Middle-Tier Extensible Data Management

Brian F. Cooper[1,2], Neal Sample[1,2], Michael J. Franklin[1,3], Joshua Olshansky[1] and Moshe Shadmon[1]

[1]RightOrder Inc.
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

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

Abstract

Current data management solutions are largely optimized for intra-enterprise, client-server applications. They depend on predictability, predefined structure, and universal administrative control, and cannot easily cope with change and lack of structure. However, modern e-commerce applications are dynamic, unpredictable, organic, and decentralized, and require adaptability. eXtensible Data Management (XDM) is a new approach that enables rapid development and deployment of networked, data-intensive services by providing semantically-rich, high-performance middle-tier data management, and allows heterogeneous data from different sources to be accessed in a uniform manner. Here, we discuss how middle tier extensible data management can benefit an enterprise, and present technical details and examples from the Index Fabric, an XDM engine we have implemented.

1. Introduction

The multi-tier architecture for distributed enterprise applications is used to provide adaptability, interoperability, and fast time to market for enterprise computing. The benefits are well known, and the multi-tier approach is quickly becoming a standard, especially for e-commerce applications [11,3]. In order to support enterprise applications, this model must be efficient, in terms of both high performance query processing, and efficient development and deployment. Unfortunately, the realization of many of the benefits of the multi-tier architecture is hindered by traditional data management technologies. These technologies suffer from two primary drawbacks: (1) they were designed and optimized for intra-enterprise, client-server applications, with clear, rigid requirements and (2) they are not deployed in a manner consistent with multi-tier, standards-based distributed architectures [1,13]. A new data management architecture is needed.

We propose an approach to this problem, called eXtensible Data Management (XDM). The goal of XDM is to provide high performance management of data that is irregularly structured, or whose structure may change over time. The system must be able to integrate new data into the database, even if the new data has a different schema or structure than the existing data. Users should be able to efficiently access all data, regardless of structure. Moreover, it should be possible over time to go beyond merely "efficient" operations to provide highly optimized access to data along frequently used paths. At the same time, because the data is irregularly structured, the data management system should provide users with assistance in formulating queries. In other words, the data management system should present a self-describing view of the data that can be queried in a robust, flexible way that is resilient to irregularity in the data.

To maximize the benefits of XDM, it is deployed using standardized, modular components in the middle-tier. This will allow the data management system to best meet the needs of applications that are themselves deployed with modular components and distributed using middle-tier application servers. Application designers have been able to utilize application servers that offer clustering, fail-over and flexibility, and we hope to extend the same benefits to XDM.

We have developed an XDM system based on a novel technology, called the Index Fabric. The Index Fabric has several important advantages over existing technology. First, it does not require a pre-existing schema for the data. Instead, the data management system is self-describing, so that the schema can be used in a descriptive manner to aid in formulating queries, rather than in a prescriptive manner to restrict the form of the data. Second, the system is dynamic, supporting the introduction of new data types and relationships. These changes do not require down-time as the system is reconfigured, and can be undertaken without interfering with existing access patterns. Third, the system is efficient and highly scalable, providing order of magnitude performance improvements over traditional systems, even as the size and complexity of the data grows. In previous work [5], we have examined the core technology of the Index Fabric, specifically in the context of managing semistructured data such as XML. Here, we examine how to deploy and exploit this technology in a multi-tier e-commerce architecture.

In this paper, we discuss how extensible, middle-tier data management can address the twin challenges of flexibility and efficiency for today's e-commerce applications. Specifically, we make several contributions:

- We present an architecture for deploying eXtensible Data Management in the middle tier of an e-commerce application.
- We discuss implementing XDM with the Index Fabric, an engine that supports schema flexibility and robust, flexible queries, while providing high performance.
- We illustrate the challenges and solutions of extensibility using a case study.
- We present experimental results that illustrate the performance improvements achievable using the Index Fabric as an XDM engine over XML data.

This paper is organized as follows. Section 2 introduces the case study that we use as a running example. Section 3 outlines the benefits of deploying extensible services in the middle tier. In Section 4, we discuss the technical details of the Index Fabric, an XDM engine that provides a unique combination of flexibility and efficiency. Section 5 will revisit the case study to show how XDM can be applied. Section 6 presents performance results for a particular extensible application using XML. In Section 7 we examine traditional solutions, and in Section 8 we discuss our conclusions.

2. Case study

We will use a hypothetical case study of "Acme Industrial Parts." This case study illustrates the challenges faced when a large diverse enterprise tries to leverage data management for e-commerce, and how an XDM system based on the Index Fabric can help.

Acme is a well-known name in industrial parts, and has several different selling venues. Acme sells its parts directly to large customers, as well as through retailers such as hardware stores. Some orders are placed via Acme's sales force, using a custom built sales application, while some are placed through a web portal. Acme needs a data management system to manage its products and provide

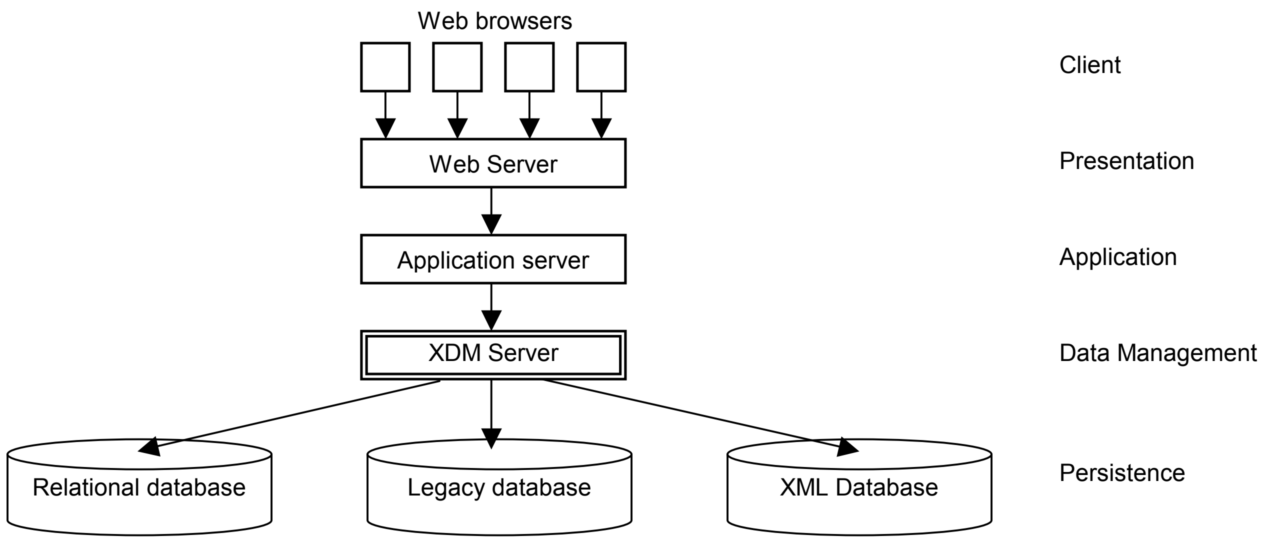


Figure 1. Multi-tiered application architecture with eXtensible Data Management layer.

information about the products to each of these venues, as well as to manage inventory and other internal product information.

Acme faces a number of challenges in trying to use traditional systems to manage its sales and inventory:

Each application has different data needs. Although some of the needs overlap, some are quite different. It is possible to create a global schema that serves all of the business needs and processes, but such a schema is likely to be complex and unwieldy. In contrast, a lowest common denominator schema can be defined containing only the elements needed by all applications, but this schema would not serve any application particularly well.

Acme can choose to build separate back-ends customized for each application. However, this limits the ability to keep all of the back-ends consistent (e.g. new products must be manually loaded in each of the databases according to the local schema). Moreover, these custom back-ends need to be integrated with the enterprise-wide inventory system.

New services are hard to deploy. For example, to add a “custom built industrial parts” service, Acme must create a new database to support that application. This requires glue code to be manually written to integrate that system with others such as the inventory system. If Acme decides not to create a new database, then it must be able to extract appropriate information for the new service from existing databases, even if none of the existing databases are a close fit with the new service.

The system must be efficient and scalable. Acme is always creating more products, producing more information related to its products, and acquiring competitors that have their own product databases. Acme cannot use a solution that is flexible if the solution does not scale.

Clearly, Acme needs a new kind of data management solution. This solution must be able to support multiple different access patterns over the same data. It should be able to store relevant information for each product, without requiring that every product record fit a uniform, global schema. The system should also support evolution of the database to provide new services in the future, and do all these things with high performance.

3. Data management in the middle-tier

There are many longstanding and well-understood arguments for moving applications to the middle tier [11,3]. The traditional client-server architecture is being rapidly replaced by the superior *n*-tiered alternative. The middle tier is an appropriate place for XDM, since there are unique benefits to deploying data services in the middle tier that cannot be realized in any other deployment. These benefits are realized compared to alternate architectures: either placing XDM at the "front-end" (application layer or above, see Figure 1) or at the "back-end" (the database.)

Integration: Middle tier data management allows us to integrate multiple schemas and sources to provide a powerful interface for all front-end applications. Deploying XDM in the middle tier avoids the need to reimplement integration in every front-end application. This allows Acme to bring together databases that may have been developed separately for each warehouse or each business unit.

Redeployment: Business components can be relocated between clients and servers without impacting the architecture of either. This means Acme can upgrade its XDM infrastructure without recompiling and redeploying front-end applications or application server. It also allows redeployment to happen despite heterogeneity of back-end architectures. Moreover, since the deployment is centralized, the management is centralized as well, and thus more efficient.

Clean separation: Data management processes can be isolated from arbitrary restrictions imposed by specific implementations. The front-end architecture is often dictated by the needs and capabilities of the various Acme user groups, and may not be appropriate for data management tasks. The back-end architecture may be dictated by the requirements of the database system or by Acme's large infrastructure investment, and may not be adaptable to the needs of XDM. Clean separation also makes it easier to deploy new applications and databases into the system, since they merely have to understand the relevant APIs in order to interoperate.

Load balancing and work distribution: Acme can enhance scalability by providing parallel clusters in the middle tier to run XDM services. Managing data at the back-end makes load balancing quite difficult, as there is no global external view of incoming requests (and knowledge of peers may be equally limited). Front-end developers should be able to focus on business logic and presentation, and not have to worry about the mechanics of load balancing.

Figure 1 shows the XDM server conceptually in a new middle tier layer. In the actual implementation, the XDM may be encapsulated in a server in its own layer, or may be integrated with the application server (e.g. as the persistence mechanism of a J2EE Enterprise Java Beans layer) The latter option is useful to eliminate network communication latency between the application server and separate XDM server.

4. The Index Fabric

We have argued that an appropriate architecture for eXtensible Data Management is to provide extensibility via a middle tier component. However, there must be a core technology to provide these services that has the required flexibility and scalability. We have implemented the Index Fabric, a data management engine that is a good substrate for XDM capabilities. The Index Fabric represents a new approach to data management that offers both flexibility and high performance. The key components of the Index Fabric are:

- The data representation, which is self-describing and provides the flexibility.
- The indexing structure, which provides the efficiency and scalability.

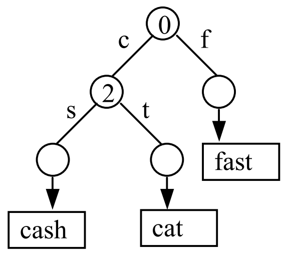


Figure 2. A Patricia trie.

4.1 Index structure

The Index Fabric is a structure that scales gracefully to large numbers of keys, and is insensitive to the length or content of inserted strings. These features are necessary to efficiently support extensible data management.

The Index Fabric is based on *Patricia* tries [15]. An example Patricia trie is shown in Figure 2. The nodes are labeled with their *depth*: the character position in the key represented by the node. The size of the Patricia trie does not depend on the length of inserted keys. Rather, each new key adds at most a single link and node to the index, even if the key is long. Patricia tries grow slowly even as large numbers of strings are inserted because of the aggressive (lossy) compression inherent in the structure.

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

More specifically, as the basic Patricia trie string index grows it is divided into block-sized subtries, and these blocks are indexed by a second trie, stored in its own block. We can represent this second trie as a new horizontal layer, complementing the vertical structure of the original trie. If the new horizontal layer is too large to fit in a single disk block, it is split into two blocks, and indexed by a third horizontal layer. An example is shown in Figure 3. The trie in layer 1 is an index over the common prefixes of the blocks in layer 0, where a common prefix is the prefix represented by the root node of the subtrie within a block. In Figure 3, the common prefix for each block is shown in “quotes”. Similarly, layer 2 indexes the common prefixes of layer 1. The index can have as many layers as necessary; the leftmost layer always contains one block.

There are two kinds of links from layer *i* to layer *i*-1: labeled far links (—►) and unlabeled direct links (-►). Far links are like normal edges in a trie, except that a far link connects a node in one layer to a subtrie in the next layer. A direct link connects a node in one layer to a block with a node representing the same prefix in the next layer. Thus, in Figure 3, the node labeled “3” in layer 1 corresponds to the prefix “cas” and is connected to a subtrie (rooted at a node representing “cas” and also labeled “3”) in layer 0 using an unlabeled direct link.

4.1.1 Searching

The search process begins in the root node of the block in the leftmost horizontal layer. Within a particular block, the search proceeds normally, comparing characters in the search key to edge labels,

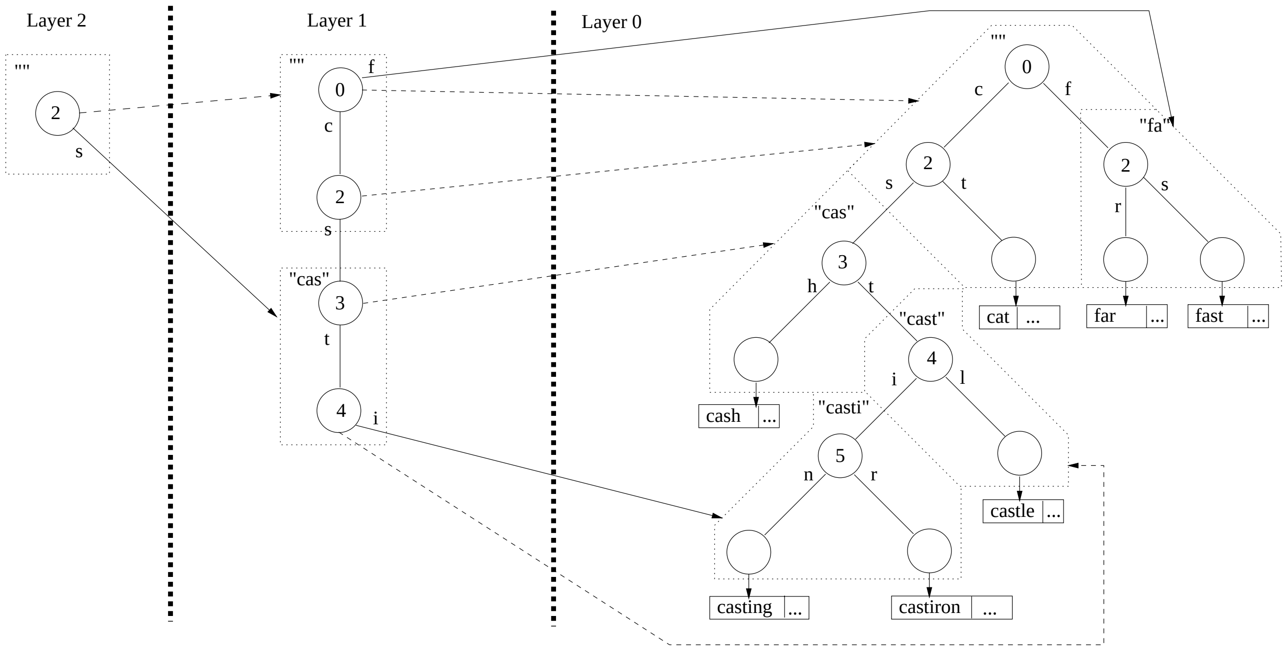


Figure 3. A layered index.

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

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

4.1.2 Updates

Updates, insertions, and deletions, like searches, can be performed very efficiently. An update is a key deletion followed by a key insertion. Inserting a key into a Patricia trie involves either adding a

[1] It is possible for the search procedure to enter the wrong block, and then have to backtrack, due to the lossy compression of prefixes in the non-leaf layers. In practice, such mistakes are rare in a well-populated tree. See [6].

single new node or adding an edge to an existing node. The insertion requires a change to a single block in layer 0. The horizontal index is searched to locate the block to be updated. If this block overflows, it must be split, requiring a new node at layer 1. This change is also confined to one block. Splits propagate left in the horizontal layers if at each layer blocks overflow, and one block per layer is affected. Splits are rare, and the insertion process is efficient. If the block in the leftmost horizontal layer (the root block) must be split, a new horizontal layer is created.

To delete a key, the fabric is searched using the key to find the block to be updated, and the edge pointing to the leaf for the deleted key is removed from the trie. It is possible to perform block recombination if block storage is underutilized, although this is not necessary for the correctness of the index. For insertion, deletion and split algorithms, the interested reader is referred to [6].

4.2 Self-describing data

Keys indexed by the system have embedded semantic hints that describe the nature of the managed data. This feature is necessary to support irregular, non-uniform and dynamic schemas, since XDM system can reflect the actual structure of the data, without having to translate it into a single, uniform schema. It is also necessary to support robust, exploratory search, since the self-describing elements can help the user in formulating queries by revealing what types of information are available in the database. The indexed keys with embedded hints are mapped to data items in the database; however, the data items themselves remain stored in their native form.

The Index Fabric embeds semantic hints by representing data as *designated strings*. A designator is a special character or string of characters that has semantic meaning. The combination of designators and the matching semantic concepts (found in the designator dictionary; see Section 4.3) makes the data self-describing. For example, Acme might assign the designator **T** to "item type," **D** to "dimensions" and **P** to "price." Then, a particular item such as a drill can be represented as the keys "**T** Drill [242]", "**D** 11in x 5 in x 7 in [242]", "**P** $64 [242]"; this encodes that item 242 is a drill, with dimensions of 11in x 5 in x 7 in, and which costs $64. (The object ID 242 may be an XML document number, a rowid in a relational database, an OID in an object oriented database, or any appropriate data pointer.) A different item may be encoded as "**T** Hammer [165]", "**C** Red [165]", "**P** $12"; this is a red hammer that costs $12 (if **C** is "Color"). These items may have come from the same data source, such as the Acme inventory database. Alternately, the second item may have its own schema because it originally resided in a different database, such as the inventory database of a company Acme acquired. By encoding both records using the same metaphor (designated keys), schema flexibility is possible, since the data engine only has to manage designated strings and does not require a uniform schema or data format. Moreover, the designators assist searches, since the data engine itself can indicate to users that some items have color information, and other items have dimension information. The system does not have to maintain explicit NULLs to indicate that a record does not have an attribute.

Note that the task of choosing appropriate self-describing semantic hints is not trivial. We are not trying to solve the problem of automatically extracting semantic information from source data or a full ontological markup language. Instead, we hope to provide the basic building blocks, in the form of designators, for managing and searching information with high efficiency. We also assume that a mechanism exists for dealing with varying formats for the data itself. For example, one price may be represented as "$5" while another is represented as "5 US$". A data cleaning step or wildcards in the query language are traditionally used to manage such discrepancies, and can be applied here as well.

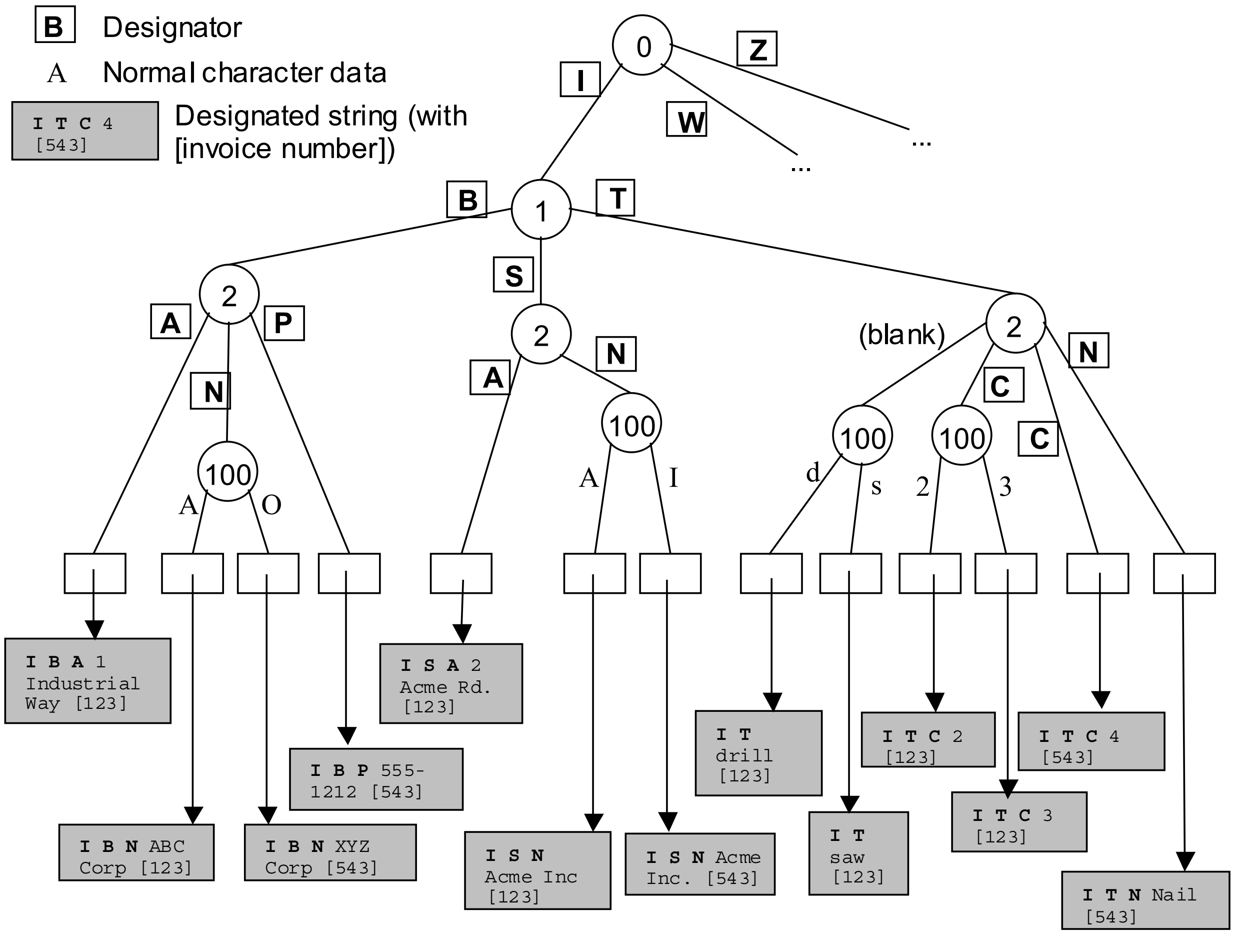


Figure 4. Patricia trie index for invoices.

Within the same XDM system, Acme can manage sales information. Figure 4 shows a portion of the index for invoices; this illustrates how designated records are represented in the Patricia trie structure. (For clarity, the multilayer structure is omitted in the figure.) Designators in this example are strings constructed from individual semantic concepts, and the following concepts are used in the figure:

`invoice` = **I**	`address` = **A**	`phone` = **P**
`buyer` = **B**	`seller` = **S**	`count` = **C**
`name` = **N**	`item` = **T**	

To search the index in Figure 4, the system constructs a search key based on the user query. Thus, to search for invoices where ABC Corp. is the buyer, the system would search for "**IBN** ABC Corp." This is done by starting at the root (the node labeled "0"), and following the appropriate edges (labeled "**I**", "**B**", etc.) until the correct data record is reached.

4.3 Dynamic structure

The data engine is able to manage new types and structures of data over time. This is important to support seamless integration of new data sources, which may have schemas different from the existing database. Moreover, it is possible to enhance the existing data by adding new tags. For example, Acme may decide to add a new attribute, "weight," to its product records. This is done simply by creating a new designator for the new attribute, and then inserting keys encoded with the new designator.

The Index Fabric tracks dynamic tags via a *designator dictionary*, which allows designators to be mapped to semantic concepts. In the case of Acme, the dictionary keeps the relationships (**T** → type, **D** → dimensions, **P** → price, …). Because new mappings can be added to the dictionary at any time, new designators can be created to support new semantic concepts. Adding new concepts is also efficient, requiring only the addition of a new mapping to the dictionary and then new designated strings to the data system. These tasks do not disrupt existing data access paths or applications. Moreover, the designator dictionary provides additional flexibility, such as the ability to update mappings (e.g. change "price" to "cost") or map multiple concepts to the same designator (e.g. map both "price" and "cost" to **P**). This flexibility is an advantage of using designators to represent semantic information.

4.4 First-class relationships

The XDM system manages important relationships between data items as first class objects. This means that relationships are explicitly materialized, and managed in the same way that data items are managed. This allows us to efficiently deal with complex relationships, since these relationships do not have to be reconstructed at query time (e.g. using joins in a conventional tuple-oriented representation). Moreover, if the relationships are first-class objects, they are managed as self-describing items. This supports query formulation, since the user or application can browse the XDM system to determine what relationships exist between the data items. For example, after reaching the node labeled "1" in Figure 4, the user can see what types of objects (in this case, buyer **B**, seller **S**, and itemlist **T**) are related to invoices.

The Index Fabric manages important relationships by representing them as designated strings, just like normal data elements. For example, to represent the fact that an item "**T** drill bit [789]" is to be used with another item "**T** drill [988]," the database administrator can direct the system to materialize the key "**T** drill [988] **T** drill bit [789]." This key is treated the same as any other designated string, and thus is a first-class object. To search for bits for a particular drill, we can search for keys prefixed by "**T** drill [988] **T** drill bit." Similarly, a search for keys prefixed by "**T** drill [988]" returns everything related to that drill item, either returning the designators describing related item types, or returning the related items themselves (depending on the user requirements.)

4.5 Efficiency and scalability

The Index Fabric provides highly efficient and scalable data management, which allows the system to support large numbers of designated keys with high performance. The system is scalable in terms of: the amount of data managed (because of the small index size), the complexity of the data (because of the support for long, designated strings), and the number of access paths through the data (because of the ability to manage multiple paths in a single, compact index). Newly integrated data can be queried efficiently along "raw access paths" that follow the structure of the data. Over time, as the importance of certain alternative access paths becomes apparent, a data administrator can add "refined access paths" that provide optimized access along these paths. (See [5] for a detailed discussion of raw and refined paths). Many access paths can be optimized in this way because new access paths are represented simply as keys in the index, and keys can be added to the index cheaply due to the compression provided by Patricia tries.

The multilayer Patricia trie index provides fast lookups and updates, even if the database is large. Moreover, the length of keys does not impact efficiency. In previous work [5] we have compared using a popular commercial relational database with and without the addition of a middle tier Index Fabric XDM system. Adding the Index Fabric increased the performance of the system by an order of magnitude or more. Moreover, as the complexity of the data and queries grew, the performance gain provided by the Index Fabric also grew, demonstrating that the flexibility of the Index Fabric XDM does not come at the cost of performance.

In the example of Acme, a scalable and efficient Index Fabric means that new services can be deployed, and an ever-increasing product line managed, without fear of bogging down the whole system. This frees the company to invent new ways to serve customers, without having to worry about whether the underlying data management system is up to the task.

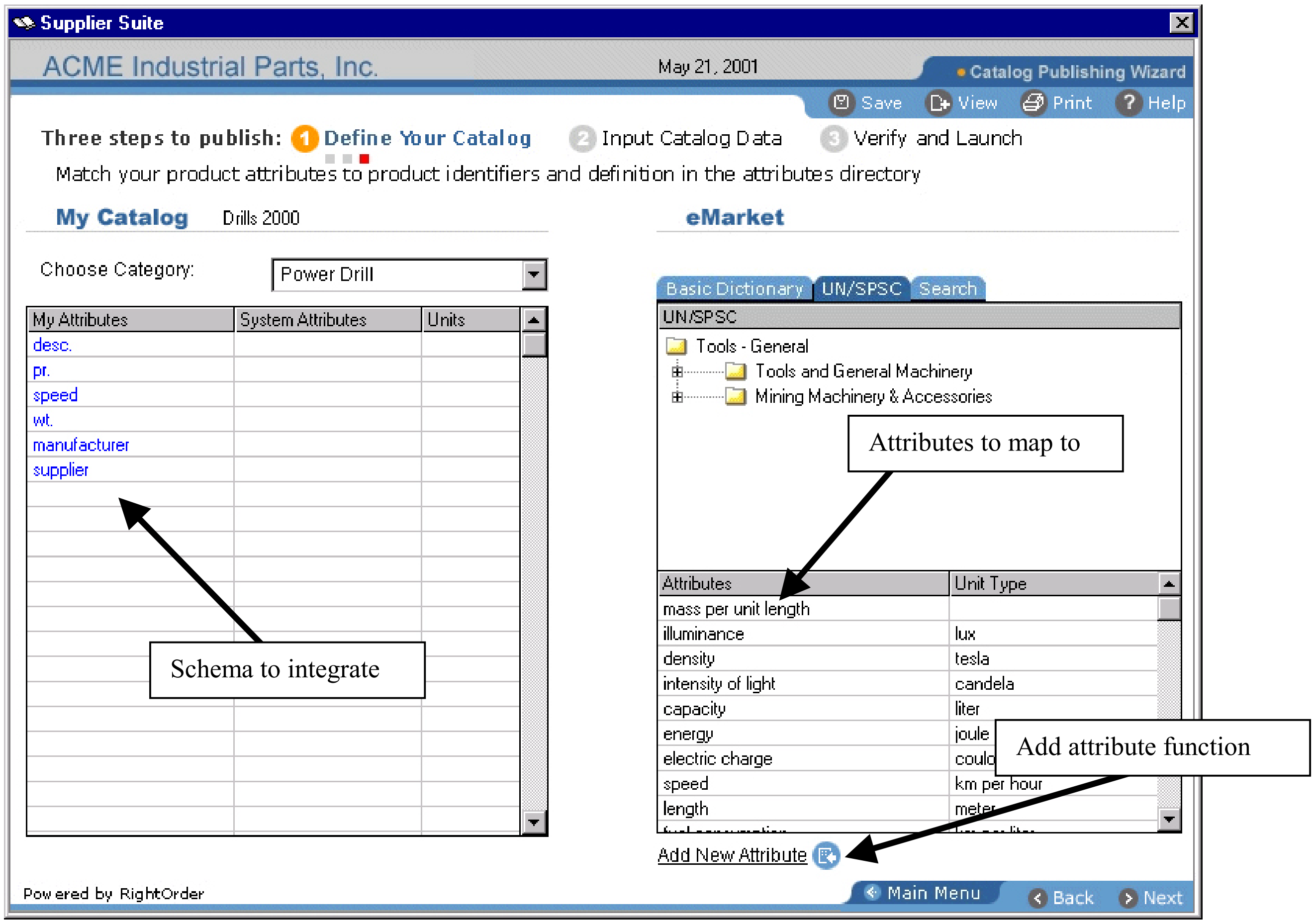
5. Acme revisited

By deploying XDM in the middle tier, Acme can support rapid development of applications. Moreover, using the Index Fabric as the basis for XDM provides a high performance, scalable solution. We have implemented the Product Directory, an application that companies such as Acme can use to manage direct sales. The Product Directory application leverages the extensibility of the Index Fabric XDM layer to provide several key features; in this section, we focus on two such features: 1) the ability to integrate new product data into an existing directory, and 2) the ability to browse the evolving structure of the database.

From time to time, Acme must add new product records to the directory. These records may not match the schema of the existing products, such as when Acme acquires a company and wants to integrate its existing product database. Integration is done using the "publish catalog" function of the Product Directory. Figure 5 shows a screenshot of the interface to the application. On the left of the screen ("My Catalog") is the schema of the new data to be integrated. On the right of the screen is the external view of the current integrated database ("eMarket").

A user can create a mapping between the attributes in the new data and attributes that already exist in external view. For example, the "speed" attribute of the new product may match the existing "speed" attribute, but the "wt." attribute may have to be mapped to an existing "weight" attribute. This mapping leverages the designator structure within the Index Fabric: each attribute is mapped to a designator, and multiple attributes can map to the same designator. As a result, once records are added to the index, they are searchable by existing applications, because the attribute names are represented internally using designators that already work for the existing applications. If the new products have attributes that do not match any existing attributes, then the "Add new attribute" function can be used. This efficiently changes the schema of the underlying database, because new designators can be added without rebuilding the whole index.

The search function is similarly flexible. Users can navigate the relationship structure of the database directly to find the products they need. This navigation follows the relationship structure that exists explicitly in the Index Fabric. The screenshot in Figure 6 illustrates a form of exploratory search, where the application gives feedback about which attributes are available and which may be relevant to the current search. The user has performed a basic search (for example, on a keyword) and



My Attributes	System Attributes	Units
desc.		
pr.		
speed		
wt.		
manufacturer		
supplier		

Attributes	Unit Type
mass per unit length	
illuminance	lux
density	tesla
intensity of light	candela
capacity	liter
energy	joule
electric charge	could
speed	km per hour
length	meter

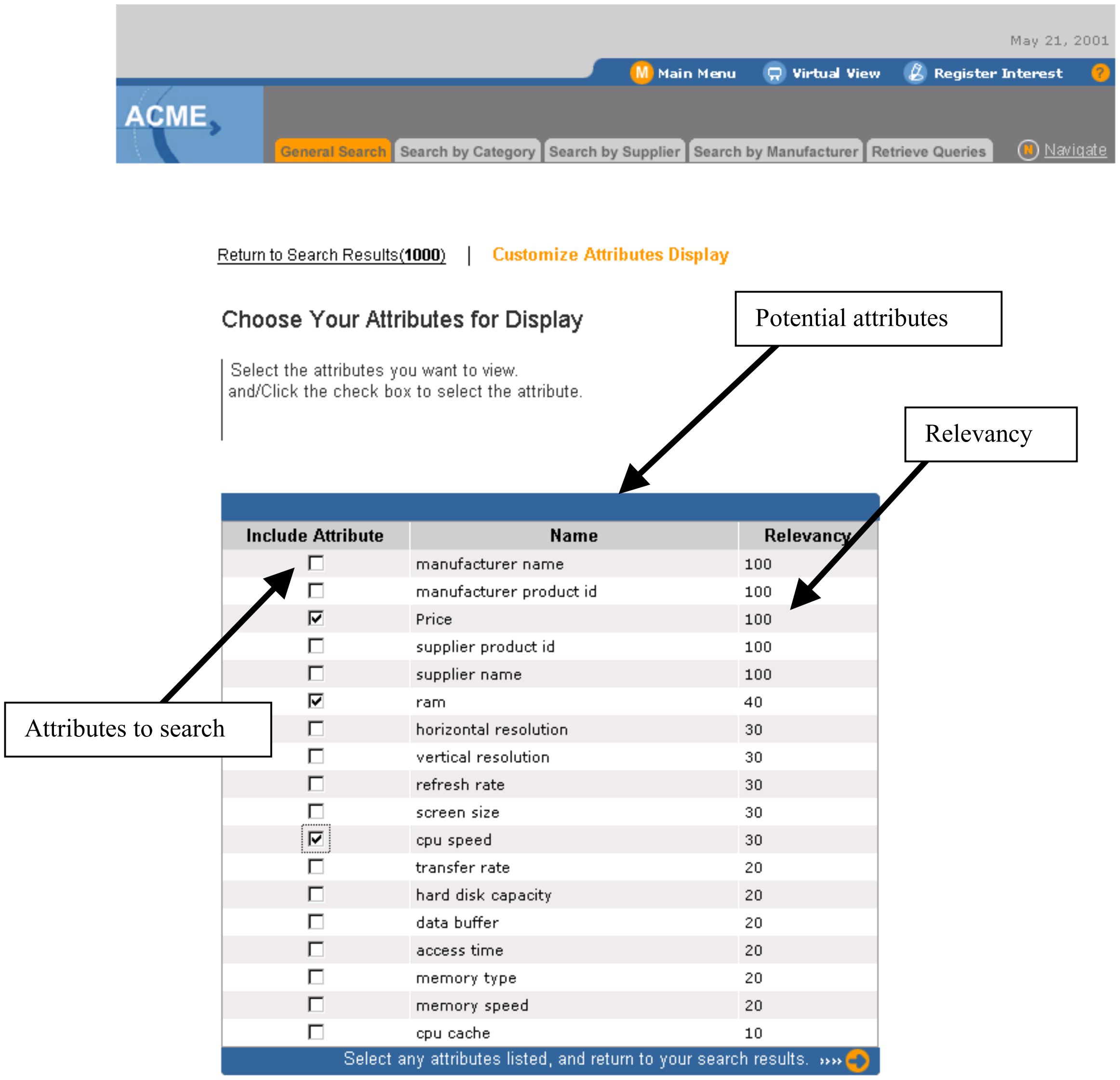
Figure 5. The publish function of the Product Directory.

the system has returned a large result set. Now the user can focus the search, based on the attributes that exist for the products in that result set. The “Include Attribute” column allows the user to decide which attributes to search on, chosen from the list in the “Name” column. “Relevancy” indicates the proportion of the result set that has the named attribute. For example, 100 percent of the returned products have a “supplier name” attribute, while only 40 percent have a “ram” attribute. The relevancy is provided from the designator information in the index, and is used to reorganize the user interface (in this case by sorting the attributes). This illustrates how the self-describing nature of the Index Fabric allows the application to dynamically adapt to the user’s current needs.

The flexibility of the Index Fabric means that this application can be built easily. The efficiency and scalability of the Index Fabric means that the application performs well, even for large product databases.

6. Experimental results

We have conducted experiments to measure the performance of the Index Fabric in an extensible data management application. The application we studied is a bibliography server with an extensible schema for publication records; this schema can be represented using XML tags. We stored an XML-



Choose Your Attributes for Display

Select the attributes you want to view.
and/Click the check box to select the attribute.

Include Attribute	Name	Relevancy
☐	manufacturer name	100
☐	manufacturer product id	100
☑	Price	100
☐	supplier product id	100
☐	supplier name	100
☑	ram	40
☐	horizontal resolution	30
☐	vertical resolution	30
☐	refresh rate	30
☐	screen size	30
☑	cpu speed	30
☐	transfer rate	20
☐	hard disk capacity	20
☐	data buffer	20
☐	access time	20
☐	memory type	20
☐	memory speed	20
☐	cpu cache	10

Select any attributes listed, and return to your search results.

Figure 6. Exploratory search interface.

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. Our performance results thus represent an "apples to apples" comparison using the same storage manager.

6.1 Experimental setup

The data set we used was the DBLP, the popular computer science bibliography [7]. The DBLP is a set of XML-like documents; 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

[2] The license agreement prohibits publishing the name of the DBMS with performance data. We refer to it as "the RDBMS." Our system can interoperate with any SQL DBMS.

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

Figure 7. Sample DBLP document.

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.

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 is somewhat regular (e.g. every publication has a title) the structure varies from document to document: the number of authors varies, some fields are omitted, and so on.

We used two different methods of indexing the XML via the RDBMS' native indexing mechanism. The first method, the *basic edge-mapping*, treats the XML as a set of nodes and edges, where a tag or atomic data element corresponds to a node and a nested relationship corresponds to an edge [9]. The database has two tables, *roots(id,label)* and *edges(parentid,childid,label)*. The *roots* table contains a tuple for every document, with an *id* for the document, and a *label*, which is the root tag of the document. The *edges* table contains a tuple for every nesting relationship. For nested tags, *parentid* is the ID of the parent node, *childid* is the ID of the child node, and *label* is the tag. For leaves (data elements nested within tags), *childid* is *NULL*, and *label* is the text of the data element. For example, the XML fragment

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

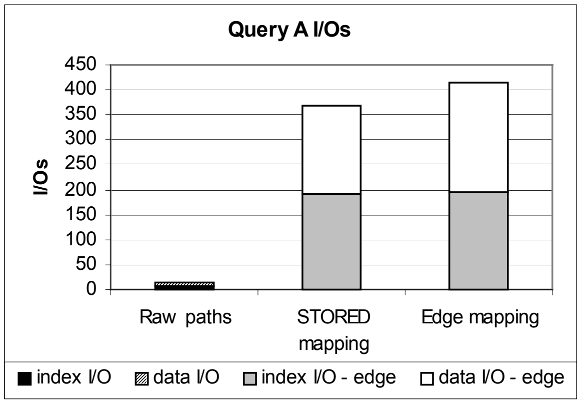
is represented by the tuple (0,book) in *roots* and the tuples (0,1,author) and (1,*NULL*,Jane Doe) in *edges*. (Keeping the leaves as part of the *edges* table offered better performance than breaking them into a separate table.) We created the following key-compressed B-tree indexes:

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

The second method of indexing XML using the DBMS' native mechanism is to use the relational mapping generated by the STORED [8] system to create a set of tables, and to build a set of B-trees over the tables. We refer to this scheme as the *STORED mapping*. STORED uses data mining to extract schemas from the data based on frequently occurring structures. The extracted schemas are used to create "storage-mapped tables" (SM tables). Most of the data can be mapped into tuples and stored in the SM tables, while more irregularly structured data must be stored in *overflow buckets*, similar to the edge mapping. The schema for the SM tables was obtained from the STORED investigators. The SM tables identified for the DBLP data are *inproceedings*, for conference papers, and *articles*, for journal papers. Conference and journal paper information that does not fit into the SM tables is stored in overflow buckets along with other types of publications (such as books.)

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

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



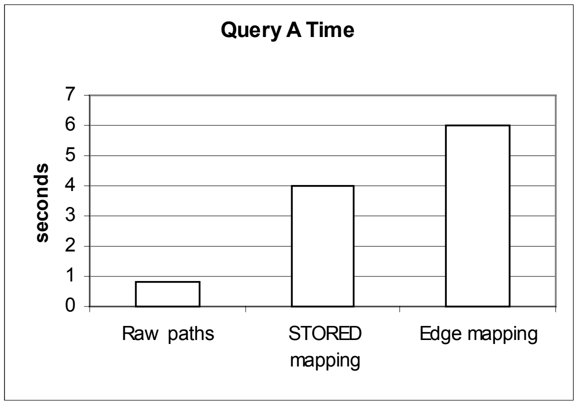


Figure 8. Performance for query A: Find book by publisher.

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

The Index Fabric contained both raw paths and refined paths for the DBLP documents (see Section 4.5 and also [5]). The index blocks were stored in an RDBMS table. All of the index schemes we studied referred to the data using document IDs. Thus, a query processor will use an index to find relevant documents, retrieve the complete documents using the IDs, and then use a post-processing step (e.g. with XSLT) to transform the found documents into presentable query results. Here, we focus on the index lookup performance.

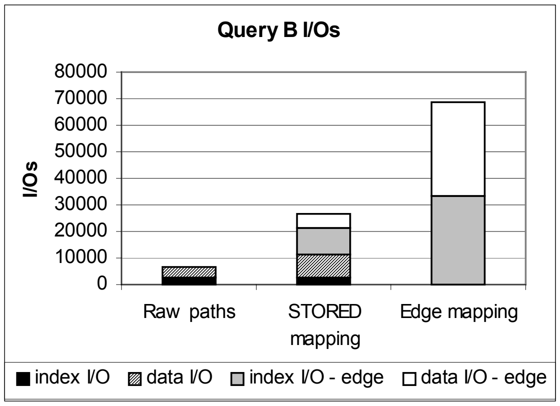
All experiments used the same installation of the RDBMS, running on an 866 MHz Pentium III machine, with 512 Mb of RAM. For our experiments, we set the cache size to ten percent of the data set size. For the edge-mapping and STORED mapping schemes, the whole cache was devoted to the RDBMS, while in the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

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

The experimental results are discussed below. In each case, our index is more efficient than the RDBMS alone, with more than an order of magnitude speedup in some instances. We discuss the queries and results next.

6.2 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



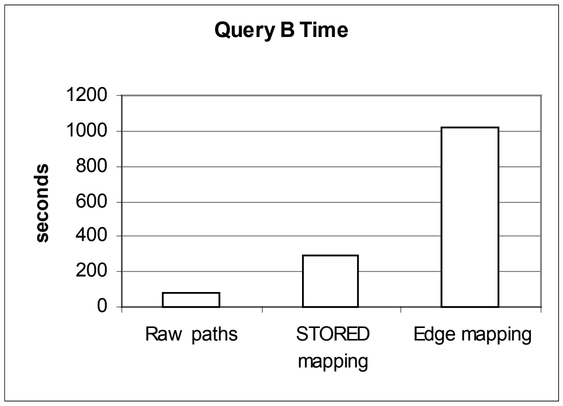


Figure 9. Performance for query B: find conference paper by author.

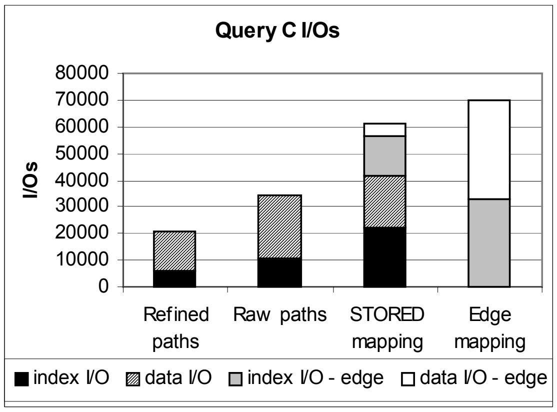
results with "publisher" tuples from the *edges* table, and joining again with the edges table to find data elements "*X*". The query cannot be answered from the storage mapped tables (SM tables) in the STORED mapping. Because books represent less than one percent of the DBLP data, they are considered "overflow" by STORED and stored in the overflow buckets.

The results for query A are shown in 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.

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.

6.3 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.



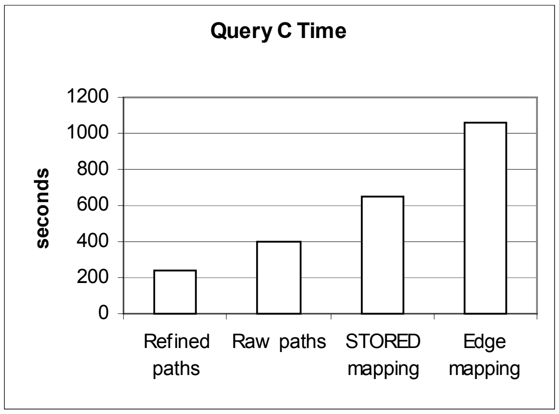


Figure 10. Performance for query C: Find all publications by authors.

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.

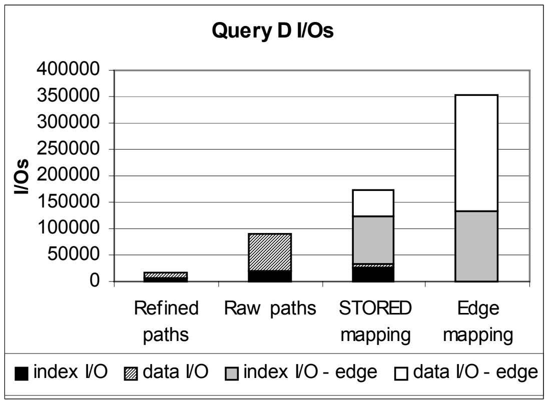
The queries over the STORED mapping are able to efficiently access the SM table (via a B-trees on the *author* attributes), achieving roughly equal performance with the raw paths. However, we must also access the overflow buckets to fully answer the query. In fact, more time is spent searching the overflow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

6.4 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



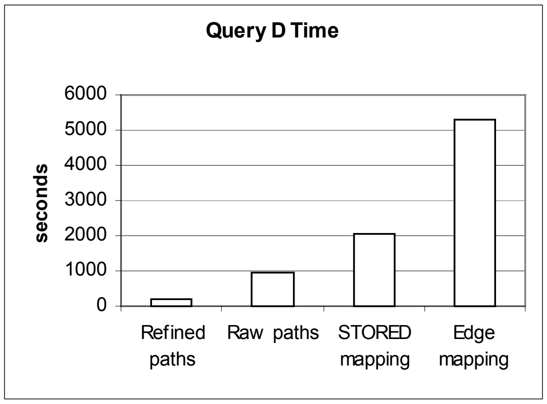


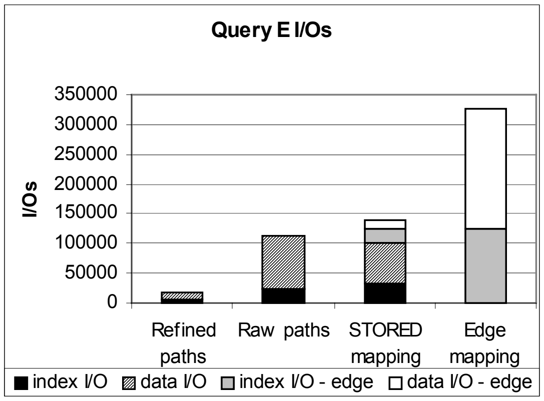
Figure 11. Performance for query D: Find all publications by co-authors.

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.

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.



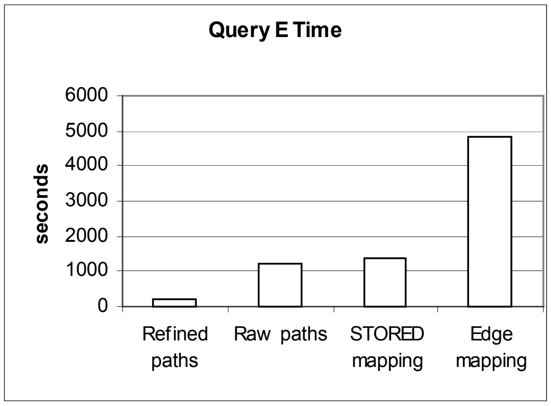


Figure 12. Performance for query E: find publications by authors and year.

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, the raw paths only provided an 18 percent decrease in I/O's versus the STORED mapping. 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.

6.5 Discussion

Our results indicate that a significant cost for a relational system managing extensible 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 extensible data.

7. Existing solutions

Other approaches have been suggested for providing adaptable data management over heterogeneous sources. One possibility is to use a wrapper/mediator architecture [2,10,14,18]. In this approach, a common data schema is defined, and a wrapper is created for each data source to translate between the common schema and the source's native interface. A mediator handles the task of accepting queries, forwarding them to the appropriate wrappers, and collating the results. Unfortunately, significant work must be expended to create a wrapper for every new source that enters the system. Moreover, it is often necessary to reconfigure or rewrite the wrappers and mediator to handle new services. Thus, while mediators can integrate heterogeneous sources, they are too inflexible for dynamic environments.

Another possibility is to provide unstructured access to the data [16,17]. This solution is similar to a web-search engine: searches are formulated as key words, and the data engine performs full-text searches over the databases [4,12]. This solution overcomes the problem of heterogeneity of structure. Moreover, the search cannot take advantage of semantic information present in the structure of the data. Thus, a great many results can be returned that are not relevant to the search, and must be filtered by the user or end application. At the same time, the underlying sources may not provide full-text search capability, eliminating this as a feasible option.

8. Conclusion

E-commerce applications are placing ever greater demands on enterprise data management infrastructure. eXtensible Data Management provides features key to rapid development and deployment of new services, while maintaining and evolving existing applications. These features include the ability to integrate multiple sources, manage dynamic and irregular schemas, provide assistance in formulating queries, and manage complex relationships, all while providing high performance and encapsulating data management. At the same time, we have argued that these data services should exist in the middle tier, to provide such benefits as source security, efficient redeployment, and enhanced modularity. An implementation of XDM, such as our Index Fabric, can provide immense benefits, both in the short term and over time, to developers of e-commerce applications.

References

[1] R. Agrawal, A. Somani and Y. Xu. Storage and querying of e-commerce data. In *Proceedings VLDB*, September 2001.

[2] Y. Arens, C. Chee, C. Hsu and C. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. In Journal of Intelligent and Cooperative Information Systems, Vol. 2, June 1993.

[3] C. Berg. The state of Java application middleware, part 1. JavaWorld, March 1999.

[4] W. Cohen. A web-based information system that reasons with structured collections of text. In *Proceedings of Autonomous Agents AA-98* (1998), 400-407.

[5] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings VLDB,* September 2001.

[6] Brian Cooper and Moshe Shadmon. The Index Fabric: Technical Overview. Technical Report, 2000. Available at http://www.rightorder.com/technology/overview.pdf.

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

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

[9] 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.

[10] H. Garcia-Molina et al. The TSIMMIS approach to mediation: data models and languages. Journal of Intelligent Information Systems, 8:117--132, 1997.

[11] L. Haas et al. Optimizing Queries across Diverse Data Sources. In *Proceedings VLDB*, September 1997.

[12] A. Howe, and D. Dreilinger. SavvySearch: A MetaSearch Engine that Learns Which Search Engines to Query. AI Magazine, 18(2), 1997.

[13] A. Jhingran. Moving up the food chain: supporting e-commerce applications on databases. SIGMOD Record, 29(4): 50-54, December 2000.

[14] W. Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *Proceedings VLDB*, September 1991.

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

[16] A. Salminen and F.W. Tompa. Pat expressions: an algebra for text search. Acta Linguista Hungarica 41, pages 277-306, 1994.

[17] VanRijsbergen, C. J. Information Retrieval. London: Butterworths, 1979.

[18] G. Wiederhold and M. Genesereth. The conceptual basis for mediation services. IEEE Intelligent Systems, pages 38-47, September/October 1997.