Database-Supported Terabyte-Sized System

Ying Lu Qing Wang Jin Wang
{luy,qing,wjin}@cs.wisc.edu
University of Wisconsin

May 13, 2000

1 Introduction

With the development of storage techniques, high capacity disk is popularly available. The explosion of available information even complicates the organization of the file systems. Terabyte-sized file system will not be uncommon. Finding a wanted file from those huge number of files can be an annoying and difficult task. It is hard, even for a good organization of files, to remember things from a few years back. After a while, not only the content of a piece of information can be forgotten, but also the existence. With such a huge sized file system, convenient and efficient access to vast amounts of information and finding the right information is indeed a big challenge.

The current directories of most OS file systems are tree-structured. Files are identified by file names. Locating of file is through means of pathnames. But the current file systems could not provide sufficient means for users to effectively and efficient access files using various search patterns or information of the content files

Current OS provides limited facilities for users to locate files, such as the find command in UNIX and explorer in Windows environment. Those facilities can only provide simple help to users to locate the files. It is desirable to have some facilities that can help user to retrieve a file in more convenient ways that effectively use the information available.

A file can be identified by both of its contents and its attributes (owner, date of creation, type, length, etc). Although a lot of indexing techniques have been developed in DB and IR, multidimensional indexing and content based retrieval is not an easy task.

The classical IR work have developed indexing and retrieval techniques for text documents. But file system contains various types of files. Content-based retrieval to files is a useful tool for user and both definition and extraction of contents are worthy of study.

File system is probably more dynamic than any database systems. The efficiency of building the index and maintaining it up-to-date (through the interaction with OS) is a challeging issue.

Due to the above motivations, we came up with the idea of a project of developing a preliminary version of the Database-Supported Terabye-sized File System.

The goal of our system is as follows.

- design an indexing structure to provide a multidimensional index to files in a file system or a directory.
- develop efficient algorithms and interface to OS so that the index can be efficiently maintained.

- develop a query interface and search mechanism so that user can find a file by giving certain conditions, such as date of modifiction, types of file, even the keywords of the file contents.

At high level, our system consists of four main components interacting with each other. They are the IR module, the Indexing module, the Query module, and the Update module.

The paper is organized as follows. In section 2, we present some related works. Section 3 gives an overview of our system. Section 4 describes the implementation details of the major components of the system. Evaluations of the system are given in section 5 and conclusions and future work are presented in section 6.

2 Related Work

2.1 Semantic File System(SFS)

SFS [3] provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific transducers. It introduces the concept of a virtual directory to provide compatibility with existing file system protocols. Virtual directory names are interpreted as queries. And it introduces the concept of transducers for extracting semantic data from different file types such as mails, c codes, and etc..

2.2 Hierarchical-Content System(HAC)

HAC [1] is a file system that combines name-based and content-based access to files at the same time. Users can create their own name spaces based on queries, on explicit path names, or on any combination interleaved arbitrarily. All regular file operations are supported in the same way. Query consistency is maintained and adapted to what the user is manually doing. Remote file systems or remote query systems can be integrated by users into their own coherent name spaces.

2.3 Glimpse

GLIMPSE [2] is a simple indexing scheme for personal file systems. Its main idea is to trade space with time by using a two-level indexing structure for finding the right files. Query time is typically slower than with inverted indexes, but they are still fast enough for many applications. GLIMPSE is particularly designed for personal information, which has characteristic of being non-uniform and including many types of documents. Its major draw back is long search time, limited file types(only text documents are supported), but it supports many types of queries, flexible interaction, low overhead, and customization.

3 System Overview

At high level, our system consists of four main component interacting with each other. They are the IR module, the Indexing module, the Query Plan generation module, and the Update module.

Functionality of basic components are :

- IR module

It is the information retrieval module. Its task is to retrieve keywords from different file types based on some IR algorithms. Different file types would require different retrieval algorithms to extract important keywords from files. Eventually, we would like this module able to extract context information so that user could request context queries instead of just keyword matching queries.

There are two main parts in this module

1. File categorization - given a file, map this file to a specific file type.
2. Data extraction - Given a file, extract important keyword or words with semantic meaning.

- Indexing Module

  Database indexing techniques is used in this module for efficient access of requested files of the user. The index trees were build using bulkloading technique in the creation time. During nightly update, the trees were modified using a list of modification data. Bulkloading is also used during updating if the amount of modification exceeds a certain threshold.

- Query Module

  This module parses the query input from the user . It does mapping of the query to tries to find a near-optimal query plan for accessing the indexing structures so that file names (with path name) can be retrieved in an efficient way. Boolean queries consists of different data patterns are allowed. Query results were returned to user in the form of a virtual directory.

- Update Module

  Nightly update is done in this module. Every night, the system is examined and all the deleted files, modified files and created files are found. According to these information, information is retrieved again for these files and the index trees are also modified correspondingly.

4 System Implementation

4.1 Information Retrieval

4.1.1 File Categorization

1. FileInfo Table:
   File categorization is the first step in the IR module. The purpose of it is to organize all the files in the file system with their basic information. In our system, this is implemented by constructing a basic information table – FileInfo Table. Inside this table, each file in the file system will take a row to record its most basic properties.
   The FileInfo Table is defined as a vector of objects which are constructed as follows:

```
class FileInfoEntry {

    String FileName;
    String FilePath;
    String FileType;

}
```

There is also a property of the file which is implicitly defined in this table – File IDentifier(File Number). There is no need to store it in FileInfoEntry object explicitly, the files' order in the table gives it out. After the construction of the FileInfo Table, a one-to-one match between the filename and a file ID is built, thus all the later reference to a file can be represented by a number instead of the long filename(with absolute path) string.

2. Implementation of File Categorization:
   The categorization is implemented by recursively scanning all the branches of the system directory tree. Each time a file is met, we record its name, path and type into a FileInfoEntry object and link it to the end of the table. To get the name and the path of a file is easy. However, for the file type extraction, some extra work needs to be done.

   There includes two steps to extract the file type:

   (a) file extension extraction:
   The extension in a file name is the main mark for what type a file is. For example, the source code of a java program will always end with ".java." But there is the problem of multiple extensions in a file name, such as ".bak" file. In such a case, the extension between the two dots will be the real file type instead of "bak"(star.java.bak would rather be marked as java file for future data extraction and indexing than bak file).

   (b) file type confirmation:
   We also need to consider that sometimes user may make mistakes. For example, they may accidently save a PS file as TXT file. In such a case, if we use file extension as the only basis to determine the file type, there will be trouble when we do the word extraction – we are ready to read a .txt file, but all the characters read in are not recognizable. So what we do in our system is to recheck the correctness of the file extension. Currently, we apply a very simple method for type confirmation – checking the contents of a file. For example, the PS file will always begin with "%%PS"; And the header of an email file will contain lines starting with "Date:", "From:", "To:" and "Subject:" separately. More accurate and powerful method for doc and source code files will be one of our future research works.

4.1.2 Plain Text Word Extraction

Based on the file types provided by the file categorization module, words in such files are extracted – e-mails, Java, c, c++ source codes, txt files, h files, ps files and pdf files. Ps and pdf files are converted to plain text files before extraction. Two Unix commands are used here to help converting – "ps2ascii" and "pdftotext".

Only useful words are extracted from these files. A "uselesswords.dat" file is predefined. It lists a bunch of "useless" words, such as "a, an, the, this, that...". These words appear frequently in files but are unlikely to be queried by users. Digits and symbols are also excluded, only meaningful words are considered to be extracted. However, there are still many unexpected non-word strings which are stored. If we could use a dictionary to check whether a string extracted is a word, the words extracted can be more meaningful. And using a dictionary could also help us to find out a verb's various tenses and a noun's plural form, therefore support more flexible queries. We will consider it as one of our future work.

Words are checked if they have already been accessed in the same file before they are stored. If a word appears more than once in one file, we only store it once.

Finally, all the words extracted are stored in a vector in which each node contains a string of word and the file IDs indicating in which files this word appears. This vector is used for building the index tree using bulk loading.

4.1.3 File Metadata Extraction

Besides plain text words, we also retrieve the information of file metadata. The reason for doing this is that we think users may want to find files that have same owner, or are last modified at the same time, or have a certain size, etc.

The metadata of a file is obtained by calling the stat() system call in Unix. Since it is a C function, and we are using Java, we apply Java Native Method to call this C function in our program. Stat() function provide us a series of metadata, and currently we only take six of them: *owner id, group id, file size, last access time, last modification time, last time of file status change.*

For now, our system accepts queries related to owner id, file size and last modification time. Owner id's are mapped to their corresponding login names in order to provide convenience for users. Last modification times are stored as integers, but we allow users to input a string indicating the month, day, year and time in a fixed format. We can convert it into the number of seconds since Jan 1, 1970 and then do search.

We haven't managed to allow queries on group id, last access time and last time of file status change. For group id, we are not able to find the mapping relation between the id and the group of people it refers. This will be one of our future work.

All these metadata are stored in vectors which are used for building index trees, like the plain text words.

4.1.4 Extraction of Semantic Data

Our system currently extracts the semantic informations of mail header and Java files. Semantic data are useful in the sense of narrowing the search area. It gives user another way of searching the information they would like to obtain in a limited area and with specific meaning to the search pattern they provide to the system. An example would be retriving the subject heading of the mails a particular person send on a particular day.

In the mail header, the sender, receiver, date, sender's email address, receiver's email address, and the content type are extracted. Currently, we only build index trees on the first 5 semantic data. The content type can be used in future work for which the non-text files in the attachment are categorized and data extraction performed on them.

We planned to extract the public and private methods, the class name, the import libraries in the Java source code files, but it is not complete at this time due to the complication of the syntax of the Java language.

4.2 Index Structure

The size of the file system and the amount of information per file in a terabyte-sized file system is huge. How to efficiently indexing the search patters for the files and provide fast access to the indices is crucial for obtaining reasonable performance of user queries. Traditionally, inverted files were used for multikey access of file structures. Inverted files were well suited for accessing files or records on the basis of boolean conditions of attribute values, but they also have drawbacks such as excessive storage space, and excessive number of disk accesses. Also updating were not handled gracefully.

B+-Tree is the indexing structure used throughout the system for storing the search patterns. B+-Tree structure was chosen due to the following considerations. First, it is simple to implement. Second, it adapts gracefully to insertions and deletions which is crucial for the indexing structures of a file system, since updating occurs frequently. Third, if the index trees were to be stored on disks, the number of disk accesses will be approximately equal to the height of the index tree. With a large fan out, the height of the index tree is not large, in fact, it is always less or equal to 4. And if the root of the tree resides in main memory then the number of disk accesses per search pattern can be limited to less than 3.

4.2.1 B+Tree Indices

The system generates one B+Tree index for each of the following patterns.

- Plain text
- Mail header information
  1. sender
  2. receiver
  3. subject
  4. date
  5. sender email
  6. receiver email
- File metadata information
  1. owner
  2. access time
  3. size of the file
  4. modification time
  5. status change time

Each index is used to store all the search patterns in different files/semantic categories.

4.2.2 Creation and Maintenance

During the initial creation of each B+Tree index, bulk loading was used to creat the B+Tree bottom-up in one go. During nightly updating, the trees are updated according to the modification data from the extraction module. In order to improve the performance of B+Tree updating, we have used a heuristic to speed up the updating of the tree. That is , if the amount of modification exceeds a certain threshold, then bulk loading is used to rebuild the tree from the leave level up instead of doing deletions and insertions into the tree one by one. This saves time when the amount of modifcation is big as searching down the tree many times would generate more disk accesses than building the tree again. Also deletion of the non-existence files were done at the leaf level instead of searching from top-to-bottom so that repeated iterations over the path of the same tree is avoided.

4.3 Query

4.3.1 Query Input

The interface of our database-supported file system is based upon queries that describe user requirements. A query is a description of desired fields and the corresponding values. We define two types of queries in our system: Simple Queries and Combination Queries.

1. Simple Queries:
   Simple queries are the basis of the combination queries. It only describes one field and one value. The syntax of simple query is like " /field_name:/value ". To get the desired flexibility and query power, our system support a very large set of fields to be queried on.

   (a) query on type: "/type:/some_value" will get files of a certain type. The following example will find all the java files in the file system

   ```
   >gfs /type:/java
   ```

   (b) query on metadata: the permitted queries include "/owner:/some_value", "/size:/some_num", "/modifyT:/some_time", "/accessT:/some_time", "/statusT:/some_time". The "owner" field can be queried without the value.

   ```
   >gfs /owner:
   ```

   will find all the file owners in the file system

   ```
   >gfs /owner:/Smith
   ```

   will find all the files owned by Smith.

   (c) query on semantic data: semantic data in some certain file types have also been extracted, indexed and ready to be queried. For example, we can query email files on semantic data field "sender", "receiver", "date", and "subject". Suppose we want to find all the email files sent by John@cs.wisc.edu, we will do:

   ```
   >gfs /sender:/john@cs.wisc.edu
   ```

   (d) query on text words: it means to find all the files containing certain word(s).

   ```
   >gfs /text:/database
   ```

   will find all the files containing word "database"

   On the basis of this most simplified query format, we make some improvements to provide the user more convenience and flexibility. This is accompolished by doing some combination queries in the simple query format. For example, to find all the files containing both the word "database" and the word "network", we only need to write:

   ```
   >gfs /text:/database;network
   ```

   Another example is to find all the emails sent by john@cs.wisc.edu on the day 11/11/1999:

   ```
   >gfs /mail:/sender=john@cs.wisc.edu; date=11/11/1999
   ```

2. Combination queries:
   They are the boolean combinations of simple queries. User can use both the AND operator '&' and the OR operator '|' to describe his/her multiple requirements.
   Example: to find all the C files owned by Smith and the Java files containing public method named "buildTable":

   ```
   >gfs (/owner:/Smith & /type:/C++) | (/type:/java & /public:/buildTable)
   ```

4.3.2 Query Processing

Query Processing include three phases: Query Parsing, Query Plan Generation, and Query execution.

- Query Parsing : it will extract simple queries and the operators from the input stream of a query string.
- Query Plan Generation : A query plan structured as a tree would be built. There can be many possible query plans, but which one is selected is not important as long as it is correct. The fact is, no matter what "shape" a tree is, all the simple queries need and will be evaluated all the same, no optimization can be done. However, since inverted index is pretty fast for simple query lookup, while at the same time the "&"(and) and "|"(or) operations are also quick, we will not worry about the optimization here.
- Query Execution : After a tree is constructed, we can evaluate a query from the leaf nodes(simple queries) and push the results bottoms up.

4.3.3 Query Output

Queries are performed in GFS through the use of virtual directories to describe a desired part of file system contents. A virtual directory is computed ahead when indices are constructed and stored in the stable storage(like disk,etc). From the point of view of a user, a virtual directory has no difference from the ordinary directories in Unix. However, each file under a virtual directory only provides a symbolic link to the real file and each subdirectory is also a virtual directory.

For the purpose of further reference of the user, the files and subdirectories of all the related virtual directories that have been queried (the query results)are cached in memory. It is named as "the history tree" of associative queries. The associative queries are a series of queries which can only contain one "gfs .." command and several "ls" and "cd" commands. However, when a totally different query is presented by the user(the new associative query set will start with another "gfs .." command), the cache will be flushed and rebuilt as the subqueries of the new query go on.

Some Unix commands are supported for the files and subdirectories in a virtual directory (query result). Currently, our system supports file copy(cp), file rename(mv) and file deletion(rm) operations. For example, supposing user have already done the following query to find all the files containing the word "database" in the current directory:

```
>gfs /text:/database
  db1.ps        db2.ps        db3.ps        db4.txt
```

then he/she wants to copy db1.ps to his/her real working directory in Unix before entering GFS:

```
>cp db1.ps ./db1.ps
```

"./" here is used to point out that the path name after it will be the ordinary path name in Unix file system. A file path starts with "/" has the same effect but the file will be searched from the root directory. Using such kind of specifications enables us to do some normal Unix operations in the GFS environment.

"Cat [filename]" can be used to display the contents of the files under the virtual directory:

```
>cat db4.txt
```

"ls" and "cd" operations are used in our system for the operations on virtual directories. When "ls" and "cd" are used to list the contents of or change the current directory to a subdirectory which does not have its contents cached in "the history tree" yet, a new subquery will be done and the result of this subquery will be cached. For example: virtual directory "gfs /owner:" corresponds to all the file owners in GFS:

```
>gfs /owner:
  qing/          wjin/          luy/
```

then user can list all the files owned by wjin, change directory to wjin/ and list the files again:

```
>ls wjin
  star.java     machine.C      mail       paper.txt
>cd wjin
>ls
  star.java     machine.C      mail       paper.txt
```

"ls" can also be used to do operations on real Unix directories, but "cd' command is a special case here. It is because we use Java process to execute all the normal Unix commands, and the status of a process will be changed if "cd [dir]" is executed inside the process, thus it is not allowed to do normal "changing directory" operation in our system. "cd [dir]" is used only for switching between the virtual directories. "CD" command without any operand is reserved for exiting from the system and returning to previous working directory of the user in Unix environment.

4.4 Update

The updating is done at every night. Basically, we determine which files are modified and which are not by comparing the last modification times.

We need two lists of files for comparison. One old list and one new list. The old list is the one which was stored at the previous night, containing the information of file ID, file path and last modification time. The new list is obtained each night by scanning the whole file system, containing the same file information.

For each file in the new list, we go through the old list. If it is found and the modification time is not changed, then just skip it; if the modification time is changed, store this file into a "modifiedFiles" vector, at the same time, mark the file as found. If it is not found, then it must be a newly created file, so store it in a "createdFiles" vector. After all the files in the new list are examined, we go through the old list again to find out those unmarked files – the deleted files, and store them into a "deletedFiles" vector. Thus, these three file lists – the createdFiles, modifiedFiles and deletedFiles are obtained and provided to each module of the system for their updatings.

We are not doing real-time updating because of two reasons. First, doing real-time updating requires us to trace the operations of users. In order to do this, we wil need to modify the kernel of Unix to put a hook in the system calls such as open(), read(), write(), create(), etc.. This would be fairly difficult for us. And second, since our database supports a terabyte-sized file system, the updating process can be slow. We are not sure whether doing real-time updating will affect the performance. So we chose to do update nightly. Of course, this causes consistency problems. Currently, we are not able to provide high consistency. This could be included in the future work.

4.4.1 Nightly Updating of FileInfo Table

After collecting all the information about the modification of the file system, the FileInfo Table will be updated. Since the table only contains the basic properties of files, the updating work is quite simple. It includes two parts:

1.) delete the no-longer-existing files:
since the order of a file in the FileInfo Table implies its File ID, we can not simply remove a non-existing file from the table. So a hole will be left at the corresponding position and only the contents of that "FileInfoEntry" object are erased. To use these holes later(reuse a file ID), we record the position information in a "free list table".

2.) insert the newly created files:
Each time before the new file information is inserted, the system will check the "free list table" to find a position and insert it into the FileInfo Table.

4.4.2 Update of Words and Metadata

During nightly updating, the words and metadata in modified files and newly created files are extracted again. Each word, along with its file number, is inserted into the index tree individually after extracted.

5 Performance Analysis

Because of the incompletion of the system, testing of big sized file system was not performed. But from the individual testing of the components, we have discovered that though B+Tree is a nice and efficient structure for indexing data. Big boolean queries would be much well supported if we could have a multikeyed indexing structure to narrow the scope of searching right from start. Instead, the B+Tree indexing only provides large subsets of partial results and intersections were needed in order to get the right results. With big booleans queries, this would degrade the performance of the query processing very much.

6 Conclusion

In this project, we have implemented the four basic components of a Database-Supported Terabyte-sized File System. During the course of the project, we have build the preliminary version of the four components of the system. Unfortunately, the integration of the whole system was not done successfully due to lack of time, due to the fact that there is a bug in the system that we could not figure out the cause of it. The four different components were indivdually tested, and are successful to some extent.

Some of the future work that are interesting and useful are :

- semantic data extraction on various file types
- stemming, error allowance
- using multi-key indexing structure for boolean queries
- consistency improvement for file modifcations

References

1. B. Gopal "Integrating Content-Based Access Mechanisms with Hierarchical File Systems".

2.Manber-etal " GLIMPSE: A Tool to Search Through Entire File Systems".

3. Gifford-etal " Semantic File Systems".