XML Semantic File System

Wei Shen, Yuan Wang, Ying Wu
Computer Sciences Department
University of Wisconsin
{weishen, yuanwang, yingwu}@cs.wisc.edu

1. Introduction

XML (eXtensible Markup Language) has recently emerged as a new standard for data representation and exchange on the Internet. Unlike HTML files, in which tags specify how data are formatted and displayed, XML files use tags on data elements to identify context of data, which provides data in more convenient and usable format. Realizing the benefits that XML could bring, many people and organizations are now storing their data in XML format on local servers. For example, Fig.1 shows an XML file stored in DBLP[1] server [Ley]. Similar examples can be found in many public gene data repositories [GJ00].

```
<?xml version="1.0" ?>
<dblp>
    <inproceedings key="conf_sigmod_GraefeD87">
        <crossref>conf/sigmod/87</crossref>
        <author>Goetz Graefe</author>
        <author>David J. DeWitt</author>
        <title>The EXODUS Optimizer Generator.</title>
        <pages>160-172</pages>
        <year>1987</year>
        <booktitle>SIGMOD Conference</booktitle>
        <url>db/conf/sigmod/sigmod87.html#GraefeD87</url>
        <cdrom>SIGMOD87/P160.PDF</cdrom>
        <ee>db/conf/sigmod/GraefeD87.html</ee>
        <cite label="ASTR76">journals_tods_AstrahanBCEGGKLMMPTWW76</cite>
        <cite label="CARE86a">conf_vldb_CareyDRS86</cite>
        …
        <cite label="YOU79">conf_vldb_YoussefiW79</cite>
        <cite label="ZANI83">conf_sigmod_Zaniolo83</cite>
    </inproceedings>
</dblp>
```

Figure 1 an XML file example

Although most XML researchers are working on designing XML query languages, query engines and search engines, more and more people have realize that XML storage management is one of several critical problems in using XML [Wid99]. Currently, there are two typical approaches to XML data:

1) All XML data are stored in regular file systems as plain text.

2) All XML data are stored in database systems, either relational DBMS [STH+99][FK99] or object-oriented database storage management system [TDCZ00].

The advantage and disadvantage of the first approach is apparent. It needs no additional implementation and it is easy for a web server to access and transfer XML data. It cannot, however, give any support to query

[1] The DBLP server provides bibliographic information on major computer science journals and proceedings which is focused on DataBase systems and Logic Programming.

engines and search engines. Query engines have to parse XML files into Document Object Model (DOM)[2] format before performing queries, and search engines can use only string matching because of lack of indices. On the other hand, database storage can avoid expensive parsing and provide DOM interface to query engines and additional indexing facilities to search engines, but it requires quite a complex system to be installed on XML data server. Traditional database storage systems do not take advantage of special features of XML data, e.g., most on-line XML data are read-only, so they will introduce much overhead in managing XML data. Furthermore, there are many small web sites that do not have (maybe do not want to have) query capability and cannot afford installing those big storage systems, but they want their data to be easily accessed by remote query engine and search engines as well as by local users. Unfortunately, database storage systems do not provide any information on finding specific XML files given queries because original XML files are thrown away after being parsed into database.

Based on observation of pros and cons of both approaches, we designed and implemented an XML semantic file system, integrating essential parts of both database storage systems and semantic file systems. The interesting features of our system include:

1) XML files are parsed and stored in well-structured disk files, which size is less than original files. The system can provide DOM to regular XML query engines because those files have tree structures inside (but compact);
2) It supports querying on original XML files, i.e., it return names of files which contain elements satisfying user queries. This is a function of semantic file systems;
3) It allows users to build indices on elements or attributes to accelerate querying and searching;
4) Our system is easy to use – it provides both command line interface to local user and programming interface to local or remote query engines and search engines;
5) The system is built on Unix file system without changing Unix kernel code. It is easy to be installed and maintained.

The rest of this paper is structured as follows. In Section 2 we introduce the related work. In Section 3 we describe the architecture of our file system and the usage. Some implementation details are given in Section 4, and in Section 5 we evaluate the performance of our system. Finally, in Section 6 we summarize our results and discuss some future work.

2. Related Work

Currently, most XML research is focused on designing XML query languages and querying systems [DFF+98][QLW98][NDM+00][GMW99]. Only a few methods have been proposed to try to solve XML data storage problem.

One approach is to modify traditional relational DBMS to store XML documents and perform queries on them [STH+99][FK99]. The advantage of this approach is that it can reuse mature and high performance database technologies and avoid implementing a whole new system. However, it has serious performance weakness on mapping XML data with complicated Document Type Description (DTD) [BBC+98] structure to relational tables and processing XML queries.

Other approaches include either 1) using certain well-implemented object storage manager, like SHORE [CDN+94] to store parsed XML data [TDCZ00]; or 2) implementing new semistructured data storage managers [MAG+97][DFS99][SLS93+][TDCZ00]. Both methods can easily support DOM interface to query engines, but they are more like database and cannot provide original file information to users.

[2] The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the contents, structure and style of XML documents.

The idea of this project was enlightened by semantic file system [GJSO91] and content-based access mechanisms on hierarchical file system [GM99]. Both systems provided basic querying capabilities on content of files by introducing virtual links into file system. They are not, however, suitable to querying XML files because the semantics of querying XML data itself is clear so that we do not need to combine pre-selected queries with files. Also conventional semantic file systems can not answer arbitrary queries which are not set when files are added to the system.

Our approach has integrated benefits from both database storage and semantic file systems in that this system supports both database queries and file system queries efficiently. It provide DOM interface to database query engines and answer file searching queries to file system users.

3. System Architecture and Usage

This system is built upon Unix file system. Fig. 2 shows the architecture of main parts of the system.

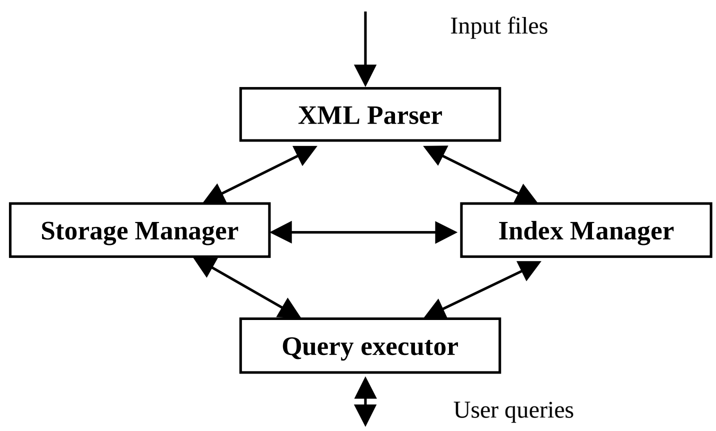


Figure 2 system architecture

The function of each part is:

1) *XML Parser* is used to parse XML source files into the system, retrieving elements and attributes information to build storage and indices.

2) *Storage Manager* gets content of source files and converts it to a special data format. Converted data is stored in numbers of small files in Unix file system.

3) *Index Manager* builds indices based on elements or attributes specified by users. Indices can be used to search both XML content in storage and original file names.

4) *Query executor* provides both command line interface to local users and programming interface (DOM) to XML query engines.

```
>add –IC year –I author papers/*.xml
> ...
```

Figure 3 Adding files

```
> cd year= “1999”
> ls
> cd author = “DeWitt”
> ls
```

Figure 4 Simple search

```
> cd year= “1999”&&author= “DeWitt”
> ls
```

Figure 5 Composite search

Commands in Fig. 3, Fig. 4 and Fig. 5 show how users can add and search files in our system. Users can use “add” command to store XML data into the system and specify elements or attributes on which indices are built (C means to build clustered index). Commands in Fig. 4 and Fig. 5 give out same results because *Query executor* has a history pool that keeps old user queries and allow user to refine query results as in traditional file systems. Query engines and search engines can also access storage through DOM interface, e.g., retrieving the value of an element by specifying the element path, like “get <dblp> <inproceedings> <paper> <author>”.

Due to the strict project deadline, we do not implement a buffer management for the file system, which can help improving cache hit in memory. In general, database buffer management have much better performance than that of regular file systems because it understands the behavior of database applications better [Sto81].

4. Implementation

Our XML Semantic File System is built on Unix file system. It consists more than 7,000 lines of C++ code. The implementation details of each subsystem are described below.

4.1 XML Parser

The *XML File Parser* is built upon IBM XML parser by instantiating its ParserHanlder class. Basically, the parser reads and scans user input XML files. It retrieves values of elements and attributes alone with data structures from files and passes them to the *Storage Manager*. When it receives record meta information from the *Storage Manager*, it sends record information, original file information (file id) and user specified element values to the *Index Manager*. The *Index Manager* then builds indices based on that information.

4.2 Storage Manager

There are two goals when we design the *XML Data Storage* subsystem:

1) It should be easy and efficient to provide DOM interface, i.e., support both tree structure traversal and direct value fetching on a user specified element path.
2) It should be compact enough because the storage itself is considered as system overhead with respect to the original files.

The structure and implementation are the trade-off of the two goals.

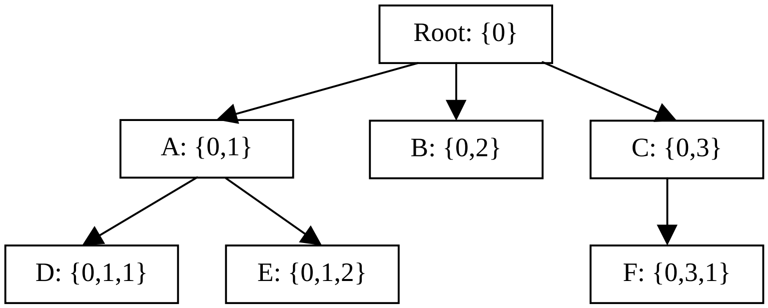
When designing the main structure of the *Storage* subsystem, we thought about two choices – direct tree structure and hash table (to build tree structure indirectly). It seems that directory tree structure is a natural and intuitive way to store parsed XML data because XML itself displays a tree structure. After careful consideration, however, we decided to use hash table based on an assumption (observation) that the most frequent operations in our system are to fetch the value of a specified element or element path. Using hash table, we can retrieve the value of those requests in 2-4 steps, no matter how large the record is. On the other hand, if we internally store records using tree structure, it needs, in average, traverse half of the record to find the required value with specified element name.

In the hash table implementation, we designed a method to describe the element and attribute position information (within the tree structure) explicitly. First, we hash element and attribute names to integers because such information is stored repeatedly in original files. This helps decreasing the size of XML data. Second, we use lists of hash keys, <integer, ..., integer> to specify the location relation between two directly connected elements, which can provide efficient tree structure operations. Fig. 6 shows a simplified tree example in our hash table implementation; Fig. 7 shows its original data in XML format. The root is denoted as “{0}”. The tag {0,1} on element A represents that A is the first child of the root, and so on.

With the above abstraction of *location*, we build two hash tables in the *storage*.

1) The first hash table contains <location, element name, and element value> in which location is the key. This hash table is required in order to support DOM because it provides accessing element values

through element path.




```
<Root>
    <A>
        <D>value</D>
        <E>value</E>
    </A>
    <B>value</B>
    <C>
        <F>value</F>
    </C>
</Root>
```

Figure 6 Example of hash table of tree representation

Figure 7 Original XML file of the tree in Figure 6

2) The second hash table is optional; it contains <element name, location list>, and helps to locate an element in the XML tree structure.

Fig. 8 shows the algorithms of two basic operations on storage, pre-order traversing and retrieving element value. As we can see, the implementation of these two hash tables represents the trade-off between compactness and efficiency. The second hash table is optional, taking about 1/5 of the storage size but can speed up accessing elements because we may have to scan the entire tree with only hash table 1.

```
Pre-order Traverse:
  current = {0} // start from the root.
  while (current != NULL ) {
    use hash table 1. to get current element.
    visit the current element.
    current = current's first child.
    while (no such location on hash table 2) {
        if (current is child of root)  return;
        current = the next sibling of current's parent.
    }
  }
```

```
Fetch(element_name) { // get all values of the
                      // specified element name.
   use hash table 2 to get location.
   use hash table 1 to get the value.
}
```

Figure 8 Algorithms to implement tree traversing and element fetching

Besides hash tables, we use other techniques in our storage system. For instance, we use *slotted page* to store variable length records. The number of physical pages in each storage files is a system parameter.

4.3 Index Manager

Indexing techniques are widely used in database systems, and are critical to the performance of all storage systems. In our system, we also provide to users the functionality to build indices to expedite query processing. We have implemented a B-Tree-like index module shown in Fig. 9. B-Tree index can support both equal-queries and range queries.

We can see that the structure of our index is very similar with that of B+ tree. Each internal node contain n key values and n+1 pointers and each pointer points to a next level node (can be either an internal node or a leaf node). A leaf node consists of a list of 3-element tuples, (pointer, meta information, key value). Within each tuple, the pointer points to a list of Rids, which indicate the addresses of those elements in the storage that have the key value.

4.4 Query Executor

The *Query Executor* provides a command line interface to file system users. It supports both equal-queries and range queries, e.g., “cd year = “1990””, and “cd year > “1995””. It also supports complex

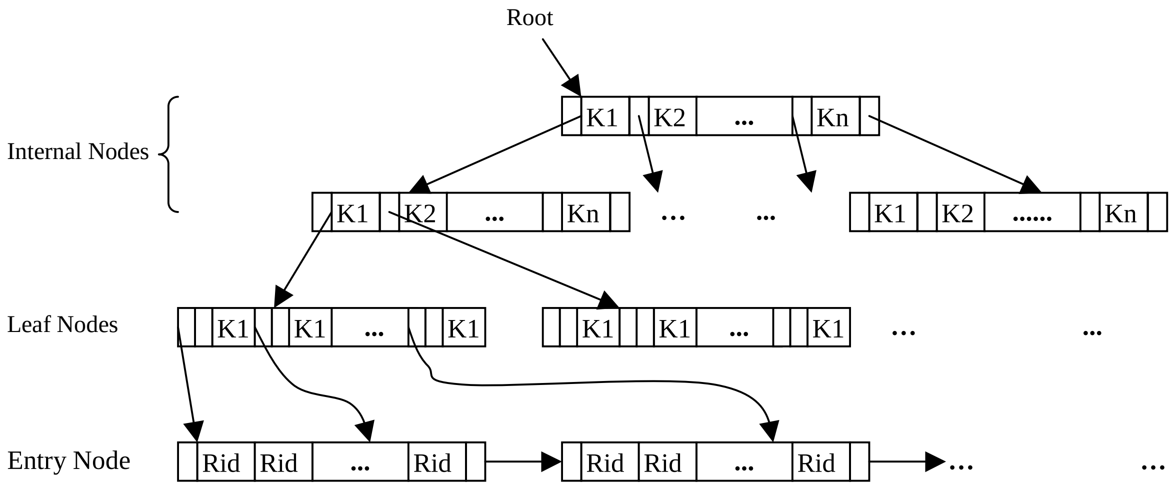


Figure 9 Index structure

queries, e.g. "cd author = "DeWitt" && year = "1999"" . When *Query Executor* gets a (selection) query, it checks every predicate to see if it can do an index-only search; otherwise, it has to scan the storage after using all possible indices.

We also implemented a history pool in *Query Executor* so that users can ask a series of queries to refine query results step by step, as we do in a traditional file system. Users can also use "cd .." or "cd" to step back.

4.5 Other Issues

In our initial design, we wanted to put the storage directly on physical disk pages, and use i-nodes to simulate tree edges in XML data structure. Realizing that it may need a quite large effort to modify Unix file system, we decided to use files to emulate such idea. We choose the size of files to be multiples of that of physical disk page. We can see the impact of the file size on our system through the performance evaluation section.

5. Performance Evaluation

Although we spent most of time on system implementation, we still performed several experiments to evaluate system performance. Because there are no similar systems we can compare with ours, we decided to investigate deep into our system along. Through following experiments, we try to check if the system achieve its goals, compactness and efficiency. Also, we want to know the characteristics of Unix file system in our system, i.e., the overhead of Unix file system introduced into our system.

Experiments are performed on an Intel® Pentium® III 550MHz PC running Linux (Redhat® 6.2). Since we do not implement a buffer management, we simply restrict our system to use only 16M memory. The data set we use is all XML files on the DBLP server.

5.1 Disk Space Cost

First, we discuss the disk space cost of our file system. Fig. 10 shows the storage size and index size with respect to the original data size. We build two indices for the XML data, one on element "author" and one on element "year".

Based on the figure, first we can see that both size of data storage and indices are about proportional to the size of original XML source. Second, the size of data storage is slightly less than that of the original XML data, which means we can provide good query interface (DOM) without requiring more disk space. Finally, the size of indices is fairly much less than data size – index can always help searching at very low

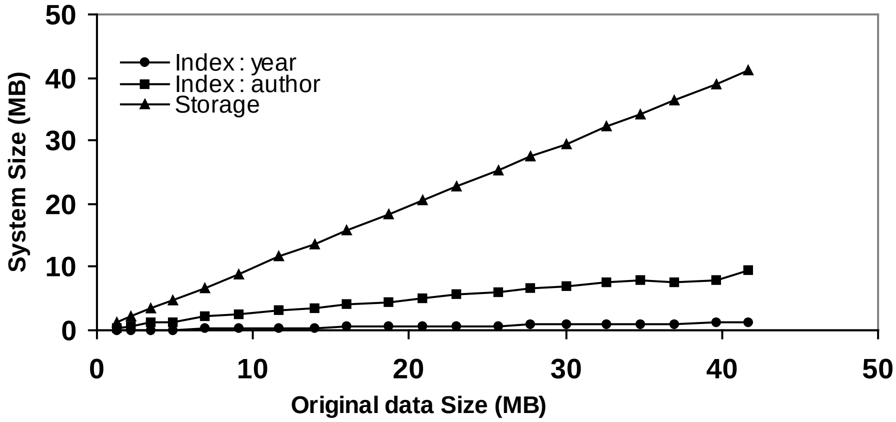


Figure 10 System size vs. Original data size

cost. Note the index on "author" is larger than that on "year" is because each record contains more information about author than year. We may conclude that our system is on the right track to accomplish efficiency on disk usage.

5.2 Initialization Cost

Before querying the XML semantic file system, users have to add XML files into the system. We conducted two experiments to investigate how time is spent during that initialization stage.

First, we analyze the time cost distribution over several subsystems in our file system. Fig. 11 shows how much time the initialization stage take on data sets with different size. Actually, we cannot control the time spent on the parser because the major part of the parser is the IBM XML parser. The total time spent on storage management and indexing is about the same as that on parsing. Considering the parsing cannot be skipped in any querying system, the time we use to build storage and indices are fairly small. Furthermore, both element values and structure are stored in the XML storage system, so original files do not need to be re-parsed when they are queried later.

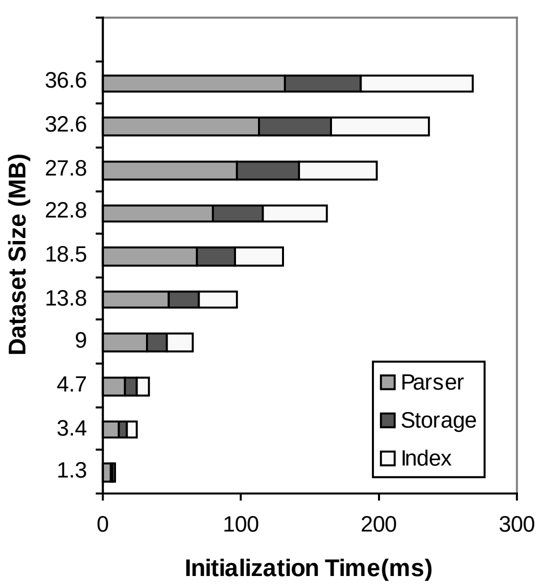


Figure 11 Initialization cost on subsystems

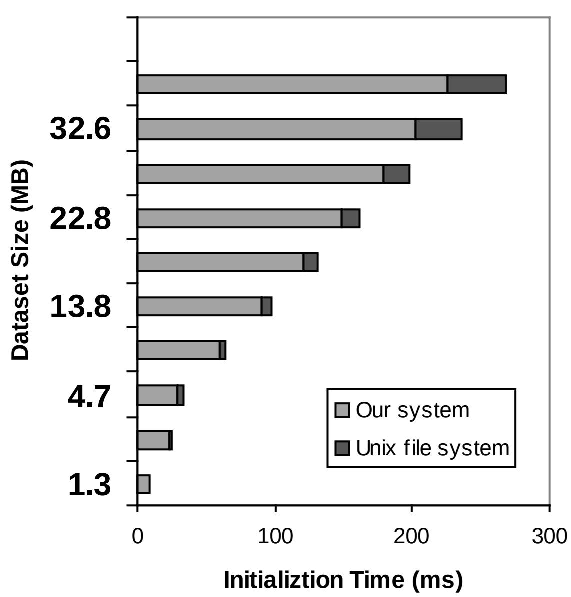


Figure 12 Initialization cost on Unix file system

Next, we analyze the time cost distribute over our system and system calls in Unix file system. As Fig. 12 shows, the Unix file system overhead is small during the initialization stage. The reason is that the storage and index construction is completed in memory and Unix file system is called only when we write the storage and indices to disks.

Both experiments indicate that the system initialization cost is not big and it is acceptable compared to the benefits it provides in later querying.

5.3 Querying Performance

The most important thing in the system is its response time to user queries. We performed selection queries to analyze the relationship between the operation time of our system cost and the cost of Unix file system calls. There are two cases when our system processes user queries – if elements of all predicates in a query have indices, the system will perform index-only search; otherwise, it has to scan the storage subsystem. Fig. 13 shows the response time of an index-only search query on a series of workloads with different size; Fig. 14 shows the case of storage-scan.

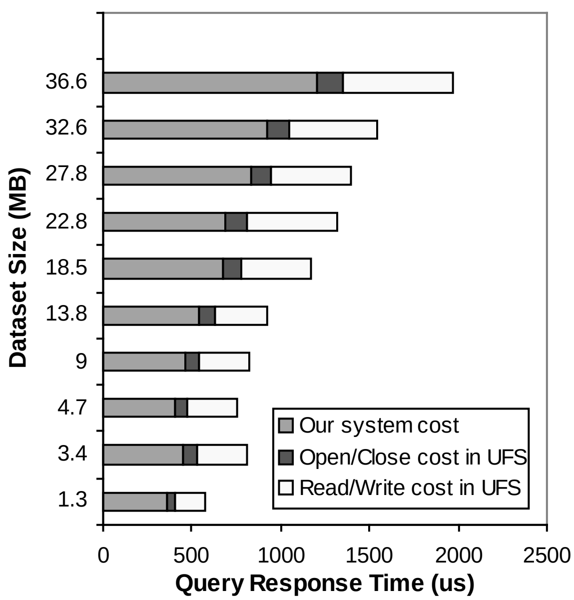


Figure 13 Index-only selection query time

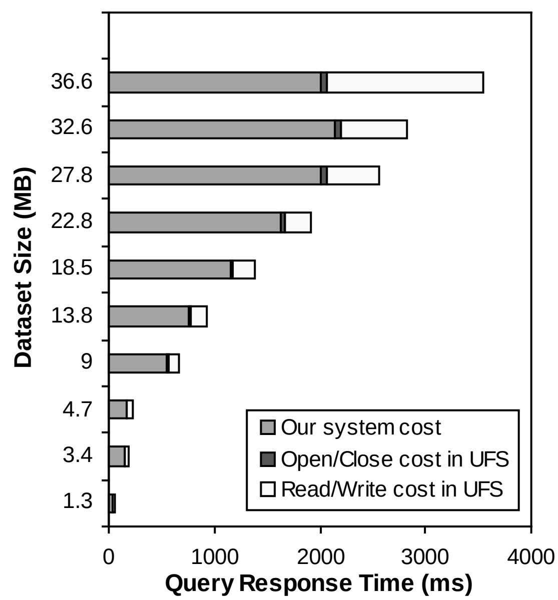


Figure 14 Storage-scan selection query time

First, the results of both cases indicate that indices can decrease query processing time dramatically. Second, the proportion of cost of Unix file system calls is big and stable in index-only search. This is because searching in the B-tree requires several file operations, which are fair expensive with respect to index searching, and the number of levels in B-tree not very sensitive to data size. On the other hand, Unix file system cost on storage-scan becomes larger within the entire cost when dataset size grows because storage-scan is performed linearly, i.e., file system cost is proportional to the number of files in the storage.

5.4 Impact of File Size

Our system is implemented on Unix file system, data in storage and index subsystems are stored in files. We conducted experiments to investigate the impact of file size on the system performance. Like experiments in previous section, Fig. 15 shows the query execution time of an index-only search and Fig. 16 shows a storage-scan query. The data set size of both experiments is 14MB, but the storage and index subsystems use different file size.

Figure 15 indicates that the file size does not affect much on index-only search, but Figure 16 shows storage-scan will take advantage of large file size.

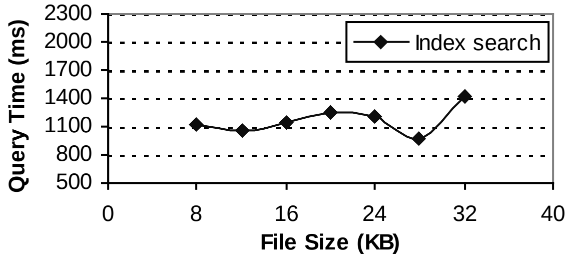


Figure 15 Index-only search query

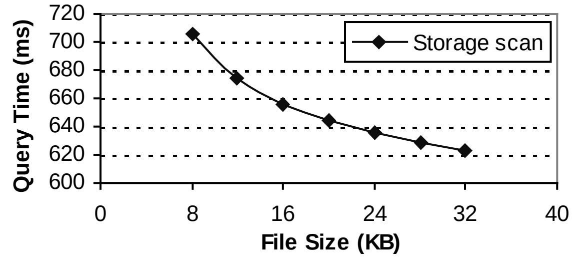


Figure 16 Storage-scan query

6. Conclusion and Future Work

Our goal is to develop an XML storage system that can serve both regular query requests from query (and search) engines and file requests from local file system users (which may include web server). Previous approaches include using traditional file systems and database storage systems. Both approaches have disadvantages – traditional file systems cannot support query engines and search engines directly because they do not have knowledge of XML data structure, while database storage systems are too expensive for a read-only an XML query system, and they cannot help locating XML source files by queries. We propose an XML Semantic File system that has functionalities of both database storage system and semantic file system. It provides DOM interface to database query engine and it can help user find XML files by processing user queries. Preliminary experiments demonstrate that our system has achieved the goal with small disk overhead, and it is also efficient in processing user queries.

In the future, we may implement a specific buffer manager in our system which will help maintaining a memory cache for storage and indices. Also, we can push the storage subsystem down to physical disk page level to avoid file operation cost.

References:

[BBC+98] J. Bosak, T.Bray, D. Connolly, E. Maler, G. Nicol, C.M. Sperberg-McQueen, L. Wood, J. Clark, "W3C XML Specification DTD", http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

[CDN+94] M. Carey, D. DeWitt, J. Naughton, M. Solomon, et. al, "Shoring Up Persistent Applications", Proceedings of the 1994 ACM SIGMOD Conference.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D Suciu, "XML-QL: A Query Language for XML", http://www.w3.org/TR/NOTE-xml-ql.

[DFS99] A. Deutsch, M. Fernandez, D. Suciu, "Storing Semistructured Data with STORED", Proceedings of the ACM SIGMOD Conference, Philadelphia, Pennslyvania, May 1999.

[FK99] D. Florescu, D. Kossman, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database", Rapport de Recherche No. 3680 INRIA, Rocquencourt, France, May 1999.

[GJ00] V.H. Guerrini, D. Jackson, “Bioinformatics and Extended Markup Language (XML).” Online Journal of Bioinformatics, Vol 1(1), 2000. http://www.cpb.uokhsc.edu/ojvr/xmlpaper.html.

[GJSO91] D. Gifford, P. Jouvelot, M. Sheldon, Jr. J. O'Toole, "Semantic File Systems", Proceedings of 13th ACM Symposium on Operating Systems Principles, pp. 16-25, Pacific Grove, California, October 1991.

[GM99] B. Gopal, U. Manber, "Integrating Content-based Access Mechanisms with Hierarchical File Systems", Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, Louisiana, February 1999.

[GMW99] R. Goldman, J. McHugh, J. Widom, "From Semistructured Data to XML: Migrating the Lore Data Model and Query Language", Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99), Philadelphia, Pennsylvania, June 1999.

[Ley] M. Ley, “Computer Science Bibliography”, http://dblp.uni-trier.de/.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, "Lore: A Database Management System for Semistructured Data", SIGMOD record, 26(3):54-66, September 1997.

[NDM+00] J. Naughton, D.J. DeWitt, D. Maire, et. al. “The Niagara Internet Query System”, Computer Sciences Department, University of Wisconsin, Submitted for VLDB 2000.

[PH98] L. Wood, P.L. Hegaret, “Document Object Model (DOM)”, online publication, http://www.c3.org/DOM/.

[QLW98] The Query Language Workshop (QL'98), http://www.w3.org/TandS/QL/QL98/pp.html.

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, J.Thomas, "The Rufus system: Information organization for semistructured data", Proceedings of the International Conference on VLDB, 1993.

[Sto81] M. Stonebraker, “Operating System Support for Database Management”, CACM 24(7), 1981.

[STH+99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[TDCZ00] F. Tian, D. DeWitt, J. Chen, C. Zhang, "The Design and Performance Evaluations of Various XML Storage Strategies", http://www.cs.wisc.edu/niagara/papers/. Submitted to VLDB 2000.

[Wid99] J. Widom, "Data Management for XML - Research Directions", Working Document, http://WWW-DB.Stanford.EDU/lore/pubs, April 1999.