

# A fast index for semistructured data

Brian F. Cooper<sup>1,2</sup>, Neal Sample<sup>1,2</sup>, Michael J. Franklin<sup>1,3</sup>, Gísli R. Hjaltason<sup>1</sup>, Moshe Shadmon<sup>1</sup>

<sup>1</sup>RightOrder Incorporated  
3850 N. First St.  
San Jose, CA 95134 USA

<sup>2</sup>Department of Computer Science  
Stanford University  
Stanford, CA 94305 USA

<sup>3</sup>Computer Science Division  
University of California  
Berkeley, CA 94720 USA

{cooperb,nsample}@db.stanford.edu, franklin@cs.berkeley.edu,  
{gislih,moshes}@rightorder.com

## Abstract

Queries navigate semistructured data via path expressions, and can be accelerated using an index. Our solution encodes paths as strings, and inserts those strings into a special index that is highly optimized for long and complex keys. We describe the Index Fabric, an indexing structure that provides the efficiency and flexibility we need. We discuss how "raw paths" are used to optimize ad hoc queries over semistructured data, and how "refined paths" optimize specific access paths. Although we can use knowledge about the queries and structure of the data to create refined paths, no such knowledge is needed for raw paths. A performance study shows that our techniques, when implemented on top of a commercial relational database system, outperform the more traditional approach of using the commercial system's indexing mechanisms to query the XML.

## 1. Introduction

Database management systems are increasingly being called upon to manage *semistructured* data: data with an irregular or changing organization. An example application for such data is a business-to-business product catalog, where data from multiple suppliers (each with their own schema) must be integrated so that buyers can query it. Semistructured data is often represented as a graph, with a set of data elements connected by labeled relationships, and this self-describing relationship

structure takes the place of a schema in traditional, structured database systems. Evaluating queries over semistructured data involves navigating paths through this relationship structure, examining both the data elements and the self-describing element names along the paths. Typically, indexes are constructed for efficient access.

One option for managing semistructured data is to store and query it with a relational database. The data must be converted into a set of tuples and stored in tables; for example, using tools provided with Oracle 8i/9i [25]. This process requires a schema for the data. Moreover, the translation is not trivial, and it is difficult to efficiently evaluate queries without extensions to the relational model [26]. If no schema exists, the data can be stored as a set of data elements and parent-child nesting relationships [17]. Querying this representation is expensive, even with indexes. The STORED system [12] uses data mining to extract a partial schema. Data that does not fit the schema well must be stored and queried in its native form.

An alternative option is to build a specialized data manager that contains a semistructured data repository at its core. Projects such as Lore [24] and industrial products such as Tamino [28] and XYZFind [29] take this approach. It is difficult to achieve high query performance using semistructured data repositories, since queries are again answered by traversing many individual element to element links, requiring multiple index lookups [23]. Moreover, semistructured data management systems do not have the benefit of the extensive experience gained with relational systems over the past few decades.

To solve this problem, we have developed a different approach that leverages existing relational database technology but provides much better performance than previous approaches. Our method encodes paths in the data as strings, and inserts these strings into an index that is highly optimized for string searching. The index blocks and semistructured data are both stored in a conventional relational database system. Evaluating queries involves encoding the desired path traversal as a search key string, and performing a lookup in our index to find the path. There are several advantages to this approach. First, there

---

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment*

**Proceedings of the 27th VLDB Conference,  
Roma, Italy, 2001**

is no need for *a priori* knowledge of the schema of the data, since the paths we encode are extracted from the data itself. Second, our approach has high performance even when the structure of the data is changing, variable or irregular. Third, the same index can accelerate queries along many different, complex access paths. This is because our indexing mechanism scales gracefully with the number of keys inserted, and is not affected by long or complex keys (representing long or complex paths).

Our indexing mechanism, called the *Index Fabric*, utilizes the aggressive key compression inherent in a Patricia trie [21] to index a large number of strings in a compact and efficient structure. Moreover, the Index Fabric is inherently balanced, so that all accesses to the index require the same small number of I/Os. As a result, we can index a large, complex, irregularly-structured, disk-resident semistructured data set while providing efficient navigation over paths in the data.

We manage two types of paths for semistructured data. First, we can index paths that exist in the raw data (called *raw paths*) to accelerate any ad hoc query. We can also reorganize portions of the data, to create *refined paths*, in order to better optimize particular queries. Both kinds of paths are encoded as strings and inserted into the Index Fabric. Because the index grows so slowly as we add new keys, we can create many refined paths and thus optimize many access patterns, even complex patterns that traditional techniques cannot easily handle. As a result, we can answer general queries efficiently using raw paths, even as we further optimize certain queries using refined paths. Maintaining all of the paths in the same index structure reduces the resource contention that occurs with multiple indexes, and provides a uniform mechanism that can be tuned for different needs.

Although our implementation of the Index Fabric uses a commercial relational DBMS, our techniques do not dictate a particular storage architecture. In fact, the fabric can be used as an index over a wide variety of storage engines, including a set of text files or a native semistructured database. The index provides a flexible, uniform and efficient mechanism to access data, while utilizing a stable storage manager to provide properties such as concurrency, fault tolerance, or security.

A popular syntax for semistructured data is XML [30], and in this paper we focus on using the Index Fabric to index XML-encoded data. XML encodes information as data elements surrounded by tags, and tags can be nested within other tags. This nesting structure can be viewed as a tree, and raw paths represent root-to-leaf traversals of this tree. Refined paths represent traversing the tree in some other way (e.g. from sibling to sibling).

We have implemented the Index Fabric as an index on top of a popular commercial relational DBMS. To evaluate performance, we indexed an XML data set using both the Index Fabric and the DBMS’s native B-trees. In the Index Fabric, we have constructed both refined and raw paths, while the relational index utilized an edge

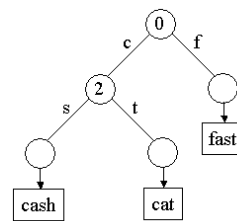


Figure 1. A Patricia trie.

mapping as well as a schema extracted by the STORED [12] system. Both refined and raw paths are significantly faster than the DBMS’s native indexing mechanism, sometimes by an order of magnitude or more. The difference is particularly striking for data with irregular structure, or queries that must navigate multiple paths.

### 1.1. Paper overview

In this paper, we describe the structure of the Index Fabric and how it can be used to optimize searches over semistructured databases. Specifically, we make the following contributions:

- We discuss how to utilize the Index Fabric’s support for long and complex keys to index semistructured data paths encoded as strings.
- We examine a simple encoding of the *raw paths* in a semistructured document, and discuss how to answer complex path queries over data with irregular structure using raw paths.
- We present *refined paths*, a method for aggressively optimizing frequently occurring and important access patterns. Refined paths support answering complicated queries using a single index lookup.
- We report the results of a performance study which shows that a semistructured index based on the Index Fabric can be an order of magnitude faster than traditional indexing schemes.

This paper is organized as follows. In Section 2 we introduce the Index Fabric and discuss searches and updates. Next, in Section 3, we present refined paths and raw paths and examine how they are used to optimize queries. In Section 4 we present the results of our performance experiments. In Section 5 we examine related work, and in Section 6 we discuss our conclusions.

## 2. The Index Fabric

The Index Fabric is a structure that scales gracefully to large numbers of keys, and is insensitive to the length or content of inserted strings. These features are necessary to treat semistructured data paths as strings.

The Index Fabric is based on *Patricia* tries [21]. An example Patricia trie is shown in Figure 1. The nodes are labeled with their *depth*: the character position in the key represented by the node. The size of the Patricia trie does not depend on the length of inserted keys. Rather, each new key adds at most a single link and node to the index,

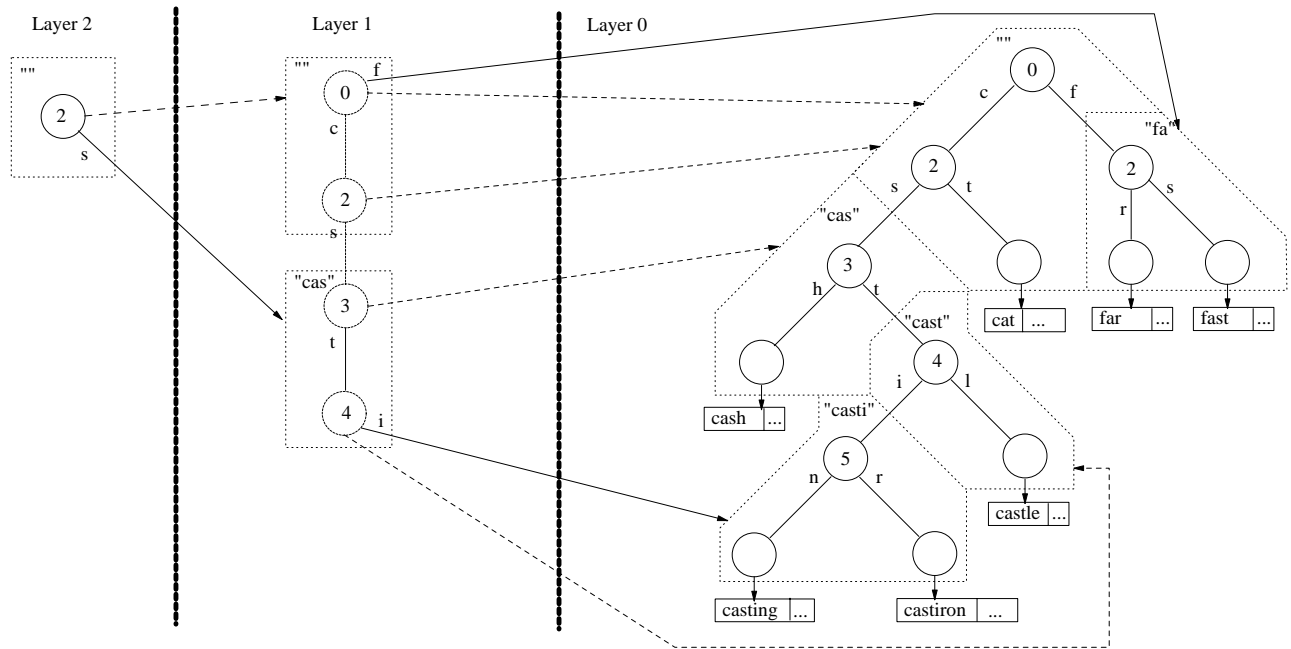


Figure 2. A layered index.

even if the key is long. Patricia tries grow slowly even as large numbers of strings are inserted because of the aggressive (lossy) compression inherent in the structure.

Patricia tries are unbalanced, main memory structures that are rarely used for disk-based data. The Index Fabric is a structure that has the graceful scaling properties of Patricia tries, but that is balanced and optimized for disk-based access like B-trees. The fabric uses a novel, layered approach: extra layers of Patricia tries allow a search to proceed directly to a block-sized portion of the index that can answer a query. Every query accesses the same number of layers, providing balanced access to the index.

More specifically, the basic Patricia trie string index is divided into block-sized subtrees, and these blocks are indexed by a second trie, stored in its own block. We can represent this second trie as a new horizontal layer, complementing the vertical structure of the original trie. If the new horizontal layer is too large to fit in a single disk block, it is split into two blocks, and indexed by a third horizontal layer. An example is shown in Figure 2. The trie in layer 1 is an index over the common prefixes of the blocks in layer 0, where a common prefix is the prefix represented by the root node of the subtree within a block. In Figure 2, the common prefix for each block is shown in “quotes”. Similarly, layer 2 indexes the common prefixes of layer 1. The index can have as many layers as necessary; the leftmost layer always contains one block.

There are two kinds of links from layer  $i$  to layer  $i-1$ : labeled far links ( $\rightarrow$ ) and unlabeled direct links ( $\dashrightarrow$ ). Far links are like normal edges in a trie, except that a far link connects a node in one layer to a subtree in the next layer. A direct link connects a node in one layer to a block with a node representing the same prefix in the next layer. Thus, in Figure 2, the node labeled “3” in layer 1

corresponds to the prefix “cas” and is connected to a subtree (rooted at a node representing “cas” and also labeled “3”) in layer 0 using an unlabeled direct link.

### 2.1. Searching

The search process begins in the root node of the block in the leftmost horizontal layer. Within a particular block, the search proceeds normally, comparing characters in the search key to edge labels, and following those edges. If the labeled edge is a far link, the search proceeds horizontally to a different block in the next layer to the right. If no labeled edge matches the appropriate character of the search key, the search follows a direct (unlabeled) edge horizontally to a new block in the next layer. The search proceeds from layer to layer until the lowest layer (layer 0) is reached and the desired data is found. During the search in layer 0, if no labeled edge matches the appropriate character of the search key, this indicates that the key does not exist, and the search terminates. Otherwise, the path is followed to the data. It is necessary to verify that the found data matches the search key, due to the lossy compression of the Patricia trie.

The search process examines one block per layer<sup>1</sup>, and always examines the same number of layers. If the blocks correspond to disk blocks, this means that the search could require one I/O per layer, unless the needed block is in the cache. One benefit of using the Patricia structure is that keys are stored very compactly, and many

<sup>1</sup> It is possible for the search procedure to enter the wrong block, and then have to backtrack, due to the lossy compression of prefixes in the non-leaf layers. This phenomenon is unique to the multi-layer Patricia trie structure. In practice, such mistakes are rare in a well-populated tree. See [10].

<pre>Doc 1: &lt;invoice&gt;   &lt;buyer&gt;     &lt;name&gt;ABC Corp&lt;/name&gt;     &lt;address&gt;1 Industrial Way&lt;/address&gt;   &lt;/buyer&gt;   &lt;seller&gt;     &lt;name&gt;Acme Inc&lt;/name&gt;     &lt;address&gt;2 Acme Rd.&lt;/address&gt;   &lt;/seller&gt;   &lt;item count=3&gt;saw&lt;/item&gt;   &lt;item count=2&gt;drill&lt;/item&gt; &lt;/invoice&gt;</pre>	<pre>Doc 2: &lt;invoice&gt;   &lt;buyer&gt;     &lt;name&gt;Oracle Inc&lt;/name&gt;     &lt;phone&gt;555-1212&lt;/phone&gt;   &lt;/buyer&gt;   &lt;seller&gt;     &lt;name&gt;IBM Corp&lt;/name&gt;   &lt;/seller&gt;   &lt;item&gt;     &lt;count&gt;4&lt;/count&gt;     &lt;name&gt;nail&lt;/name&gt;   &lt;/item&gt; &lt;/invoice&gt;</pre>
--	--

**Figure 3. Sample XML.**

keys can be indexed per block. Thus, blocks have a very high out-degree (number of far and direct links referring to the next layer to the right.) Consequently, the vast majority of space required by the index is at the rightmost layer, and the layers to the left (layer 1,2,... $n$ ) are significantly smaller. In practice, this means that an index storing a large number of keys (e.g. a billion) requires three layers; layer 0 must be stored on disk but layers 1 and 2 can reside in main memory. Key lookups require at most one I/O, for the leaf index layer (in addition to data I/Os). In the present context, this means that following any indexed path through the semistructured data, no matter how long, requires at most one index I/O.

## 2.2. Updates

Updates, insertions, and deletions, like searches, can be performed very efficiently. An update is a key deletion followed by a key insertion. Inserting a key into a Patricia trie involves either adding a single new node or adding an edge to an existing node. The insertion requires a change to a single block in layer 0. The horizontal index is searched to locate the block to be updated. If this block overflows, it must be split, requiring a new node at layer 1. This change is also confined to one block. Splits propagate left in the horizontal layers if at each layer blocks overflow, and one block per layer is affected. Splits are rare, and the insertion process is efficient. If the block in the leftmost horizontal layer (the root block) must be split, a new horizontal layer is created.

To delete a key, the fabric is searched using the key to find the block to be updated, and the edge pointing to the leaf for the deleted key is removed from the trie. It is possible to perform block recombination if block storage is underutilized, although this is not necessary for the correctness of the index. Due to space restrictions, we do not present insertion, deletion and split algorithms here. The interested reader is referred to [10].

## 3. Indexing XML with the Index Fabric

Because the Index Fabric can efficiently manage large numbers of complex keys, we can use it to search many complex paths through the XML. In this section, we

discuss encoding XML paths as keys for insertion into the fabric, and how to use path lookups to evaluate queries. As a running example, we will use the XML in Figure 3.

### 3.1. Designators

We encode data paths using *designators*: special characters or character strings. A unique designator is assigned to each tag that appears in the XML. For example, for the XML in Figure 3, we can choose **I** for `<invoice>`, **B** for `<buyer>`, **N** for `<name>`, and so on. (For illustration, here we will represent designators as boldface characters.) Then, the string “**IBNABC Corp**” has the same meaning as the XML fragment

```
<invoice>
  <buyer><name>ABC Corp</name></buyer>
</invoice>
```

The designator-encoded XML string is inserted into the layered Patricia trie of the Index Fabric, which treats designators the same way as normal characters, though conceptually they are from different alphabets.

In order to interpret these designators (and consequently to form and interpret queries) we maintain a mapping between designators and element tags called the *designator dictionary*. When an XML document is parsed for indexing, each tag is matched to a designator using the dictionary. New designators are generated automatically for new tags. The tag names from queries are also translated into designators using the dictionary, to form a search key over the Index Fabric. (See Section 3.5.)

### 3.2. Raw paths

Raw paths index the hierarchical structure of the XML by encoding root-to-leaf paths as strings. Simple path expressions that start at the root require a single index lookup. Other path expressions may require several lookups, or post-processing the result set. In this section, we focus on the encoding of raw paths. Raw paths build on previous work in path indexing. (See Section 5).

Tagged data elements are represented as designator-encoded strings. We can regard all data elements as leaves in the XML tree. For example, the XML fragment

```
<A>alpha<B>beta<C>gamma</C></B></A>
```

(a) <invoice> = I  
 <buyer> = B  
 <name> = N  
 <address> = A  
 <seller> = S  
 <item> = T  
 <phone> = P  
 <count> = C  
 count (attribute) = C'

(b)

Document 1	Document 2
IBN ABC Corp	IBN Oracle Inc
IBA 1 Industrial Way	IBP 555-1212
ISN Acme Inc	ISN IBM Corp
ISA 2 Acme Rd.	ITC 4
IT drill	ITN nail
ITC' 2	
IT saw	
ITC' 3	

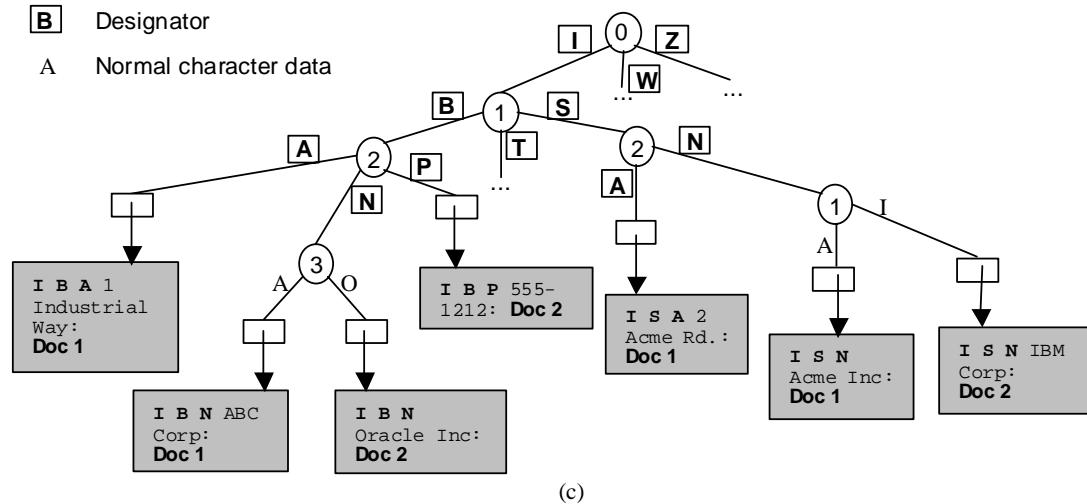


Figure 4. Raw paths.

can be represented as a tree with three root-to-leaf paths: <A>alpha, <A><B>beta and <A><B><C>gamma. If we assign **A**, **B** and **C** as the designators for <A>, <B> and <C> respectively, then we can encode the paths in this XML fragment as “**A** alpha”, “**A B** beta” and “**A B C** gamma.” This is a *prefix encoding* of the paths: the designators, representing the nested tag structure, appear at the beginning of the key, followed by the data element at the leaf of the path. This encoding does not require a pre-existing, regular or static schema for the data.

The alternative is *infix encoding*, in which data elements are nodes along the path. An infix encoding of the above fragment would be “**A** alpha **B** beta **C** gamma.” Here, for clarity, we will follow the convention of previous work, which is to treat data elements as leaves, and we will focus on the prefix encoding.

Tags can contain attributes (name/value pairs.) We treat attributes like tagged children; e.g. <A B="alpha">... is treated as if it were <A><B>alpha </B>.... The result is that attributes of <A> appear as siblings of the other tags nested within <A>. The label “**B**” is assigned different designators when it appears as a tag and an attribute (e.g. **B**=tag, **B'**=attribute).

At any time, a new document can be added to the raw path index, even if its structure differs from previously indexed documents. The root-to-leaf paths in the document are encoded as raw path keys, and inserted into the fabric. New tags that did not exist in the index

previously can be assigned new designators “on-the-fly” as the document is being indexed. Currently, this process does not preserve the sequential ordering of tags in the XML document. We have developed a system of alternate designators to encode order, but do not have space to discuss those techniques here.

### 3.2.1. Raw path example

The XML of Figure 3 can be encoded as a set of raw paths. First, we assign designators to tags, as shown in Figure 4(a). Next, we encode the root-to-leaf paths to produce the keys shown in Figure 4(b). Finally, we insert these keys in the Index Fabric to generate the trie shown in Figure 4(c). For clarity, this figure omits the horizontal layers and some parts of the trie.

### 3.3. Refined paths

Refined paths are specialized paths through the XML that optimize frequently occurring access patterns. Refined paths can support queries that have wildcards, alternates and different constants.

For example, we can create a refined path that is tuned for a frequently occurring query over the XML in Figure 3, such as “find the invoices where company X sold to company Y.” Answering this query involves finding <buyer> tags that are siblings of a <seller> tag within the same <invoice> tag. First, we assign a

designator, such as “**Z**,” to the path. (Recall that designators are just special characters or strings shown here in boldface for clarity.) Next, we encode the information indexed by this refined path in an Index Fabric key. If “Acme Inc” sold items to “ABC Corp,” we would create a key of the form “**Z ABC Corp Acme Inc**.” Finally, we insert the keys we have created into the fabric. The keys refer to the XML fragments or documents that answer the query. (See Section 3.4.)

This encoding scheme is similar to that used for raw paths, with designators and data elements in the same key. In a sense, we are overloading the metaphor of encoding paths as strings to support optimizing specific queries by encoding specialized paths. Raw and refined paths are kept in the same index and accessed using string lookups.

Adding new documents to the refined path index is accomplished in two steps. First, the new documents are parsed to extract information matching the access pattern of the refined path. Then, this information is encoded as an Index Fabric key and inserted into the index. Changes to refined paths are reflected in simple key updates.

The database administrator decides which refined paths are appropriate. As with any indexing scheme, creating a new access path requires scanning the database and extracting the keys for insertion into the index. Our structure grows slowly as new keys are inserted. Thus, unlike previous indexing schemes, we can pre-optimize a great many queries without worrying about resource contention between different indexes.

### 3.4. Combining the index with a storage manager

Because the Index Fabric is an index, it does not dictate a particular architecture for the storage manager of the database system. The storage manager can take a number of forms. The indexed keys can be associated with pointers that refer to flat text files, tuples in a relational system, or objects in a native XML database. In any case, searching the fabric proceeds as described, and the returned pointers are interpreted appropriately by the database system. In our implementation, both the index blocks and the actual XML data are stored in a relational database system. Thus, we leverage the maturity of the RDBMS, including concurrency and recovery features.

### 3.5. Accelerating queries using the Index Fabric

Path expressions are a central component of semistructured query languages (e.g. Lorel [2] or Quilt [6]). We focus on *selection* using path expressions, that is, choosing which XML documents or fragments answer the query, since that is the purpose of an index. We assume that an XML database system could use a standard approach, such as XSLT [31], to perform *projection*.

A *simple path expression* specifies a sequence of tags starting from the root of the XML. For example, the query “Find invoices where the buyer is ABC Corp” asks for XML documents that contain the root-to-leaf path

“`invoice.buyer.name.`ABC Corp`.`” We use a *key lookup operator* to search for the raw path key corresponding to the simple path expression.

Raw paths can also be used to accelerate *general path expressions*, which are vital for dealing with data that has irregular or changing structure because they allow for alternates, optional tags and wildcards. We expand the query into multiple simple path expressions, and evaluate each using separate key lookup operators. Thus, the path expression  $A.(B_1|B_2).C$  results in searches for  $A.B_1.C$  and  $A.B_2.C$ . This means multiple traversals but each traversal is a simple, efficient lookup.

If the query contains wildcards, then it expands to an infinite set. For example,  $A.(%)*.C$  means find every  $\langle C \rangle$  that has an ancestor  $\langle A \rangle$ . To answer this query, we start by using a *prefix key lookup operator* to search for the “A” prefix, and then follow every child of the “A” prefix node to see if there is a “C” somewhere down below. Because we “prefix-encode” all of the raw paths, we can prune branches deeper than the designators (e.g. after we see the first non-designator character.)

We can further prune the traversal using another structure that summarizes the XML hierarchy. For example, Fernandez and Suciu [15] describe techniques for utilizing partial knowledge of a graph structure to prune or rewrite general path expressions.

Queries that correspond to refined paths can be further optimized. The query processor identifies the query as corresponding to a refined path, and translates the query into a search key. For example, a query “Find all invoices where ABC Corp bought from Acme Inc” becomes “**Z ABC Corp Acme Inc**.” The index is searched using the key find the relevant XML. The search uses the horizontal layers and is very efficient; even if there are many millions of indexed elements, the answer can be found using at most a single index I/O.

## 4. Experimental results

We have conducted performance experiments of our indexing mechanism. We stored an XML-encoded data set in a popular commercial relational database system<sup>2</sup>, and compared the performance of queries using the DBMS’ native B-tree index versus using the Index Fabric implemented on top of the same database system. Our performance results thus represent an “apples to apples” comparison using the same storage manager.

### 4.1. Experimental setup

The data set we used was the DBLP, the popular computer science bibliography [11]. The DBLP is a set of XML-like documents; each document corresponds to a

---

<sup>2</sup> The license agreement prohibits publishing the name of the DBMS with performance data. We refer to it as “the RDBMS.” Our system can interoperate with any SQL DBMS.

```

<article key="Codd70">
  <author>E. F. Codd</author>,
  <title>A Relational Model of Data for Large
    Shared Data Banks.</title>,
  <pages>377-387</pages>,
  <year>1970</year>,
  <volume>13</volume>,
  <journal>CACM</journal>,
  <number>6</number>,
  <url>db/journals/cacm/cacm13.html#Codd70</url>
  <ee>db/journals/cacm/Codd70.html</ee>
  <cdrom>CACMs1/CACM13/P377.pdf</cdrom>
</article>

```

**Figure 5. Sample DBLP document.**

single publication. There are over 180,000 documents, totaling 72 Mb of data, grouped into eight classes (journal article, book, etc.) A document contains information about the type of publication, the title of the publication, the authors, and so on. A sample document is shown in Figure 5. Although the data is somewhat regular (e.g. every publication has a title) the structure varies from document to document: the number of authors varies, some fields are omitted, and so on.

We used two different methods of indexing the XML via the RDBMS’ native indexing mechanism. The first method, the *basic edge-mapping*, treats the XML as a set of nodes and edges, where a tag or atomic data element corresponds to a node and a nested relationship corresponds to an edge. The database has two tables, *roots(id,label)* and *edges(parentid,childid,label)*. The *roots* table contains a tuple for every document, with an *id* for the document, and a *label*, which is the root tag of the document. The *edges* table contains a tuple for every nesting relationship. For nested tags, *parentid* is the ID of the parent node, *childid* is the ID of the child node, and *label* is the tag. For leaves (data elements nested within tags), *childid* is *NULL*, and *label* is the text of the data element. For example, the XML fragment

```
<book><author>Jane Doe</author></book>
```

is represented by the tuple (0,book) in *roots* and the tuples (0,1,author) and (1,*NULL*,Jane Doe) in *edges*. (Keeping the leaves as part of the *edges* table offered better performance than breaking them into a separate table.) We created the following key-compressed B-tree indexes:

- An index on *roots(id)*, and an index on *roots(label)*.
- An index on *edges(parentid)*, an index on *edges(childid)*, and an index on *edges(label)*.

The second method of indexing XML using the DBMS’ native mechanism is to use the relational mapping generated by the STORED [12] system to create a set of tables, and to build a set of B-trees over the tables. We refer to this scheme as the *STORED mapping*. STORED uses data mining to extract schemas from the data based on frequently occurring structures. The extracted schemas are used to create “storage-mapped tables” (SM tables). Most of the data can be mapped into tuples and stored in the SM tables, while more irregularly structured data must be stored in *overflow buckets*, similar

Query	Description
A	Find books by publisher
B	Find conference papers by author
C	Find all publications by author
D	Find all publications by co-authors
E	Find all publications by author and year

**Table 1. Queries.**

to the edge mapping. The schema for the SM tables was obtained from the STORED investigators [13]. The SM tables identified for the DBLP data are *inproceedings*, for conference papers, and *articles*, for journal papers. Conference and journal paper information that does not fit into the SM tables is stored in overflow buckets along with other types of publications (such as books.)

To evaluate a query over the STORED mapping, the query processor may have to examine the SM tables, the overflow buckets, or both. We created the following key-compressed B-tree indexes:

- An index on each of the *author* attributes in the *inproceedings* and *articles* SM tables.
- An index on the *booktitle* attribute (e.g., conference name) in the *inproceedings* table.
- An index on the *id* attribute of each SM table; the *id* joins with *roots(id)* in the overflow buckets.

For both the edge and STORED mapping it was necessary to hand tune the query plans generated by the RDBMS, since the plans that were automatically tended to us inefficient join algorithms. We were able to significantly improve the performance (e.g. reducing the time to execute thousands of queries from days to hours).

The Index Fabric contained both raw paths and refined paths for the DBLP documents. The fabric blocks were stored in an RDBMS table. All of the index schemes we studied index the document IDs. Thus, a query processor will use an index to find relevant documents, retrieve the complete documents, and then use a post-processing step (e.g. with XSLT) to transform the found documents into presentable query results. Here, we focus on the index lookup performance.

All experiments used the same installation of the RDBMS, running on an 866 MHz Pentium III machine, with 512 Mb of RAM. For our experiments, we set the cache size to ten percent of the data set size. For the edge-mapping and STORED mapping schemes, the whole cache was devoted to the RDBMS, while in the Index Fabric scheme, half of the cache was given to the fabric and half was given to the RDBMS. In all cases, experiments were run on a cold cache. The default RDBMS logging was used both for queries over the relational mappings and queries over the Index Fabric.

We evaluated a series of five queries (Table 1) over the DBLP data. We ran each query multiple times with different constants; for example, with query B, we tried 7,000 different authors. In each case, 20 percent of the query set represented queries that returned no result because the key was not in the data set.

		I/O - Blocks						Time - Seconds								
	Edge Map		STORED		Raw path		Refined path		Edge Map		STORED		Raw path		Refined path	
	value	$\Delta$	value	$\Delta$	value	$\Delta$	value	$\Delta$	value	$\Delta$	value	$\Delta$	value	$\Delta$	value	$\Delta$
<b>A</b>	416	1.0	370	1.1	13	<b>32.0</b>	-	-	6	1.0	4	1.5	0.83	<b>7.2</b>	-	-
<b>B</b>	68788	1.0	26490	2.6	6950	<b>9.9</b>	-	-	1017	1.0	293	3.5	81	<b>12.6</b>	-	-
<b>C</b>	69925	1.0	61272	1.1	34305	2.0	20545	<b>3.4</b>	1056	1.0	649	1.6	397	2.7	236	<b>4.5</b>
<b>D</b>	353612	1.0	171712	2.1	89248	4.0	17337	<b>20.4</b>	5293	1.0	2067	2.6	975	5.4	208	<b>25.4</b>
<b>E</b>	327279	1.0	138386	2.4	113439	2.9	16529	<b>19.8</b>	4835	1.0	1382	3.5	1209	4.0	202	<b>23.9</b>

Table 2. Experimental results.

The experimental results are summarized in Table 2. (The  $\Delta$  column is speed-up versus edge mapping.) In each case, our index is more efficient than the RDBMS alone, with more than an order of magnitude speedup in some instances. We discuss the queries and results next.

#### 4.2. Query A: Find books by publisher

Query A accesses a small portion of the DBLP database, since out of over 180,000 documents, only 436 correspond to books. This query is also quite simple, since it looks for document IDs based on a single root-to-leaf path, “book.publisher.X” for a particular X. Since it can be answered using a single lookup in the raw path index, we have not created a refined path. The query can be answered using the basic edge-mapping by selecting “book” tuples from the *roots* table, joining the results with “publisher” tuples from the *edges* table, and joining again with the *edges* table to find data elements “X”. The query cannot be answered from the storage mapped tables (SM tables) in the STORED mapping. Because books represent less than one percent of the DBLP data, they are considered “overflow” by STORED and stored in the overflow buckets.

The results for query A are shown in Table 2, and represent looking for 48 different publishers. The raw path index is much faster than the edge mapping, with a 97 percent reduction in block reads and an 86 percent reduction in total time. The raw path index is also faster than accessing the STORED overflow buckets, with 96 percent fewer I/Os and 79 percent less time. Note that the overflow buckets require less time and I/Os than the edge mapping because the overflow buckets do not contain the information stored in the SM tables, while the edge mapping contains all of the DBLP information and requires larger indexes.

These results indicate that it can be quite expensive to query semistructured data stored as edges and attributes. This is because multiple joins are required between the *roots* and *edges* table. Even though indexes support these joins, multiple index lookups are required, and these increase the time to answer the query. Moreover, the DBLP data is relatively shallow, in that the path length from root to leaf is only two edges. Deeper XML data, with longer path lengths, would require even more joins and thus more index lookups. In contrast, a single index lookup is required for the raw paths.

#### 4.3. Query B: Find conference papers by author

This query accesses a large portion of the DBLP, as conference papers represent 57 percent of the DBLP publications. We chose this query because it uses a single SM table in the STORED mapping. The SM table generated by STORED for conference papers has three author attributes, and overflow buckets contain any additional authors. In fact, the query processor must take the union of two queries: first, find document IDs by author in the *inproceedings* SM table, and second, query any *inproceedings.author.X* paths in the *roots* and *edges* overflow tables. Both queries are supported by B-trees. The edge mapping uses a similar query to the overflow buckets. The query is answered with one raw path lookup (for *inproceedings.author.X*) and we did not create a refined path.

The results in Table 2 are for queries with 7,000 different author names. Raw paths are much more efficient, with an order of magnitude less time and I/O’s than the edge mapping, and 74 percent fewer I/Os and 72 percent less time than the STORED mapping. We have plotted the I/Os in Figure 7 with the block reads for index blocks and for data blocks (to retrieve document IDs) broken out; the data reads for the Index Fabric include the result verification step for the Patricia trie. For the STORED mapping, Figures 6 and 7 separate I/Os to the edge-mapped overflow buckets and I/Os to the SM tables.

Although SM tables can be accessed efficiently (via a B-trees on the *author* attributes), the need to go to the overflow buckets to complete the query adds significant overhead. The performance of the edge mapping, which is an order of magnitude slower than the raw paths, confirms that this process is expensive. This result illustrates that when some of the data is irregularly structured (even if a large amount fits in the SM tables), then the performance of the relational mappings (edge and STORED) suffers.

#### 4.4. Other queries

Query C (find all document IDs of publications by author X) contains a wildcard, since it searches for the path “(%)\*.author.X.” The results in Table 2 represent queries for 10,000 different author names.

Query D seeks IDs of publications co-authored by author “X” and author “Y.” This is a “sibling” query that

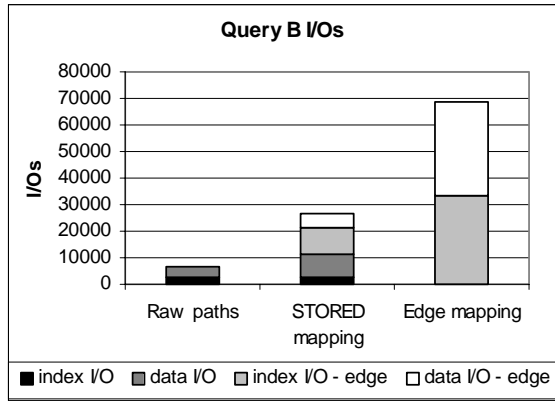


Figure 6. Query B: find conference paper by author.

looks for two tags nested within the same parent tag. The results in Table 2 are for queries on 10,000 different pairs of authors, and the I/Os are shown in Figure 7.

Query E (find IDs of publications by author  $X$  in year  $Y$ ) also seeks a sibling relationship, this time between  $\langle \text{author} \rangle$  and  $\langle \text{year} \rangle$ . The difference is that while  $\langle \text{author} \rangle$  is very selective (with over 100,000 unique authors), there are only 58 different years (including items such as “1989/1990”). Consequently, there are a large number of documents for each year. The results in Table 2 are for 10,000 author/year pairs.

The results shown in Table 2 illustrate that irregularly structured data is a significant obstacle to managing semistructured data in a relational system. For the STORED mapping, the SM tables can be accessed efficiently, but the queries cannot be fully answered without costly access to the overflow buckets. The edge mapping (which treats all data as irregularly structured) is even less efficient, since every query must be evaluated using expensive self-joins. Thus, even though there are multiple raw path lookups for queries C, D and E, the raw paths outperform the relational mappings in each case. Moreover, the refined paths offer a significant optimization, especially for complex queries.

## 5. Related work

The problem of storing, indexing and searching semistructured data has gained increasing attention [1,5,6,23]. Shanmugasundaram et al [26] have investigated using DTD’s to map the XML data into relational tables. The STORED system extracts the schema from the data itself using data mining [12]. Both [26] and [12] note that it is difficult to deal with data that has irregular or variable structure. Florescu and Kossmann have examined storing XML in an RDBMS as a set of attributes and edges, using little or no knowledge of the document structure [17], for example, the edge mapping we examine here. Other systems store the data “natively” using a semistructured data model [24,28,29]. Evaluating path expressions in these systems usually requires multiple index lookups [23]. Raw paths are

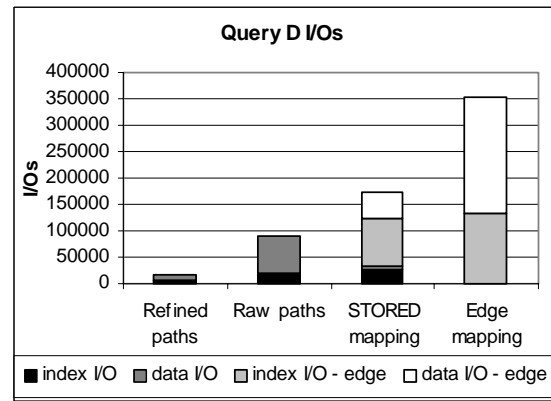


Figure 7. Query D: Find publications by co-authors.

conceptually similar to DataGuides [18].

A join index, such as that proposed by Valduriez [27], precomputes joins so that at query time, specific queries are very efficient. This idea is similar in spirit to our raw and refined paths. However, a separate join index must be built for each access path. Moreover, a join index is sensitive to key length, and is usually only used for a single join, not a whole path.

Path navigation has been studied in object oriented (OO) databases. OO databases use sequences [4,22] or hierarchies of path indexes [32] to support long paths, requiring multiple index lookups per path. Our mechanism supports following paths with a single index lookup. Also, OO indexes support linear paths, requiring multiple indexes to evaluate “branchy” queries. Our structure provides a single index for all queries, and one lookup to evaluate the query using a refined path. Third, semistructured data requires generalized path expressions in order to navigate irregular structure. Although Christophides et al. [8] have studied this problem, their work focuses on query rewriting and not indexes, and our mechanism could utilize their techniques (or those of [15]) to better optimize generalized path expressions over raw paths. Finally, OO indexes must deal with class inheritance [7], while XML indexes do not.

Text indexing has been studied extensively in both structured and unstructured databases. Suffix arrays and compressed suffix arrays [16], based on Patricia tries, provide partial-match searching rather than path navigation. Several data and query models for structured data besides XML have been studied [3]; our techniques can be adapted for these other models. Others have extended text indexes and multidimensional indexes to deal with structured data [20]; our structural encoding is new, and we deal with all of the structure in one index.

The Index Fabric is a balanced structure like a B-tree [9], but unlike the B-tree, scales well to large numbers of keys and is insensitive to the length or complexity of keys. Diwan et al have examined taking general graph structures and providing balanced, disk based access [14]. Our structure is optimized specifically for Patricia tries.

## 6. Conclusions

We have investigated encoding paths through semistructured data as simple strings, and performing string lookups to answer queries. We have investigated two options: raw paths, which assume no *a priori* knowledge of queries or structure, and refined paths, which take advantage of such knowledge to achieve further optimization. Our techniques rely on the Index Fabric for high performance string lookups over a large set of non-uniform, long, and complex strings. While the indexing mechanisms of an RDBMS or semistructured data repository can provide some optimization, they have difficulty achieving the high performance possible with our techniques. Our experimental results confirm that implementing our techniques on top of an RDBMS offers a significant improvement over using the RDBMS's native indexes for semistructured data. This is especially true if the query is complex or branchy, or accesses "irregular" portions of the data (that must be stored in overflow buckets). Clearly, the Index Fabric represents an effective way to manage semistructured data.

## Acknowledgements

The authors would like to thank Alin Deutsch, Mary Fernandez and Dan Suciu for the use of their STORED results for the DBLP data. We also want to thank Donald Kossmann for helpful comments on a draft of this paper.

## References

- [1] S. Abiteboul. Querying semi-structured data. In *Proc. ICDT*, 1997.
- [2] S. Abiteboul et al. The Lorel query language for semistructured data. *Int. J. on Digital Libraries* 1(1): 68-88, 1997.
- [3] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text traversal. *SIGMOD Record* 25(1): 67-79, 1996.
- [4] E. Bertino. Index configuration in object-oriented databases. *VLDB Journal* 3(3): 355-399, 1994.
- [5] P. Buneman et al. A query language and optimization techniques for unstructured data. In *Proc. SIGMOD*, 1996.
- [6] D. Chamberlain, J. Robie and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proc. WebDB Workshop*, 2000.
- [7] S. Choenni et al. On the selection of optimal index configuration in OO databases. In *Proc. ICDE*, 1994.
- [8] V. Christophides, S. Cluet and G. Moerkottke. Evaluating queries with generalized path expressions. In *Proc. SIGMOD*, pages 413-422, 1996.
- [9] D. Comer. The ubiquitous B-tree. *Computing Surveys* 11(2): 121-137, 1979.
- [10] B. Cooper and M. Shadmon. The Index Fabric: A mechanism for indexing and querying the same data in many different ways. Technical Report, 2000. Available at <http://www.rightorder.com/technology/overview.pdf>.
- [11] DBLP Computer Science Bibliography. At <http://www.informatik.uni-trier.de/~ley/db/>.
- [12] A. Deutsch, M. Fernandez and D. Suciu. Storing semistructured data with STORED. In *Proc. SIGMOD*, 1999.
- [13] Alin Deutsch. Personal communication, January 24, 2001.
- [14] A. A. Diwan et al. Clustering techniques for minimizing external path length. In *Proc. 22<sup>nd</sup> VLDB*, 1996.
- [15] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. ICDE*, 1998.
- [16] P. Ferragina and G. Manzini. An experimental study of a compressed index. In *Proc. ACM-SIAM SODA*, 2001.
- [17] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. INRIA Technical Report 3684, 1999.
- [18] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proc. 23<sup>rd</sup> VLDB*, pages 436-445, 1997.
- [19] Alon Itai. The JS Data Structure. Technical report, 1999.
- [20] H. V. Jagadish, N. Koudas and D. Srivastava. On effective multi-dimensional indexing for strings. In *Proc. SIGMOD*, 2000.
- [21] Donald Knuth. *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition*. Addison Wesley, Reading, MA, 1998.
- [22] W. C. Lee and D. L. Lee. Path Dictionary: A New Approach to Query Processing in Object-Oriented Databases. *IEEE TKDE*, 10(3): 371-388, May/June 1998.
- [23] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. 25<sup>th</sup> VLDB*, 1999.
- [24] J. McHugh et al. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3): 54-66, 1997.
- [25] Oracle Corp. *Oracle 9i database*. <http://www.oracle.com/ip/dep/otn/database/9i/index.html>.
- [26] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. 25<sup>th</sup> VLDB*, 1999.
- [27] P. Valduriez. Join Indices. *TODS* 12(2): 218-246, 1987.
- [28] Software AG. *Tamino XML database*. <http://www.softwareag.com/tamino/>.
- [29] XYZFind. *XML Database*. <http://www.xyzfind.com>.
- [30] W3C. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, October 6, 2000. See <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [31] W3C. *XSL Transformations (XSLT) 1.0*. W3C Recommendation, November 16, 1999. See <http://www.w3.org/TR/1999/REC-xslt-19991116>.
- [32] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proc. VLDB*, 1994.