Index Structures for Matching XML Twigs Using Relational Query Processors

Zhiyuan Chen
Microsoft Research
zhchen@microsoft.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), or 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, including 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 realized using the access methods of the underlying database system. Finally, we present an experimental evaluation to understand the performance tradeoff between index space and twig 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., [8, 4, 30]) 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 [10, 9, 26, 27], and in native XML databases [11, 23, 22]. Prior solutions to this problem use a combination of indexing [12, 21, 6, 5, 14], link traversal [20, 13] and join techniques [34, 1, 3, 18].

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

[1] IDREFs are encoded and queried as values in XML. A link through an IDREF is treated as a value-based join of the IDREF value(s) and the (corresponding) ID value(s). Hence, we do not consider IDREFs as part of the twig pattern.

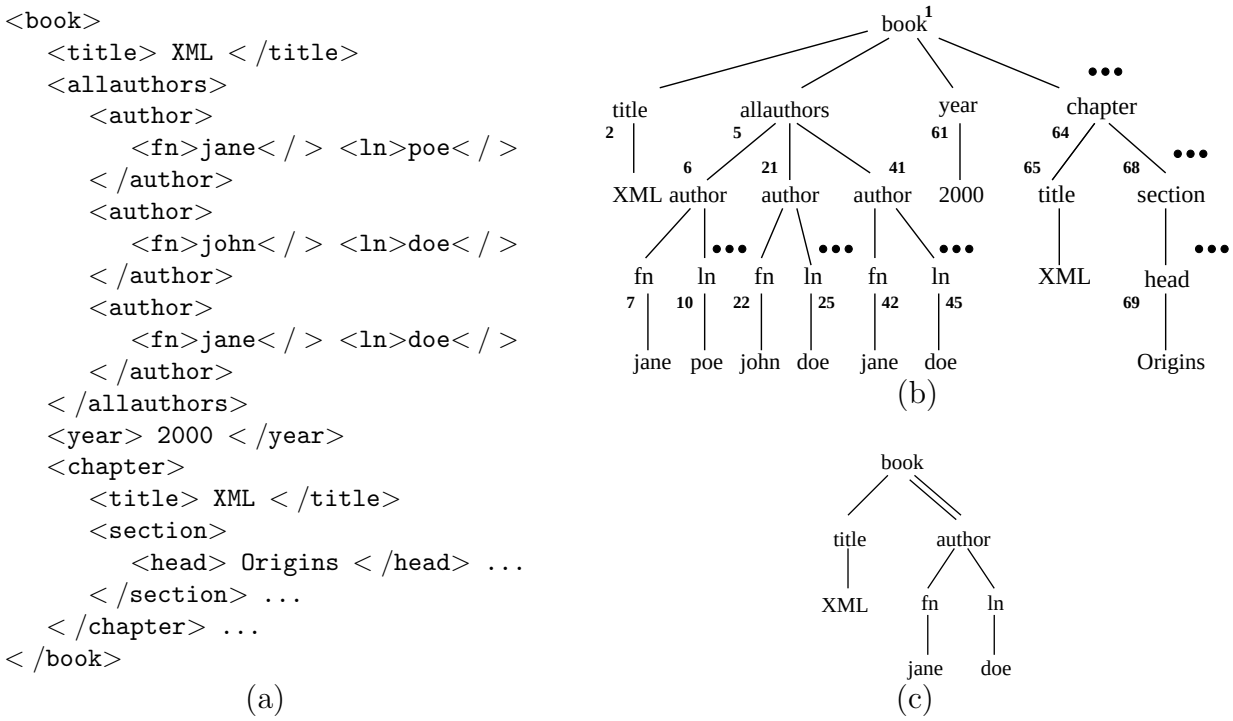


Figure 1: (a) An XML database fragment, (b) Tree representation, (c) Query twig pattern

structures should support queries having "//" (i.e., ancestor-descendant relationships of unbounded depth) efficiently (though not necessarily in a single lookup). Support for twig queries means that the index structures should be able to process 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., [12, 21, 6, 5, 14, 29, 24]), relational join indices [28], and object-oriented path indices (see, e.g., [2, 16, 31]) 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 6 for more details). Further, some existing index structures [6, 18] require either special index structures or join algorithms not available in today's systems, while others [29, 24] 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 and object-oriented path indices, and relational join indices, and to understand the performance tradeoff between index space and twig matching time.

The rest of this paper is organized as follows. In Section 2, we formally define the indexing problems we address in this paper. In Section 3, we define the family of indices, and in Section 4, we discuss how the space used by our index structures can be optimized. In Section 5, we present our experimental results. In Section 6, we discuss related work, and in Section 7, we present our conclusions.

2 Preliminaries and Problem Definition

2.1 Data Model and Query Twig Patterns

An XML database is a forest of rooted, ordered, labeled trees[2], 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 [30], Quilt [4] and XML-QL [8] 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). Note that an ancestor-descendant edge is needed between the `book` element and the `author` element. This query twig pattern would match the data tree in Figure 1. 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. Finding all matches of a query twig pattern in an XML

[2]IDREFs are encoded and queried as values in XML. A link through an IDREF is treated as a value-based join of the IDREF value(s) and the (corresponding) ID value(s), and is not considered part of the XML tree structure.

database is clearly a core operation in XML query processing, both in relational implementations of XML databases [10, 9, 26, 27], and in native XML databases [11, 23, 22].

2.2 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.

2.3 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 [34, 1, 3]. Alternatively, if there are few XML books, one

could first find all book ids matching "`/book[title = XML]`". Then, one could 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.

3 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`.

3.1 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 *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

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

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 value index in Lore [20] returns the ID of an attribute or element given its tag name and value. This index is essentially a B+-tree index on the `SchemaPath` and `LeafValue`,

Index	Subset of `SchemaPath`	Sublist of `IdList`	Indexed Columns
Value [20]	paths of length 1	only last ID	`SchemaPath`, `LeafValue`
Forward link [20]	paths of length 1	only last ID	`HeadId`, `SchemaPath`
DataGuide [12]	root-to-leaf path prefixes	only last ID	`SchemaPath`
Index Fabric [6]	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

where `SchemaPath` consists of paths with length one (i.e., the tag name), and the last ID in `IdList` is returned. The forward link index [20] in Lore returns the ID of an element or attribute given its tag name and the ID of its parent. This is essentially a B+-tree index on `HeadId` and `SchemaPath`, where `HeadId` is the start ID of the path, `SchemaPath` has length one, and the last ID in `IdList` is returned. Similarly, the DataGuide [12] returns the last ID of the `IdList` for every root-to-leaf prefix path. Finally, the IndexFabric [6] 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.

It is important to note that in our implementation of these indices, we only consider relational adaptations (using B+-trees) because some space-efficient structures such as Patricia tries used in [6] are not present in current commercial relational databases. However, since many commercial systems such as DB2 implement prefix compression on indexed columns to reduce the key size, *regular* B+-tree indices are also space efficient when the schema path lengths are not too long.

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, summarized in Figure 3, which can answer the FreeIndex and BoundIndex problems with one index lookup, respectively.

3.2 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 extension is to efficiently support 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[3], 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

[3]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 [34] can be used, to enable containment queries.

ReverseSchemaPath	LeafValue	IdList
B	null	[1]
TB	XML	[1,2]
UB	null	[1,5]
AUB	null	[1,5,6]
FAUB	null	[1,5,6,7]
FAUB	jane	[1,5,6,7]
LAUB	null	[1,5,6,10]
LAUB	poe	[1,5,6,10]
	...	

Figure 4: The 4-ary Relation Adapted for ROOTPATHS

we just *reverse* the SchemaPath values to be indexed (e.g., FAUB instead of BUAF in 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.

Figure 4 shows the 4-ary representation adapted for the ROOTPATHS index. As shown, the HeadID column can be dropped since only paths starting from the root are stored (hence all tuples have HeadID = 1, and this does not have to be explicitly stored). Further, the SchemaPaths are reversed to enable the efficient evaluation of PCsubpath queries with an initial "//".

A B+-tree index 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 would be 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. Of course, fully specified PCsubpaths (without an initial "//") can also be handled using this index.

HeadId	ReverseSchemaPath	LeafValue	IdList
1	**B**	null	[]
1	**TB**	null	[2]
1	**TB**	XML	[2]
1	**UB**	null	[5]
1	**AUB**	null	[5,6]
1	**FAUB**	null	[5,6,7]
1	**FAUB**	jane	[5,6,7]
1	**LAUB**	null	[5,6,10]
1	**LAUB**	poe	[5,6,10]
	...		
5	**U**	null	[]
5	**AU**	null	[6]
5	**FAU**	null	[6,7]
5	**FAU**	jane	[6,7]
5	**LAU**	null	[6,10]
5	**LAU**	poe	[6,10]
	...		

Figure 5: The 4-ary Relation Adapted for DATAPATHS

3.3 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. Figure 5 shows the adaptation of the 4-ary representation for DATAPATHS.

DATAPATHS index can solve both the FreeIndex and the BoundIndex problems in one index lookup.[4] 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

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

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 discuss lossless and lossy compression techniques in the next section.

4 Compressing ROOTPATHS and DATAPATHS

The ROOTPATHS and DATAPATHS indices can be quite large, depending on the size and depth of the XML database, because node ids are duplicated in IdList and SchemaPaths are duplicated in DATAPATHS.

In this section, we explore lossless and lossy compression techniques for reducing the index sizes. The lossless compression schemes do not negatively impact query functionality (i.e., exactly the same query plan can be used), while the lossy compression schemes trade off space for query functionality. Also, for all compression techniques, there is a tradeoff of the decompression overhead at run time and space savings. For example, we could use dictionary-encoding to compress the LeafValues. However, the dictionary is likely quite large and cannot fit in memory, incurring I/O overhead for index lookup. Thus, we only consider compressing IdList, HeadId and SchemaPath in this paper.

4.1 Compressing IdLists

The IdList attribute of ROOTPATHS and DATAPATHS maintains a list of node identifiers, typically generated using depth-first or breadth-first numbering, for the nodes in the schema path. One lossless compression technique is to store only the offset of each identifier with respect to the previous identifier in the IdList, as is done in compressed inverted indices in IR. This corresponds to a differential encoding of the IdList, and is likely to lead to a significant savings in space because the ids in the list are strongly correlated by parent-child

relationships.

With some knowledge about the query workload, it is also possible to prune the `IdLists`. For example, a node that is never returned as part of the result of any twig pattern in the workload, and is not a branching point of any twig pattern, can be eliminated from the `IdList` (i.e., replaced by a NULL). An extreme example is when the query workload contains only simple rooted path patterns (i.e., no branching or recursion) that return the path root nodes; this occurs when one is only filtering XML documents based on the *existence* of a pattern, rather than returning each pattern match; this is the query class handled by the Index Fabric. In this case, each `IdList` in `ROOTPATHS` contains one node. This compression of `IdLists` results in loss in functionality. One can only match queries in the workload, and the index is not useful for ad hoc path patterns.

4.2 Compressing `SchemaPaths`

In a well-structured XML database, the number of distinct schema paths is quite small compared to the number of root-to-leaf paths. For example, the DBLP database has 235 distinct schema paths, and the XMark [32] database has 902 distinct schema paths. This naturally suggests that one can dictionary-encode each of the schema paths, representing them as small integer ids. The effect of such an encoding on the 4-ary relation of Figure 2 is depicted in Figure 6, where the `SchemaPath` attribute has been replaced by the `SchemaPathId` attribute.

This compression of schema paths, however, results in some loss in functionality. One can no longer match a PCsubpath pattern that begins with a "`//`", e.g., "`//author/fn[. = jane]`". This loss of functionality is due to the fact that the schema path identifier is indivisible, and one cannot compute its prefixes or suffixes. Thus, reducing the space used by the index can result in an increase in query evaluation time, by eliminating some (potentially) efficient query processing plans.

HeadId	SchemaPathId	LeafValue	IdList
1	**1**	null	[]
1	**2**	null	[2]
1	**2**	XML	[2]
1	**3**	null	[5]
1	**4**	null	[5,6]
1	**5**	null	[5,6,7]
1	**5**	jane	[5,6,7]
1	**6**	null	[5,6,10]
1	**6**	poe	[5,6,10]
	...		
5	**20**	null	[]
5	**21**	null	[6]
5	**22**	null	[6,7]
5	**22**	jane	[6,7]
5	**23**	null	[6,10]
5	**23**	poe	[6,10]
	...		

Figure 6: `SchemaPath` Compression in the 4-ary relation

4.3 Pruning `HeadIds`

While a FreeIndex lookup is useful for any PCsubpath pattern, a BoundIndex lookup is useful only when one knows a set of `HeadId` values, say, because of a previous index lookup of a PCsubpath in the twig pattern, and the optimizer's choice of index-nested-loops as the join algorithm. This observation is the basis for reducing the size of `DATAPATHS`.

If we know the query workload, then we can prune out entries from the `DATAPATHS` index whose `HeadId` corresponds to a data node that is not a query branch point. This technique is sensitive to the query workload. One can still use the index to match queries not in the workload (using `IdLists`), but the index-nested-loop join strategy will not be possible.

5 Experimental Evaluation

We now present an experimental evaluation of the `ROOTPATHS` and `DATAPATHS` indices with the existing index structures in the same family. We also compare our approach against Access Support Relations [16] and Join Indices [28], which were originally proposed for indexing

paths in object-oriented and relational databases, respectively. We evaluated the following features of the new index structures:

- Benefit of indexing both `SchemaPath` and `LeafValue`.
- Benefit of returning full `IdLists`.
- Benefit of reversing `SchemaPath` for recursive queries.
- Benefit of supporting index-nested-loop join.
- Effects of space compression.

5.1 Experimental Setup

Since XML data may often be stored in relational database systems, we chose to run our experiments on top of IBM's DB2 relational database. We used both a real (DBLP [7], which is shallow) and a synthetic (XMark [32], which is deep) data set for our experiments. We assume the XML data is stored in an Edge Table [10], which stores every edge in the XML data and we assume each node is assigned a unique id. [5]

We now describe the details of our relational implementation and experimental setup before presenting our experimental results.

5.1.1 Database Settings and Query Workload

We used a 100MB scaled XMark data [32] and a 50 MB DBLP data [7]. Our experiments were performed using a 1.7 GHz Pentium machine running Windows 2000, with 1GB memory and a single 37 GB disk. We used DB2 version 7.2, and ran the experiments with a 40MB buffer pool with operating system cache turned off in order to study the effects of using a

[5] For other storage formats where the XML data is stored in multiple tables, we assume each node is assigned a unique id within a table, and the node id stored in all index structures consists of a table id and the node id.

non main-memory resident data set. We also turned off the Windows file system cache so that data items evicted from the database buffer pool were not cached in the file system. We collected detailed statistics on all relations and indices before running our queries. The experimental results reported are the total query execution time of 10 independent runs with a warm cache, excluding the query optimization time. This simulates the case where many read-only XML queries are run concurrently against the data. The results for a cold cache are similar and omitted for space reasons. The cost of translating the XPath query to SQL is considered part of the query optimization cost. In all experiments, the cost of translating a tag name to the internal representation is negligible because the translation table can fit in a single page and can be assumed to always reside in memory.

We used a workload of XPath queries, and varied the parameters of the query such as the number of branches, the selectivity of each branch, and the depth of branches. Figure 10 summarizes these queries. The details of individual queries can be found in Figures 7 and 8.

5.1.2 Details of Relational Implementation

We implemented seven 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 [20] (these indices are the most useful indices reported in [10]), Access Support Relations `(ASR)`, and Join Indices `(JI)`.

Since commercial database systems (such as DB2 and Oracle) do not currently implement Patricia trie, we use regular B+-tree indices in this paper to simulate Index Fabric. Many commercial systems such as DB2 has implemented prefix compression on indexed columns to reduce the key size. Thus when the schema paths are not too long, *regular* B+-tree indices are also space efficient.

Query	*Query*	*Result Size Per Branch*
$Q1_x$	/site/regions/namerica/item/quantity[. = 5]	1
$Q1_d$	/inproceedings/year[. = '1950']	1
$Q2_x$	/site/regions/namerica/item/quantity[. = 2]	3128
$Q2_d$	/inproceedings/year[. = '1979']	1647
$Q3_x$	/site/regions/namerica/item/quantity[. = 1]	11062
$Q3_d$	/inproceedings/year[. = '1998']	10258
$Q4_x$	/site[people/person/profile/@income = 46814.17] /open_auctions/open_auction[@increase = 75.00]	1 55
$Q5_x$	/site[people/person/profile/@income = 46814.17] [people/person/name = 'Hagen Artosi'] /open_auctions/open_auction[@increase = 75.00]	1 1 55
$Q6_x$	/site[people/person/profile/@income = 9876.00] /open_auctions/open_auction[@increase = 75.00]	2038 55
$Q7_x$	/site[people/person/profile/@income = 9876.00] [regions/namerica/item/location = 'united states'] /open_auctions/open_auction[@increase = 75.00]	2038 7519 55
$Q8_x$	/site[people/person/profile/@income = 9876.00] /open_auctions/open_auction[@increase = 3.00]	2038 5172
$Q9_x$	/site[people/person/profile/@income = 9876.00] [regions/namerica/item/location = 'united states'] /open_auctions/open_auction[@increase = 3.00]	2038 7519 5172
$Q10_x$	/site/open_auctions/open_auction [annotation/author/@person = 'person22082'] /time	 3 59486
$Q11_x$	/site/open_auctions/open_auction [annotation/author/@person = 'person22082'] [bidder/@increase = 3.00] /time	 3 5172 59486

Figure 7: Single-branch and twig queries used in our experiments

Query	*Query*	*Result Size Per Branch*
$Q12_x$	`/site//item[incategory/category = 'category440']` `/mailbox/mail/date`	41 20946

$Q13_x$	`/site//item[incategory/category = 'category440']`
	`/mailbox/mail/date`
	`/mailbox/mail/to`
		41
		20946
		20946

$Q14_x$	`/site//item[quantity = 2]` `[location = 'United States']`	1543 16294

$Q15_x$	`/site//item[quantity = 2]`
	`[location = 'United States']`
	`/mailbox/mail/to`
		1543
		16294
		20946

Figure 8: XMark branching twig queries with one recursion

The original proposals for ASRs [16] and Join Indices [28] present techniques for materializing a subset of the paths given a query workload. However, since our focus is on evaluating ad hoc queries, we implemented ASRs and Join Indices by materializing all relevant paths present in the data.

Since the DataGuide and the Index Fabric do not store `IdLists`, they cannot be directly used to answer twig queries. Consequently, 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.[6] We also experimented with various query plans for branching queries, where DataGuide and Index Fabric were used only for some of the query branches, and the link and value indices were used for other branches. We chose the best among these as characterizing the performance of the DataGuide and Index Fabric approaches. We refer to these combined strategies as DG+Edge and IF+Edge.

We could not use the structural join algorithms of [34, 1, 3, 18] since none of these algorithms has been implemented in commercial database systems.

[6]Note that we cannot use the Reverse DataGuide [19] for this purpose, since it can only return branch point ids given the *leaf to root path*.

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 9: 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 10: Queries

Figure 9 gives the space requirement for the various index structures. The space for `DATAPATHS` and `ROOTPATHS` is the result after differential-encoding on `IdList`. Since XMark data is more deeply nested than DBLP, the space requirements for `DATAPATHS` increase proportionally.

5.2 Experimental Results

We first compare our index structures with existing XML index structures. We then present a comparison with ASRs and Join Indices.

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 11 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 very 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`.

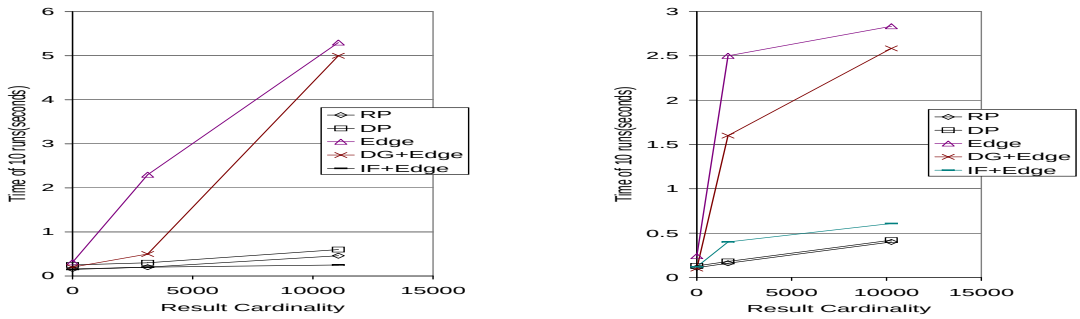
Edge performs badly because it performs a join operation for each step along the path. As the selectivity of paths decreases, it increases the cost of each join. The bad performance of Edge is a simple justification for using a single index lookup instead of resorting to more expensive joins.

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, a separate lookup has to be performed for the schema path (using the DataGuide) and for the data value (using the value index), and the results 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 $Q6_x$ and $Q7_x$ to evaluate the performance of queries with a mix of selective and



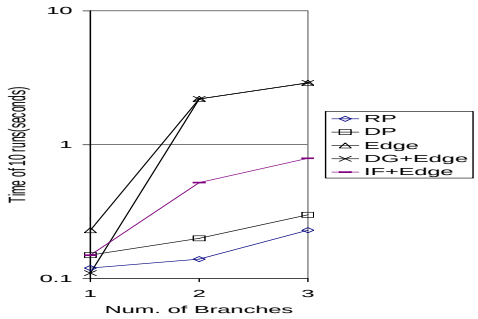
(a) XMark (b) DBLP

Figure 11: Increasing selectivity for single path queries

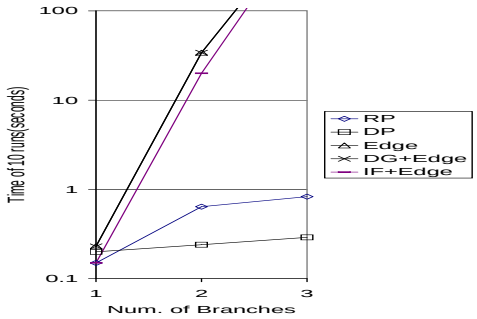
unselective branches, and $Q8_x$ and $Q9_x$ for queries with all unselective branches. For all these queries, the branch point is high in the query. The results for DBLP are similar and omitted due to space restrictions.

Figures 12(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 in both regards (note the log time scale on the graphs).

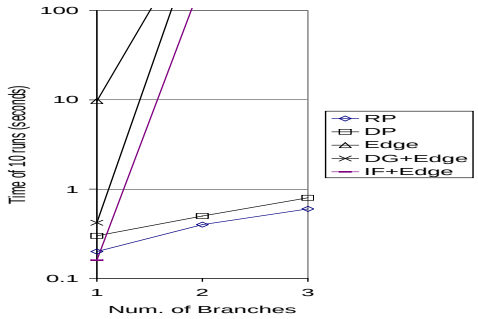
`ROOTPATHS` and `DATAPATHS` perform so well because they store `IdLists`. Hence, they can do an index lookup for each path, extract the ids of the branch point from the `IdLists`, and do a join on the branch points to produce the desired result. With increasingly unselective predicates, more ids will 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 12(a) and 12(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`, but is larger and more expensive to access.



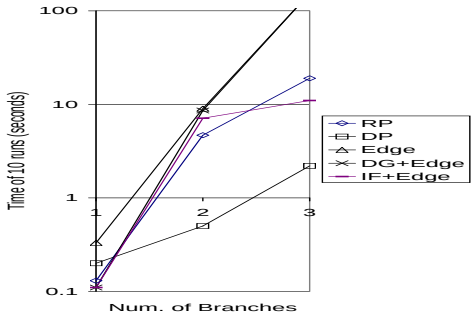
(a) Twig queries with selective branches



(b) Twig queries with selective and unselective branches



(c) Twig queries with unselective branches



(d) Twig queries with low branch points.

Figure 12: XMark twig queries without recursion

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 of the 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` approaches (the graphs

only show the execution time). Also, the relational optimizer 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. We thus believe that `IdLists` are valuable both for reducing the overhead of performing joins, and also 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 $Q10_x$ and $Q11_x$ for the XMark data, which have one selective path and other unselective paths, for this set of experiments. The performance results are shown in Figure 12(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.

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 queries with a "//" because such queries can be converted into B+-tree prefix match queries on `ReverseSchemaPath`s. The detailed results are omitted due to space constraints.

5.2.5 Space Optimizations

Although `DATAPATHS` performs orders of magnitude better than existing approaches, one possible concern is its space overhead. The lossless compression strategies reduced the space requirement by about 30%, which gives rise to the space requirement shown in Figure 9. We now study the effects of other lossy compression strategies.

We implemented `SchemaPaths` compression, which reduces the space overhead by an additional 10MB for the XMark data, and has no savings for the DBLP data. For this marginal savings in space, `SchemaPaths` compression may not be desirable because it does not support recursive ("//") queries. We implemented `HeadId` pruning based on workload information (i.e., all queries used in our experiments), and the index size dropped considerably to 141MB (1.4 times the data size) for the XMark data and 38.4MB (77% of data size) for the DBLP data. Note, however, such pruning disables index-nested-loop join for queries not in the workload and branching at other positions. Thus there might be a performance penalty for such queries and so this compression should be used judiciously.

5.2.6 Comparison with ASRs and Join Indices

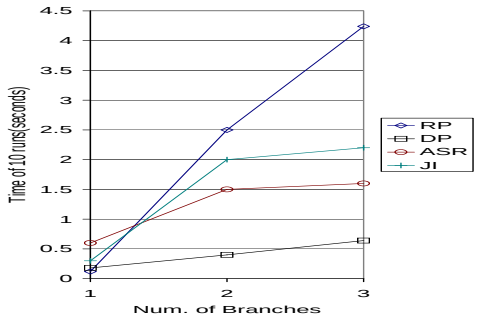
We now compare our index structures against ASR and Join Indices. ASR and Join Indices are similar to DATAPATHS in the sense that all of them encode nodes along paths. However, there are three differences between them.

First, both ASR and Join Indices assume the schema is known a priori. Therefore, ASR and Join Indices require schema discovery as a pre-requisite step and have manageability problems when new data, not conforming to the previous schema, is added.

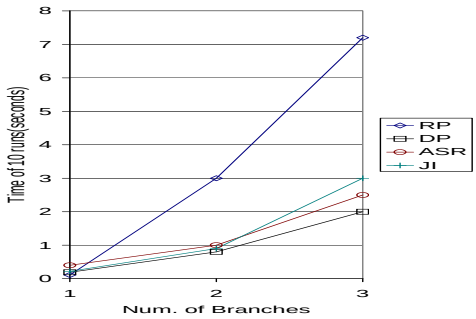
Second, our index structures encode both schema and data using the same framework, while ASR and Join Indices encode schema as relation names. This gives our index structures two advantages over ASR and Join Indices. First, this drastically reduces the number of relations and indices, and the management overhead. For example, in order to support ad hoc queries, both ASR and Join Indices created 902 and 235 tables for XMark and DBLP respectively. Our index structures each have only one index.

More importantly, indexing schema and data together enables the efficient evaluation of "//" queries, when the recursion matches many subpaths, because both ASR and Join Indices need to access many relations, one for each matching subpath. This is less efficient than accessing a single index structure because in a unified index structure, the cost of accessing the index is logarithmic to the data size, but the cost of accessing many small indices is linear to the number of indices. To investigate this, we ran experiments for the queries shown in Figure 8 which contain a "//" as branch point and matches six subpaths in the data. Again, we vary the number of branches as well as selectivity of different branches. $Q12_x$ and $Q13_x$ consist of both selective and unselective branches, and $Q14_x$ and $Q15_x$ consist of unselective branches. The results for all selective branches are similar, so they are omitted. We also exclude the overhead to decide which relations to access for ASR and Join Indices. So their real performance would be worse than shown here.

Figure 13 shows the results. The performance of Edge table, DG+Edge, and IF+Edge are



(a) Selective and unselective branches



(b) Unselective branches

Figure 13: XMark queries having a "//" as branch point

not shown because they are about an order worse than our index structures. The results show that the performance of DATAPATHS is up to a factor of 5 better than ASR and Join Indices because the latter techniques have to access 6 different relations to retrieve a single branch in the query. This difference decreases as the queries contain only unselective branches, because now the cost of joining these branches dominates the cost of index access. ROOTPATHS has bad performance because index-nested-loops join is much more efficient than merge join for these queries.

Note that the same argument applies to other index structures that answer a recursive query by translating the recursion into several equality path conditions (e.g., XRel [33]). Hence we do not compare our index structures with these indices in this paper.

Finally, ASRs and Join Indices require more space than DATAPATHS. ASR uses more space because it cannot compress IdLists, which are stored in separate columns. However, the space saving is less than that achieved by the differential encoding of IdLists (i.e., 30%, see Section 5.2.5) because DATAPATHS need to store SchemaPath. Join Index needs even more space than ASRs for the following reason. Join Index only store the starting and ending node id along a subpath. In order to return intermediate nodes on this path, Join indices have to support both forward lookup to return the ending node and backward lookup to return the starting node. As a result, Join Indices need to build two B+-tree indices per subpath,

while ASRs only need to build one.

6 Related Work

There has been a flurry of recent work on storing and querying XML documents. Some of the proposed approaches use relational database systems for this purpose [10, 9, 26, 27], while others propose building native XML database systems [11, 23, 22]. A central problem in both of these approaches is to efficiently index XML documents so that queries can be answered efficiently. Prior work in this area has focused on efficient ways to harness existing relational index structures [10, 9, 26, 13, 29, 24], and on the development of new index structures [12, 21, 6, 5, 15, 14]. The work in this paper builds upon prior work and develops a new family of index structures that can support a wide range of query access patterns, with different space-time tradeoffs, that can be tightly integrated with relational query processors.

The works in [12, 21, 5, 14] focus on indexing XML paths, *excluding the data values at the ends of the paths*. They rely on complex in-memory graph structures to represent and query XML structures. For example, consider the evaluation of the query "`/book[title = 'XML']`". The index structures can be used to determine the "`/book/title`" ids, but some other auxiliary index structure (e.g., value indices [20]) will have to be used to determine the ids of all elements having the value `'XML'`. The ids of these elements will then have to be joined together to produce the final result. (There are other possible evaluation strategies, but all of them require a potentially expensive join operation or multiple index lookups because the data value is indexed separately from the path.) While such query evaluation strategies may be efficient in some scenarios, often this strategy will be much worse than a single index lookup (as we verified experimentally). Further, the trivial extension to these index structures that treats values as part of the structure would not work, since there are usually far more distinct values than distinctive paths; adding values to the structure will

blow up the size of the in-memory structure and require complex memory management.

The Index Fabric [6] indexes XML paths and data items together. Thus, the Index Fabric can support root-to-leaf path queries in a single index lookup. It can also support branching queries if the query workload is known beforehand; it precomputes all possible branching queries and encodes them as single path queries. However, in the presence of ad hoc queries or 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 [29] and PRIX [24] techniques have been proposed for indexing XML twigs using relational access methods such as B+ trees. These techniques 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 for the general evaluation of XML queries.

XML path indexing is also related to the problem of join indexing in relational database systems [28] and path indexing in object-oriented database systems (see, e.g., [2, 16, 31]). 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 shall show the limitations of these previous approaches, especially for recursive queries, experimentally in Section 5.

Most related to our approach are other recent approaches for indexing XML paths using a relational database [33, 25]. The ToXin approach [25] intends to build XML indices similar

to Access Support Relations (ASR) [16] and Join Indices [28]. However, they have the same problem as ASR/Join Indices, which are inefficient for recursive queries, as shown in Section 5. The XRel approach [33] also stores paths in relational tables, except that it stores the actual paths in a different table and only stores path ids with the data. This normalization saves space, but the ramification of the decision to store path ids (instead of actual paths) with the data is that 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.

There has been some interesting recent work on building approximate XML indices [21, 15]. The basic idea is to tradeoff accuracy of the index structure for associated space savings. The approximation techniques specified in [15] are quite general and could potentially be applied to our index structures to further reduce the index space requirement; of course, this would also imply that our index structures would no longer return exact results and a post-filtering step would be necessary.

Related to the problem of indexing XML documents is the evaluation of XML containment queries. XML containment queries are used to efficiently determine ancestor-descendant relationships (such as "`a//b`"). Novel join algorithms [34, 18, 1, 3] and adaptations of R-trees [17] have been proposed for this purpose. Such techniques can be used to stitch together the intermediate results produced using our index structures. It is important to note that containment query-processing techniques are *not* path indexing techniques.

Of course, XML indexing is just one aspect of XML query processing. A closely related problem is the optimization of XML queries by choosing from available indices. The Lore system presents an optimization approach [20] in which they choose among three XML graph traversal strategies (top-down, bottom-up, and hybrid). The index structures proposed in this paper can support all three strategies, and can thus be used with a Lore-style optimizer. Our focus, however, is to map XML queries to relational index lookups that are visible to the relational query optimizer, so that we can leverage the vast body of work on relational

query optimization in order to optimize XML queries.

7 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. The good performance of our proposed techniques can be attributed to the following factors: (a) combined indexing of XML schema paths and data values, (b) use of `IdLists` to determine branch points, and (c) support for general relational query processing strategies (such as index-nested-loops join). Based on our experiments using the DBLP dataset and the XMark benchmark, we determined that these new index structures outperform the use of existing indices by *orders of magnitude* for most twig queries, while remaining competitive for single-path (non-branching) queries.

It is important to keep in mind that this performance improvement comes at the cost of additional index space, and a higher index update cost. Updating the `ROOTPATHS` and `DATAPATHS` indices requires updating multiple index entries. For example, for `ROOTPATHS`, inserting an author with a certain name to an existing book requires inserting all prefixes of the "`/book/author/name`" path. However, `ROOTPATHS` and `DATAPATHS` themselves could be used to speed up the lookup of the entries to update. For example, if we want to delete an author with a certain name from an existing book (whose ID is known) from `ROOTPATHS`, we could use the author name and the schema path "`/book/author/name`" to locate the authors with the given name, and extract the book IDs from the matching entries to examine whether the book ID matches the book ID to delete. Note that using Edge table, DataGuide, or Index

Fabric all requires joins to locate the index entries.

Directions for future work include investigating efficient update algorithms for `ROOTPATHS` and `DATAPATHS`, and exploring additional index space compression strategies (e.g., dictionary encoding the leaf values). Another interesting avenue is to explore the use of multi-dimensional access methods, such as R-trees, to deal with complex conditions on values and thus index a larger class of XML path expressions.

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] N. Bruno, N. Koudas, D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[4] D. D. Chamberlin, J. Robie, D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB Workshop*, 2000.

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

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

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

[8] A. Deutsch, et al. XML-QL: A query language for XML. Submission to the W3C. Available from http://www.w3.org/TR/NOTE-xml-ql.

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

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

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

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

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

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

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

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

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

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

[19] H. Liefke, D. Suciu. XMill: an Efficient Compressor for XML Data. In *SIGMOD*, 2000.

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

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

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

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

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

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

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

[27] I. Tatarinov, et al. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

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

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

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

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

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

[33] 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.

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