Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 60 (2007) 283–302
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Index structures for matching XML twigs using relational query processors

Zhiyuan Chen [a,*], Johannes Gehrke [b], Flip Korn [c], Nick Koudas [d], Jayavel Shanmugasundaram [b], Divesh Srivastava [c]

[a] *Information Systems Department, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, United States*
[b] *Department of Computer Sciences, Cornell University, 4105B Upson Hall, Ithaca, NY 14853, United States*
[c] *AT&T Labs–Research, 180 Park Ave, P.O. Box 971, Florham Park, NJ 07932-0971, United States*
[d] *Department of Computer Science, Bahen Center for Information Technology, University of Toronto, 40 St. George Street Rm BA5240, Toronto ON M5S 2E4, Canada*

Received 14 March 2006; received in revised form 14 March 2006; accepted 14 March 2006
Available online 18 April 2006

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 relational 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.

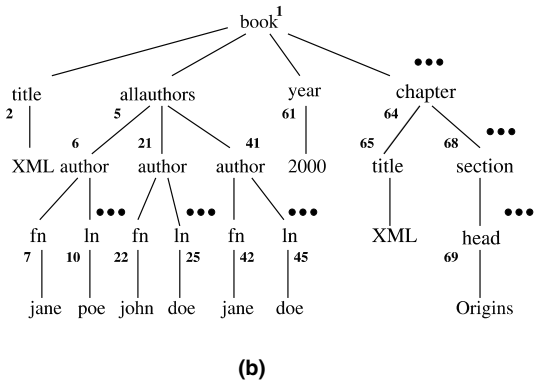
Keywords: XML; Index; Relational database

* Corresponding author.
E-mail addresses: zhchen@umbc.edu (Z. Chen), johannes@cs.cornell.edu (J. Gehrke), flip@research.att.com (F. Korn), koudas@cs.toronto.edu (N. Koudas), jai@cs.cornell.edu (J. Shanmugasundaram), divesh@research.att.com (D. Srivastava).

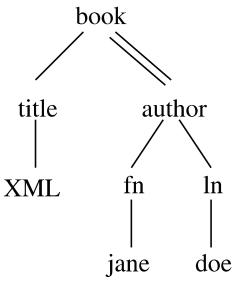
0169-023X/$ - see front matter
doi:10.1016/j.datak.2006.03.003

```
<book>
   <title>XML</title>
   <allauthors>
      <author>
         <fn>jane</ >
         <ln>poe</ >
      </author>
      <author>
         <fn>john</ >
         <ln>doe</ >
      </author>
      <author>
         <fn>jane</ >
         <ln>doe</ >
      </author>
   </allauthors>
   <year>2000</year>
   <chapter>
      <title>XML</title>
      <section>
         <head>Origins</head>
         ...
      </section> ...
   </chapter> ...
</book>
```

(a)



(b)



(c)

Fig. 1. (a) An XML database fragment, (b) tree representation, (c) query twig pattern.

1. Introduction

XML employs a tree-structured model for representing data. Quite naturally, queries in XML query languages (see, e.g., [24]) 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 Fig. 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 [6,7,21], and in native XML databases [8,11,18]. Prior solutions to this problem use a combination of indexing [3,4,9,12,17], link traversal [10,16] and join techniques [1,15,28].

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 spec-

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

ified, 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 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., [3,4,9,12,17,19,23]), relational join indices [22], and object-oriented path indices (see, e.g., [2,14,25]) 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 [4,15] require either special index structures or join algorithms not available in today's systems, while others [19,23] 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 can be tightly integrated with a relational query processor, 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 first review related work in Section 2. We then formally define the indexing problems we address in Section 3. In Section 4, we define the family of indices and propose two novel index structures. We discuss lossless and lossy compression techniques for these indices in Section 5. We present our experimental results in Section 6 and conclude in Section 7.

2. Related work

The works in [3,10,9,12,13,17] 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 [4] 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 [23] and PRIX [19] 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 [22] and path indexing in object-oriented database systems [2,14,25]. 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 experimentally, especially for recursive queries, in Section 6.

There are also recent approaches for indexing XML paths using a relational database [20,27]. The ToXin approach [20] builds XML indices similar to Access Support Relations (ASR) [14] and Join Indices [22], thus have the same problem as ASR/Join Indices. The XRel approach [27] 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 Fig. 1(a), its tree representation is shown in Fig. 1(b). Each non-leaf node is associated with a unique numeric identifier, shown beside the node.

Queries in XML query languages like XQuery [24] 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 Fig. 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 Fig. 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 [1,28]. 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:

3.3.1. 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 Fig. 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.

3.3.2. 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 consist 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 Fig. 1(b) is given in Fig. 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.

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]
	...		

Fig. 2. The 4-ary relation.

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

4.1.1. Family of indices

Given the 4-ary relational representation of XML database D, the family of indices includes 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 Fig. 3. For example, the IndexFabric [4] 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, con-

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 [9]	root-to-leaf paths and prefixes	only last ID	SchemaPath
Index Fabric [4]	root-to-leaf paths	first or last ID	SchemaPath, LeafValue
ROOTPATHS	root-to-leaf paths and prefixes	full IdList	LeafValue, reverse SchemaPath
DATAPATHS	all paths	full IdList	LeafValue, HeadId, reverse SchemaPath

Fig. 3. Members of family of indices.

sider 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, respectively, with one index lookup.

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 Fig. 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 "//title[.='XML']" in a single index lookup. This is done by looking up on the key ('XML', T*). Similarly, PCsubpath patterns

[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 [28] can instead be used to additionally enable containment joins.

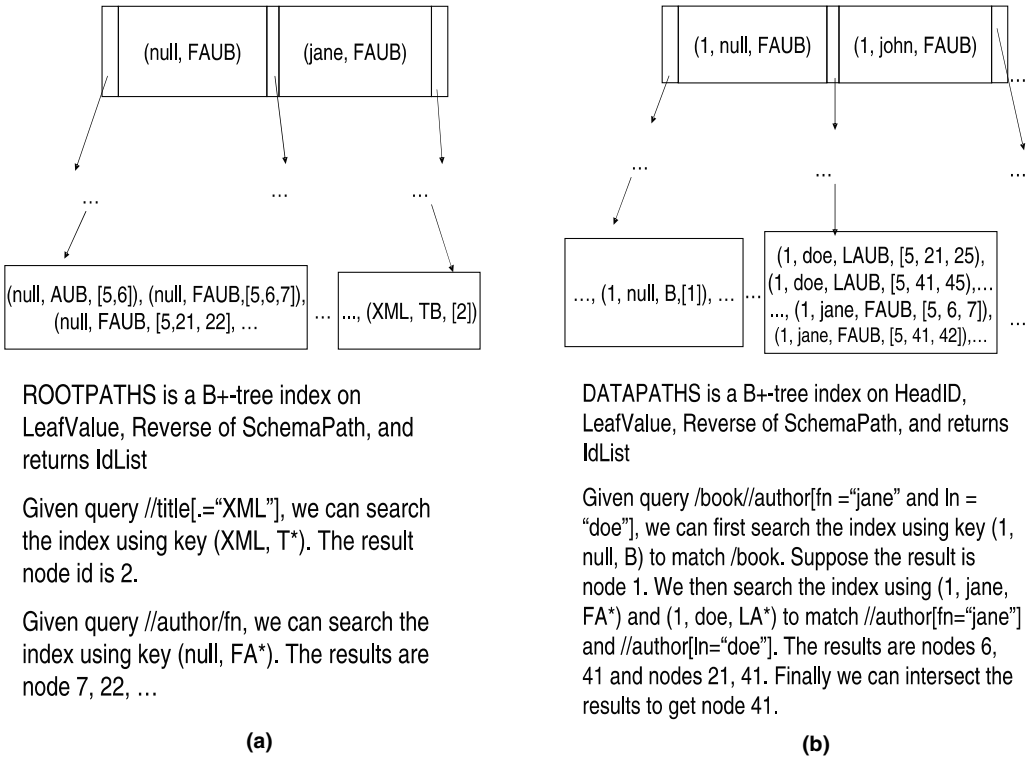


Fig. 4. Illustration of proposed index structures: (a) ROOTPATHS and (b) DATAPATHS.

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. Fig. 4(a) illustrates the ROOTPATHS index structure.

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. Fig. 4(b) illustrates the DATAPATHS index structure.

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

[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).

5. 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 between 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 to be quite large and cannot fit in memory, incurring additional I/O overhead for index lookup. Thus, we only consider compressing IdList, HeadId and SchemaPath in this paper.

5.1. 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.

5.2. 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 [26] 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.

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.

5.3. 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 chooses 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.

6. Experimental evaluation

We present an experimental evaluation of the ROOTPATHS and DATAPATHS indices with the existing index structures in the same family.

6.1. Experimental setup

We assume the XML data is stored in an Edge Table [7] 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 100 MB scaled XMark data [26] and a 50 MB DBLP data [5]. Our experiments were performed using a 1.7 GHz Pentium machine running Windows 2000, with 1 GB memory and a 37 GB disk, and a 40 MB buffer pool with operating system cache turned off. The reported query 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.

6.1.1. Queries

We used a workload of 15 XPath queries on XMark and three 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. Fig. 5 summarizes the characteristics of these queries. The details of these queries can be found in Figs. 6–9.

6.1.2. Indexing strategies

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 [16], Access Support Relations (ASR) [14], and Join Indices (JI) [22]. We use regular B+-tree indices in this paper to simulate Index Fabric since 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 used (possibly many lookups in) the reverse link index on the Edge Table to determine the branch point ids from the leaf ids, and chose the best query plan. We refer to these combined strategies as DG + Edge and IF + Edge.

The original proposals for ASRs [14] and Join Indices [22] 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

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

Fig. 5. Summary of characteristics of queries used in experiments.

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

Fig. 6. Single-branch queries used in our experiments.

Query	*Query*	*Result Size Per Branch*
$Q4_x$	/site[people/person/profile/@income = 46814.17]	1
	/open_auctions/open_auction[@increase = 75.00]	55
$Q5_x$	/site[people/person/profile/@income = 46814.17]	1
	[people/person/name = 'Hagen Artosi']	1
	/open_auctions/open_auction[@increase = 75.00]	55
$Q6_x$	/site[people/person/profile/@income = 9876.00]	2038
	/open_auctions/open_auction[@increase = 75.00]	55
$Q7_x$	/site[people/person/profile/@income = 9876.00]	2038
	[regions/namerica/item/location = 'united states']	7519
	/open_auctions/open_auction[@increase = 75.00]	55
$Q8_x$	/site[people/person/profile/@income = 9876.00]	2038
	/open_auctions/open_auction[@increase = 3.00]	5172
$Q9_x$	/site[people/person/profile/@income = 9876.00]	2038
	[regions/namerica/item/location = 'united states']	7519
	/open_auctions/open_auction[@increase = 3.00]	5172

Fig. 7. Multi-branch queries with high branch points used in our experiments.

ASRs and Join Indices by materializing all relevant paths present in the data. ASR was implemented as a number of relations, one for each prefix of a root-to-leaf SchemaPath. Each relation has $k+1$ columns (k is the length of the prefix path), where the first k columns store IDs of nodes on the prefix SchemaPath, and the

Query	*Query*	*Result Size Per Branch*
$Q10_x$	/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

Fig. 8. Multi-branch queries with low branch points 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

Fig. 9. Recursive queries used in our experiments.

last column stores leaf values. For each relation, we built $k-1$ indices, where the ith index ($1 \leqslant i \leqslant k-1$) is on the $k+1, i, i+1, \ldots, k$th columns, and supports lookup on leaf value and/or the `HeadId` of the ith node on the path. For example, given a relation on ``'/book/author''``, an index is built on `(author value, book id, author id)`, which supports lookup on `author` value and/or `book id`.

The Join Index (JI) Hierarchy is similar to ASR except that it does not store intermediate nodes on a SchemaPath. Thus we materialized the complete join index hierarchy by creating one relation per substring of every root-to-leaf SchemaPath. Each relation consists of three columns: the starting node ID, the ending node ID, and the leaf value. In order to return intermediate nodes on a SchemaPath, JI has to support both forward lookup on starting ID and backward lookup on ending ID. For example, in order to return "author" node in "/book/author[name = 'Zhiyuan Chen']", two lookups are needed: one to return the "name" ID in the relation corresponding to "/book/author/name" given a "name" value, and the other to return the "author" ID in the relation corresponding to "/author/name" given the previously returned "name" ID. To support both types of lookups, we built two B+-tree indices on each relation: one as a clustered index on the leaf value, the starting ID, and the ending ID, the other on the leaf value, the ending ID, and the starting ID.

A twig query can be answered by ASR or JI by looking up each branch in the query, then using sort merge, hash, or index nested loops join methods to join the results of all branches. In case the query contains a "//", if the schema is known, we can look up all SchemaPaths matching the query and union the results for each SchemaPath. Further, we can directly use the JI relation corresponding to the query if the "//" is at the beginning of the query. In our experiments, we assume the ideal case for ASR and JI, when the above schema-based rewriting is applied without any additional overhead.

Fig. 10 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.

Fig. 11 reports the time to build various index structures. The time to build these indexes increases with the sizes of indexes. For ROOTPATHS index, it takes about 3 min to build the index for DBLP and about 4.5 min for XMark. For DATAPATHS index, it takes about 3 min for DBLP and about 16 min for XMark. These two indexes are built by two steps. In the first step, a XML document is parsed and the HeadID, SchemaPath, LeafValue, and IdList are extracted. In the second step, a regular B+-tree index is built based on extracted 4-ary relation. We observe that the second step takes most of the time (the first step takes less than 1 min in all cases) because the parsing and extraction operations in the first step can be done in memory and does not require any sorting. Thus the efficiency of index building mostly depends on the efficiency of DB2 to build a B+-tree index.

6.2. Experimental results

6.2.1. 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

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

Fig. 10. Space (in MB) for different indices.

Data set	*RP*	*DP*	*Edge*	*DG+Edge*	*IF+Edge*	*ASR*	*JI*
XMark	268	970	286	380	376	1044	1850
DBLP	180	187	239	299	340	209	716

Fig. 11. Index building time (in seconds) for different indices.

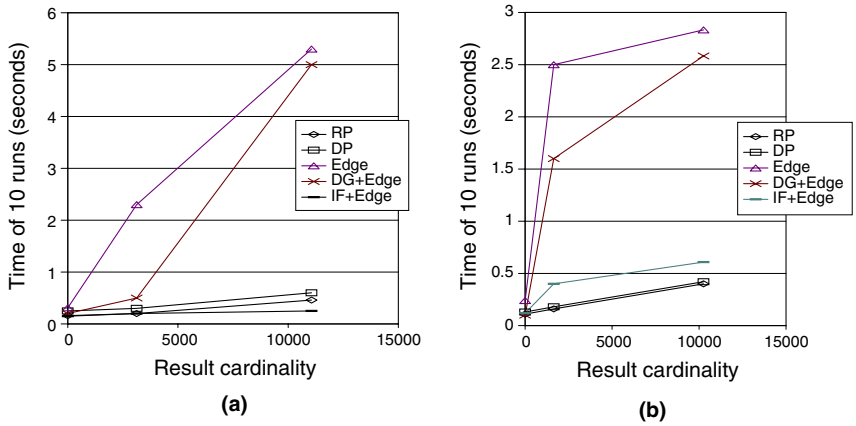


Fig. 12. Increasing selectivity for single path queries: (a) XMark and (b) DBLP.

relatively unselective ($Q3_d, Q3_x$). Fig. 12 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.

6.2.2. 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 (i.e., close to the query root) in the query. The results for DBLP are similar and omitted.

Fig. 13(a)–(c) shows 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 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 Fig. 13(a) and (c) is that in these cases the selectivities are roughly

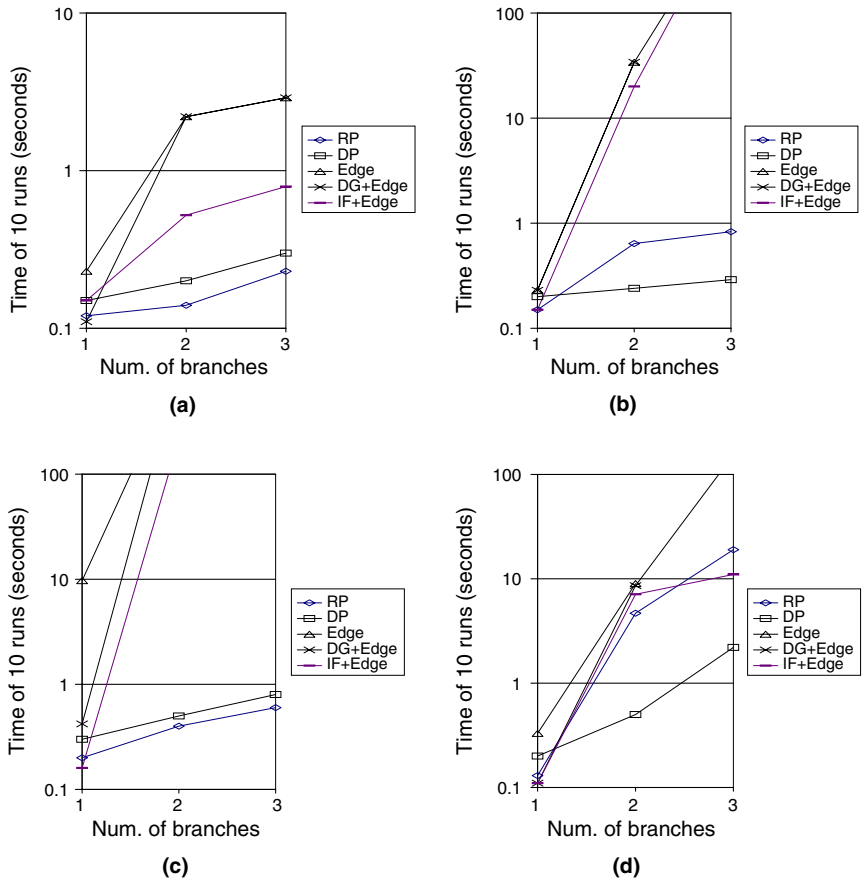


Fig. 13. XMark twig queries without recursion: (a) twig queries with selective branches, (b) twig queries with selective and unselective branches, (c) twig queries with unselective branches and (d) twig queries with low branch points.

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 min. 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* these queries was longer than the time it took to *execute* the ROOTPATHS and DATAPATHS queries (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.

6.2.3. 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 Fig. 13(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.

6.2.4. Recursive queries

We now examine the performance of evaluating recursive ("//") queries. The recursive queries are exactly the same as queries used in Fig. 7 except that each query now starts with a "//". To examine the overhead for recursive queries, we compare the performance of ROOTPATHS and DATAPATHS for queries which do not have a recursion. (Other indices cannot be used here.)

Fig. 14 shows the results for queries with only selective branches (variants of $Q4_x$, $Q5_x$) or only unselective branches (variants of $Q8_x$, $Q9_x$). The results for queries with both selective and unselective branches are similar and omitted. The results show that ROOTPATHS and DATAPATHS have little additional overhead (less than 5%) for processing queries with a "//". This is because such queries can be converted into B+-tree range queries on ReverseSchemaPaths. The Edge approach can also support such queries, but suffers from the previous overheads of multiple joins for both paths and twigs.

6.2.5. Space optimizations

Although DATAPATHS performs orders of magnitude better than existing approaches, one possible concern is its space overhead. The lossless compression strategy (differential encoding of *IDLists*) reduced the space requirement by about 30%, which gives rise to the space requirement shown in Fig. 10. We now study the effects of other lossy compression strategies.

We implemented SchemaPaths compression, which reduces the space overhead by an additional 10 MB 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

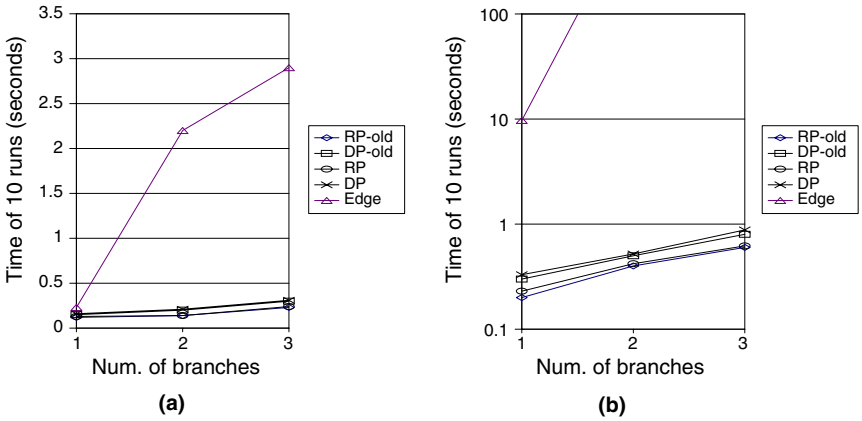


Fig. 14. Overhead for recursive queries: (a) twig queries with selective branches and (b) twig queries with unselective branches.

index size dropped considerably to 141 MB (1.4 times the data size) for the XMark data and 38.4 MB (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.

6.2.6. 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 Fig. 9 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.

Fig. 15 shows the results. The performance of Edge table, DG + Edge, and IF + Edge are not shown because they are about an order of magnitude 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 six 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.

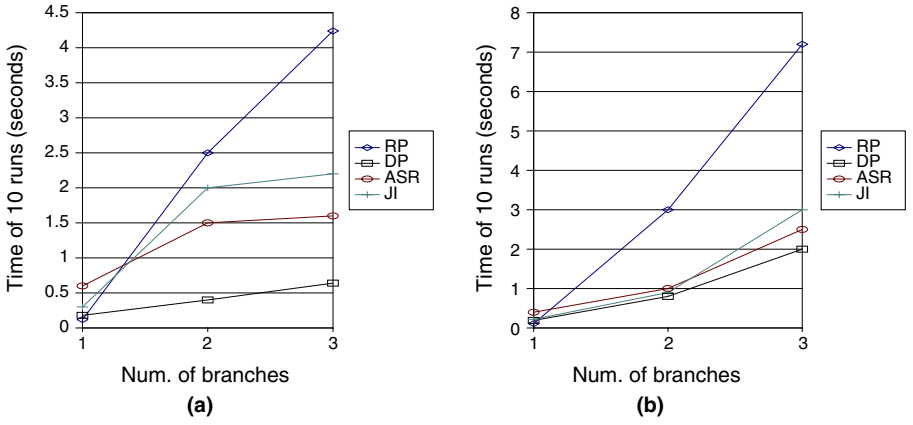


Fig. 15. XMark queries having a "//" as branch point: (a) selective and unselective branches and (b) unselective branches.

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 [27]). 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 6.2.5) because DATAPATHS need to store SchemaPath. Join Index needs even more space than ASRs for the following reason. Join Index only stores 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.

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. A promising direction for future work is to devise efficient update techniques for the proposed index structures.

References

[1] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, D. Srivastava, 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, vol. 1 (2), 1989.
[3] C.-W. Chung, J.-K. Min, K. Shim, APEX: an adaptive path index for XML data, in: SIGMOD, 2002.
[4] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon, A fast index for semistructured data, in: VLDB, 2001.
[5] DBLP. Available from: <http://www.informatik.uni-trier.de/~ley/db/index.html>.
[6] A. Deutsch, M. Fernandez, D. Suciu, Storing semistructured data with STORED, in: SIGMOD, 1999.
[7] D. Florescu, D. Kossman, Storing and querying XML data using an RDMBS, IEEE Data Engineering Bulletin 22 (3) (1999) 27–34.
[8] R. Goldman, J. McHugh, J. Widom, From semistructured data to XML: Migrating the Lore data model and query language, in: WebDB Workshop, 1999.
[9] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: VLDB, 1997.
[10] T. Grust, Accelerating XPath location steps, in: SIGMOD, 2002.
[11] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu, TIMBER: a native XML database, VLDB Journal 11 (4) (2002) 274–291.
[12] R. Kaushik, P. Bohannon, J.F. Naughton, H.F. Korth, Covering indexes for branching path queries, in: SIGMOD, 2002.
[13] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, 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] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: VLDB, 2001.
[16] J. McHugh, J. Widom, Query optimization for XML, in: VLDB, 1999.
[17] T. Milo, D. Suciu, Index structures for path expressions, in: ICDT, 1999.
[18] J. Naughton, D.J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, R. Chen, The Niagara Internet Query System, IEEE Data Engineering Bulletin 2 (2) (2001).
[19] P. Rao, B. Moon, PRIX: indexing and querying XML using pruffer sequences, in: ICDE, 2004.
[20] F. Rizzolo, A. Mendelzon, Indexing XML data with ToXin, in: WebDB Workshop, 2001.
[21] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, in: VLDB, 1999.
[22] P. Valduriez, Join indices, in: ACM TODS, 1987, vol. 12(2).
[23] H. Wang, S. Park, W. Fan, P. Yu. ViST: a dynamic index method for querying XML data by tree structures, in: SIGMOD, 2003.
[24] World Wide Web Consortium, XQuery: a query language for XML. Available from: <http://www.w3.org/TR/xquery>.
[25] Z. Xie, J. Han, Join index hierarchies for supporting efficient navigations in object-oriented systems, in: VLDB, 1994.
[26] XMark, The XML benchmark project. Available from: <http://monetdb.cwi.nl/xml>.

[27] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura, XRel: a path-based approach to storage and retrieval of XML documents using relational databases, in: ACM TOIT, vol. 1(1), 2001, pp. 110–141.
[28] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G.M. Lohman, On supporting containment queries in relational database management systems, in: SIGMOD, 2001.

Zhiyuan Chen received the Ph.D. degree in computer science from the Cornell University in 2002. Presently, he is an assistant professor in the information systems department at University of Maryland, Baltimore County. His research interests include XML and semi-structured data, privacy-preserving data mining, data integration, automatic database tuning, and database compression.

Johannes Gehrke is an Associate Professor in the Department of Computer Science at Cornell University and the Technical Director of Data-Intensive Computing of the Cornell Theory Center. He obtained his Ph.D. in computer science from the University of Wisconsin-Madison in 1999. Johannes' research interests are in the areas of data mining, data stream processing, data privacy, and applications of database and data mining technology to marketing and the sciences. Johannes has received a National Science Foundation Career Award, an Arthur P. Sloan Fellowship, an IBM Faculty Award, the Cornell College of Engineering James and Mary Tien Excellence in Teaching Award, and the Cornell University Provost's Award for Distinguished Scholarship. He is the author of numerous publications on data mining and database systems, and he co-authored the undergraduate textbook Database Management Systems (McGrawHill (2002), currently in its third edition), used at universities all over the world. He has also given courses and tutorials on data mining and data stream processing at international conferences and on Wall Street, and he has extensive industry experience as technical advisor.

Flip Korn is a member of the Database Research Department at AT&T Labs-Research in Florham Park, NJ. He received a Ph.D. degree from the University of Maryland, College Park in 1998. His current research interests are in the area of database methods for network management, with a focus on processing data streams.

Nick Koudas is a faculty member at the University of Toronto, department of computer science. He holds a Ph.D. from the University of Toronto, an M.Sc. from the University of Maryland at College Park, and a B.Tech. from the University of Patras in Greece. He serves as an associate editor for the Information Systems journal, the IEEE TKDE journal and the ACM Transactions on the WEB. He is the recipient of the 1998 ICDE Best Paper award. His research interests include core database management, data quality, metadata management and its applications to networking.

Jayavel Shanmugasundaram is an Assistant Professor in the Department of Computer Science at Cornell University. He obtained his Ph.D. degree from the University of Wisconsin, Madison, his masters degree from the University of Massachusetts, Amherst, and his bachelors degree from the Regional Engineering College, Tiruchirappalli, India, all in Computer Science. Prior to joining Cornell University, he spent two years at the IBM Almaden Research Center in San Jose, California. Jayavel's research interests include Internet data management, information retrieval, and query processing in emerging system architectures. He is the author of several publications and patents on these topics, and his research ideas have been implemented in commercial data management products. Jayavel is an invited expert and co-editor of the XQuery and XPath Full-Text language currently being developed by the World Wide Web Consortium (W3C). He has received an NSF CAREER Award, an IBM Faculty Award, and the James and Mary Tien Excellence in Teaching Award.

Divesh Srivastava is the head of the Database Research Department at AT&T Labs-Research. He received his Ph.D. from the University of Wisconsin, Madison, and his B.Tech. from the Indian Institute of Technology, Bombay. His current research interests include data quality, IP network data management and XML databases.