

Indexing Open Schemas

Neal Sample^{1,2}, Moshe Shadmon²

¹Department of Computer Science,
Stanford University, Stanford, CA 94305, USA

²RightOrder Incorporated
404 Saratoga Ave. Santa Clara, CA 95050
nsample@stanford.edu, moshesh@rightorder.com

Abstract. Significant work has been done towards achieving the goal of placing semistructured data on an equal footing with relational data. While much attention has been paid to performance issues, far less work has been done to address one of the fundamental issues of semistructured data: schema evolution. Semistructured indexing and storage solutions tend to end where schema evolution begins. In practice, a real promise of semistructured data management will be realized where schemas evolve and change. In contrast to fixed schemas, we refer to schemas that grow and change as *open schemas*. This paper addresses the central complications associated with indexing open and evolving schemas: we specify the features and functionality that should be supported in order to handle evolving semistructured data. Specific contributions include a map of the steps for handling open schemas and an index for open schemas.

1 Introduction

Storing and managing semistructured data has received attention in both industrial and academic circles. Significant work has been done to achieve the goal of placing XML data (among other semistructured formats) on an equal footing with relational data. Some research projects and commercial products have the explicit objective of bringing the expanding body of semistructured data inside relational databases to achieve the goal [1,2,3], while others have attempted to build high-performance “native” semistructured databases and indexes [4,5,7].

While much attention has been paid to performance issues, far less work has been done to address one of the fundamental issues of semistructured data: *schema evolution*. In relational databases, the schema represents the logical structure of data elements. It is typically an expensive operation to alter schemas, and may have consequences that bubble up from the relational store to the application layer. Semistructured indexing and storage solutions tend to end where schema evolution begins. Data stores and indexes are built assuming that the data may enter and leave the system as some form of XML, but that the XML’s schema will be somewhat static, most likely corresponding to a fairly rigid DTD. This is an overly limiting assumption.

In practice, the real promise of semistructured data management will be realized when schemas evolve and change. In contrast to fixed schemas, we refer to schemas that grow and change as *open schemas*. For example, in the most well defined industries there are still competing standards and representations for information. Even when there are standard record formats, records can be incomplete due to missing or incorrectly entered information. However, these semi-rigid and incomplete data sets are actually the easiest available sources to integrate. In practice, data records and content objects vary widely among entities in both scope and format. This type of data provides a challenge for traditional databases to store and search, and the complications grow as actors enter and leave. While DTDs and DDML/XSchema can accurately *describe* evolving semistructured data sets, they say nothing about effectively storing and indexing such data [14].

This paper addresses the central complications associated with indexing open and evolving schemas. Our objective is twofold, both theoretical and practical. We specify the features and functionality that should be supported in order to handle evolving semistructured data. Some features are well understood because they are required with or without an open schema, so we will just give a cursory examination of those details. Our specific contributions:

1. Map out steps for handling open schemas – we cover the fundamental requirements for managing data with an evolving schema. These steps are generally applicable and may be interpreted broadly for any semistructured data format.
2. An index for open schemas – we illustrate the Index Fabric, an index well suited for evolving schemas. The Index Fabric is compact, high-performance, and balanced over semistructured and irregular data sources, making it ideal for schemas that change over time.

This rest of this paper consists of three main sections. First, we explain the steps for indexing an open schema, from key generation to query support. Second, we present the Index Fabric as a substrate for supporting indexes over data with an open schema. Finally, we summarize our contributions and close with a few remarks about open issues.

2 Managing Open Schemas

In this section we cover the steps required to index data with an open schema. These steps cover each level of the process, from generating keys to changing data to updating indexes after source data changes.

2.1 Key encoding

Interesting data items are located by searching an index for matching keys. Keys consist of attribute names and data values for corresponding attributes. In a relational database, attributes are the same as columns in a table. With semistructured data, there is no assumption that there is a regular or complete mapping to a relational table. If there is such an obvious mapping, either to a relational table or to a fully

specified and static DTD, then indexing is not difficult. However, an evolving schema places new requirements on the indexing methodology. In order to build the index, there must be an association between attribute names (e.g., XML tags) and data values (e.g., XML PCDATA). We propose to map attribute names to *designators*, and to keep information about designators in a dictionary. This designator dictionary maps designators (used in an index) to and from arbitrary attribute names [5].

Keys are an encoding of designators and data values derived from objects in the database.

2.1.1 Designator creation

Any semistructured indexing strategy must support dynamic creation of new designators. As new attribute names (e.g., previously unseen XML tags) enter the system they should be integrated into the existing framework with a minimum of effort. This means that an in-flight index with an existing set of designators should support the definition of new designators, and the insertion of keys tagged with those designators should not require a complete index rebuild. Creating new designators is the first requirement when supporting evolving schemas.

2.1.2 Designator dictionary

There must be an explicit mechanism to map between the self-describing labels (e.g., tags) in the data objects and designators representing those labels. This mechanism, the *designator dictionary*, must support the following functions:

- *Semantic marking of designators*: when the system encounters two objects from different domains and has knowledge of the distinction, it should create distinct designators for syntactically similar but semantically different attributes (e.g. as attribute names (label, tag, etc.) versus designators, as from one domain versus another). For example, “<record>” in one domain may refer to a collection of data, while “<record>” in another domain refers to an album of music.
- *Lookup of an existing attribute name*: to find the corresponding designator(s) and associated semantic markings. If the attribute name does not exist, an error condition should be returned.
- *Creation of new designators*: for new attribute names. This function should support marking the new designators with semantic information.
- *Lookup of an existing designator*: to find the corresponding attribute name and associated semantic markings.
- *Pattern matching over attribute names*: ideally, the dictionary would support regular expression lookup to find attribute names.

The designator dictionary is the bridge between the external view of data and any internal representations used for indexing/storage/etc. It can also play a crucial role in relating semantically equivalent concepts and searching over data types that do not adhere to identical schemas. For instance, augmenting the designator dictionary with semantic information can assist when searching for attributes that do not have a common name, but do have a common meaning. The designator dictionary overcomes one of the limitations of existing databases, namely the fixed name space. With the

designator dictionary identical and different names are mapped to designators simplifying schema evolution and avoiding name conflicts when data is integrated.

2.1.3 Encoding keys

Data elements in a semistructured data document may be of any length, and attribute names are fundamentally unlimited as well. Ideally, a key of any length should be supportable. With keys over semistructured data, it is needed that the keys should allow search and still reflect hierarchy of any depth. Obviously, the index implementation of these long keys that embed relationships should not impact performance. Moreover, keys may include multiple designators with no corresponding data. For example, the path “*invoice.buyer.name*.‘Cisco’” could be represented as “**I.B.N**.Cisco” where “**I**,” “**B**,” and “**N**” are designators. At the same time a key such as “**I.325.B.N**.Cisco” allows search for the buyer name of invoice number “325.” These are examples of raw and refined paths explained in more detail in [5].

2.2 Data interface

There should be an abstract “pointer” type, which could be a unique document identifier, a pair (docid, offset), a physical row identifier, or any type of pointer appropriate for a particular data storage manager. If the index is designed generically to handle arbitrary pointer types, then any data storage mechanism can be used. Separating the indexing mechanism from the storage mechanism is important for generality, at the cost of potential performance. (However, performance results presented in [5] indicate that external indexing can be quite effective compared to less-flexible native indexes.)

This requirement implies a modular separation between any index and the data manager. In other words, different data storage managers can be used without modifying the index. This means that schemas may evolve, and that even new storage managers may be introduced without altering existing stores. The index itself is agnostic to storage and treats keys generated from different data stores uniformly. This level of indirection is critical for complete flexibility, but would likely be somewhat limited in installations where performance is favored at the cost of fixing the underlying storage manager. The features supported by the data layer should be:

- Insert document; returns a document ID (pointer)
- Accept a pointer from the index and return the corresponding document or fragment

2.3 Inserts, updates, and deletions

Indexing new document types can be done automatically, if there is an appropriate set of indexing rules in place. When a new document enters the system, new keys are built from the document, and those keys are inserted in the index. A simple, generic strategy would be to generate keys for all root-to-leaf paths within a semistructured document, generating new designators for previously unseen tags. Arbitrarily

complicated indexing strategies are possible, though not be completely generic and restricted specific to particular document types. While the generation of schema specific rules tends to run counter to the notion of complete generality, with semistructured data it is understood that document specific optimizations are allowed. Below, we outline inserting, deleting, and updating documents, and how that would be handled assuming a simple index of root-to-leaf paths.

2.3.1 Parser

Documents must be parsed to create a set of keys for index insertion. The first step is to use the designator dictionary to map attribute names to designators (and create new designators as necessary). The second step is to extract the root-to-leaf paths in the document, then encoding those paths as designated keys. For optimizations, there should also be an interface for specifying alternate, document-specific paths, so that the parser can extract them from documents. The interface to specify alternatives keys to generate and index could be programmatic (e.g., Java applets) or declarative (e.g., XSLT [6]).

2.3.2 Insertion

Once a document is parsed, the set of keys for that document must be inserted into the index. When the document itself is inserted into the data layer, an ID is return, as are any more specific pointers to pieces of the document. The keys are put into an index and refer to the document using the “pointer” type mentioned section 2.2.

2.3.3 Deletion

Of course, not just schemas evolve and change over time. Individual documents may grow and change over their lifetime. As such, there must be a way to remove individual keys from the index, not just complete documents. This implies that there must also be a mechanism to identify any key belonging to a specific document, so that it can be removed. Analyzing local deletions to the document can discover the keys to be removed from the index. Finally, there must be an interface for specifying that a complete document should be removed, which then identifies all associated keys and deletes them from the index. This can be done using the same parsing rules that generated the initial key set, or may exist as a materialized set stored offline, depending on performance requirements.

2.3.4 Updates

The concept of “update” is somewhat ill-defined for semistructured documents. The simplest approach is to remove the affected document, change it, and then re-insert (and re-index) the document. However, this brute force approach likely entails significant overhead if an update is relatively small compared to the overall document. If indexing rules are straightforward, the changed set of keys should be discoverable from updated portions of a complete document.

2.4 Query support

Querying an open schema can be problematic for certain applications that make assumptions about underlying data. For instance, if query expects to discover data with the form *A.B.C*, but the schema has evolved to include data with *A.B.C.D*, how should that be handled? An obvious solution is to push filtering logic into the client application such that it accepts any *A.B.C.**, but discards anything uninteresting. This type of filtering has been touted as an advantage of using semistructured data, but is not always so clearly desirable in practice. For performance reasons, it may be much better to return just the portion *A.B.C*. Furthermore, if clients are brittle and validate returned values against a restricted DTD, previously legitimate results may be rejected. This indicates that query support should include filter operations at the server and mechanisms that avoid retrieving irrelevant data. This may be standing filters for specific clients, XSLT, or some other mechanism.

2.4.1 Query language

An index should support any front-end semistructured query language, as even query languages are in a state of flux and evolution. This means there should be a well-defined internal language for query operations. Query compilers specific to front-end languages will translate queries into internal language queries. A key step will be the translation of attribute names (e.g. tags) to designators.

2.4.2 Query operators

There are basic operators (which provide simple index lookups) and more complicated operators (to provide joins, unions, etc.). We begin by defining a complete but simple set of operators (e.g., able to answer queries but not necessarily in the most efficient way) and build on that set as necessary.

Simple operators

- *Single key lookup*: given a designator encoded key, find the pointer associated with that key.
- *Prefix key lookup*: given a prefix of a designator-encoded key, find the pointers associated with all keys that have that prefix.
- *Data retrieval*: given a pointer, retrieve the data (e.g., document or fragment) associated with that pointer.

Complex operators

- *Set union*: given two sets of pointers, return the set union.
- *Set intersection*: given two sets of pointers, return the set intersection.
- *Set difference*: given two sets of pointers, return the set difference.
- *Projection*: given a document, extract the desired information. (As mentioned above, this can use a standard tool, like XSLT.)
- *Join*: given two sets of pointers, join the associated data according to a specific condition.

- *Select*: given a set of pointers, filter out those that do not meet a specific condition.
- *Discover structure*: retrieve unknown structure subordinated to a given path.

2.4.3 Query optimization

There are many different ways to use an index to answer even simple queries. For example, a search for a document with a buyer “X” (e.g. <buyer>X</buyer>) and seller “Y” (e.g. <seller>Y</seller>) can be supported in several ways, including:

- Lookup documents with buyer “X.” Lookup documents with seller “Y.” Take the intersection.
- Lookup documents with buyer “X.” Perform a select for documents with seller “Y.”
- Lookup documents with seller “Y.” Perform a select for documents with buyer “X.”

A query optimizer can select an efficient plan for evaluating queries. An optimizer is especially important for searches over root-to-leaf paths, since naïve plans for complex queries (e.g., queries with wildcards) can be very bad. The optimizer may take into account statistics about the data. Statistics about attribute name distribution/structure can be especially important for optimizing queries over root-to-leaf paths.

However, query optimization is a complex problem and still an active research problem in the area of semistructured data. As a result, it is desirable to begin with a query planner that produces good query plans given simple indexing. While investments in query planning can occur in the engine, the indexing strategy outlined so far makes it easy to directly leverage information about expected queries. For instance, the query for “documents with a buyer ‘X’ and seller ‘Y’” may be easily handled with a single key lookup if the parser knows to generate *buyer.seller* composite keys whenever a document has both components.

3 Indexing Open Schemas

Proper indexing is an important component of any database; indexes may access particular elements quick. There are many trade-offs to consider when choosing which indexes to build. With relational data sources, indexing decisions are made based on expected queries, index update costs, space requirements, and expected benefit, among many other criteria.

When dealing with open schemas, making the right decisions about indexing is more difficult. In this section, we present the Index Fabric, a novel indexing structure suited to both semistructured and hierarchical data. The Index Fabric is also uniquely suited to open schemas. First, the index does not depend on a schema or DTD, and second because it explicitly maintains all element relationships and does so at a very low cost. This means that queries with path-expressions, a crucial component of semistructured queries, are supported in the Index Fabric.

3.1 Piecewise indexing

There are two schools of thought in indexing semistructured data: *complete path* and *piecewise* indexing. A complete path index is an index that can search for a full document path in a single lookup. Examples of complete path indexes are the Index Fabric [5,7], or a meta-index of paths like Lore's DataGuide [10]. Piecewise indexing over semistructured data refers to techniques that break paths down into smaller pieces. Examples of piece-wise indexing include myriad flavors of edge maps [12], and include examples like STORED's overflow bins.

3.1.1 Breaking apart the document

Piecewise approaches are essentially generic. The main technique is to deconstruct complete documents into an edge map. An edge map is generally a set of tuples (e.g., <parent, child, type, label, docid>) derived from a document. An index, such as a B-tree or an inverted list, is built over the resulting set of tuples. Self-joins against the table of edge map tuples are performed (using the index for speed) to perform path lookups. Breaking apart the original document is the key to the generality of the edge map: there is no restriction placed on the document's structure. This means that the approach is uniquely suited to indexing open schemas.

The edge map graph may be used to exclusively for path query support, or may actually be used to reconstruct the original document (if the document is not stored in a complete form elsewhere in the database). Either approach has one clear drawback: poor performance. Long paths mean multiple self-joins on the edge map table. Also, wildcards and other more restricted pattern-matching operators imply a discovery process; a pure edge map may generate very large intermediate results without additional supporting information.

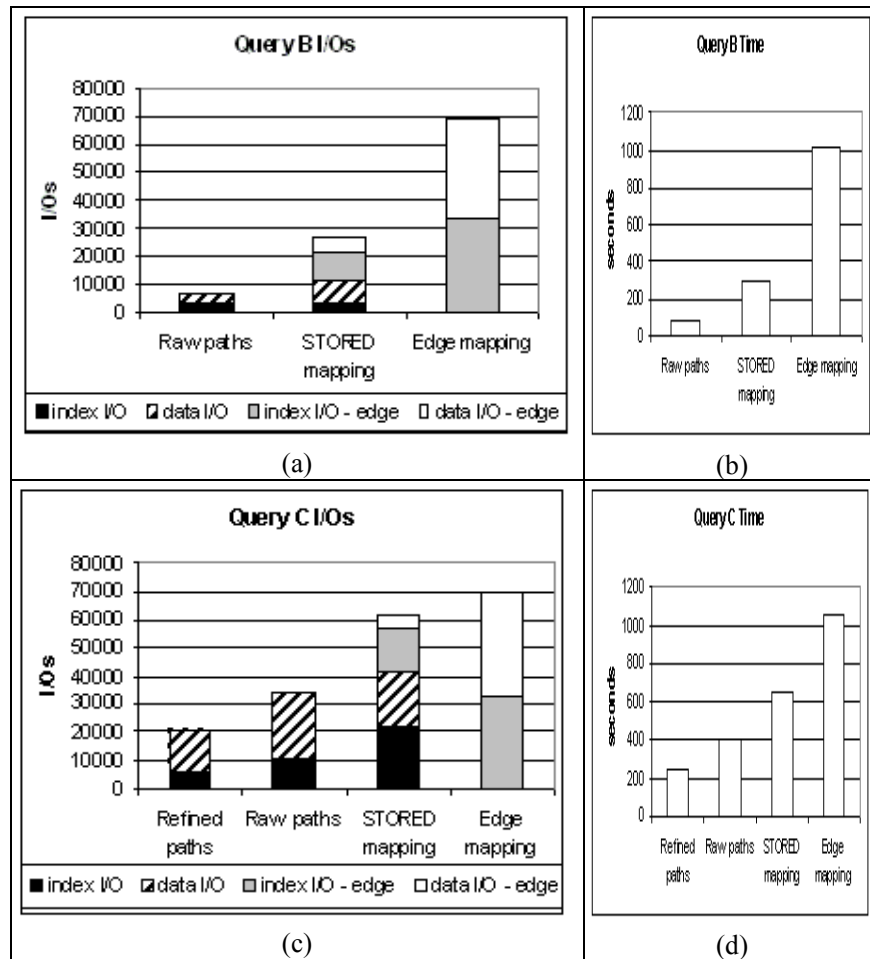
Edge map approaches are generally poor performers. This does not imply that they have no utility, however. A hybrid approach, using tables for common and repeated structures within documents combined with an edge map for the less common and "ragged" document edges has been shown to greatly outperform simple edge maps. STORED is one such hybrid approach [1]. The STORED system uses data mining techniques to extract a partial schema from a set of documents. Data that does not fit the schema well is stored and queried in its native form, via an edge map.

3.1.2 Static edge map performance

We have extensively compared both pure edge maps and the STORED system to the Index Fabric for queries over semistructured documents [5]. These comparisons were over a fixed data set with a schema that did not grow or change over time. This means that STORED was not disadvantaged during the experiments by an evolving schema. For both simple and complex queries, the Index Fabric outperformed the edge map and the STORED approach. Relative I/Os and query times can be seen in Table 1. In Table 1, (a) and (b) show the Index Fabric (raw paths) outperforming STORED and edge maps. Cells (c) and (d) show that even more improvements are possible when using query specific key strings ("refined paths"). What should be noted is that STORED's table scans and index I/Os were comparable to the Index Fabric, and that

it was not until STORED was forced to look into the edge map for paths not part of a main table that performance really suffered.

Table 1. Index Fabric performance results



The most promising result from this previous work was not that the Index Fabric was faster than STORED in an apples to apples query comparison; the most promising result was that the Index Fabric was as fast as the STORED approach over the fully relational portion of the data set. The implication is that the Index Fabric completed queries over truly semistructured sources in comparable time to some of the best relational techniques. A full explanation of the experiment is found in [5], we simply replicate some of the raw results here to underscore claims about the high performance of the Index Fabric. Performance over a static set of documents is a benefit, but recall that the real concern is performance under an evolving schema.

Even the somewhat improved performance of the STORED system comes at a real cost: flexibility. By mining a partial schema from a document set, the STORED system is optimized for the set of documents that it is trained on. The central issue of this paper, open and evolving schema, runs counter to the STORED process. Our assumption that the data changes over time invalidates the STORED mining process, and could quickly render mined schema irrelevant.

3.2 Complete indexing - Index Fabric

Section 3.1.2 puts forward the argument that the Index Fabric is a fast index for semistructured data, and that it was shown to be faster than one of the best recent hybrid relational approaches. We conclude section 3 with a discussion about why the Index Fabric is also uniquely suited for use with evolving schemas.

3.2.1 Complete tree indexing

At the heart of the Index Fabric is a Patricia trie string index [5, 7]. Other string-based indexes have been proposed using Patricias, including Sybase's Compact B-tree Index in the iAnywhere product [8] and the String B-tree [9]. These alternative indexes are built like B-trees with very long strings, but the B-tree blocks are individually compressed using Patricia structures. This is a subtle yet fundamental difference between them and the Index Fabric. Because the Index Fabric maintains a full Patricia over the entire key set, it can be used like Lore's DataGuide [10]. Vertical tree traversals from the root of the Index Fabric over the tree representing the data in the index are possible in an efficient manner. In the String B-tree or iAnywhere B-tree, such a traversal is not efficient in many cases. A fuller description of the Index Fabric can be found in [5,7,11].

This fully connected vertical Patricia is important for constraining search when the structure of the search results is not known at search time. For example, in hybrid edge map approach like STORED, finding matches over even a simple XPath query like `/catalog/cd/*` could be very expensive if the schema has changed. Imagine every document looked something like document in Figure 1 when the STORED schema was mined.

```
<catalog>
  <cd country="USA">
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <price>10.90</price>
  </cd>
  <cd country="UK">
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <price>9.90</price>
  </cd>
  <cd country="USA">
    <title>Greatest Hits</title>
    <artist>Dolly Parton</artist>
    <price>9.90</price>
  </cd>
</catalog>
```

Figure 1. Sample XML document

When the schema changes because even one of the `<cd>` tags adds a `<saleprice />`, the overflow edge map table kicks in for every query like `/catalog/cd/*`. The complete Patricia at the core of the Index Fabric prevents the need for an overflow edge map, and also explicitly indicates what tags can reside along any given path during a single lookup, without any joins. This property can be leveraged for inexpensive tests for existence of long paths as well. Looking for something like `/catalog/cd[saleprice<10.0]` can be very expensive when few `<cd>`s in the catalog have that property, but it cannot be determined until each join is performed and the overflow edge map is consulted for every `<cd>`. A visual example of this property can be seen in Figure 2(a) and (b).

In Figure 2(a), we see the original document structure represented in the Index Fabric's Patricia, before the addition of any `<saleprice>` tags. In Figure 2(b), we see the new document structure represented in the Index Fabric's Patricia, after the addition of some `<saleprice>` tag. With keys encoded in the Index Fabric, XPath queries for `/catalog/cd[saleprice<10.0]` are immediately directed along the appropriate path in Figure 2(b), as soon as such keys enter the index. With an edge map or hybrid approach, a significant portion of the document base needs to be checked before results are returned. (We should note that researchers have implemented better search techniques than edge maps in particular instances. Lore's DataGuide can be searched top-down, bottom-up, or in a hybrid fashion to reduce the expense of consulting the index [10]. However, such techniques are not typically available in the logic of commercial systems.)

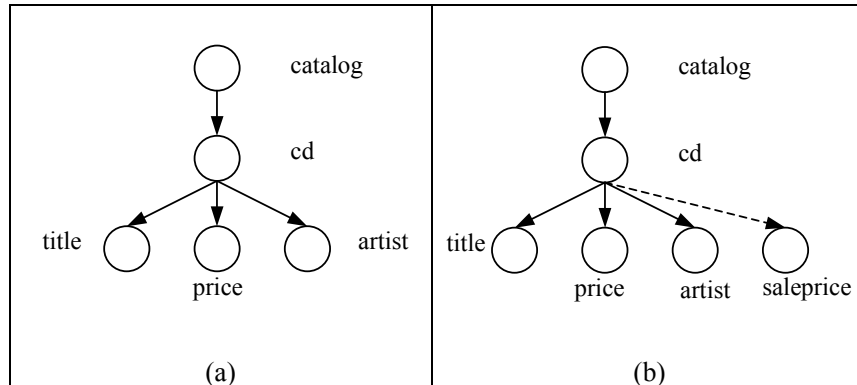


Figure 2. XML structural summary

3.2.2 Index Fabric properties

The index Fabric is a layered vertically in the same manner that a B-tree is layered vertically. While a non-trivial implementation in practice and beyond the scope of this paper, this horizontal balancing process makes it possible to access either “shallow” or “deep” paths with equal effort. There is a single I/O per horizontal layer in the Index Fabric just as there is in a B-tree tree. The Index Fabric is significantly smaller than a corresponding B-tree because the Patricia at the core is a lossy string index. The high-compression of the Patricia means that the overall size of the index is low, but also means that there is a high fan-out in upper layer blocks, ensuring that the index remains shallow. (In the Index Fabric, because of the lossy Patricia, there is a constant cost per key, regardless of key length. This means that the Index Fabric is not sensitive to the *size* of the keys, only the *number* of keys.) These properties combine to yield the performance advantages seen in Table 1 above, presented in [5]. More recently, we have demonstrated near linear speedups for the same queries when parallelizing and distributing portions of the index [11].

There is always the question of performance when indexing hierarchical data. With semistructured and hierarchical data, root-to-leaf paths tend to be long, raising difficulties for complete path keys using standard indexing. The use of designators in the index allows the unification of paths that share the same prefix, even if the indexed data is of different types and stored differently. For example, a query for particular attributes of a product in a catalog where the indexed data is *catalog.product.attribute.value* allows a query to follow the *catalog.product* path and navigate to the needed attributes according to the designators found at the end of the *catalog.product* path. If a desired attribute exists (or is missing) for the particular path, this information is readily available.

The Index Fabric is a self-describing index. This means that properties of the indexed elements are stored per element rather than in an external schema. This makes the description of the data always complete regardless of the diversity between the data elements and obviates the need to update a schema when new elements are introduced. The indexed data reflects the schema, allowing the schema to evolve

with the insertion of new types of keys. This feature, along with the fact that the index is able to treat all keys in a uniform manner, with the ability to create a balanced and small index, makes the Index Fabric perfectly suited to index data where schemas are not fixed.

3.2.3 Index Fabric, a complete indexing system

In order for the Index Fabric to leverage its performance advantages in the face of open schemas, we show that it meets the requirements of the indexing scheme set forth in section 2. First, the Index Fabric should be compatible with an open process for key encoding, including the creation of new designators for new attribute names entering the system. Second, the Index Fabric should support a generic data interface, not restricted to a single table, source, or set of documents. Third, update operations should be handled gracefully, without requiring significant downtime or index rebuilding as schemas change. Finally, the Index Fabric should support many different query patterns, including those mentioned in section 2.4.2.

Key encoding

Adding new keys to the Index Fabric is a straightforward process derived from the one outlined in section 2.1. The Index Fabric is a string index that encodes arbitrary bit-strings. These strings can be any length, and the index is content agnostic. Because there are no formatting or content restrictions placed on indexed string, new designators may be added at any point to represent new attribute names. The Index Fabric maps all keys to designated strings regardless of their storage data types. Thus, the data can be very different but paths assigned with the same designators are comparable as if the data was normalized to fit a single schema. In addition, the bit-strings are uniform allowing comparison of keys by their value rather than simply by type. This is in stark contrast to a fully structured environment, like a relational system, where the data needs to be normalized and the addition of new attributes means the creation of new indexes and often implies altering tables.

Data Interface

The Index Fabric manages long, arbitrary keys. In the index, there is some portion of the key materialized by the Patricia structure coupled with an arbitrary pointer object. This pointer may be of any size, and may eventually refer to any object. This works in an obvious way through indirection. B-trees in relational systems frequently have (*value, pointer*) pairs in the leaves, where a pointer is something specific to that implementation, like a “row ID.” After searching a B-tree for some values, a set of homogenous pointers is returned. The Index Fabric also returns a set of pointers, however the pointers themselves may be of multiple types. Recall that keys appended with pointers are inserted into the Index Fabric, without restriction on the pointer’s content or structure. The translation of arbitrary pointers to data happens outside the index.

Handling updates

Updates to the Index Fabric are based directly on output from a document parser. When new documents are added to a database, they pass through a parser that generates the initial set of keys. The initial set of keys is inserted into the index along with a pointer to the original document or appropriate fragment. An update to the index may be triggered by an update to a document, and is modeled as two operations: a delete followed by an insert. Deletions are not value-based; they require a key string and a pointer. This is standard requirement of any non-unique index.

Query support

Section 2.4 outlines a set of functional requirements to support a structured query language. The Index Fabric efficiently supports each of these operations. The simple operators are *single key lookup*, *prefix key lookup*, and *data retrieval*. As noted previously, single key lookup is done in one probe of the index, regardless of key complexity. Prefix key lookup is handled in a similar way. A single horizontal probe into the index locates the specified portion (the prefix) of the query in the Patricia. From this point, a vertical traversal collects all subordinate elements. Referring to Figure 2(b), a query for `/catalog/cd` would then return a set of four key types, `{catalog.cd.title, catalog.cd.artist, catalog.cd.price, catalog.cd.saleprice}`. The Index Fabric directly supports single and prefix key lookups directly.

Complex operations should be supported by the index as well. Some operations are the direct responsibility of the index (like set operations), while others are used in result post-processing (such as projection). Compared to other indexing alternatives, set operations can be somewhat expensive to perform in the Index Fabric because the Index Fabric's allowance for multiple pointer types. For example, set intersection in a B-tree is a simple process. It can be done by taking two sets of pointers, sorting each, then scanning the pointer sets in order to find the elements that exist in each set. This process is simplified in B-trees by the assumption that all pointers are of the same type (and size).

In the Index Fabric, pointers may be different types (and potentially sizes) to support federated indexes over multiple sources. One solution to the set intersection problem is to partition the sets of pointers by types, essentially sorting by "pointer type" first, and then sorting those partitions by value. This partitioned solution is a reasonable way to perform operations over sets of multiple pointer types.

Optimizing queries using the Index Fabric

It has already been shown that the Index Fabric is a fast and small index for semistructured data [5]. We further present that the Index Fabric is appropriate for specialty optimizations. In section 2.4.3, we presented the case of generating special key strings that aid in evaluating specific queries. The example presented was the query for "documents with a buyer 'X' and seller 'Y'." By generating new composite keys of the form *buyer.seller* whenever a document has both components, the search can be optimized. The Index Fabric is not limited to any particular key type or structure.

The real insight to adding this support comes from the designator dictionary. It may not be obvious as to why arbitrarily adding a key like *buyer.seller* is problematic. But what happens when a user enters an XPath query like */buyer/seller*? Does this path correspond to a real data fragment, or does it exist in the index simply as an optimized key? The designator dictionary makes it possible to distinguish between keys that represent actual document structure and those built for the purpose of optimized access. The optimized *buyer.seller* can be represented with *buyer'.seller'*, distinguished from an actual path, but appropriate for the optimized query.

3.3 Open schema performance

When the schemas change, the utility of the *a priori* analysis done with an approach like STORED is diminished. The STORED algorithm chooses materializations based on high support within the training data set, which can change radically with schema evolution. Where the schema is a moving target, we are left with edge map approaches to storage and indexing. The authors of [13] point out that conventional horizontal row representations fail when schemas constantly evolve. They advance the alternative approach of a vertical representation where data is stored as tuples consisting of object identifier and an attribute/value pair.

Figure 3 shows a horizontal table and its corresponding representation in the vertical format as explained in [13]. (The symbol \emptyset represents a null value.)

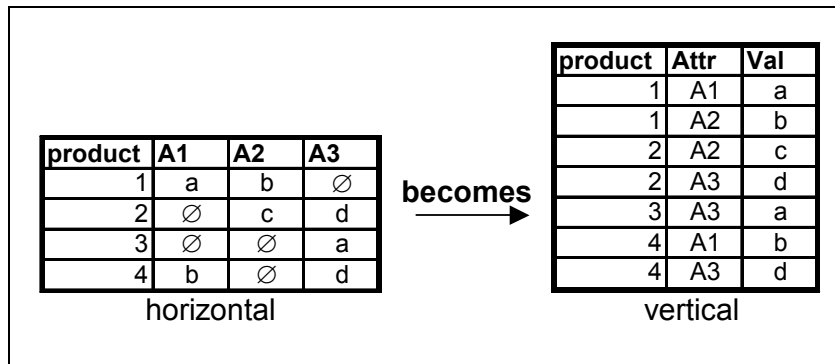


Figure 3: Horizontal and vertical table representations

With the vertical approach to data storage, the schema is completely open even in a relational store - one simply adds new tuples corresponding to new attributes. According to [13], the best approach to indexing of the vertical table is to build B-tree indexes over each of the three columns.

We compared query times of the vertical table approach for handling open schemas with the Index Fabric. Our approach to storage for the Index Fabric was similar to [13], with one alteration; the data was stored in 3 tables shown in

Figure 4: An attribute name table, an attribute value table, and a product ID table. The Index Fabric engine was embedded transparently within a commercial relational DBMS.

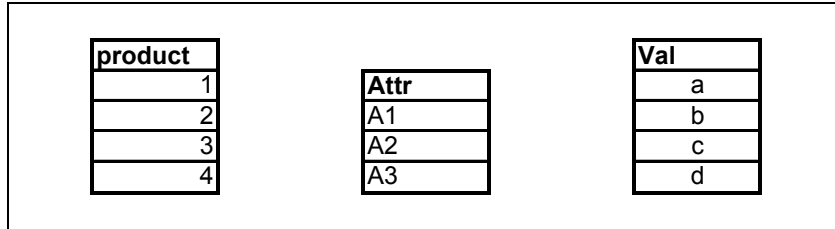


Figure 4: Index Fabric approach - the data tables

The keys in the Index Fabric are the concatenation of 3 fields: *Attr.Val.product* (these keys are shown in the “key string” table embedded in

Figure 5). This storage and indexing structure facilitates schema flexibility; one simply adds new attributes, new values and new products, while maintaining the relationships between the products and the attribute/value pairs through the index.

As noted in earlier publications, the leaf layer of the Index Fabric has a Patricia structure over the data keys [5, 7, 11]. The key strings are not materialized explicitly, rather the index points to the data from which the keys are constructed. Every path in the index has 3 pointers: (a) at the attribute name position in the trie there is a link to the attribute name column in the attribute name table, (b) at the attribute value position in the trie there is a link to the attribute value column in the attribute value table, and (c) at the product ID position in the trie there is a link to the product column in the product table.

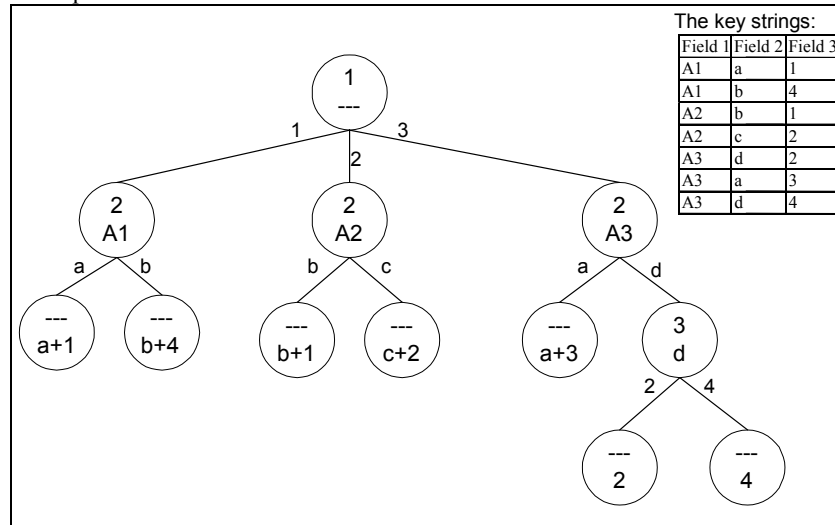


Figure 5: The Index Fabric structure

Figure 5 shows the Patricia structure over this sample set of keys (these keys come from the data table presented in Figure 3). The circles in

Figure 5 represent elements in the index. The links between the circles represent the labels of the Patricia links. Each circle contains two values. The top value stands for a node position. The bottom value stands for a link to a data element; rather than

show the pointers here, we show the key value of the data element. Two data links are represented by two key values. If the top value is a dashed line, the circled element represents a link to data (but is not a node). If the bottom value is a dashed line, the circled element represents a node without a link to the data. For example, the leftmost circles of the tree include the root node having the value 1 and no data link, a node with a value 2 and a link to a data record whose key value is “A1” (the first attribute name in the attribute name table). The third left-most circle represents two links to data elements: a link to a data element whose key is “a” (the first value in the attribute value table) and a link to the data element whose key is “1” (the first product in the product table).

Searching in the trie is simple: to search for the key *A2.b.1*, we follow the root node having the value “1”, the link labeled “2”, the node having the value “2”. We also follow a link to “A2” in the attribute name table, the link labeled “b” to the value “b” in the attribute value table, and the product “1” in the product table. A query to retrieve product IDs by attribute name “A3” and attribute value “d” would follow the path *A3.d.** to the sub-trie rooted at the node with the value “3”.

Our experiments run on a 700 MHz dual processor Intel Pentium III machine with 2GB physical memory. The operating system was RH Linux 6.2. The installed database is one of the most widely used commercial databases¹. For the vertical partition approach (similar to [13]), the database cache was set to 320 MB. For the Index Fabric, the database cache was set to 100MB and the embedded engine code was running in a JVM configured with 128 MB of memory from which 15MB was used for caching² (only 243 MB total, compared to 320MB). The size of the 3 indexes created for the vertical layout was 5.6 GB; the Index Fabric size was 1.6 GB.

The data in both setups consisted of 1 million products, with 100 attributes per product³. The queries were for product ID, by attribute/value pairs. For the vertical layout, the queries had from 1 to 4 different attribute value pairs. In order to see the trend with Index Fabric we ran queries with 1, 2, 5, and 10 attribute value pairs. We queried for both common and uncommon attributes, assuming an evolving schema. It should be noted that these relatively “flat” multiple attribute queries are equivalent to contiguous path queries (no intervening wildcards) often seen in semistructured queries.

¹ Our licensing agreements do not allow us to publish the DBMS brand.

² This cache was used to cache the non-leaf blocks of the Index Fabric as well as some of the leaf blocks and data records.

³ We tested different data sets with different numbers of attributes and received similar results.

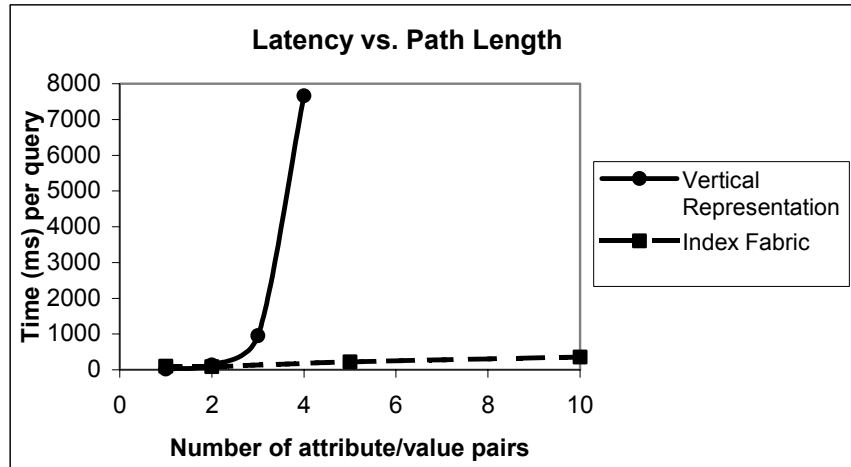


Figure 6: Latency results

With the Index Fabric, searching for increased numbers of attributes scales linearly. The same search over the vertical table rapidly degrades to unusable, even with a small number of attributes. This is an indicator of the high cost for multiple attribute queries, and also for long document path construction. Searching a completely open schema then degrades quickly, especially as the number of queried attributes grows.

Several differences combine to account for the results seen in **Figure 6**, primarily that the vertical approach joins every attribute name with an attribute value. This generates, in many cases, operations on large intermediate result sets. For example, a query for products with 'price=100' first considers all products that have a price attribute. With the Index Fabric, the composite index eliminates the join. Maintaining a similar index with relational DBMS is not practical with an evolving schema; the space requirements balloon since attribute values would be duplicated for every relevant attribute name. Since the attribute value field is large (contains names of products, suppliers, descriptions etc.), it constrains the options for the system and effective indexes. The size of the indexes built over the vertical table can potentially be *two orders* of magnitude larger than the indexes built over the horizontal table. The Patricia structure of the Index Fabric offers high compression, minimizing the size of the index.

As mentioned above, the size of the indexes created for the vertical layout was 5.6 GB, the Index Fabric size was 1.6 GB. In addition, with Index Fabric, the attribute names, the attribute values and the products are each only stored once in different tables. With the relational DBMS, in order to maintain composite indexes, the attribute values (and the attribute names) would need to be stored adjacent to the product IDs, causing the system to revert to the horizontal orientation. With the Index Fabric all the keys addressing the same product have the same row IDs, whereas with the vertical schema attribute/value pairs of the same product address different rows in the vertical table. With vertical tables, once the row IDs for each attribute/value pair were found (by a potentially costly join operation), the actual tuples are read to find if

multiple values belong to the same product. Said differently, each attribute/value pair returns a set of row IDs. To decide which of the row IDs point to tuples satisfying the query, the system needs to retrieve, for each row ID, the product ID from the vertical table. This is what caused the vertical system to behave exponentially with increased number of attribute/value pairs.

4 Conclusions

We have addressed some of the complications associated with indexing open and evolving schemas at both a theoretical and practical level. We mapped out the steps for handling open schemas, covering each of the major issues: key encoding, data interfaces, updating the data, and query support. These steps are generally applicable and may be interpreted broadly for any semistructured data format.

We also present an index well suited for use with open schemas, the Index Fabric. The Index Fabric is a self-describing index. It is based on two cardinal advantages: the use of designators to store properties and structure per object's key, augmented by a compact, high-performance index structure. These features combine to make the Index Fabric an ideal substrate for semistructured and irregular data sources that change over time. Previous experiments demonstrated that the Index Fabric is a successful index for semistructured data, and the preliminary experiments presented here demonstrate that the Index Fabric is an effective index for data with an evolving schema.

5 Bibliography

- 1 A. Deutsch, M. Fernandez and D. Suciu. Storing semistructured data with STORED. SIGMOD, 1999.
- 2 J. Cheng, J. Xu. XML and DB2. ICDE 2000.
- 3 "XML Support in Oracle8i and Beyond." Oracle technical whitepaper, November 1998. http://otn.oracle.com/tech/xml/htdocs/xml_twp.html
- 4 "Tamino White Paper." A Software AG report, October 2001. <http://www.softwareag.com/tamino/download/tamino.pdf>
- 5 B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In Proc. VLDB, 2001.
- 6 J. Clark, ed. XSL Transformations (XSLT). November 1999. <http://www.w3.org/TR/xslt>
- 7 B. Cooper and M. Shadmon. The Index Fabric: A mechanism for indexing and querying the same data in many different ways. Technical Report, 2000. <http://www.rightorder.com/technology/overview.pdf>
- 8 P. Bumbulis and I. Bowman. A Compact B-tree. SIGMOD, 2002.
- 9 P. Ferragina and R. Grossi. "The String B-Tree: A New Data Structure for String Search in External Memory and its Applications." Journal of the ACM, 1998.
- 10 R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. VLDB, 1997.
- 11 B. Cooper, N. Sample, and M. Shadmon. A parallel index for semistructured data. ACM SAC, 2002.

- 12 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.
- 13 R. Agrawal, A. Somani, and Y. Xu: Storage and Querying of E-commerce Data. In Proc. VLDB 2001.
- 14 Notes on the Xschema/DDML W3C submission process, <http://purl.oclc.org/NET/ddml>