Paper ID AMERICAS 138

A fast index for semistructured data

Brian Cooper[1,2*], Neal Sample[1,2], Michael J. Franklin[1,3], Gísli R. Hjaltason[1] and Moshe Shadmon[1]

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

[1]RightOrder Incorporated
3850 N. First St.
San Jose, CA 95134 USA

[2]Department of Computer Science
Stanford University
Stanford, CA 94305 USA

[3]Computer Science Division
University of California
Berkeley, CA 94720 USA

Category: Research

Topic area: Core DB Technology Area

Topics: 1. Semi-structured Data

2. Optimization and Performance

* Contact: Brian Cooper

E-mail: cooperb@DB.Stanford.EDU

Mail: Gates Hall 4A, room 435
Stanford, CA 94305-9040 USA

Phone: (650) 248-4259

A fast index for semistructured data

Brian Cooper[1,2], Neal Sample[1,2], Michael J. Franklin[1,3], Gísli R. Hjaltason[1] and Moshe Shadmon[1]

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

[1]RightOrder Incorporated
3850 N. First St.
San Jose, CA 95134 USA

[2]Department of Computer Science
Stanford University
Stanford, CA 94305 USA

[3]Computer Science Division
University of California
Berkeley, CA 94720 USA

Abstract

Semistructured data contains many complex relationships between data items; morever, the structure of these relationships is irregular and can frequently change. Queries over such data navigate these relationships via path expressions. Our solution is based on encoding paths as strings, and inserting those strings into a special index that is highly optimized for long and complex keys. This provides the flexibility needed to query and manage semistructured data without sacrificing high performance. We describe the Index Fabric, an indexing structure that provides the efficiency and flexibility we need. The Index Fabric contains "raw paths," paths that can be used to optimize ad hoc queries over semistructured sources, and "refined paths," indexing paths optimized for specific queries. Although we can use knowledge about the queries and structure of the data to create refined paths, no such *a priori* 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 native 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 it can be queried by buyers. 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. In order to support high performance queries, it must be possible to efficiently navigate the relationship structure. Typically, indexes must be constructed to support efficient access.

One option for managing semistructured data is to store and query it using a relational database. The data must be converted into a set of tuples and stored in structured tables. For example, Oracle 8i/9i provides a set of tools to ease the process of creating a mapping between the semistructured data and the relational system [22]. Even if a schema exists for the data, this process is not trivial, and it is difficult to efficiently evaluate queries without extensions to the relational model [23]. The problem is even harder if there is no explicit schema. It is possible to extract a partial schema from the data using data mining techniques, as done with the STORED system, although data that does not fit the schema well must be stored in its native form as a set of edges between data elements [12]. Another possibility is to store the whole data set in a relational database as a set of data elements and edges, and ignore the issue of schema altogether [16]. Querying a relational

representation of edges and elements is quite inefficient, since navigating a long or complex path requires multiple joins of elements along the path. Even if joins can be optimized using indexes, traversing a path requires multiple random lookups in several different indexes, which (as shown in Section 4) is still expensive.

An alternative option for managing semistructured data is to build a specialized data manager that contains a semistructured data repository at its core. Projects such as Lore [21] and industrial products such as Tamino [25] and XYZFind [26] 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 [20]. 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 through the data as strings, and inserts these strings into an index that is highly optimized for string storage and searching. The index blocks and semistructured data both are stored in a conventional relational database system. Evaluating queries involves encoding the desired 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 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 offers 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 [18] 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/O's. 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 provide high optimization for 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 single, simple searching mechanism that can be tuned for different needs.

Our work is focused on indexing the data to provide fast searches. 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.

1.1. Indexing XML encoded data

A popular syntax for semistructured data is XML [27], and in this paper we focus on using the Index Fabric to index XML-encoded data. XML represents semistructured data using data elements surrounded by tags, which themselves can be nested within other tags. For example, consider the following fragment:

```
<invoice number="233">
   <buyer>Alpha Corp</buyer>
   <seller>Beta, Inc</seller>
   <item>Hammer</item>
   <item>Wrench</item>
</invoice>
```

This fragment represents a simple invoice, in which Beta Inc sold a hammer and a wrench to Alpha Corp. This fragment may be one part of a large XML database containing millions of invoices. One possible query is "Find all invoices where Beta Inc sold to Alpha Corp." This query searches for `<invoice>` documents where `<buyer>Alpha Corp</buyer>` is a sibling of `<seller>Beta Inc</seller>`. We can answer this query by performing a lookup for the raw path "`invoice.buyer.`Alpha Corp'`" to retrieve a set of IDs of documents where Alpha Corp was a buyer. Next, we perform another lookup for the raw path "`invoice.seller.`Beta Inc'`" to get another set of document IDs. Taking the intersection of these two sets gives us the set of documents that answers the query.

Alternatively, we may choose to create a refined path for the query. First, we parse the XML documents to find those that have `<buyer>` and `<seller>` tags as children of the same `<invoice>` tag. When we find such documents, we extract the text tagged by `<buyer>` and `<seller>`, and create a single refined path key to insert in the index. For example, the XML fragment above produces the key "`invoice.buyer=`Alpha Corp'.invoice.seller=`Beta Inc'`." Then, we execute the query "Find all invoices where *X* sold to *Y*" by looking up "`invoice.buyer=`*X*`.invoice.seller=`*Y*" in the Index Fabric.

This approach differs from previous work on both native and RDBMS-based XML storage. For example, Lore's indexes store pairwise links between data elements (e.g. parent-child relationships between tags), and searches are conducted by following a series of these pairwise links, requiring multiple index lookups [20]. Lore also offers a *DataGuide* index [17] a form of path index similar to our raw paths. Since the DataGuide is itself a semistructured data object, traversing a path may require multiple object lookups. With the Index

Fabric, a semistructured repository could reduce the number of lookups to one (with refined paths) or one per searched path (with raw paths) instead of multiple lookups per searched path.

The other conventional approach is to store and search semistructured data using a structured database system. Traditional indexes can accelerate queries if the data is essentially structured, such as relational data encoded in XML. However, irregular portions of the data must be decomposed into a series of pairwise links to create homogenous tuples that can be managed by a traditional index. These links must be laboriously reconstructed at query time to support path navigation. With our mechanism, the semistructured data could still be stored in a relational system, but queried directly in an efficient manner.

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. Under the Index Fabric, we have constructed both refined and raw paths, while the relational index utilized a basic edge mapping as well as an extracted schema generated by the STORED [12] system. Both refined and raw paths are significantly faster than the DBMS's native indexing mechanism, with up to an order of magnitude improvement. The improvements are particularly striking for data with irregular structure, or queries that must navigate multiple paths.

1.2. 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 is up to 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 discuss 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

Representing and searching complex paths as strings requires a specialized indexing mechanism that supports quick lookups of strings regardless of their length or complexity. 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. This provides the needed component that makes it possible to treat semistructured data paths as strings.

The Index Fabric is based on a compressed form of tries called *Patricia* tries [18]. A Patricia trie differs from a standard trie in that nodes with one child are compressed into their parent node, so that all nodes have at least two children. An example Patricia trie is shown in Figure 1, where the nodes are labeled with their *depth*: the character position in the key represented by the node. Because not every character of the key is examined during the search, the record that is ultimately found must be checked against the search key. For example, if we search for "cut" in Figure 1, we will reach the "cat" record. 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 regardless of the actual key length. Furthermore, unlike B-trees, Patricia tries grow slowly even as large numbers of strings are inserted because of the aggressive (lossy) compression inherent in the structure.

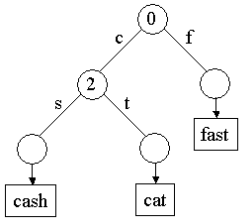

Figure 1. A Patricia trie.

Although researchers have long known about Patricia tries, such structures have rarely been used to manage large amounts of data, especially disk-based data, because they are unbalanced and best suited for usage in main memory. What is needed 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 Index Fabric is such a structure. It uses a novel layered trie approach to transform a Patricia trie into a disk-based index: 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 subtries, and these blocks are indexed by a second trie, stored in its own blocks. 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 indexes the subtries of layer 0 by indexing the common prefixes of layer 0, where a common prefix is the prefix represented by the root node of a subtrie. (In the figure, the common prefix for each block is shown in "quotes".) Thus, in the figure, "cas" is the common prefix of the subtrie rooted at the node labeled "3" in layer 0, and the trie in layer 1 can be searched for "cas" to reach this subtrie. Similarly, the trie in layer 2 indexes the common prefixes of layer 1. Although Figure 2 shows a three layer trie, we can create as many layers as necessary. The leftmost layer always contains one block, which is the root of the horizontal layer structure.

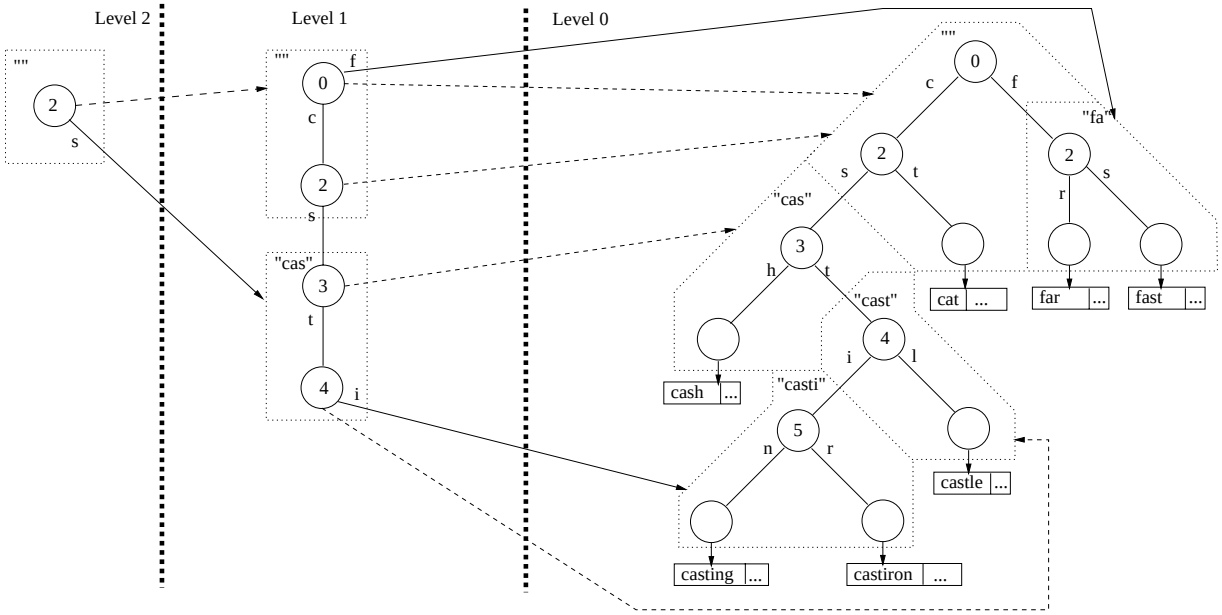

Figure 2. A layered index.

The trie at each horizontal layer refers to the subtries in the next layer to the right using two kinds of pointers. The first, called a *far link* (→), is labeled, while the second, a *direct link* (⇢) is unlabeled. Far links are similar to normal edges in a Patricia trie, except that a far link connects a node in one layer to a subtrie in the next layer. In contrast, a direct link connects a node in one layer with a block containing a node at the same depth and representing the same prefix in the next layer. Thus, in Figure 2, the node labeled "3" in layer 1, which corresponds to the prefix "cas," is connected to a subtrie (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 root block, which resides in the leftmost horizontal layer. Within a particular block, the search proceeds like a normal Patricia search down the vertical trie, 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 layer. In this way, the search proceeds from layer to layer until the lowest layer (layer 0) is reached and the desired data is found. The search algorithm is shown in Figures 3 and 4.

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. For example, if we search for "flag" in Figure 2, we will reach the rightmost node labeled "2" in the lower layer. Because there is no outgoing edge labeled "a," we know that "flag" does not exist in the index. Otherwise, the path is followed to the data. As noted above, it is necessary to verify that the found data matches the search key (at the end of procedure SEARCH).

```
address SEARCH(key x) {
   B := root block of index;
   while B is not a pointer to data do {
       B := SEARCH_BLOCK(x, B);
       if B = NULL then
           return NULL;
   }

   if key(B) = x then
       return B;
   else
       return NULL;
}
```

Figure 3. Procedure SEARCH.

The search process examines one block per layer[1]. In a disk-based index, in which 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. The horizontal tree serves to balance the search process, because every search must examine the same number of layers. One benefit of using the Patricia structure is that keys are stored very compactly, and many keys can be indexed per block. As a result, blocks have a very high out-degree, or number of far and direct links referring to child blocks in 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 be easily stored in main memory. Key lookups require at most one I/O, for the leaf index layer (in addition to data I/O's). 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 consists of a deletion of an old version of a key followed by an insertion of a new version. Inserting a key into a Patricia trie involves either adding a single new node or adding an edge to an existing node. Thus, the insertion requires a local change in layer 0 which can be handled within a single block. The affected block must be found, with a simple search utilizing the horizontal index, so it can be updated. If the affected block overflows, it must be split, requiring a new node at layer 1. This change is also confined to a single layer 1 block. Splits propagate left in the horizontal layers if at each layer blocks overflow. However, in each case, a single block per layer is affected. Moreover, splits are rare. Thus, 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. This is how the Index Fabric grows horizontally.

[1] The compression introduced by the Patricia means that it is possible for SEARCH to make a "mistake," entering an incorrect block and then having to backtrack. In practice, such mistakes are rare in a well-populated tree. See [10].

```
address SEARCH_BLOCK(key x, block B) {
    n := root node of B;
    i := depth(n); /* the position in the search key represented by the node */
    B' := NULL; /* this will be the block in the next layer */

    while B' = NULL and i < length(x) and
      a near or far edge n → n' labeled x[i] exists {
        if n → n' is a far link then { /* n' is in the next layer */
            B' := address of block pointed to by n → n';
            } else { /* n → n' is a near link */
            n = n';
            i := depth(n);
        }
    }

    if B' = NULL and there exists a direct edge n → n' then
        B' := address of block pointed to by the direct edge n → n';

    /* at this point, B' may be a pointer to another block, a pointer to data,
       or NULL */
    return B';
}
```

Figure 4. Procedure SEARCH_BLOCK.

Deletion of a key is performed in a similar manner. 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. If this results in a node with a single child, that node is removed. 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 the 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. The fabric allows us to quickly look up an entire path, or a prefix of a path. In this section, we discuss encoding XML paths as keys for insertion into the fabric; then we discuss using path lookups to evaluate queries in Section 3.5. There are two types of paths that we encode: paths reflecting the hierarchical organization of XML (raw paths, Section 3.2) and traversals along other paths that are specialized for particular queries (refined paths, Section 3.3). Both types of paths are encoded using self-describing elements called *designators*, (Section 3.1). As a running example, we will use the XML in Figure 5.

3.1. Designators

In order to represent different paths encoded as strings in the same index, we encode the paths using *designators:* special characters or character strings that are prefixed to data elements in the path string. We choose a unique designator for each tag that appears in the XML. For example, for the XML in Figure 5, we can choose **I** for `<invoice>`, **B** for `<buyer>`, **N** for `<name>`, and so on. (For purposes of illustration, here

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

Figure 5. Sample XML.

we will represent designators as boldface characters.) Then, the string "**IBN**ABC 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 as normal characters, though from our perspective 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 looked up in the dictionary to find the matching designator. New designators are generated automatically for new tags. The tag names from queries are also translated into designators using the dictionary, so that we can form a search key over the Index Fabric. (Queries are discussed in Section 3.5.)

3.2. Raw paths

Raw paths index the hierarchical structure of the XML by encoding root-to-leaf paths as strings. Raw paths therefore support, with a single index lookup, path expressions that involve traversing the XML from the root to the leaf. Other path expressions may require several lookups, or post-processing the result set. Nonetheless, raw paths can provide an important optimization step for ad hoc queries over data with irregular or changing structure. In this section, we focus on the encoding of raw paths. Raw paths draw on previous work in path indexing, but add the vital component of fast lookups using the Index Fabric. (Related work is addressed in Section 5).

As mentioned above, tagged data elements are represented as designator-encoded strings. Data elements in XML are represented as strings, and thus encoding them into index keys is very natural. We can regard all data elements as leaves in the XML tree. For example, consider the following XML fragment:

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

This fragment 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.

The alternative to this encoding is an *infix encoding*, in which data elements are represented as nodes along the path. In this case, the XML fragment above results in a single key, encoded as "**A** alpha **B** beta **C** gamma." Both the prefix and infix encodings are supportable in our fabric structure, and each is useful in different situations. For example, the path query "`A.B.C.`gamma'" (e.g., ignore the "alpha," "beta" and any other data elements along the path) is best supported by a lookup for the prefix encoded key "**A B C** gamma." In contrast, the path expression "`A.`alpha'.B.`beta'.C.`gamma'" (e.g., the "alpha" and "beta" must exist along the path) is best supported by a lookup for the infix encoded path "**A** alpha **B** beta **C** gamma."

We can take advantage of the aggressive key compression available in the Index Fabric to encode paths as (potentially very long) strings. For example, in the prefix encoding, it is useful to start the data element portion of the encoding at a well known position in the string, say, 100. Thus, the key "**A B** beta" is actually the string "**AB**", followed by 98 blanks, followed by "beta." However, because the Index Fabric is based on Patricia tries, the index automatically compresses out the superfluous blanks, while retaining the fact that the "beta" began at position 100. Similarly, in the infix encoding, we can start each element along the path at a predefined position. We might represent the key "**A** alpha **B** beta **C** gamma" with the "**A** alpha" starting at position 0, the "**B** beta" at position 100, and the "**C** gamma" at position 200. This technique becomes useful when we wish to navigate the tree to discover which tags are present, because we know where to find the designators (representing tags) in each key. This issue is revisited in Section 3.5.1.

Note that our path encoding does not rely on a pre-existing schema for the data. As a result, the data can have irregular and variable structure without causing difficulty for our techniques. Also, because the Index Fabric scales well to large numbers of keys, it is possible to support both prefix and infix encodings simultaneously in the same index. In this paper, for clarity and brevity, we will follow the convention of previous work in semistructured data, which is to treat data elements as leaves, and thus we will focus on the prefix encoding.

XML tags can contain attributes, which are name/value pairs. Our approach is to treat attributes as if they were tagged children. For example, the tag `<A B="alpha">...` can be treated as if it were written `<A><B>alpha</B>...` to allow us to index the "B" as a child of `<A>`. The result is that attributes of a tag `<A>` appear as siblings of the other tags nested within `<A>`. However, we still want to be able to identify "B" as an attribute rather than an actual tag. This is done by assigning different designators to the label "B" when it appears as a tag and an attribute (e.g. **B**=tag, **B'**=attribute). The designator dictionary maps both of these designators to "B" and indicates which designator represents "B" as a tag and which represents "B" as an attribute. (Of course, "B" may appear only as an attribute; then only one designator needs to be mapped to it, but this designator should still be marked in the dictionary as representing an attribute and not a tag.)

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. If the document contains a new tag that did not exist in the index previously, this tag can be assigned a new designator "on-the-fly" as the document is being indexed. Currently, this process does not preserve the sequential ordering of tags in the XML document. For example, item "saw" comes before item "drill" in document 1 of Figure 5. We have developed a system of alternate designators that encode order in the fabric structure, but do not have space to discuss those techniques here.

3.2.1. Raw path example

The XML of Figure 5 can be encoded as a set of raw paths. We start by assigning designators to tags:

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

Next, we encode the root-to-leaf paths to produce the following keys (padded with blanks as described):

Document 1		Document 2	
I B N ABC Corp	**I T** drill	**I B N** Oracle Inc	**I T C** 4
I B A 1 Industrial Way	**I T C'** 2	**I B P** 555-1212	**I T N** nail
I S N Acme Inc	**I T** saw	**I S N** IBM Corp	
I S A 2 Acme Rd.	**I T C'** 3		

Finally, we insert these keys in the Index Fabric to generate the trie shown in Figure 6. For clarity, this figure omits the horizontal layer structure. The figure also shows branches from the root marked **W** and **Z**; these edges correspond to refined paths, which we describe next.

3.3. Refined paths

Refined paths represent specialized traversals of the XML that are appropriate for frequently occurring access patterns. The data elements may be organized in the most efficient manner for these patterns, even if that organization does not correspond to the original XML tree structure and may exclude some of the data from the original XML. These access patterns support queries which can have wildcards, alternates and different constants. For example, consider the XML in Figure 5, which shows invoices with buyers, sellers and items. A frequently occurring query over this XML may be "find the invoices where company *X* sold to company *Y*." In other words, the relevant XML documents will have the following form:

```
<invoice>
  <buyer>
    <name>X</name>
  </buyer>
  <seller>
    <name>Y</name>
  </seller>
</invoice>
```

for specific values of *X* and *Y*. Such a query must do more than just traverse the XML once from the root to a leaf. Instead, answering this query involves finding `<buyer>` tags that are siblings of a `<seller>` tag within the same `<invoice>` tag. We can create a refined path that is tuned for this query. First, we assign a

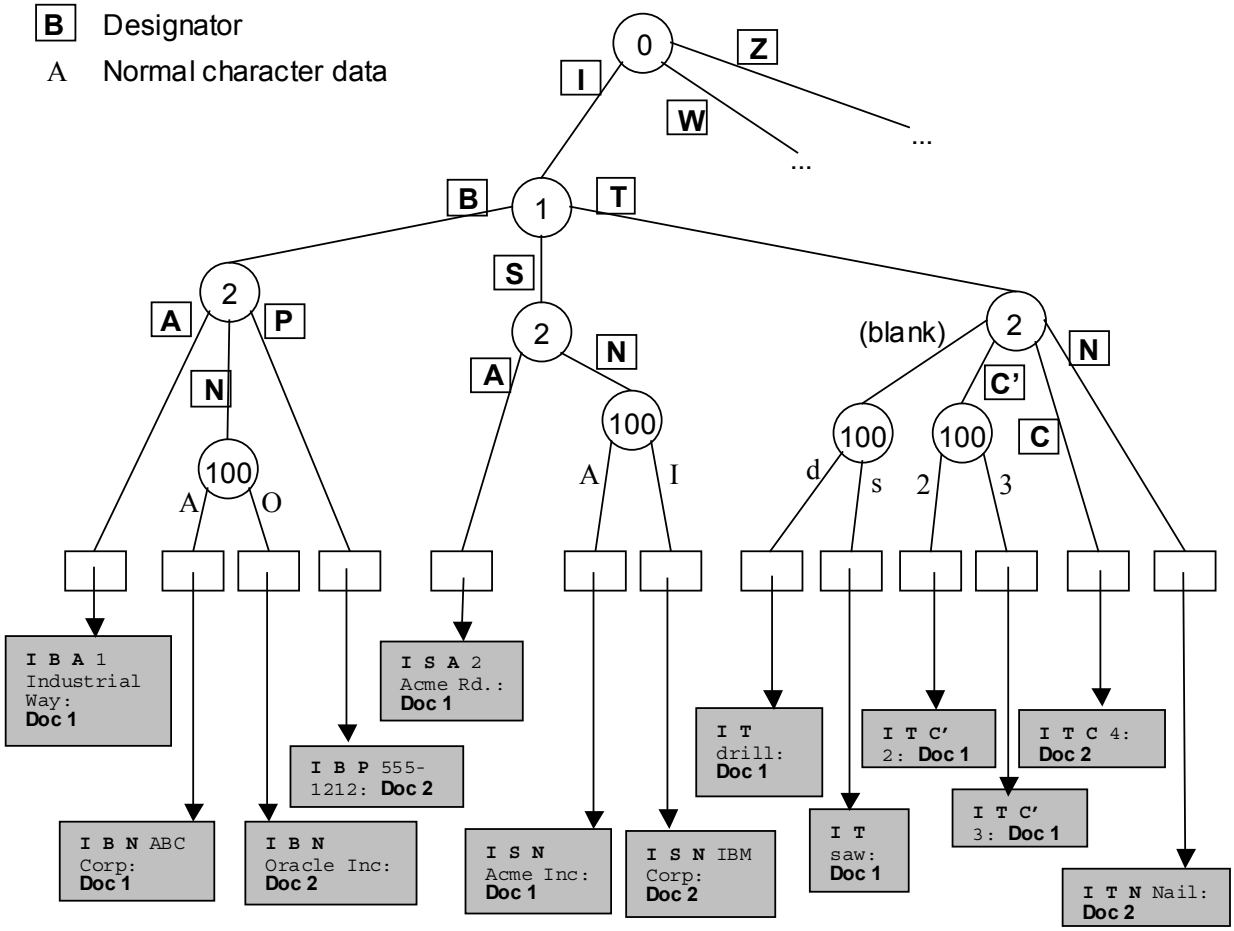

Figure 6. Raw paths.

designator to the path. For this example, we can choose "**Z**." (Recall that designators are just special characters or strings, and are shown here in boldface for clarity.) Next, we encode the information indexed by this refined path in an Index Fabric key in any appropriate way that allows us to distinguish the buyer from the seller. For example, we could choose to put the buyer at position 100 and the seller at position 200, e.g. "**Z** [99 blanks] *X* [99blanks] *Y*." Then, if "Acme Inc" sold items to "ABC Corp," we would create a key of the form "**Z** ABC Corp Acme Inc" (padded with blanks as described). Finally, we insert the keys we have created into the fabric, with the leaves of the trie referring to the XML that is the answer to the query. Thus, following the "**Z** ABC Corp Acme Inc" path in the trie would lead to a pointer that refers to document 1. The refined path could lead to multiple XML documents or fragments if they are all answers to the query. (A discussion of how the Index Fabric refers to data is in Section 3.4.)

Note that we are using an encoding scheme similar to that used for raw paths; that is, designators and data elements in the same key, padded with blanks (that are not stored by the Patricia trie) as necessary. In a sense, we are overloading the simple metaphor of encoding paths as strings to support encoding new paths that do not actually exist in the hierarchical structure of the XML. We have created these new paths for the purpose of

optimizing queries. The fact that we choose different designators for raw paths and refined paths allows us to keep both types of keys in the same index and to treat them in a uniform manner (as strings).

Other refined paths can be built to support more complicated queries, such as queries with wildcards. We may want to support searches for companies in invoices, regardless of whether they are a buyer or a seller. Then, we are looking for structures of the form "<invoice><?><name>*X*</name></?></invoice>" where *X* is a company name and <?> can be either <buyer> or <seller>. We can create a path designated **W** to support this query. The **W** path is constructed by examining new documents for <name> tags, and if the tag occurs as a child of an <invoice> tag, then the data element *X* is extracted and indexed. Due to space limitations, we do not show the trie structure for the refined paths.

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. Refined paths can be maintained efficiently, even for documents with irregular structure, as changes are reflected in simple key insertions.

It is up to the data administrator to decide which refined paths are appropriate. Each new path is created by inserting a new set of keys into the fabric. As with any indexing scheme, creating a new access path requires scanning the database to extract the appropriate information for constructing the keys. Because our structure grows slowly as new keys are inserted, multiple refined paths referring to the same XML data can coexist in a single Index Fabric. Thus, unlike previous indexing schemes, we can pre-optimize a great many queries without worrying about the negative performance impacts of resource contention between different indexes. In addition to representing tags, refined paths can be created to encode details such as attributes within a tag, the textual order that elements appear in the XML, or specific branching structures. Moreover, the refined paths deal with the difficulty of managing variable or irregular structure within the data at document parse time, reducing the difficulty faced by the query processor at query time.

3.4. Combining the Index Fabric 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. As a result, the storage manager can take a number of forms. One simple possibility is that the XML documents are stored as text files. In this case, searches over the Index Fabric can return pointers to the files and offsets within the files. Another possibility is that the underlying storage manager is a semistructured database with its own internal representation of parsed XML. Then, searching the Index Fabric can provide pointers that are meaningful in this native representation. A third possibility is that the XML has been stored in relational tables. In this case, the Index Fabric can point to tuples that form the results of the query. In any case, searching the fabric proceeds as described above, and the returned pointers are interpreted by the database system in the appropriate way. In our implementation, both the Index Fabric blocks and the

actual XML data are stored in a relational database system, leveraging the stability of a commercial system. This includes using the concurrency and recovery features offered by the DBMS to support index updates.

3.5. Accelerating queries using the Index Fabric

We can support path expressions by using a *key lookup* operator for a full key to find the corresponding data record. The search for the key proceeds efficiently, using the horizontal structure of the index. In addition, a *prefix key lookup* operator supports looking up the prefix of a key, and returning all paths with that prefix. For example, for the prefix "**A B**", we can find "**A B C** alpha", "**A B D** beta", "**A B** delta" and so on. We use the specified prefix to traverse the horizontal layers of the layered trie to the leaf layer. We then traverse the subtrie representing the prefix to find all of the pointers to data. The subtrie is traversed wholly within the vertical index of the leaf layer, examining multiple blocks if necessary.

Path expressions are a central component of any semistructured query language (such as Lorel [2] or Quilt [5]). We focus on *selection*, 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 [28], to perform *projection*.

3.5.1. Answering queries using raw paths

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 a root-to-leaf path of the form "`invoice.buyer.name.`ABC Corp'." We then use the *key lookup operator* to perform a lookup in the Index Fabric for the raw path corresponding to the simple path expression.

The fabric can also be used to accelerate *general path expressions*, which are key to dealing with data that has irregular or changing structure. General path expressions allow for alternates, optional tags and wildcards. For example, the general path expression `A.(`B_1`|B2)` searches for <A> tags that have <B_1> or <B_2> nested within them. We can expand the query into multiple simple path expressions, and evaluate each expression individually using separate *key lookup* operators. This means multiple traversals but each traversal is a simple, efficient lookup. Thus, the path expressions `A.(`B_1`|`B_2`).C` results in searches for `A.`B_1`.C` and `A.`B_2`.C`.

If the query expands to an infinite set, then it is harder to answer efficiently. This happens when the query includes wildcards. For example, `A.(%)*.C` means find every <C> that has an ancestor <A>. To answer this query, we start by using the *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 depth 100; see Section 3.2.)

We can further prune the traversal using another structure that summarizes the XML structure. Fernandez and Suciu [15] describe techniques for utilizing partial knowledge of a graph structure to prune or rewrite general path expressions. For example, with the query `A.(%)*.C`, we can use knowledge of the graph

structure to prune traversals of paths prefixed with `A.B` if we know that no `<B>` tag has a `<C>` descendent. Such techniques can be used to optimize searches with raw paths, while still utilizing the high performance lookups offered by indexing the raw paths in the Index Fabric.

3.5.2. Answering queries using refined paths

Queries that correspond to refined paths can be further optimized. First, the query processor identifies the query as corresponding to an existing refined path. The query is translated into a search key in the same way that the XML was encoded. Thus, a query "Find all invoices where ABC Corp bought from Acme Inc" becomes "**Z** ABC Corp Acme Inc." We then traverse the trie using this search key to find the relevant XML. Because we are using the horizontal layers of the trie, the search 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 a series of experiments to measure the performance of our indexing mechanism. To do this, we have stored an XML-encoded data set in a popular commercial relational database system[2], 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. Thus, our performance results represent an "apples to apples" comparison using the same commercial storage manager.

4.1. Experimental setup

The data set used in our experiments was the DBLP, the popular computer science bibliography [11]. The DBLP is a set of XML-like documents, where each document corresponds to a 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 7. Although the data tends to be somewhat regular (e.g. every publication has a title) the structure varies from document to document, as 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, called 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* generated for the root node of each document, and a *label*, corresponding to the root tag of the document. The *edges* table contained a tuple for every nesting relationship. For nested tags, *parentid* is the ID

[2] Our license agreement for the DBMS software prohibits publishing the name of the software along with performance numbers. We will refer to it as "the RDBMS." Our Index Fabric implementation 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 7. Sample DBLP document.

of the parent node, *childid* is the ID of the child node, and *label* is the tag. For leaves (data elements nested within tags), *parentid* is the ID of the parent node, *childid* is *NULL*, and *label* is the text of the data element. For example, the XML fragment

```
<book><author>Jane Smith</author></book>
```

is represented by the tuple (0,book) in *roots* and the tuples (0,1,author) and (1,*NULL*,Jane Smith) in *edges*. We experimented with an edge mapping where data elements were broken out from the *edges* table into a third table, *leaves(parentid,data)*; however, keeping the leaves as part of the *edges* table offered better performance[3]. We created the following B-tree indexes, using key compression:

- One index on *roots(id)*, and one index on *roots(label)*.
- One index on *edges(parentid),* one index on *edges(childid)*, and one index on *edges(label)*.

The second method of indexing XML using 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*. The STORED mapping represents using knowledge of the document structure to map the XML to relational tables, as opposed to the "naïve" basic edge mapping. STORED uses a data mining algorithm to extract schemas from the data based on frequently occurring structures, and these extracted schemas are used to create "storage-mapped tables" (SM tables). Data that corresponds to these extracted schemas can be mapped directly into tuples and stored in the SM tables, while more irregularly structured data must be stored in *overflow buckets*, which preserve the tree structure of the XML. The schema for the SM tables was obtained from the STORED investigators [13], while the overflow buckets used the same tables (*roots* and *edges*) as the basic edge-mapping. The SM tables identified for the DBLP data are *inproceedings*, for conference papers, and *articles*, for journal papers.

Note that these SM tables only store part of the conference and journal paper information. For example, the SM tables have room for three authors, but there is not enough "support" for the data mining algorithm to create columns for the fourth author, the fifth author, and so on (if they exist). Conference and journal paper information that does not fit into the SM tables are stored in overflow buckets along with other types of

[3] This conclusion is also noted by Florescu and Kossmann [16]. We chose the edge mapping in preference to the *attribute mapping* described in [16] first because the edge mapping is better for general path expressions (e.g. with wildcards) and second because it matches the format of STORED overflow buckets.

publications that do not have high support, such as books. To evaluate a query over the STORED mapping, the query processor may therefore have to examine the SM tables, the overflow buckets, or both. We created the following B-tree indexes, using key compression:

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

For both the edge mapping and STORED mapping it was necessary to hand tune the query plans generated by the query processor, since the plans that were automatically generated tended to use hash joins. By forcing the system to use index joins instead, we were able to significantly improve the performance of the RDBMS (e.g. reducing time to execute thousands of queries from 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 and retrieved for processing when necessary. All of the index schemes (basic edge mapping, STORED mapping, raw paths and refined paths) 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. We are primarily concerned here with comparing the performance of the indexes.

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 over the DBLP data. The queries are summarized in Table 1. In our experiments, 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. We describe each query in more detail, along with the performance results, in the next sections.

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

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.

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 Figure 8, which represent looking for 48 different publishers. In this figure, the block reads for index blocks and for data blocks (to retrieve document IDs) are broken out; the data reads for the Index Fabric also includes the result verification step required by the Patricia trie. As Figure 8 shows, 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/O's and 79 percent less time. Note that the overflow buckets require less time and I/O's than the edge mapping because the overflow buckets do not contain information stored in the SM tables, while the edge mapping contains all of the DBLP information and requires larger indexes.

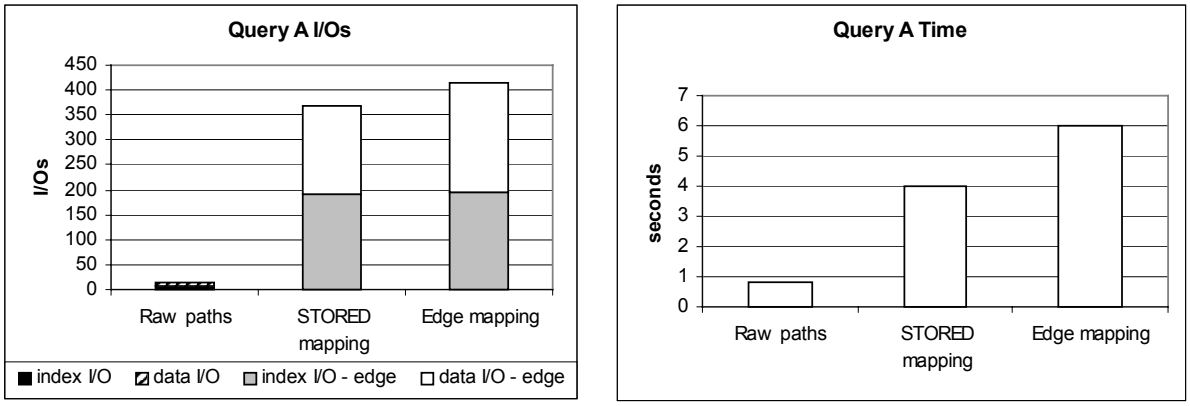

Figure 8. Performance for query A: Find book by publisher.

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. However, 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 as that used for the overflow buckets. The query is answered with one lookup in the raw paths (for `inproceedings.author.`*X*) and we did not create a refined path.

The results, for queries with 7,000 different author names, are shown in Figure 9. For the STORED mapping, this and all following figures separate I/O's to the edge-mapped overflow buckets and the SM tables. The figure shows that that raw paths are much more efficient, with 90 percent fewer I/O's and 92 percent less time than the edge mapping, and 74 percent fewer I/O's and 72 percent less time than the STORED mapping.

The queries over the STORED mapping are able to efficiently access the SM table (via a B-trees on the *author* attributes), achieving roughly equal performance with the raw paths. However, we must also access the overflow buckets to fully answer the query. In fact, more time is spent searching the overflow buckets than the SM tables, because of the need to perform multiple joins (and thus index lookups). 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.

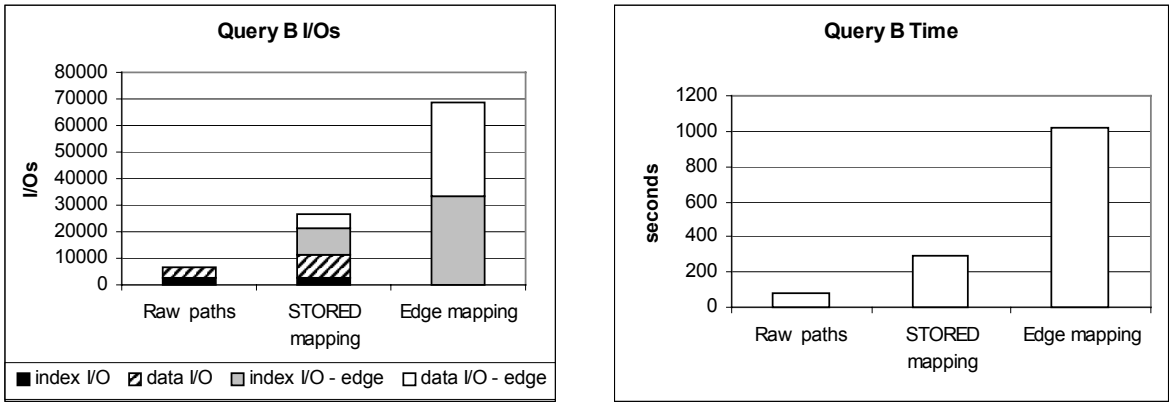

Figure 9. Performance for query B: find conference paper by author.

4.4. Queries C, D and E

Queries C, D and E are more complex than the previous queries, in that they require examining more document classes. Moreover, queries D and E are "branchy," since they examine sibling relationships. In each case, the performance of the relational mappings suffers due to the need to examine irregularly structured data.

Query C (find all document IDs of publications by author *X*) contains a wildcard, since it searches for the path "`(%)*.author.`*X*." Candidate documents contain an `<author>` tag. We must execute one query for each document type (`<inproceedings>`, `<article>`, `<book>`, etc.) for both the raw paths and the

STORED mapping, and then compute the union of the returned document IDs. The added difficulty suggests that we can further optimize this query using a refined path of the form "**Z** AuthorName". To create the refined path, we scan the documents looking for `<author>` tags, and extract the author's name. The query can be answered using a single refined path lookup despite the variability introduced by the wildcard.

The results are shown in Figure 10, and represent queries for 10,000 different author names. The raw paths are significantly faster than either relational mapping, requiring 51 percent fewer I/O's than the edge mapping, and 44 percent fewer I/O's than the STORED mapping. The refined paths offer even better performance: 71 percent fewer I/O's than the edge mapping, and 66 percent fewer I/O's than the STORED mapping. Figure 10 shows that most of the I/O's for querying the STORED mapping were in the SM tables, not the overflow buckets. This is partially because the query processor must access multiple SM tables. Moreover, the wildcard allows us to optimize the query over the overflow buckets (and edge mapping) somewhat; we no longer need to examine the *roots* table to verify the kind of document, avoiding a lookup in the *label* B-tree on the *roots* table. This optimization is only possible if we know that all documents are publications. If the database contained other documents besides publication records, e.g. invoices, or we simply did not know that all documents were publications, then we would have to check the document type. Even with this optimization, our techniques offer better performance than the STORED mapping and edge mapping.

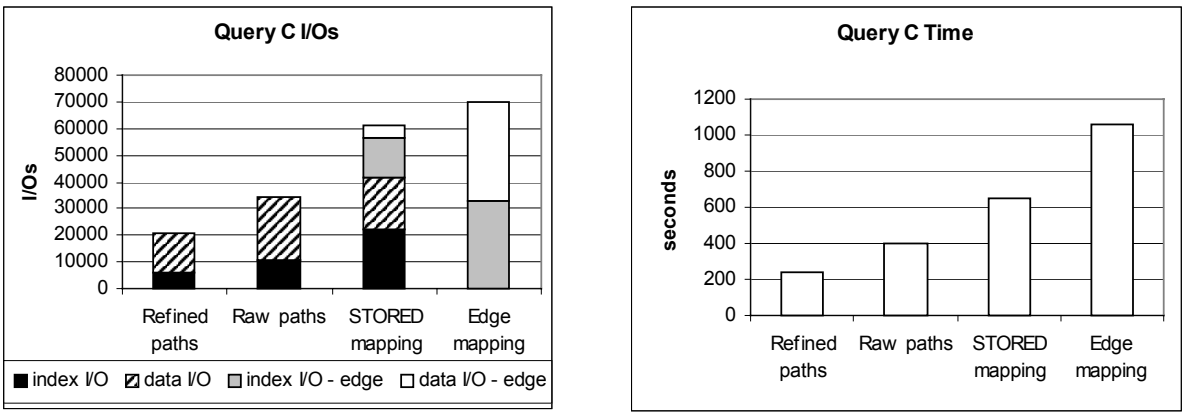

Figure 10. Performance for query C: Find all publications by authors.

Query D seeks IDs of publications co-authored by author "*X*" and author "*Y*." This query can be answered in the edge mapping by searching for "*X*" and "*Y*" as data items in the *edges* table, and then joining again with the edge table to see if "*X*" and "*Y*" are nested in `<author>` tags with a common parent. The STORED mapping can find some results by self-joining the *inproceedings* and *articles* SM tables, but must also join those tables with the overflow buckets tables if there are more than three authors. The overflow buckets must be searched for other kinds of publications (e.g. books).

Query D can be answered by the raw paths by traversing "*P*.author.*X*" and "*P*.author.*Y*" where *P*={book, inproceedings, article…} and taking the intersection of the result sets. However, a better query plan is to perform the lookup for "*P*.author.*X*," retrieve the set of document IDs, and use these to prune our lookups on

"*P*.author.*Y*." Specifically, for each document ID found for "*P*.author.*X*," we perform a lookup in the raw path index to see if "*P*.author.*Y*" also refers to that document ID. (In our experiments, the pruning plan requires 30 percent fewer I/O's than the simple plan of looking up both authors and taking the intersection.) We also further optimized this query with a refined path: "**Z** author1 author2", where author1 and author2 are co-authors and author1 lexicographically precedes author2.

The results, for queries on 10,000 different pairs of authors, are shown in Figure 11. Again, the raw paths perform quite well, even though multiple lookups are required, with a 47 percent improvement in I/O's versus the STORED mapping, and a 74 percent improvement in I/O's versus the edge mapping. The refined paths were even more efficient, with seven times fewer I/O's than the raw paths, and an order of magnitude reduction in time and I/O's versus either relational mapping. This demonstrates that while raw paths are good even for "branchy" queries, refined paths can offer significant optimization for complex path traversals.

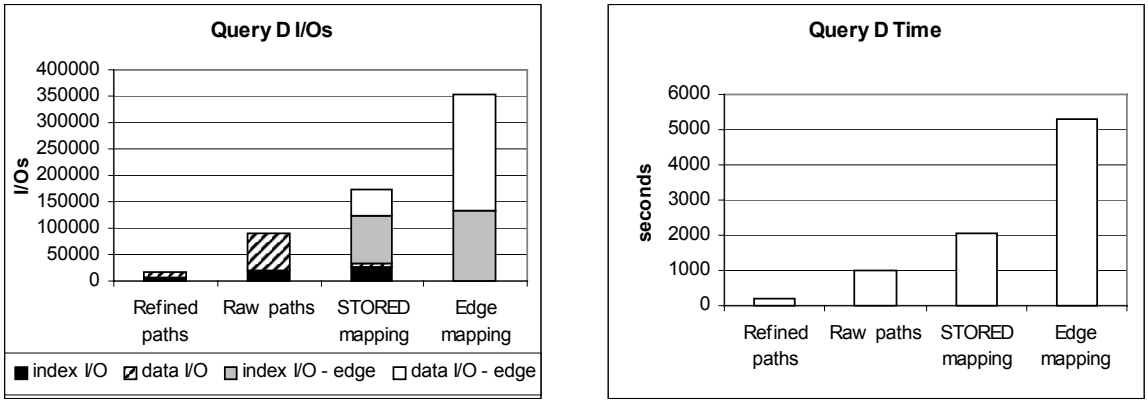

Figure 11. Performance for query D: Find all publications by co-authors.

Query E (find IDs of publications by author *X* in year *Y*) also seeks a sibling relationship, this time between <author> and <year>. The difference is that while <author> 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. We can evaluate query E like query D. For the raw paths, we perform a lookup on "*P*.author.*X*" (where P is the publication type), and use the resulting document IDs to prune the search for "*P*.year.*Y*." (We use the author results to prune years because of the high selectivity of the author name.) This query plan offers a 68 percent improvement in I/O's over the simple plan of looking up all documents for "*P*.year.*Y*" and intersecting with the results of the "*P*.author.*X*" query.

The results for 10,000 author/year pairs is shown in Figure 12. Queries over the raw paths required 53 percent fewer I/O's than the edge mapping. However, raw path queries were slower than the STORED mapping queries (with 10 percent more I/O's.) We believe this is due to the fact that the results of the year query were not clustered, and required multiple I/O's to retrieve the result set. We currently are examining how to better cluster the data to improve performance. Refined paths provided more than an order of magnitude improvement in time and I/O's over both relational mappings.

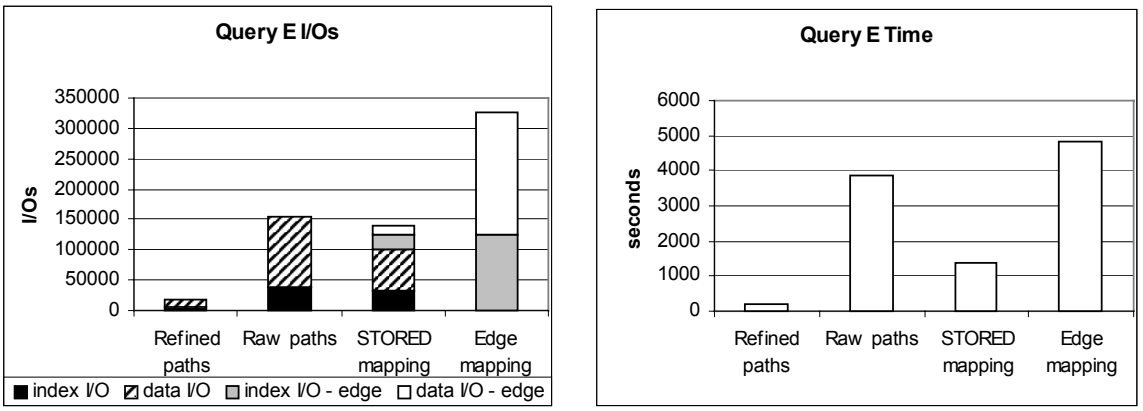

Figure 12. Performance for query E: find publications by author and year.

4.5. Discussion

Our results indicate that a significant cost for a relational system managing semistructured data comes from handling irregular structure. In each of our experiments, a significant cost for the STORED mapping resulted from accessing the overflow buckets. While accessing the SM tables themselves can be done efficiently, those tables cannot fully answer the queries. This is because a large portion of the data, even within the journal article and conference paper classes, was irregularly structured and considered "overflow." In fact, approximately half of the DBLP data had to be stored in the overflow buckets (even though 99 percent of the documents were either journal articles or conference papers). The irregularities in the structure of the data make it difficult to achieve high performance. This effect is even more pronounced in the edge mapping, which treats all of the data as "irregular," and has much worse performance than the STORED mapping.

These results suggest that what is required is an efficient mechanism for handling irregularly structured data. Full path indexing that does not rely on an *a priori* schema provides this mechanism, if the paths can be looked up efficiently. Moreover, a large number of different paths must be indexed. Our mechanism, which provides these properties, is effective at achieving efficient queries over semistructured data.

5. Related work

The problem of storing, indexing and searching semistructured data has been gaining increasing attention [1][4][5][20]. As stated in Section 1, there are two main approaches. One approach is to convert the data into a structured form, (e.g. relational), and store it in a relational DBMS. In this approach, techniques differ based on how much knowledge of the data's schema is used by the system. Shanmugasundaram et al [23] have investigated using DTD's (essentially, XML schemas) to map the XML data into relational tables. The STORED system assumes no prior knowledge of the schema, but extracts the schema from the data itself using data mining [12]. Both of these investigators have noted that it is difficult to deal with data that has irregular or variable structure, requiring either extensions to the relational model or overflow buckets. The Index Fabric,

which indexes all paths using the same simple metaphor of designator-encoded strings, does not have such difficulties. In contrast, Florescu and Kossmann have examined storing XML in an RDBMS, utilizing little or no knowledge of the document structure, as a set of attributes and edges [16]. While they note that relational databases can quickly evaluate queries over XML mapped to relational tables in this way, our results show that much better performance can be achieved using the Index Fabric.

Another approach is to store the data "natively" using a semistructured data model [21][25][26]. Although indexes can be used to support path expressions, evaluating such expressions requires multiple index lookups [20]. Even if a path index such as a DataGuide [17] is used, the database system has difficulty matching the efficiency of the Index Fabric, which is tuned for long, complex keys. Moreover, a specialized repository does not take advantage of the maturity of commercial relational systems. Our techniques can be used with either a relational system or a specialized repository to provide high performance.

A join index, such as that proposed by Valduriez [24], precomputes partial query answers so that at query time, specific joins are very efficient. This idea is similar in spirit to our raw and refined paths, although our index supports longer paths than individual pairwise joins between elements. Moreover, our layered index structure allows us to index many different (possibly long and complex) paths without causing degraded efficiency or resource contention. A join index must be dedicated to a particular task, is sensitive to key length, and competes for resources with other indexes in the system.

Path navigation has been studied before in the context of object oriented (OO) databases. OO indexing techniques, while similar to our mechanism, differ in several important respects. First, Bertino [3] suggests breaking up long paths into shorter components, and creating separate indexes on each, to minimize update costs. In our work, updates are inexpensive because of the efficient structure of the Index Fabric, and paths can be as long and complex as necessary. OO databases use sequences [3][19] or hierarchies of path indexes [29] to support longer paths, requiring multiple index lookups per path. Our mechanism supports following paths with a single index lookup. Second, OO indexes support linear paths (simple path expressions), requiring the combination of multiple indexes to evaluate "branchy" queries. Our structure provides a single index, even for branchy queries, and one lookup to evaluate the query using a refined path. Although multiple lookups of raw paths may be necessary for "branchy" queries, they are done in the same index, removing the need for multiple indexes. Third, Chawathe et al. [6] have reported that multiple indexes in OO databases can experience "index interaction," in which the utility of indexes is reduced even as they consume precious system resources. By providing a single, uniform index, we avoid index interaction and resource competition. Fourth, the nature of semistructured data requires the capability of supporting generalized path expressions in addition to the simple paths supported by OO indexes. 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 Fernandez and Suciu [15]) to provide better optimization of generalized path expressions over raw paths. Finally, OO indexes must deal with class inheritance [7], while XML indexes do not.

The Index Fabric structure itself aims to preserve the balance properties of the 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. Other researchers have examined taking unbalanced general graph structures and providing balanced, disk based access (e.g. [14]). Our structure is optimized specifically for disk based access of Patricia tries. Moreover, the horizontal layers in the fabric allow us to skip over large portions of the trie to focus on the localized component that can answer our query.

6. Conclusions

To cope with the irregular structure and complex relationships of semistructured data, query languages must use path expressions, which support navigating relationships between elements and dealing with structural variability. To achieve high performance, a semistructured database system must have an indexing mechanism that supports quick traversals over long and complex paths. We have investigated encoding these paths as simple strings, and performing lookups of these strings to answer queries. Such a technique requires an index that has very high performance for string lookups, even if there are a large number of strings, and the strings are non-uniform, long, and complex. The Index Fabric is a layered index that offers the features we require. 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 indexing scheme offers 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. Using these paths, we can accelerate any ad hoc query. We have presented experimental results that 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 alternative to traditional approaches to semistructured data indexing. Moreover, because our index can be used with existing data management systems, we can leverage the advantages of a mature DBMS, such as fault tolerance or concurrency, to effectively 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 documents, and for the assistance and advice provided by Dr. Deutsch in the use of those results in our experiments. We would also like to thank Donald Kossmann for his helpful comments on a draft of this paper.

References

[1] Serge Abiteboul. Querying semi-structured data. In *Proceedings ICDT*, 1997.

[2] Serge Abiteboul et al. The Lorel query language for semistructured data. *Int. J. on Digital Libraries* 1(1): 68-88, 1997.

[3] Elisa Bertino. Index configuration in object-oriented databases. *VLDB Journal* 3(3): 355-399, 1994.

[4] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings SIGMOD*, pages 505-516, 1996.

[5] Don Chamberlain, Jonathan Robie and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proceedings WebDB Workshop*, 2000.

[6] Sudarshan S. Chawathe, Ming-Syan Chen and Philip S. Yu. On index selection schemes for nested object hierarchies. In *Proceedings 20th VLDB*, pages 331-341, 1994.

[7] Sunil Choenni, Elisa Bertino, Henk Blanken and Thiel Chang. On the selection of optimal index configuration in OO databases. In *Proc. ICDE*, pages 526-537, 1994.

[8] Vassilis Christophides, Sophie Cluet and Guido Moerkottke. Evaluating queries with generalized path expressions. In *Proceedings SIGMOD*, pages 413-422, 1996.

[9] D. Comer. The ubiquitous B-tree. *Computing Surveys* 11(2): 121-137, 1979.

[10] Brian Cooper and Moshe 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. See 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, Sanjeeva Rane, S. Seshadri and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings 22nd VLDB*, pages 342-353, 1996.

[15] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. ICDE*, 1998.

[16] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. INRIA Technical Report 3684, March 1999.

[17] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings 23rd VLDB*, pages 436-445, 1997.

[18] Donald Knuth. *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition.* Addison Wesley, Reading, MA, 1998.

[19] W. C. Lee and D. L. Lee. Path Dictionary: A New Approach to Query Processing in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3): 371-388, May/June 1998.

[20] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proceedings 25th VLDB*, 1999.

[21] Jason McHugh et al. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3): 54-66, 1997.

[22] Oracle Corp. *Oracle 9i database*. http://www.oracle.com/ip/deploy/database/9i/index.html.

[23] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings 25th VLDB*, 1999.

[24] Patrick Valduriez. Join Indices. *TODS* 12(2): 218-246, 1987.

[25] Software AG. *Tamino XML database*. http://www.softwareag.com/tamino/.

[26] XYZFind. *XML Database.* http://www.xyzfind.com.

[27] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, October 6, 2000. See http://www.w3.org/TR/2000/REC-xml-20001006.

[28] World Wide Web Consortium. *XSL Transformations (XSLT) 1.0*. W3C Recommendation, November 16, 1999. See http://www.w3.org/TR/1999/REC-xslt-19991116.

[29] Zhaohui Xie and Jiawei Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings 20th VLDB*, pages 522-533, 1994.