path index structures. We also propose two novel index structures: ROOTPATHS and DATAPATHS.

4.1. Framework

We first introduce some notation. *Data paths* in the XML data consists of two parts: (i) a *schema path*, which consists solely of schema components, i.e., element tags and attribute names, and (ii) a *leaf value* as a string if the path reaches a leaf. Schema paths can be dictionary-encoded using special characters (whose lengths depend on the dictionary size) as designators for the schema components.

In order to solve the BoundIndex problem (which is a more general version of the FreeIndex problem), one needs to explicitly represent data paths that are arbitrary *subpaths* (not just prefix subpaths) of the root-to-leaf paths, and associate each such data path with the node at which the subpath is rooted. Such a relational representation of

HeadId	SchemaPath	LeafValue	IdList
1	**B**	null	[]
1	**BT**	null	[2]
1	**BT**	XML	[2]
1	**BU**	null	[5]
1	**BUA**	null	[5,6]
1	**BUAF**	null	[5,6,7]
1	**BUAF**	jane	[5,6,7]
1	**BUAL**	null	[5,6,10]
1	**BUAL**	poe	[5,6,10]
	...		
5	**U**	null	[]
5	**UA**	null	[6]
5	**UAF**	null	[6,7]
5	**UAF**	jane	[6,7]
5	**UAL**	null	[6,10]
5	**UAL**	poe	[6,10]
	...		

Figure 2. The 4-ary relation

all the data paths in an XML database is (`HeadId, SchemaPath, LeafValue, IdList`), where `HeadId` is the id of the start of the data path, and `IdList` is the list of all node identifiers along the schema path, except for the `HeadId`.

As an example, a fragment of the 4-ary relational representation of the data tree of Figure 1(b) is given in Figure 2, where the element tags have been encoded using boldface characters as designators, based on the first character of the tag, except for `allauthors` which uses **U** as its designator.

We define the family of indices solving the FreeIndex and BoundIndex problems as follows:

Family of Indices: Given the 4-ary relational representation of XML database D, the family of indices include all indices that:

1. store a subset of all possible `SchemaPaths` in D;
2. store a sublist of `IdList`;
3. index a subset of the columns `HeadId`, `SchemaPath`, and `LeafValue`.

Given a query, the index structure probes the indexed columns in (3) and returns the sublist of `IdList` stored in the index entries.

Many existing indices fit in this framework, as summarized in Figure 3. For example, the Index-Fabric [5] returns the ID of either the root or the leaf element (first or last ID in `IdList`), given a root-to-leaf path and the value of the leaf element.

There are also many possible indices belonging to the family that have not been explored yet. For example, all existing indices return the first or last IDs in the `IdList`, but do not return other IDs. Also, none of them index both `HeadID` and `SchemaPaths` with length larger than one. Consequently, none of the existing index structures can answer the FreeIndex or BoundIndex problem with a single index lookup. For example, consider the query "`/book/allauthors/author[fn = 'jane' and ln = 'doe']`". The FreeIndex problem requires the "author" ID given "`/book/allauthors/author[fn = 'jane']`". Using Index Fabric, one can find all IDs of "fn" satisfying "`/book/allauthors/author[fn = 'jane']`", but the author ID is not returned.

We now propose two novel index structures in this family, `ROOTPATHS` and `DATAPATHS`, which can answer the FreeIndex and BoundIndex problems with one index lookup, respectively.

4.2. `ROOTPATHS` Index

`ROOTPATHS` is a B+-tree index on the concatenation of `LeafValue` and the reverse of `SchemaPath`, and it returns the complete `IdList`. Only the prefixes of the root-to-leaf paths are indexed (i.e., only those rows with `HeadID` = 1).

There are two main differences between `ROOTPATHS` and the Index Fabric. The first difference is that `ROOTPATHS` stores the prefix paths in addition to root-to-leaf paths. This efficiently supports queries that do not go all the way to a leaf (e.g., "`/book`"). The second extension is to store the entire `IdList`, i.e., all node identifiers along the schema path[2], as opposed to storing only the document-id or leaf-id of the path as is done in the Index Fabric. The `IdList` extension is key to evaluating branching queries efficiently using relational query processors, at an additional space cost, because it gives the ids of the branch points in a single index lookup.

We now show how a regular B+-tree index can be used to support PCsubpath queries with initial "`//`". We need to permit *suffix* matches on the `SchemaPath` attribute (with exact matches on the `LeafValue` attribute, if any). The key observation is that, although B+-trees are not efficient at suffix matches, they are very efficient for prefix matches. Consequently, if we just *reverse* the `SchemaPath` values to be indexed (e.g., `FAUB` instead of `BUAF` in

2 The node identifiers used in this paper are simple numeric values, which suffice for subsequent sort-merge joins, and index-nested-loop joins. Alternative identifiers such as those in [30] can be used, to enable containment queries.

Index	Subset of SchemaPath	Sublist of IdList	Indexed Columns
Value [17]	paths of length 1	only last ID	SchemaPath, LeafValue
Forward link [17]	paths of length 1	only last ID	HeadId, SchemaPath
DataGuide [10]	root-to-leaf path prefixes	only last ID	SchemaPath
Index Fabric [5]	root-to-leaf paths	only first or last ID	SchemaPath, LeafValue
ROOTPATHS	root-to-leaf path prefixes	full IdList	LeafValue, reverse SchemaPath
DATAPATHS	all paths	full IdList	LeafValue, HeadId, reverse SchemaPath

Figure 3. Members of Family of Indices

Figure 2), a regular B+-tree can be used to support suffix matches. This observation has also previously been used in the string indexing community for matching string suffixes.

A B+-tree index on the concatenation LeafValue·ReverseSchemaPath in the ROOTPATHS relation can be used to directly match PCsubpath patterns with initial recursion, such as "//author[fn='jane']" in a single index lookup. This is done by looking up on the key ('jane', FA*). Similarly, PCsubpath patterns with initial recursion, but without a condition on the leaf value, such as "//author/fn" can be looked up on the key (null, FA*). Neither the Index Fabric nor the DataGuide can support the evaluation of such queries efficiently. Fully specified PCsubpaths (without an initial "//") can also be handled using this index.

4.3. DATAPATHS Index

The DATAPATHS index is a regular B+-tree index on the concatenation of HeadId, LeafValue and the reverse of SchemaPath (or the concatenation LeafValue·HeadId·ReverseSchemaPath), where the SchemaPath column stores all subpaths of root-to-leaf paths, and the complete IdList is returned.

DATAPATHS index can solve both the FreeIndex and the BoundIndex problems in one index lookup.[3] For example, consider query "/book//author[fn = 'jane' and ln = 'doe']". One can use the index to probe all book-ids that match "/book", which is a FreeIndex problem. Using these book-ids as HeadId values, one can solve the BoundIndex problem by probing author-id matches to each of the two PCsubpaths "//author[fn = 'jane']" and "//author[ln = 'doe']", rooted at the book-ids. Finally the intersection of these two sets of author-id matches is the answer of the query. Alternative plans, enabled by the DATAPATHS index, are also possible. Note that the initial recursion in these PCsubpaths necessitate the use of ReverseSchemaPath in the BoundIndex.

The DATAPATHS index is bigger than ROOTPATHS, but is exactly what is needed to solve the BoundIndex problem in one index lookup.

We also explore lossless and lossy (meaning the compressed index can only be used to answer a given workload) compression methods to reduce the space usage of ROOTPATHS and DATAPATHS. We implemented differential encoding compression for IdList. For detail discussion about compression, please refer to [3] for details.

5. Experimental Evaluation

We present an experimental evaluation of the ROOTPATHS and DATAPATHS indices with the existing index structures in the same family. Please refer to [3] for comparison of our approach against Access Support Relations [14] and Join Indices [24].

5.1. Experimental Setup

We assume the XML data is stored in an Edge Table [8] in IBM's DB2 version 7.2 relational database management system. The Edge table approach is selected because any XML data can be stored using this approach. We used a 100MB scaled XMark data [28] and a 50 MB DBLP data [6]. Our experiments were performed using a 1.7 GHz Pentium machine running Windows 2000, with 1GB memory and a 37 GB disk, and a 40MB buffer pool with operating system cache turned off. The reported query

3 In our implementation, we added a virtual root as the parent of all XML documents, thus the index can solve FreeIndex as well (by letting the HeadId be the virtual root).

execution time are the average of 10 runs with a cold cache, excluding query optimization time. The cost of translating the XPath query to SQL is considered part of the query optimization cost.

We used a workload of 15 XPath queries on XMark and 3 queries on DBLP (because DBLP is too shallow for testing), and varied the parameters of the query such as the number of branches, the selectivity of each branch, and the depth of branches. Figure 5 summarizes these queries. Details of these queries can be found at [3].

We implemented five different indexing strategies for our experiments: `ROOTPATHS (RP)` and `DATAPATHS (DP)` (both with differential encoding on `IdList`), simulated DataGuide `(DG)` and simulated Index Fabric `(IF)` using B+-tree index, Edge Table index with the value index, forward link, and backward link index as described in [17]. We use regular B+-tree indices in this paper to simulate Index Fabric. DB2 has implemented prefix compression on indexed columns to reduce the key size. Thus *regular* B+-tree indices are also space efficient.

Since the DataGuide and the Index Fabric do not store `IdLists`, they cannot be directly used to answer twig queries. Consequently, we experimented with various query plans where we used the DataGuide/Index Fabric to look up ids at the end of root-to-leaf paths, then we used (possibly many lookups in) the reverse link index on Edge Table to determine the branch point ids from the leaf ids, and choose the best query plan. We refer to these combined strategies as DG+Edge and IF+Edge.

Figure 4 gives the space requirement for the various index structures. Since XMark data is more deeply nested than DBLP, the space requirements for `DATAPATHS` increase proportionally.

5.2. Experimental Results

5.2.1. Indexing Schema Paths and Values Together We examine the benefit of indexing schema paths and data values together by choosing a single fully-specified path query, and varying it from highly selective ($Q1_d$, $Q1_x$), to moderately selective ($Q2_d$, $Q2_x$), to relatively unselective ($Q3_d$, $Q3_x$). Figure 6 shows the performance of various index structures (XMark on the left, DBLP on the right). The Index Fabric and `ROOTPATHS` are among the best approaches, while `DATAPATHS` is only slightly worse. Meanwhile the Edge and DataGuide+Edge approaches perform badly with decreasing selectivity.

The good performance of Index Fabric is expected because it is optimized for simple path queries. `ROOTPATHS` suffers a slight overhead because it stores `IdLists` instead of just Ids, and also incurs the cost of invoking a user-defined function to extract the ids. Similarly, `DATAPATHS` is slightly worse than `ROOTPATHS` because it has the overhead of storing both `IdLists` and `HeadId`.

Edge performs badly because it performs a join operation for each step along the path. As the selectivity of paths decreases, it increases the cost of each join. The bad performance of Edge is a simple justification for using a single index lookup instead of resorting to more expensive joins.

The most interesting aspect of the figure, however, is the bad performance of DataGuide+Edge. The main reason for this behavior is that schema paths are indexed separately from the data values. Consequently, the results for schema paths and data values have to be joined together. As the selectivity of paths decreases, the cost of each join increases, resulting in bad performance.

5.2.2. Returning `IdLists` We now examine the performance benefits of returning `IdLists` for twig queries. We study three groups of queries, one in which all branches are selective, one is which all branches are unselective, and one in which there are selective and unselective branches. For each group, we vary the number of branches.

We used queries $Q4_x$ (2 branches) and $Q5_x$ (3 branches) to evaluate the performance of queries with all selective branches. In addition, we also used a single path selective query (chosen as the first branch common to $Q4_x$ and $Q5_x$) as a baseline for comparison. Similarly, we used $Q6_x$ (2 branches) and $Q7_x$ (3 branches) to evaluate the performance of queries with a mix of selective and unselective branches, and $Q8_x$ (2 branches) and $Q9_x$ (3 branches) for queries with all unselective branches. For all these queries, the branch point is high in the query. The results for DBLP are similar and omitted due to space restrictions.

Figures 7(a), (b), and (c) show the performance results for the different groups of queries. `ROOTPATHS` and `DATAPATHS` scale gracefully both with respect to the number of branches and with respect to the selectivity of these branches. However, the Index Fabric, DataGuide and Edge approaches perform badly

Data set	*RP*	*DP*	*Edge*	*DG+Edge*	*IF+Edge*	*ASR*	*JI*
XMark	119	431	127	169	167	464	822
DBLP	80	83	106	133	151	93	318

Figure 4. Space (in MB) for different indices

Query	*Branches*	*Result Size Per Branch*	*Depth of Branches*	*Recursions*
$Q1_x$ to $Q3_x$	1	1-11062	–	0
$Q1_d$ to $Q3_d$	1	1-10258	–	0
$Q4_x$ to $Q9_x$	2-3	1-7519	High	0
$Q10_x$ to $Q11_x$	2-3	3-59486	Low	0
$Q12_x$ to $Q15_x$	2-3	41-20946	Low	1

Figure 5. Queries

in both regards (note the log scale on the graphs).

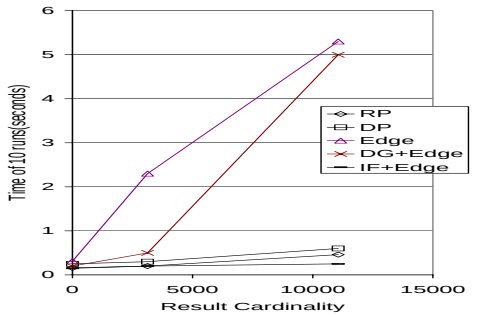
`ROOTPATHS` and `DATAPATHS` perform so well because they store `IdLists`. Hence, they can do an index lookup for each path, extract the ids of the branch point from the `IdLists`, and do a join on the branch points to produce the desired result. With increasingly unselective predicates, more ids will need to be extracted, thereby explaining the slightly higher running times as the selectivity of paths decreases. In all cases, however, the running time of the two approaches is well under a second. The reason that `DATAPATHS` performs slightly worse than `ROOTPATHS` in Figures 7(a) and 7(c) is that in these cases the selectivities are roughly the same and thus the speedup from index-nested-loops join cannot be exploited. (The index-nested-loops join strategy is effective when one branch is selective whereas the other branches are unselective.) Since a sort-merge join is performed for both, `DATAPATHS` offers no benefit over `ROOTPATHS`.

In contrast, the performance of the Edge table, DG+Edge, and IF+Edge approaches is many orders of magnitude worse, both when the number of branches increases and when the selectivity of the branches decreases. In fact, for unselective queries with three branches, the execution time for these approaches was more than 10 minutes. This phenomenon occurs because, in the absence of `IdLists`, these approaches have to perform expensive joins to determine the relationship between the path leaves and the branch points. Since the branch points were high for this set of experiments, they had to perform a 5-way join for each branch. While the joins are expensive enough to do for selective branch queries, performance degrades dramatically in the presence of unselective branches.

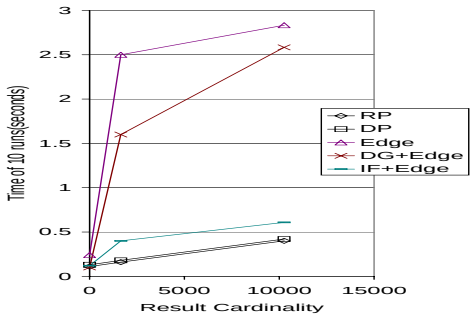
It is also interesting to note some limitations of relational systems in evaluating many joins. The time that DB2 took to *optimize* the queries was longer than the time it took to *execute* the queries using the `ROOTPATHS` and `DATAPATHS` (the graphs only show the execution time). The relational optimizer also understandably made some wrong decisions for queries with a large number of joins, which further contributed to the bad performance of Index Fabric, DataGuide and Edge. Thus `IdLists` are valuable both for reducing the overhead of performing joins, and for simplifying the generated query to enable better optimization.

5.2.3. Benefit of Index-nested-loop Join We now vary the branching point of the twig queries so that they branch closer to the leaves (recall that we used branching points close to the root for the previous set of experiments). We use $Q10_x$ (2 branches) and $Q11_x$ (3 branches) for the XMark data. Both queries have one selective path and other unselective paths. The performance results are shown in Figure 7(d). The results for DBLP are similar and are omitted.

As before, `DATAPATHS` performs uniformly well, while Index Fabric, DataGuide and Edge perform poorly as the number of branches increase. The performance of these three approaches, while still up to orders of magnitude worse than `DATAPATHS`, is better than the case when the branches are deeper because the number of joins required to determine the branch point is lower for this set of experiments.

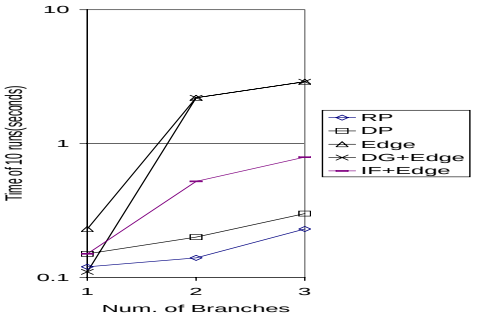


(a) XMark

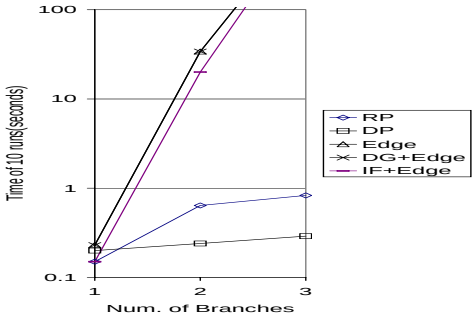


(b) DBLP

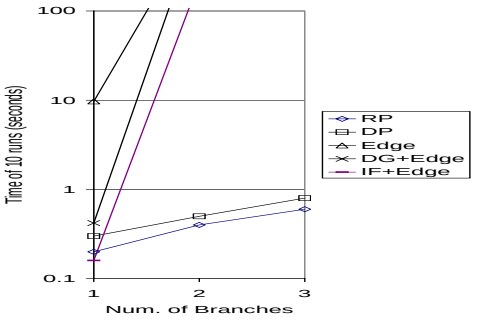
Figure 6. Increasing selectivity for single path queries



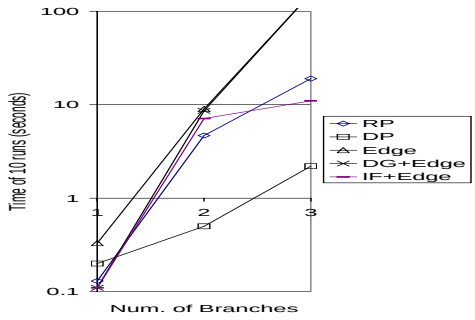
(a) Twig queries with selective branches



(b) Twig queries with selective and unselective branches



(c) Twig queries with unselective branches



(d) Twig queries with low branch points.

Figure 7. XMark twig queries without recursion

The most surprising result here is the relatively bad performance of `ROOTPATHS` (it is even worse than IF+Edge at a point). The reason for this degradation of performance is that `ROOTPATHS` does not support the index-nested-loop join strategy while the other indices do. The index-nested-loop join strategy is much better for this set of queries because (a) one branch is very selective, (b) other branches are unselective, and (c) each selective branch matches with only very few unselective branches. Condition (c) was not satisfied earlier for the queries with deep branches because they branch at nodes closer to the root, which usually have a large number of descendants.

5.2.4. Recursive Queries We now examine the performance of evaluating recursive ("`//`") queries. The recursive queries are exactly the same

as queries used in Section 5.2.2 except that each query now starts with a "//". To examine the overhead for recursive queries, we compare the performance of ROOTPATHS and DATAPATHS for original queries which do not have a recursion. (Other indices cannot be used here.) We found that ROOTPATHS and DATAPATHS have less than 5% overhead for processing recursive queries because such queries can be converted into B+-tree prefix match queries on ReverseSchemaPaths. Please refer to [3] for details.

6. Conclusion

We have described a family of index structures, with different space-time tradeoffs, for the efficient evaluation of ad hoc, recursive, twig queries. The proposed index structures are enabled by a simple relational representation of the XML data paths. This permits conventional use of existing relational index structures (e.g., B+-trees) for the twig indexing problem, and can thus be tightly coupled with a relational optimizer and query evaluator. We will study efficient update techniques in the future.

References

[1] S. Al-Khalifa, et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] E. Bertino, W. Kim. Indexing techniques for queries on nested objects. In *IEEE TKDE*, 1(2), 1989.

[3] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugsundaram, D. Srivastava. Index Structures for Matching XML Twigs Using Relational Query Processors. *Cornell Technical Report TR2004-1961*, 2004. Available at http://techreports.library.cornell.edu.

[4] C.-W. Chung, J.-K. Min, K. Shim. APEX: An adaptive path index for XML data In *SIGMOD*, 2002.

[5] B. F. Cooper, et al. A fast index for semistructured data. In *VLDB*, 2001.

[6] DBLP. http://www.informatik.uni-trier.de/~ley/db/index.html.

[7] A. Deutsch, M. Fernandez, D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.

[8] D. Florescu, D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[9] R. Goldman, J. McHugh, J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *WebDB Workshop*, 1999.

[10] R. Goldman, J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[11] T. Grust. Accelerating XPath location steps. In *SIGMOD*, 2002.

[12] R. Kaushik, et al. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[13] R. Kaushik, et al. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[14] A. Kemper, G. Moerkotte. Access support in object bases. In *SIGMOD*, 1990.

[15] D. D. Kha, M. Yoshikawa, S. Uemura. An XML indexing structure with relative region coordinate. In *ICDE*, 2001.

[16] Q. Li, B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, 2001.

[17] J. McHugh, J. Widom. Query optimization for XML. In *VLDB*, 1999.

[18] T. Milo, D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[19] U. of Michigan. The TIMBER system. Available from http://www.eecs.umich.edu/db/timber/.

[20] J. Naughton, et al. The Niagara Internet Query System. In *IEEE Data Engineering Bulletin*, 24(2), 2001.

[21] P. Rao, B. Moon. PRIX: Indexing and Querying XML Using Pruffer Sequences. In *ICDE*, 2004.

[22] F. Rizzolo, A. Mendelzon. Indexing XML data with ToXin. In *WebDB Workshop*, 2001.

[23] J. Shanmugasundaram, et al. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[24] P. Valduriez. Join indices. In *ACM TODS*, 12(2), 1987.

[25] H. Wang, S. Park, W. Fan, P. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *SIGMOD*, 2003.

[26] World Wide Web Consortium. XQuery: A query language for XML. Available from http://www.w3.org/TR/xquery.

[27] Z. Xie, J. Han. Join index hierarchies for supporting efficient navigations in object-oriented systems. In *VLDB*, 1994.

[28] XMark The XML benchmark project. http://monetdb.cwi.nl/xml.

[29] M. Yoshikawa, et al. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. In *ACM TOIT*, 1(1): 110–141, 2001.

[30] C. Zhang, et al. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.