INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

CASH: A Category - based File Management System

By

Lan Yang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

April 27, 2005

Abstract

File management remains a hot topic because file creation, retrieval and deletion are such basic aspects for all computer users in their daily life. Here, we are dedicated to develop a file management system, CASH, to effectively and efficiently organize, manage, maintain, and search tremendous amount of electronic files based on latest database technologies, so users can easily find the exact files they seek. CASH is a category-based shell, which is developed from TCSH. In this system, users manage their data by using files and categories, instead of files and directories in conventional file systems. Unlike directory-based file systems, CASH allows a file to be classified to more than one category, and a category could also have multiple parent categories. These features make it much more flexible to place or locate a file. CASH develops a set of commands, and defines several rules to create, update, and delete a category hierarchy, and the categorization of files. Moreover, it extends some existing Unix commands, whose execution will affect the information of some files, to update the database immediately.

Acknowledgements

First, I would like to sincerely thank Dr. Mengchi Liu, who is my supervisor, for his guidance, encouragement, and patience. I am so lucky to have such a wonderful and nice advisor. Without his direction and insights, this thesis would not have become a reality.

Special thanks to my parents and elder sister. We always feel so close, though they live at the other end of the earth. Their encouragement and well-being are the biggest supports for me.

I am very grateful to my friend, Johan, to read and edit my thesis. I also would like to thank Yang Cao, WanXia Wei for all of their suggestions and discussions with me. And, I will thank QunXiao Wang and ZhiHong Li for sharing their useful information with me. My final acknowledgement will go to all of my friends, both in Ottawa and in Vancouver, for sharing my tears and happiness in these years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Computers are now a necessity in our daily life. To many people, a computer is a world of files. File searching is such a basic aspect of working with computers while we hardly noticed its existence before. But now, with the development of the Internet, we get easier and faster access to much more information to what we have been accustomed. On the other hand, hard disks with lower prices but with more storage capacity, encourage users to keep all files in which they are interested more or less. Therefore, file searching is no longer a problem that we can ignore. A recent file system study [13] showed that the challenge has shifted from deciding what to save or keep to finding specific information when it is desired. To meet this challenge, users need better tools for personal file organization and searches.

Nowadays, many very powerful document management systems (DM) have emerged, such as Documentum, FileNet, doQuments, and Hummingbird. Typically, they will have imaging and workflow, archiving, previewing and Web interface facilities. Besides resolving collaboration, consistency and even security of the data through the web, one of their key issues is how to organize corporate documents into categories of interest. However,

most of these systems are mainly developed for enterprises to quickly find, share and access all document types, both scanned files and electronic computer-generated files. To many personal computer users, such powerful functions are somewhat of a waste to them, and they would rather choose a DM system with a lower price, which has less but enough functions for them. On the other hand, for some users, especially professional computer users, documents are not the only information with which they are concerned. Therefore, they need a more general tool to manage all of their files.

Regardless of what operating systems, most users have to organize files by placing them in hierarchies of folders or directories, and more generally, such hierarchies pervade files and information storage systems [2]. In such hierarchical structures, a file can appear in only one location at a time, and users locate a file mainly by its path and name. **File Placing** and **File Searching** are two main aspects when users are working with files. Placing a file in the right directory helps users to find it much more easily. Users normally use or create a directory in a hierarchy with a name that could remind them of what files are under it. But for users with a lot of files and directories, finding or creating an appropriate directory in hierarchies for a file is not as simple as it was thought to be. First, there may be a directory with the same name somewhere, but, it contains totally different kinds of files. Second, there may be a directory with a different name somewhere, but it contains similar information. Third, a file could involve several knowledge fields, a directory name usually can show only one of them. Fourth, for a computer with a lot of files and folders, it is very difficult to locate the target file if users don't remember the path or filename. Moreover, if users don't know how such hierarchies are structured, then it would be a painful process to get anything useful from them. All of these problems motivate us to investigate new models for user-oriented file management systems. This is the exact point on which we develop CASH. Moreover, users are often frustrated with long path entries, especially when desired files are at too low a level in the tree structure. This is the other problem that we solved in our system.

CASH is based on TCSH, which has been a very popular shell that Unix/Linux users are using nowadays. CASH is a category-based management system. In CASH, each file can only have one physical location, but it can have multiple logical locations, called categories, in the category hierarchy. This feature allows users to reach a file by multiple paths, which increases the possibility of destination files being found. On the other hand, users always know what information is in a file when they save it. And, they will always know the topic or clue of a desired file, because this is the purpose for the search. It requires little effort to place and to find their files in the right location. To work with CASH, users hardly need to worry about the physical locations of their files. Given proper categorization, the system will locate them automatically.

1.2 Objectives

The main goal of the work presented in this thesis is to help Unix/Linux-based computer users to work with their files, including file grouping, finding and retrieving, based on the Unix file system. In other words, we will try to free users from the burden of memorizing or recalling where their files are, what their names should be, and also from the puzzling tasks of where to place their files, or, which directory is best suited for them. Our approach is a category-based file management system.

To achieve this goal, we tried to improve our system by implementing the following three aspects:

Firstly, CASH allows users to create a more logical and accurate category hierarchy, where they can easily find an appropriate place for their files. We developed a set of commands, based on TCSH, to create a new structure for Unix file systems. Compared to the tree structure in traditional file systems, our structure is not a tree but an acyclic graph. Users require little effort to create or modify their own hierarchical category structures for their files. In a category hierarchy, any category can have more than one parent category,

and the knowledge field of a child category is a specialization of those of its parent categories. Given a more accurate category, users can easily decide where to place their files in this logical category hierarchy, and then worry less about where to physically store them on their disks.

Secondly, in order to help users to work with their category hierarchies more easily, we tried to avoid the absolute or relative path entries as in the tree structure. By giving a simple category name without a path, users can reach it directly. If there are several candidates, the system allows users to make the choice.

Finally, CASH extends or improves the functions of some existing Unix/Linux commands with which users are familiar. Besides keeping all original functions for these commands, we developed some functions to make the database remain consistent with the results from these commands. In addition, since we have applied a database tool in realizing our approach, we tried to use it to improve on the performance of some existing Unix/Linux commands, such as `find`.

1.3 Outline of Thesis

The thesis is organized as follows:

- Chapter 2 introduces background knowledge and explains the basis for our design.
- Chapter 3 gives an overview of related work; it especially introduces an approach for Unix file management: Semantic File Systems.
- Chapter 4 presents our approach to file management in UNIX: CASH. It explains why and how we designed this new system.
- Chapter 5 discusses implementation issues. It introduces the basic rules of applying this system, and illustrates our approach with a running example that is used throughout this chapter.

- Chapter 6 concludes this thesis.
- Chapter 7 gives more details for each command that we developed in CASH.

Chapter 2

Background

2.1 File Systems

In a computer, a file system is a method of naming, storing and organizing computer files to make them easily found and accessed. It involves where to store files (e.g. hard disk, CD-ROM), how to store them (eg. executable or not), and also includes a format for specifying the path to a file through the structure of directories. So formally speaking, a file system is a set of rules that are implemented for the storage, hierarchical organization, manipulation, navigation, and retrieval of files [10].

Every operating system, such as DOS, Windows, OS/2, Macintosh, and Unix/Linux-based all have their own file systems, which help them to organize and to keep track of files. The basic similarity of these file systems is that they all organize files into a hierarchical tree structure, and the internal nodes of the hierarchy are called directories or folders while the leaves are files. This determines that a file within them can only belong to one location. Figure 2.1 shows a typical directory-based file system.

Although each operating system provides its own file management system, we can develop and apply other file management systems, which provide more features to help us handle files more easily. This is what has been done in this thesis.

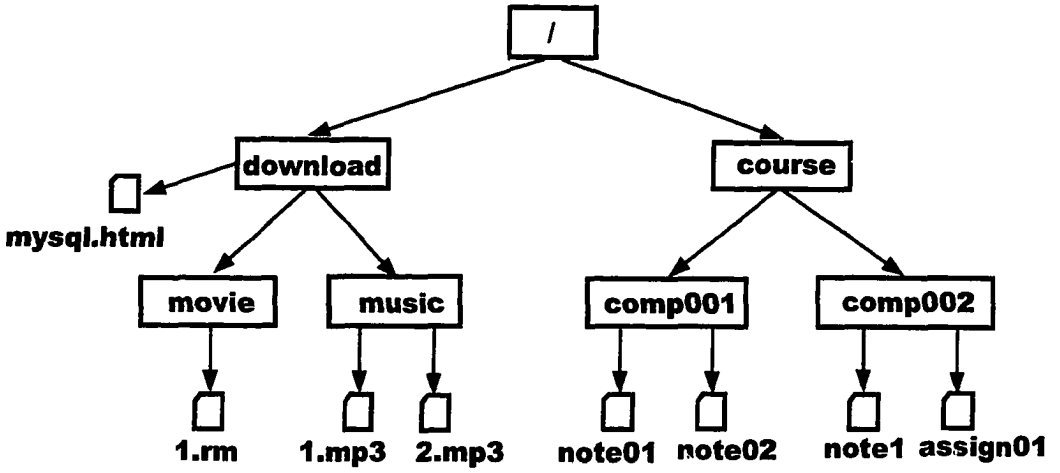

Figure 2.1: Tree structure

2.1.1 File Placing

Directories or folders are used to group related files, and their names are used to remind users of what files are under them. Any file can only belong to one directory in the traditional hierarchical file system, even though they may have more than one feature with which users may be concerned, and be relevant to multiple directories.

To mitigate this problem, most file systems allow files to logically appear in multiple folders by providing some *file pointer* mechanisms such as aliases or links. However, these can introduce extra problems while solving existing problems. By creating confusing distinctions among file names, file locations and file identities, users are required to understand whether they are acting on a link or a target file before executing commands. Otherwise, they could lose or modify their files unexpectedly. These features make it difficult to manage files. Moreover, these distinctions are managed differently in each type of file system; although they are intended to address similar problems, and mechanisms such as aliases in MacOS, shortcuts in Windows, and hard and symbolic links in Unix/Linux all have different semantics.

2.1.2 File Searching

Generally speaking, there are two basic strategies for searching files in conventional file systems: location-based searching, text-based searching.

Location-based Searching In location-based searching, "users take a guess at the folder where they think a file may be located, go to that location, and then browse the list of files or an array of icons in the location until they find the file that they are looking for. The process is iterated as needed" [1]. Users would prefer to be able to go to the right location on their first try. Given a list of potential files in the target location, they often identify the desired file by date, name or some other characteristics that they think it should have.

Text-based Searching Most operating systems provide text-based search utilities, such as `find` and `grep` in Unix/Linux. These utilities usually rely on keywords, file names or other physical characteristics of files (i.e. file type, size, etc.). Unfortunately all of these features are illogical and easy for users to forget. That is why users do not use the text-based search frequently.

Nardi's research [1] reported that users overwhelmingly preferred a location-based search, and, they also preferred browsing lists of files to recognize the desired one rather than trying to remember exact file names. On the other hand, users seem to prefer to actively search for a file that they have previously placed in a particular location rather than sitting there waiting for the computer to return a list of files that may or may not be relevant, because this makes them feel that they control the search. But to retrieve files on remote machines, users may more willingly use text-based search techniques, because the space being searched is beyond their control, and they have no idea where the desired files could be, or, what they could be called.

However, users still have to waste a lot of time with file searching, even though they apply both of the above searching strategies. In location-based searching, users usually

cannot find a file on their first try, because it can be reached from only one path. In the text-based searching, users can not recall the non-meaningful clue for searching in most cases. CASH provides a category hierarchy, which helps to prevent users from browsing too many potential directories and files by providing multiple file paths, and also provides an approach for users to locate files according to their logical features or topics instead of physical features with traditional text-based searching.

2.2 Yahoo Desktop and Google Desktop

Yahoo is a search directory, and Google is a search engine. Yahoo store some basic details about a web page in the database, such as categories, page description and so on. So it mainly rely on category search and keyword search, which only is determined by the basic information held in the database. Google holds a copy of each page on the internet [19]. Google search depends on the keyword and relevancy analysis.

Yahoo Desktop Search tool allows users to search through their PC hard drives for files and folders in the same way that the Yahoo internet search engine searches the web. It is able to search over 200 different file types, such as PDFs and music files. Besides it has included a sorting facility so that users can perfect their results by various attributes, such as date and file size, Yahoo's tool also provides a preview window to help users to check the content before they click on each search result. Google has released a new Google Desktop Search tool, which allows people to scan their computers for information in the same way they use Google to search the web. After it creates an index of all a user's searchable information and stores it on his/her computer, Desktop Search will update continually for most file types; Google desktop search provides the full text of users' email, files, viewed web pages, and chats, but its file search is limited to files on a user's primary hard drive.

The both search tools require Windows XP or Window 200 Sp 3+ [17, 18].

2.3 Locate in Unix

To search a file, the `find` utility has to navigate the given directories with all of their sub-directories one by one. It is very time-consuming when there are a user works with a great number of files. So users could use `updatedb` to save all their files to the database, then they can use `locate` to retrieve a desired file faster. In this database, only file names and their paths are stored, so users can only search a file or a directory by providing its name.

2.4 Links in Unix/Linux

Compared to DOS, Unix/Linux has a more powerful file management system. For example, links in it may cross the hierarchy, and allow one or more filenames to refer to the same physical file. So, it is possible for users to find a file by using different paths.

There are two different links in Unix/Linux: **Hard Link** and **Symbolic Link.**

HARD LINK: When a user creates a file, there is one link to it. A hard link is a pointer to an existing file. In other words, hard links create a number of different names that all refer to the same contents. Commands executed upon any of these different names will then operate upon the same file contents.

The syntax to make a hard link to an existing file is as follows:

`ln` *original-name new-name*

This will create a new item in the working directory, and *new-name* could be an additional name that users can use to refer to the file with *original-name.* The user can use the standard Unix/Linux `rm` command to delete a link. After a link has been deleted, the file contents will still exist as long as there is at least one remaining link, and the user can still access it through these remaining links. Moreover, regular users can only create a hard link to files, except the super user can create it for a directory [7].

SYMBOLIC LINK: A symbolic link, also called a soft link, was developed because of the limitations of hard links. It is a special kind of file that points to another file, much like a ***shortcut*** in Windows or an ***alias*** in Macintosh. Unlike a hard link, a symbolic link does not contain the data in the target file. It simply points to another entry somewhere in the file system. This difference gives symbolic links certain qualities that hard links do not have. For example, a symbolic link can work between different file systems. In addition, it can be created for a directory.

The syntax to make a symbolic link is as follows:

`ln -s` *original-name new-name*

After a *new-name* is created, the user can perform an operation on it or execute it, just as he does when working with *original-name*. When *new-name* is deleted or moved, *original-name* and its remaining symbolic links will not be affected. But if the *original-name* is deleted, renamed or moved to another location, all of its symbolic files have to be deleted or moved; otherwise, they will not function properly. In other words, users should manually manage and maintain symbolic links.

2.5 Inode

A Unix file system consists of two parts: a set of data blocks where the contents of files and directories are kept, and an index to these blocks. Each entry in the file system has an entry in the index. Each entry in the index is called an *inode*. The inode is a table which contains the file/directory's attributes along with pointers to disk blocks containing data. The attributes include the type and the size of contents, a set of permissions bits, owner and group information, the number of links, and so on.

All hard links for a specific file have the same inode information, this feature allows hard links to only work within a file system. Given several file names, it is easy for users to judge which files are linked to one another by using command "`ls -i`" to show their

inode numbers. On the other hand, we can determine to which file a symbolic link is linked by using command:`ls -l`. But if given a file, which may have hard links or symbolic links, we can only know how many hard links it has, but don't know if it has symbolic links. Furthermore, no command can explicitly tell users or where its links are located.

Consequently, links introduce some new problems while improving the flexibility of file access. They make it difficult to keep track of files. Also, users have to take time to maintain links carefully and always make sure that if they act on a link or a target file upon execution. That is why sometime they delete or modify some useful files by mistake.

2.6 SHELL

Shell is both a command language and a programming language that provides an interface to the Unix/Linux operating system. Its features include parameter passing, variables and string substitution, etc. Its programming language include constructs, such as `'while'`, `'if then else'`, `'case'` and `'for'` to control the flow. Two-way communication is possible between the shell and commands. String-valued parameters, such as file/directory names and options, may be passed to a command. A code returned by commands may be used to determine control-flow, and the standard output from a command may be used as shell input for another command.

Shell executes a program when users give it a command in response to its prompt. The format for a command line is:

```
command  [option] [arg1] [arg2]  ... [argn]
```

Commands. Some useful Unix/Linux commands consist only of the name of the command. Commands that require arguments give a short error message when users use them without enough arguments.

Options. "An *option* is a special argument that modifies the effects of a command [7]". For any command, users can specify more than one possible option, so that they can execute a command in several different ways. Options have specific definitions and are interpreted by the program that the command calls.

Arguments. An *argument* is a filename, string of text, number, or any other object on which a command acts [7]. For example, arguments for a `rm` command are names of the files/directories that a user wants to delete.

For example, the following command means to remove directory `sample` with all subdirectories encountered.

```
rm  -r sample
```

2.6.1 Introduction to Some Popular Shells

Unix/Linux has several standalone command-interpreting shells, including Bourne Shell, C Shell, Korn Shell, Turbo C Shell, etc. Users can switch back and forth between different shells, but they can only use one of them at one time. A *shell script* is a file that contains commands to be executed by the shell. The commands in a shell script can be commands that users can enter in response to a shell prompt. Different Shells have some differences in their programming languages. Many users prefer the Bourne Shell's programming language to C Shell's, and it is the basis of the Korn Shell programming language [7].

The C shell originated on Berkeley Unix, and it is a shell with C language-like syntax. C Shell has more versatile variables than those of the Bourne Shell. Users can customize C Shell variables to make it more tolerant to mistakes and easier to use. The alias mechanism makes it easy to rename the name of an existing command to a new name with which users are accustomed. In addition, the history mechanism allows users to edit and rerun previous command lines.

Korn Shell provides both a programming language like the Bourne Shell and many of

the interactive features of the C Shell. It implements the hallmark features of the C Shell that make it popular as an interactive command interpreter, just as mentioned above. It also develops several user's interface features, including command-line editing. Scripts written for the Bourne Shell can run under Korn Shell without modification. Also, scripts run more quickly under the Korn Shell than under the Bourne Shell because more commands are built in [7].

Turbo C Shell is a popular shell nowadays, and it is an enhanced but completely compatible version of the Berkeley Unix C shell. It is a command language interpreter used both as an interactive login shell and a shell script command processor. It includes a command-line editor, programmable word completion, spelling correction, a the history mechanism, job control and a C-like syntax.

From the developing process of Shell, we can see that experts tried several ways to reduce or simplify command-editing, such as programmable word completion, and history mechanism. However, in all existing file systems, files or directories (*folders*) can have the same name if and only if they are not in the same directory. To specify any file, the default mechanism is used to provide its absolute or relative path in the tree structure. It is a tedious and time-consuming job. Therefore, in our design, we tried to solve this problem with the help of a database, since command-editing and file retrieval are such basic and important aspects when most Unix/Linux users use Shell.

2.6.2 Some Existing Useful Environment Variables in Shell

Shells provide some useful features to make command-line input simpler and friendlier. Users simply need to customize the environment in their own system to take advantage of these features. The most useful ones are PATH and CDPATH.

PATH: In Unix/Linux, DOS, and Windows, users are allowed to set a list of paths in their own PATH variables. To execute a command, the system will search this command in

these paths automatically, besides the current working directory. If we don't have this environment variable set correctly, Unix/Linux won't search other paths to find the command that we want. Typically, users can customize PATH variables by adding frequently used paths to them. The paths in this list are searched sequentially, so if two directories have executable programs or scripts with the same names, the first one encountered is selected.

CDPATH: In many conventional operating systems, we use *cd* command to change the current directory, and we also need to provide an absolute or relative path of a directory in order to change it. To avoid this tedious job, in Unix/Linux we can set a group of frequently accessed directories to our CDPATH variable. Then we don't have to type anything but a simple directory name, which should be a subdirectory of any of those listed in CDPATH, then we can reach the directory that we desire. For example if we add */usr/local* to the CDPATH, then we can *cd* to the directory called */usr/local/mysql* by simply executing the command: *cd mysql.*

However, all these variables still do not solve the problem of homonym. For example, I have another directory path */tool/mysql*, and I also add */tool* behind */usr/local* to my CDPATH list. Then, if I type *cd mysql* under some other categories except *tool*, I can never reach */tool/mysql*, as */usr/local* appears before */tool* in the CDPATH list. The same problem exists with PATH. Thus, PATH and CDPATH only partially alleviate the problem rather than solve it completely.

2.7 MYSQL

MySQL is now very popular open source database tool currently available. It is a well-respected product that is more than capable of commercial operation. In fact, the entire Google search engine is built upon MySQL technology [11]! MySQL is a relational database management system. It is based on a client/server model, and a user's interface

to it is the MySQL client program. Mysql has the following advantages compared to other database tools:

- ***Fast***: According to the Mysql Benchmarks, it is faster than AdaBas D, PostgreSQL, Solid and Empress on Linux systems. It is also faster than Access2000, Informix. MS-SQL, Sybase, and Oracle8.0.3, on Windows platforms [9, 12].

- ***Easy***: It is easy to download and install Mysql. The installation procedure is straightforward and well documented, and users have fewer problems installing it.

- ***Free***: Mysql is licensed under the GNU General Public License (GPL), which allows users to use it for free.

- ***Scalable***: There is no limit to the number of tables, databases, or users. Users can use it for small personal databases up to big heavy-traffic industrial databases.

- ***Secured***: Since the Mysql server is fully networked, and can be accessed from anywhere on the Internet, Mysql has full access control to allow a user to can determine the level of access that they'd like other users to have.

- ***Portable***: Mysql is available on so many platforms; therefore, it's easy for users to develop an application or work with a database that can be moved to a different platform.

In CASH, we use Mysql to store information of files and categories. Mysql not only makes our system work more efficiently, but also helps us to develop it more easily.

Chapter 3

Related works

This chapter discusses related work. Section 3.1 outlines early research in this field: semantic file systems. Section 3.2 Introduces file categorization mechanism currently available in the file management field.

3.1 Semantic File Systems (SFS)

"A semantic file system (SFS) is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers*" [6]. By introducing the concept of a virtual directory, it is compatible with existing file system protocols.

SFS is a halfway between a traditional hierarchical file system and a traditional database.

Associative Access. Associative access is designed to make it easier for users to share information by helping them to discover and locate programs, documents, and other relevant objects [6]. For example, a file can be located according to several attributes, such as its author, words contained, type and title. These attributes are generated by *transducer*. Semantic file systems provide a friendly user interface, and also an application programming

interface to its associative access facilities.

Virtual Directory. Semantic file systems integrate associative access into a tree structured file system through the concept of a *virtual directory*. Virtual directory names are used as conditions for queries and thus provide flexible associative access to files and directories. In the following example, command line 1 locates all files that export the procedure `sorted`, and then further narrow the results to those files which have the extension `c`.

```
1. % cd /sfs/exports:/sorted
2. % ls -F
   bubble.c@   bubble.o@
3. % cd ext:/c
4. % ls -F
   bubble.c@
```

Transducer. A transducer is a program. It works as a filter, and assigns attributes to a given input file. There are different transducers for each file-type for which the user desires a classification. Figure 3.1 shows a sample transducer input and output. For each file in the semantic file system, the file type must be identified, and then an appropriate transducer can run on the file. Different file types or transducers can have some attributes in common, such as `author:` in 3.1, while they can have their own special attributes. For example, `exports:` may be output from only one specific transducer: `Object Transducer`. The information from these transducers is passed to an indexer database, which does attribute lookups when a search is requested. Through the use of specialized transducers, a semantic file system can understand different file types, such as documents, programs, mail, and other files contained by the system.

Automatic Indexing. The automatic indexing of files and directories is called *semantic*, which means that user programmable *transducers* use information about the semantics of

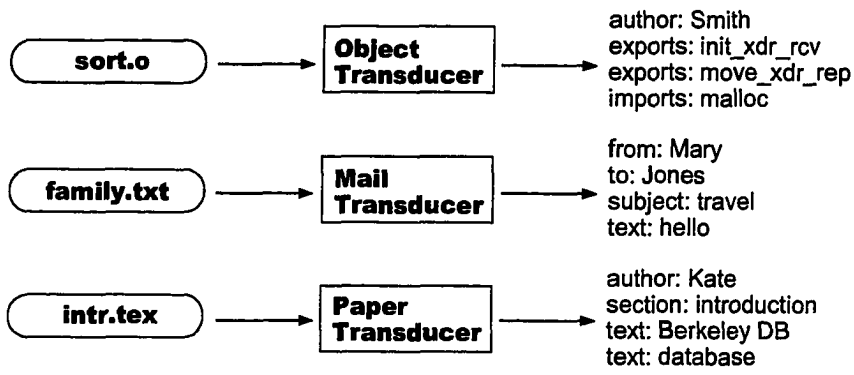

Figure 3.1: Sample Transducer Input and Output

updated or newly created file system objects to extract the properties for indexing. Automatic indexing is performed when files or directories are created or updated.

3.1.1 Semantic File System Semantics

Semantic file systems can implement as many semantics as needed. Files stored in a semantic file system are interpreted by file type specific transducers to produce a set of descriptive attributes, which help later retrieval of the files. The semantics of a semantic file system can be readily extended and improved because users can write new transducers.

Attribute. An attribute is a *field-value* pair, where a *field* describes a property of a file (such as its author, or its subject), and a *value* is a string or an integer. A file could have many attributes that have the same field name but with different field values. For example, a text file would have as many `author:` attributes as authors it has. In SFS, field names end with a colon.

Entity. The unit of associative access is called an *entity*, which is generalized from files that contain multiple objects (such as mail files).

Query. A query is a collection of desired attributes that permits a high degree of selectivity in locating entities of interest. The result of a query is a set of files and/or directories that contain the entities described in the query. The values of a field can be useful when narrowing a query to eliminate entities that are not of interest [6].

A semantic file system is *query consistent* when it guarantees query results that correspond to its current contents. Through the use of virtual directories, queries are performed in a semantic file system to describe a desired view of file system contents. From the viewpoint of a client program, a virtual directory is indistinguishable from an ordinary directory. In a semantic file system, the query facilities appear as virtual directories at each level of the directory tree. A *field virtual directory* is named by a field, and has one entry for each possible value of its corresponding field. The entries in a *field virtual directory* are *value virtual directories*. A *value virtual directory* as one entry for each entity defined by a field-value pair. So in the following example, the virtual directory `/sf/author:` corresponds to the `author:` field. The field virtual directory `/sfs/author:` would have one entry for each author that has written a file in `/sfs`.

```
%ls -F /sfs/author:
Mary/   Smith/
```

Thus the value virtual directory `/sfs/author:/Smith` contains entries for files in `/sfs` that are written by `Smith`. Each entry is a symbolic link to the file. For example:

```
%ls -F /sfs/author:/Smith
abstract.txt@   thesis.tex@   schedule.pdf@
```

The semantic file system has been implemented in Unix/Linux and is syntactically compatible with the traditional Unix/Linux hierarchical file system. A directory called */sfs* is added at the root of the Unix/Linux file system, as the root from which all queries begin. From this directory, queries can be made by specifying pairs of directory names, which are

separated by a colon. The first directory name specifies the request field. The second directory name specifies the value of the field, which we want to search for. Further searches may be done inside of the resulting directory to narrow the search.

The semantic file system is fast enough to be usable and provides an enhanced search ability, allowing people to locate files they would otherwise have had difficulty finding. It is an effective solution to the problem of content-addressable file access. Furthermore, such a file system can effectively be layered on top of a traditional hierarchical Unix/Linux file system, using overloading of symbols and inclusion of virtual directories.

however, we think the semantic file system still relies the tree structures. Given a right dictory , it will help users to more quickly recognize which one is what they want, but users still need to remember in which directory their files could be, then check potential virtual directories.

3.2 File Classification (Categorization)

Mack et al. [5] divided the basic tasks in knowledge management into 4 stages: Capture/Extract, Analyze/Organize, Find, Create/Synthesize, and distribute/Share. Many powerful file management systems pay a lot of attention to file classification (text classification). Text classification has direct benefits to the Analyze/Organize and Find stages. It can be generated manually, automatically or semi-automatically.

In the commercial field, the manual Analyze/Organize step requires a great deal of manual intervention and is very time consuming. So automatic content analysis and classification is the better choice, although they will lose accuracy. There are a number of mechanisms that can help people create taxonomies and place informational objects within their categories, although the amount of automation can vary. For example, some accept a training set within an existing taxonomy, and place files in categories based on similarities. Others allow human cataloguers to create sophisticated rules to specify certain words and

phrases, which will place a page in a category and be used for automatic content analysis. However, the complexity of language parsing, combined with the non-textual data type and large number of proprietary file formats, limits the effectiveness of content analysis [16]. Moreover, automatic mechanisms ignore a very important way that users think about their data: **Context**, which is defined as "the interrelated conditions in which something exists or occurs." [15]. Context is often the way that people remember things [16]. And users usually use context to locate data [14]. The same information could have different context for different users. So, automatic mechanisms are not accurate enough for all users.

Anne Kao [4] has proven that user assisted text classification results in a more correct and precise description of each file that a fully automatic system would. Some programs simply allow users to manually classify files to a specific category. It is not limited by file types, and is very accurate although it is time-consuming.

Chapter 4

Our Approach

CASH is a category-based shell. It is developed from TCSH, so it inherits all benefits from TCSH.

4.1 What Is a Category?

A category is a user created name used to describe the commonalities of all files that belong to it. A category can be any noun or noun phrase, but should be meaningful to its users. It helps users to organize their files and reminds them of what their files are about. In our approach, a category usually shows one common aspect of a group of files. Sometimes, It also can be an arbitrary value that reminds users of one of features of files, but does not have to be a word in corresponding files. In Figure 4.1, *TCSH*, *BASH* and *IEEE* could be potential categories that users can make, because their names show one aspect of a group of files. Since they are meaningful, it is easy for users to remember, or recall them.

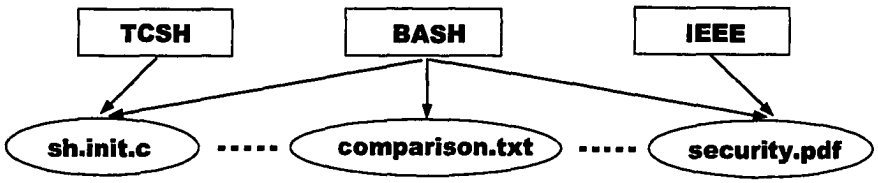

Figure 4.1: Multiple categorizations

4.2 Category-based shell: CASH

CASH is compatible with Unix/Linux file systems, which means that a file in it can have only one physical location, so it can work on existing OSs, and also can take advantage of ongoing OS improvements. In addition, it keeps the current file access capabilities, and users feel comfortable to work with their files in a manner with which they are accustomed.

CASH generalizes existing OS directories and folders, links in Unix/Linux, and shortcuts in Windows into a generic notion of categories. Categories in CASH are the primary means for organizing, grouping, and retrieving files. Restrictively, a directory, folder or a link can be treated as a category. However, CASH extends the conventional tree hierarchies by introducing category hierarchies. In a category hierarchy, files can belong to multiple categories, and a category can also have multiple parent categories while having multiple child categories. In addition, users feel free to create their own hierarchical or flat category structures as long as there is no circle in it. To realize this function, we develop a set of new commands for users to organize their categories and files. On the other hand, to help users to create their category hierarchy more easily, we allow them to load their current directory structure as the basis for their category hierarchies.

In CASH, we provide fast access to any category just by entering a simple category name without an absolute or relative path. Our mechanism allows users to directly reach any category through a friendly interface. Moreover, users can edit aliases for every category, this helps them to more conveniently find the categories that they want.

In addition, CASH is designed for ordinary computer users, and normally, they will not deal with too many files a day. In this case, when a user creates, or copies a file, they always know the topics with which they are concerned in this file. Since these topics could be very good clues when the user searches this file in the future, they can be accurate basis for its categorization. We know that users usually take a few hours or even a few days to get useful information, no matter if they edit it themselves or search and copy it from somewhere else, they should not mind taking a few more seconds just to arrange files well and place them in right categories, because it is for their future searching convenience. Therefore, in our approach, we adopt manual categorization, which is the most accurate method to classify their files.

4.3 Flat Category Structure

In a flat category structure, every category is independent, and there is no relationship between any pair of categories. The knowledge field for each category can be disjointed or overlapping.

It is easy to construct a flat category structure; Figure 4.2 gives a flat category structure example. It shows that a flat category structure can provide categorization for files. For example, */code/JAVA/assign.java* is categorized to category *Berkeley DB* and *JAVA* because it is a program written by JAVA language and including the JAVA API functions for Berkeley DB.

However, it is not convenient to work with a flat category structure. When users want to associate a file with some categories, they need to list all existing categories in their system because they are not sure that if they already had these categories, or what are the exact names for these categories. Then they have to search them one by one to find out the related categories, or create new ones. If there are too many categories in their system, It is difficult to search for possible categories. In this case, users may prefer to create a new

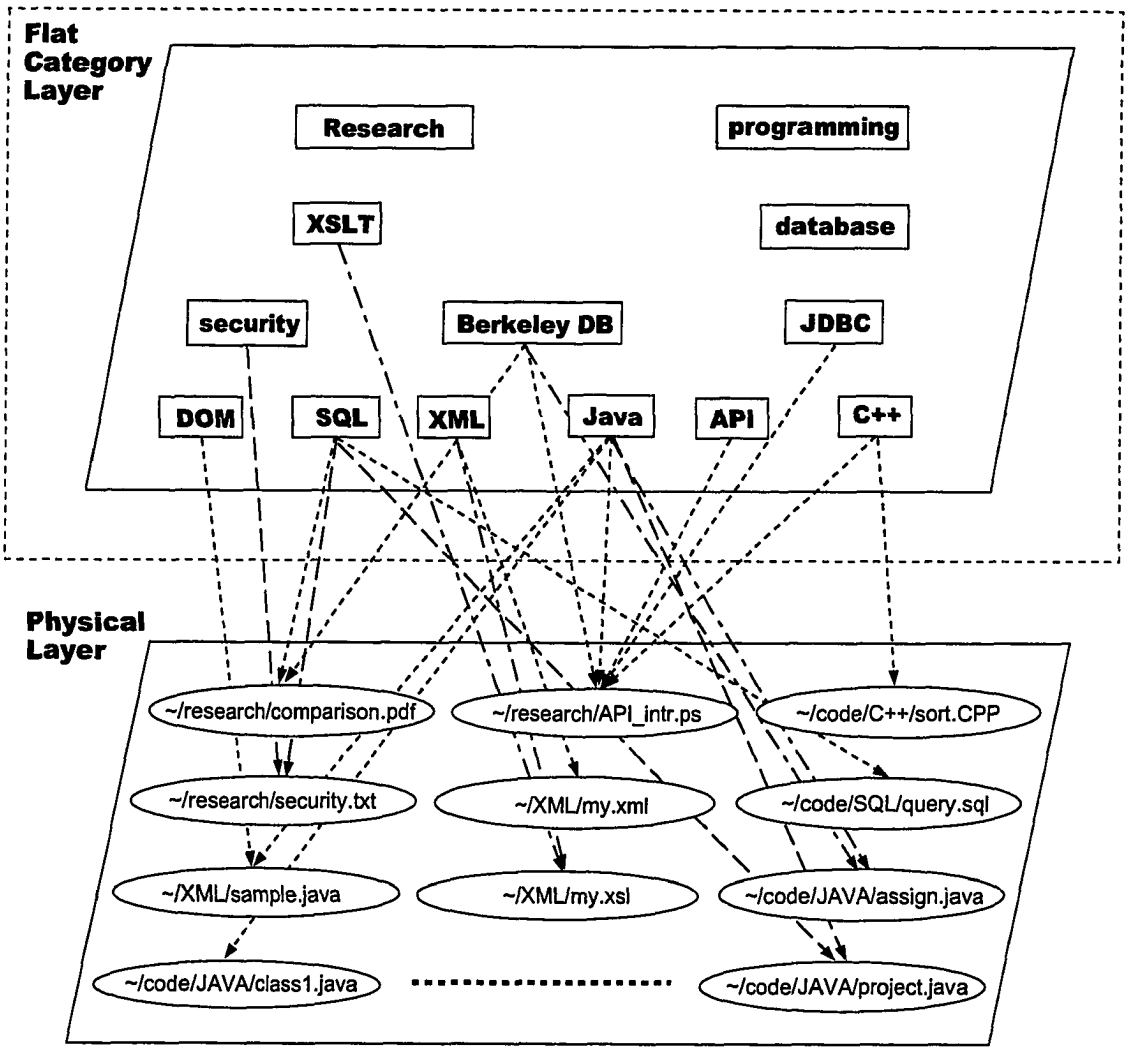

Figure 4.2: Flat category structure

category instead of searching for it. This could increase the category redundancy, and the more categories there are, the more difficult the searching is. This is why CASH provides hierarchical category structure.

4.4 Hierarchical Category Structure

The *tree* structure in all operating systems is a kind of hierarchical structure. Hierarchically organizing files has become a natural habit for computer users, and they also use it very well. This is one reason that we apply a hierarchical category structure in CASH. The other reason is that a category hierarchy solves the problem that occurred in the flat category structure. As we known, in all hierarchical structures, the knowledge field from the upper level to the lower level becomes from general to specific, so the hierarchy itself provides a logical path, and so acts as a guide helping users to find the potential categories.

Moreover, in a category hierarchy, any category can have one or multiple parent categories, which could be from the same or different levels. The default parent category is *root*. Also, a category can have zero or as many sub-categories as needed, which also can belong to different levels. At the same time, a file can be classified to multiple categories. Users can access a file or category through different paths in category hierarchies.

Therefore, restrictively speaking, a category hierarchy is not a hierarchical tree, but a hierarchical graph. To avoid loops in category navigation, it can only be an acyclic graph. Figure 4.3 shows an example of hierarchical category structure. There are two layers in the whole system structure: **Hierarchical Category Layer** and **Physical Layer**. A hierarchical category layer is a personalized category hierarchy, and a physical layer can be composed of all physical files. In this figure, category *DOM* has two parent categories: *XML* and *JAVA*, because it is related to the both knowledge fields. So if users want to find some files about *DOM*, they can get them from path */Research/XML* or */Programming/JAVA*.

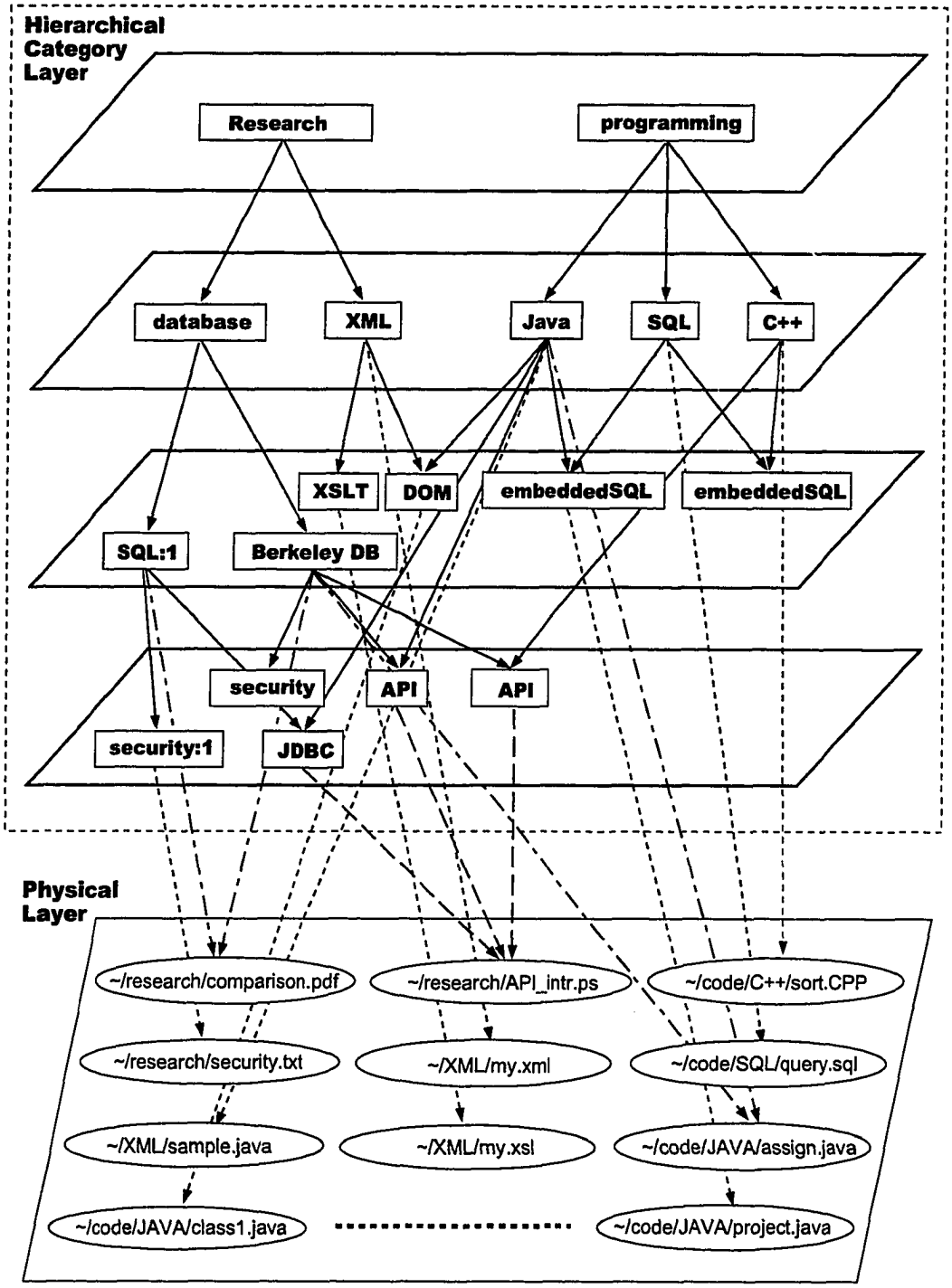

Figure 4.3: Hierarchical category structure

4.5 Links versus Category Management Approach in CASH

Our system provides a brand new mechanism to organize files. It has all the benefits of links but not their problems. The only limitation is that it works in the Unix file system.

Links are mainly used to create multiple entrances for files or directories. Usually, users would like to create links across different directories, which helps users to get to a directory or reach a file through different paths; otherwise these additional entrances aren't that helpful. In CASH, every category can have as many aliases as they need, users can get to a category by all its aliases. In addition, each category can have multiple parent categories, which allows users to navigate to a category through different paths. Therefore, from this point, a category management system can take over all benefits that links have.

As we said above, for non-superusers, a HARD LINK provides multiple entrances to a file. In CASH, to create one more path for a file, what users need to do is to only classify this file to another category that they want, instead of creating a link in Unix/Linux. Then we say that this file has a relationship with the given category, and can be listed under it. Unix/Linux only provides the same command `rm` for both physical file deletion and link deletion. If users just want to delete one extra entrance to a file, they will just delete one link for this file, called *logical deletion*, instead of deleting the physical file. So, they should make sure that the given link is not the last pointer to this file before deleting it. Otherwise the file could be physically removed from the disk. CASH solves this problem well by providing two different commands for logical deletion or physical deletion, so users can control the result of a removing operation. To remove one path of a file, users use `decat` to de-classify this file from one category. To physically remove a file, they can use `rmcat`. So users feel that it is much safer to work with files by using CASH.

A SYMBOLIC LINK is usually used to create an extra path or shortcut to a file or directory. If an original file or directory is deleted or moved, its entire symbolic links will be remained, but not work properly. As a result, users have to manually update or delete

useless or incorrect symbolic links. But, Unix/Linux does not provide explicit information to tell users where a file's links could be. Consequently, it is difficult for them to maintain links. The analogue of ***directory*** in CASH is ***category***. To create multiple paths for a category, users only need to create multiple parent-child relationships between other categories and itself. For example, category *JDBC* already had a parent category named *JAVA*. If users create a new parent-child relationship between *Database* and *JDBC*, then they can reach *JDBC* through two different paths. Once a category is deleted, all the relationships between it and other categories will be deleted automatically. Therefore, users don't need to make an effort to maintain their category hierarchies. CASH is timesaving.

4.6 Category System

Category system works as a medium between users and their file systems. Given a computer, if a single user wants to rearrange his/her files to make a new application convenient, he/she has to create/delete or move directories and files all around. But for multiple users with multiple applications, one directory structure in this computer usually can't make each user satisfied. So they usually have to duplicate directories for their own uses, which is a waste. The category system exactly helps users to solve this problem. Each user can have a personalized category structure for their files. Each file has both a category path and a directory path. If the logical location in the category hierarchy of a file is given in any file operation, the category system will transform it, and get its physical location in the directory structure, according to the user's personalized category structure. From this point of view, a category system works as a translator.

4.7 System Features

In conventional file systems, *file* and *directory* are two main elements that users use to manage and access their data. In our system, we use a category to generalize directories and symbolic links to overcome the problems with existing OS. CASH has the following features:

1. Every file can be classified to multiple categories.

Quite often, a file may involve several aspects and thus can be classified to several categories. In Figure 4.1, a file *comparison.txt*, which is copied from IEEE, and is about the comparison between TCSH and BASH. Then we would place it in categories TCSH, BASH, and IEEE.

2. A category can have multiple parent categories.

In a directory-based file systems, a directory is used to group a collection of files and directories, and its name often shows one knowledge field in common of all the information under it. Usually, the meaning of a parent directory usually is more general than those of its child directories. But a more specific knowledge field could belong to several more general knowledge fields. Therefore, in CASH, we allow a category to have multiple parent categories, each of which characterizes one aspect of it. This feature mainly helps users to more accurately define a knowledge field for a category, then to free them from the puzzle of where to place their files. In Figure 4.4, category *embeddedSQL* could be related to both *C++* and *SQL*, or *JAVA* and *SQL*. So users need a pair of categories to define it.

With such a logical and accurate category hierarchy, users will be very clear where to place or locate a file.

3. Categories with the same name can share one parent category.

In directory-based file systems, a sub-directory's name is unique under a parent directory. So sometimes, we have to make an effort to find a synonym for an existing directory name because we want to create a new directory with similar information as this directory

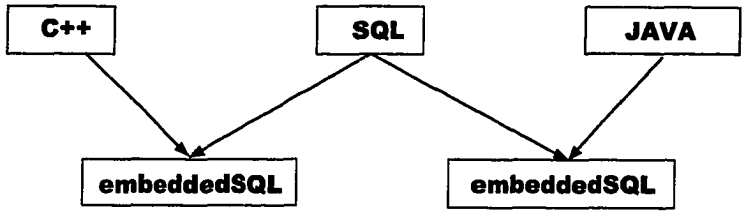

Figure 4.4: Multiple parent categories structure

has. In our system, users never have this problem, just as Figure 4.4 shown above, *SQL* have two sub-categories named *embeddedSQL*. This resolves the problem of naming categories. For those duplicated categories, which have completely the same parent categories, users have to differentiate them with various *Desp* information. If users enter a non-unique category under a path, the system will list all of its duplicated categories along with their parent categories and descriptions, so users will be very clear as to which category they should choose.

4. Files with the same name can share one category.

In CASH, files with the same name can share one category. In this case, the system will list all these files with their categorizations and descriptions, then users can make a choice.

5. A file and a sub-category under a category can share the same name.

In a directory-based system, files and child-directories are not allowed to use the same name in a directory. There is no such a limitation in our system. CASH allows that categories and files can have the same names no matter where their logical positions are. But users need to tell the system if they want to process with files or categories.

6. Auto-searching. In CASH, to reach a file or category, an absolute or relative path is not necessarily provided. We provide auto-searching function to make path entry simpler and faster. Auto-searching means to automatically search for a simple category or a simple file name in the database. If the result is not unique, users are allowed to choose a specific one from the result list. We will introduce the application of this function in Chapter 5.

7. Aliases for category.

A category can have a few aliases, which are treated the same as the category. For example, we could generalize the knowledge about SQL and Berkeley DB into category *database*, but *DB* is also a very popular word to represent it. So users can create *DB* as an alias of *database*, and these two category names will be treated equally. This means that a category shares the same position in a category hierarchy with all of its aliases.

4.8 How to Construct Hierarchical Category Structure?

Taxonomy construction is a challenging problem that works best with sufficient domain knowledge. Fully automatic construction often leads to unsatisfactory results. Consequently, most taxonomies are built and maintained manually. On the other hand, even with the same files, different users can have different classifications because they have different viewpoints or focal points. In addition, our system is designed for users to manage their own files. They usually know or can predict what information will be in their computers. Especially, users always know the exact content that they are concerned with for every file when they create or copy them. So users are the best candidates to construct their own hierarchical category structure in our system. In our system, we provide a command, which helps users to load directory structures in file systems as their initial category hierarchies, so they don't need to construct their category structures from scratch.

General Rules For Constructing Category Hierarchies A category hierarchy can be developed top-down as users build a directory tree structure. There are three kinds of nodes in the whole category hierarchy: **Root**, **Internal nodes** and **Leaves**. **Root** is the category, which only has child categories, but no parent category, and is the highest level in a hierarchy; An **internal node** is a category that has both parent categories and child categories. A **leaf** is a category, which only has parent categories but no child categories. A

parent category usually has more general meaning than its child categories, and a category along with all its child categories, called sub-hierarchy, can describe a knowledge field, which is meaningful or related to users. For example, we can build a sub-hierarchy for "people" to show a relationship between people who could be related to our files, such as authors of paper, and senders of emails. We also can create a sub-hierarchy to show the hierarchical structure of our interests and routine business. Moreover, if we are students or professors, we can construct one sub-hierarchy for our courses. Generally speaking, users can build as many sub-hierarchies as knowledge fields that they are interested in or related to. Then if it is needed, they can logically link some of these sub-hierarchies to form a more accurate and complete structure to map all contents for their files.

A category hierarchy is good or right as long as its user feels it is logical and convenient, and can map his files well. However, no matter how users want to build their own hierarchies, the only limitation is that it can never have a circle in it. We will show a sample in the next chapter.

Chapter 5

Implementation Issues

5.1 System Architecture

This system architecture is shown in Figure 5.1. CASH provides a command line interface, and it follows the general command syntax of existing shells. After a command is entered, CASH determines if it involves the category system or changes information of some files according to a pre-defined command list. If it doesn't, then the system just throws it to TCSH. If this command is related to category operation, or affects a file's information, then the category system will be aroused to take over the job. Figure 5.2 shows the algorithm of our system.

5.2 Built-in Commands

A shell allows execution of Unix/Linux commands, both synchronously and asynchronously. Unix/Linux shells also provide a small set of built-in commands (builtins) implementing functionality impossible (e.g. cd, break, continue, and exec), or inconvenient (history, kill, or pwd, for example) to obtain via separate utilities. In our approach, we develop some new built-in commands for users to implement the category management system.

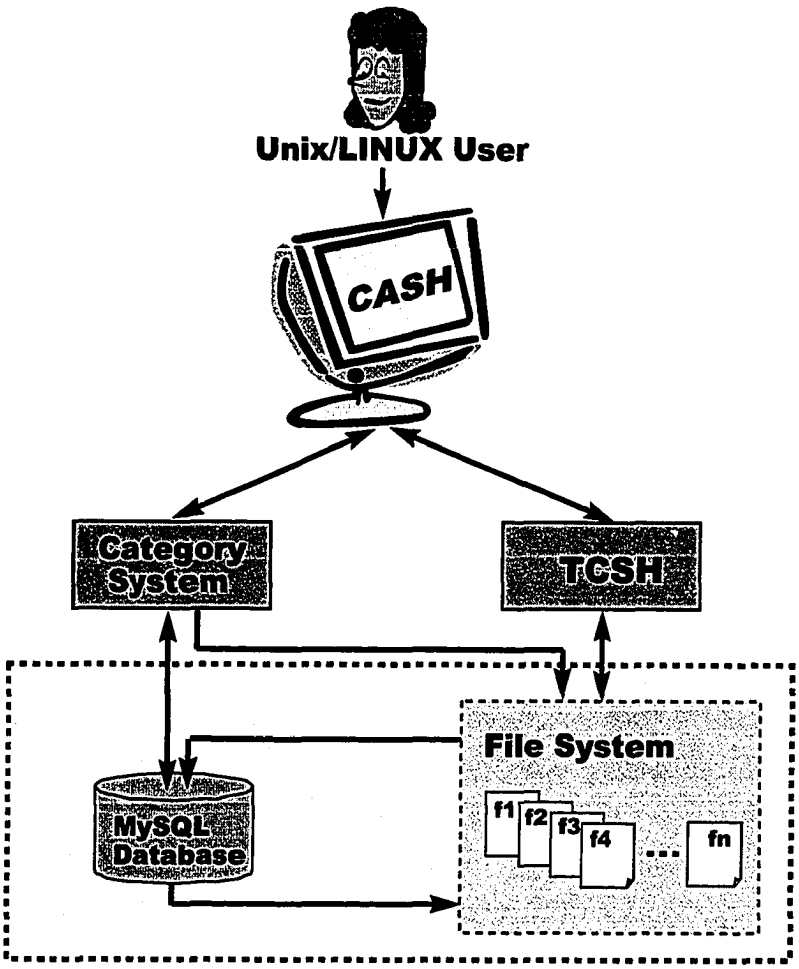

Figure 5.1: System architecture

System Algorithm.

```
1.  initialization;
2.  REPEAT
3.      READ keyboard;
4.  UNTIL a command (c) is entered;
5.  IF c is a built-in command THEN
6.      IF c is a command related to category system THEN
7.          creates the connection to the database;
8.          executes the corresponding operation;
9.          closes the connection to the database;
10.     ELSE
11.         executes the corresponding TCSH command;
12.         IF c is not the "exit" command THEN
13.             GOTO 2.
14.         ELSE
15.             EXIT;
16.         END IF
17.     END IF
18. ELSE
19.     locates c in the disk;
20.     creates a new process;
21.     run c in this new process;
22.     GOTO 2.
23. END IF
```

Figure 5.2: System algorithm.

5.3 Database Design

In CASH, we use Mysql to store the information for all directories, files, categories, and a category hierarchy with four tables.

5.3.1 Category Information

Two tables are used to store category information: Category-Names and Category-Hierarchy. Each record of **Category-Names table** is composed of **CID**, **CName**. **CID** is the unique ID of a category, and **CName** is the name of a category or an alias. All aliases for one category have the same ID. Figure 5.3 shows a sample category table.

Category-Names:

CID	CName
0	/
1	course590
2	paper
3	note
4	course595
5	assignment
6	note
7	email
7	letter
8	JAVA
9	SQL
10	JDBC

Figure 5.3: Category-Names Table

Since *email* and *letter* have the same ID, they are aliases. There are two *note* in this

table, but they have a different ID.

5.3.2 Hierarchy Information

The hierarchical information of a category structure in the database is stored in the **Category-Hierarchy table**. This table is composed of four fields: **CID, Parents, Children**, and **Desc**. **CID** is the ID of a category, and it is originally defined in Category-Names table. **Parents** is a list of pairs of parent ID and parent category name. **Children** is a list of pairs of a child ID and a child category name. And, **Desc** is a short text to illustrate the knowledge field of a category. It is helpful to remind users of what information is in this category. Figure 5.4 is a sample category hierarchy information stored in database:

Category-Hierarchy:

CID	Parents	Children	Desc
0		<1|course590><4|course595> <7|email><8|JAVA><9|SQL>	
1	<0|/>	<3|note><2|paper>	
2	<1|course590>		only for winter term
3	<1|course590>		
4	<0|/>	<6|assignment><5|note>	
5	<4|course595>		
6	<4|course595>		
7	<0|/>		
8	<0|/>		from Smith
9	<0|/>		
10	<8|JAVA><9|SQL>		

Figure 5.4: Category-Hierarchy Table

INDEX: In our database design, we use *Index* for each of the tables. Indexes are used to find rows with specific column values fast. Without an index, MySQL has to start with the first record and then read through the whole table to find the relevant rows. The larger the table, the poorer the performance. If the table has an index for the columns in question, MySQL can quickly determine the desired position in the middle of the data file without having to look at all the data. "If a table has 1,000 rows, this is at least 100 times faster than reading sequentially. Note that if you need to access almost all 1,000 rows, it is faster to read sequentially, because that minimizes disk seeks" [8].

Index helps to speed up directories and files' finding, which makes it possible for our approach to realize the automatic directory and file recognition in command line editing.

5.3.3 Directory Information

CASH saves all useful information about directories in the **Directories table**. Each record of this table is as follows:

DINO: Each directory has a unique ID, which is the inode number of this directory;

DPath: the path of a directory;

Mode: the mode of the directory;

Mtime: the last modify time of a directory;

Size: the size of a directory;

Links: the number of links;

Uid: user id for a directory;

Gid: group id for a directory;

We will explain why we need this table in the following section. Figure 5.5 shows sample information in this table:

Directories:

DINO	DPath	Mode	Mtime	Size	link	Uid	Gid
3640	/export/home/lyang	16877	Dec 01 10:10	4096	4	lyang	other
251091	/export/home/lyang/course590	16832	Dec 02 17:20	2048	4	lyang	other
251094	/export/home/lyang/course590/paper	16877	Dec 05 21:09	1024	4	lyang	other
...	...	...	...	...	...	...	...

Figure 5.5: Directories Table

5.3.4 File Information

All files properties and their categorizations are stored in **Files table**. File properties can be used to reflect features of the files themselves, such as size, of the activities over the file, such as "last-modify Dec 15 8:22"; Most file storage and management systems already provide some sort of file property (or "metadata") mechanisms, which record details such as the owner of a file, its length, and when it was last accessed. In Unix/Linux, a file's properties are saved in its Inode. File properties are primary information used for file finding no matter in DOS or Unix/Linux. However, we usually waste a lot of time waiting for search results, because there isn't the support of a database. For example, to seek a file under a possible directory, the `find` utility has to read this directory and all of its sub-directories one by one. This operation is very time-consuming if there are lots of directories and files will be involved. I believe that every Unix user has the same experience. Therefore, we store all files' properties including their physical locations in this table, and the database mechanism can help us get what we want faster. But it may waste a lot of disk space if we save the absolute name for each file, because each path length of Unix/Linux-based systems can be up to 1024 characters. This is why we will store the directory ID instead of an absolute directory path here. Files table has the following structure:

FName: the name of a file, without path;

DID: the unique directory ID;

Mode: the mode of the file;

Ctime: the creation time of a file;

Atime: the last-accessed time of a file;

Mtime: the last modify time of a file;

Size: the size of a file;

Links: the number of links;

Uid: user id for a file;

Gid: group id for a file;

Inode: the number of inode;

Desc: a short description of a file. Usually it is impossible to show many files on one screen, but it is possible to display many corresponding abstracts for files. This helps users make a faster judgment when looking for files.

Categories: a list of category names, which are associated with the file; Figure 5.6 shows sample file information.

Files:

FName	DID	Mode	Ctime	Atime	Mtime	Size
insert.CPP	251091	33152	Jan 6 12:45	Jan 7 02:45	Jan 6 12:45	36280
sort.CPP	251091	33152	Jan 16 12:40	Jan 9 07:30	Jan 20 13:55	25185
sample.java	251094	33152	Jan 7 01:09	Mar 23 09:23	Mar 09 02:05	1939
...	...	...	...	...	...	...

Uid	Gid	Links	Inode	Desp	Categories
1201	1	1	5956		<4|course595><9|SQL>
1201	1	1	5950	a bubble-sorting	<5|assignment>
1201	1	1	5960		<4|course595><10|JDBC>
...	...	...	...	...	...

Figure 5.6: Files Table

5.4 Command Design

While designing commands, we try our best to make every command easy for users to remember and embed frequently used functions in our new commands. We inherit corresponding old names, and add *cat* at the end of them to distinguish the new commands from the old ones. For example, `rmcat` is similar to `rm` in Unix/Linux, and it is used to remove a category.

Furthermore, we allow some commands to apply two sets of path formats: **Directory Format** and **Category Format**. If an argument is given in *directory format*, the system will locate it in the directory structure by the given directory path. Then, if an argument is given in *category format*, the system will locate it in category hierarchy by the given category path. The default path format is *category format* in our system. For example, it is quite common to classify files, which are in directory format, to categories when users just initiate their category structures, because most of files have no categorization yet. The syntax of the command is as follows:

```
classify [option] [arg1] [arg2] ...  [arg3]
```

For example, if users want to classify */tcsh/sh.c* to category *course595* and *C++*, they can use the following command:

```
classify -d  /tcsh/sh.c course595,C++
```

(Option `'-d'` indicates that */tcsh/sh.c* is a file in directory format.)

But after users have done most of the file classification, then, it is more convenient for them to remember and to use category paths to adjust or to modify a file's categorization. For example, If users want */tcsh/sh.c* to belong to one more category *database*, they can enter:

```
classify -f /C++/sh.c database
```

(Option `'-f'` indicates that */C++/sh.c* is a file.)

Moreover, to help users to build their category hierarchy, we develop a command

named `load`, so users can choose to load the whole directory structure or some of its sub-structures as original drafts for their category hierarchies if they think it is helpful. These extended commands are basically divided into two parts: one part is for category hierarchy construction, and the other part is for file classification.

5.4.1 Rules for Command Process

(1). The fault-tolerance of category paths obeys the rules for the recognition of a file or directory path in Unix/Linux.

For example:

a) `./../research/../comparison.pdf = research/comparison.pdf`

b) `/../../research/comparison.pdf = /research/comparison.pdf`

(2). Duplicated categories in one category are allowed, so if there are non-unique categories existing in a given command, users have to make a choice according to the listed options. For example, the following command means to list all child categories and files in *embeddedSQL*,

```
lscat /programming/SQL/embeddedSQL
```

Since there are two categories named *embeddedSQL* in *SQL*, the system will prompt an option list:

```
<0> SQL, C++/embeddedSQL
<1> SQL, JAVA/embeddedSQL
Please choose a number, or 'Q' to quit!  [ ]
```

Then users can reach the specific category that they specifically desire.

(3). Given a category with an absolute path name, our system will apply the traditional path verification mechanism. In this case, the auto-searching function is disabled. Consider the following example:

```
lscat /programming/DOM
```

According to 4.3, there is no category named *DOM* under *programming*, thus, an error message will be prompted.

(4). Given a simple category or a category with a relative path, the system will see whether it is available under the current working category. If it doesn't, auto-searching function is enabled to globally search the first category in the given path. Once it is found, the system will start to verify the path from it instead of the current working category. For example, suppose the current working category is *XML*, and when the following command is entered:

```
lscat programming/JAVA
```

Since *programming* is not in *XML* according to Figure 4.3, the system will automatically search *programming* in the database first. After finding it, the system will continue to check whether *JAVA* is under *programming*. The system only applies auto-searching for the first category in a category path, so if users enter:

```
lscat programming/DOM
```

then an error message will be prompted, because there is no a *DOM* under *programming* in Figure 4.3.

(5). Files can be searched automatically in CASH, and Rule (3) and Rule (4) also apply to file path searching and checking.

(6). The option `'-f'` is used to tell the system that the given arguments are files when executing some commands in which category arguments and file arguments are both acceptable. We allow child categories and files in a category to share the same name, and auto-searching for both files and categories are provided. To avoid user confusion, our system thinks the entered arguments are categories by default for these commands. For example,

```
lscat programming/JAVA/DOM/sample.java
```

CASH will search *sample.jave* as a category. To tell the system that it represents a file, users should add an option `'-f'` in their commands. So the following command will list the information of *sample.jave*.

```
lscat -f programming/JAVA/DOM/sample.java
```

(7). The Option '-d' is used to tell the system that the given arguments are provided in a directory format when executing some commands, in which category arguments and directory arguments are both acceptable. We have shown an example in section 5.4.

5.5 Command Overview

Generally speaking, commands developed in this system are mainly used for category displaying, directory loading, hierarchy building and modification, and file classification. We also extended some existing shell commands, which can affect the data in the users' databases, such as cp, rm, etc. We give a basic introduction to these commands in this section.

5.5.1 Commands for Displaying

5.5.1.1 Prompts

The system provides two sets of prompts displaying: **Directory Prompt** and **Category Prompt**. The default is the directory prompt. The user is free to switch between both.

1. The **category** utility displays the category prompt. An example of **category** is as follows:

```
rose-home{1}category
CASH-/>
```

2. The **directory** utility displays the directory prompt. An example of **directory** is as follows:

```
CASH-/>directory
rose-home{1}
```

5.5.1.2 Loading a Directory

The **load** utility is used to load existing directories and all of its sub-directories as pre-defined categories, and it allows a file to automatically inherit its classification in the directory structure. Symbolic links for a directory could be considered to be its aliases. This utility helps users to rapidly build their hierarchical category structures. The command format is as follows:

load *dir*

dir is a directory, which will be loaded. Figure 5.7 shows the algorithm for the `load` utility.

Example:

1. Load all files under the directory *sample* and its sub-directories;

```
CASH-/>load ~/sample
loading: /export/home/lyang/sample
loading: /export/home/lyang/sample/research
loading: /export/home/lyang/sample/research/XML
loading: /export/home/lyang/sample/research/database
loading: /export/home/lyang/sample/code
loading: /export/home/lyang/sample/code/C++
loading: /export/home/lyang/sample/code/JAVA
loading: /export/home/lyang/sample/code/SQL
loading: /export/home/lyang/sample/code/SQL/embeddedSQL
```

Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12 show the information in the database after the execution of the load command.

5.5.1.3 Changing the Current Working Category

The **cdcat** utility changes the current working category. It has the following format:

cdcat *cat*

Function: Load *dir* with all of its sub-directories under a category list *cats*

1. IF directory *dir* exists THEN
2. CALL load_dir with *cats* and *dir*;
3. ELSE
4. prompts an error message;
5. END IF
6. return;

Function: Load *dir* under a category list *cats*

PROCEDURE **load_dir** {*cats*, *dir*}

1. writes the inode information of *dir* to **Directories** table;
2. CALL new_category with *cats*, *dir*, NULL and 1; // Figure 5.8
3. FOR each file *f* under directory *dir*
4. categorizes *f* to *dir*;
5. writes *f*'s inode information and categorization to **Documents** table;
6. END FOR
7. FOR each sub-directory *sub-dir* under *dir*
8. CALL **load_dir** with *dir*, *sub-dir*;
9. END FOR

END PROCEDURE

Figure 5.7: Algorithm for `load`

Function: To create a new category *new_cat* with *parents* as its parent category list, *children* as its child category list, and the flag *update_related_cats* determines if the information of category in *parents* and *children* need to be updated.

PROCEDURE **new_category** {*parents*, *new_cat*, *children*, *update_related_cats*}

1. adds *new_cat* to **Category_Names** table;
2. adds *new_cat* to **Category_Hierarchy** table, takes *parents* as its parent category list, and *children* as its child category list;
3. IF update_related_cats=1 THEN
4. FOR each category *c* in *parents*
5. appends category *new_cat* to *c*'s child category list;
6. END FOR
7. FOR each category *c* in *children*
8. appends category *new_cat* to *c*'s parent category list;
9. END FOR
10. END IF

END PROCEDURE

Figure 5.8: Algorithm for new category creation.

CID	Cname
0	/
1	sample
2	research
3	XML
4	database
5	code
6	C++
7	JAVA
8	SQL
9	embeddedSQL

Figure 5.9: CategoryTBL after `load`

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	<2|research><5|code>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>		
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|code>		
7	<5|code>		
8	<5|code>	<9|embeddedSQL>	
9	<8|SQL>		

Figure 5.10: HierarchyTBL after `load`

DINO	DPath	Mode	Mtime	Size	link	Uid	Gid
5912	/export/home/lyang/sample	16877	Dec 01 08:30	512	4	lyang	other
5940	/export/home/lyang/sample/research	16832	Dec 02 08:40	512	5	lyang	other
197389	/export/home/lyang/sample/research/XML	16877	Dec 02 09:01	512	2	lyang	other
197390	/export/home/lyang/sample/research/database	16877	Dec 02 10:10	512	2	lyang	other
6260	/export/home/lyang/sample/code	16832	Dec 02 12:30	512	5	lyang	other
197380	/export/home/lyang/sample/code/C++	16877	Jan 01 07:21	512	2	lyang	other
197386	/export/home/lyang/sample/code/JAVA	16877	Jan 02 10:23	512	2	lyang	other
197387	/export/home/lyang/sample/code/SQL	16877	Jan 11 14:20	512	3	lyang	other
197388	/export/home/lyang/sample/code/SQL/embeddedSQL	19877	Jan 02 16:30	512	2	lyang	other

Figure 5.11: DirectoryTBL after `load`

cat is a specific category that the current working category will be changed to.

Example:

1. Change the current working category to *sample*;

```
CASH-/>cdcat sample
CASH-sample>
```

5.5.1.4 Displaying the Current Working Category

The user can check their current working category path by using the **`pwdcat`** utility. An example of **`pwdcat`** is as follows:

```
CASH-sample>pwdcat
/sample
```

5.5.1.5 Listing Files or Categories

The user can use **`lscat`** utility to list the information of files or categories. The format is as follows:

FName	DID	Mode	Ctime	Atime	Mtime	Size
insert.CPP	197380	33152	Jan 6 11:45	Jan 7 12:15	Jan 6 11:45	74590
sort.CPP	197380	33152	Jan 6 12:50	Jan 7 12:45	Jan 6 12:50	6415
gui.java	197386	33152	Mar 6 02:05	Mar 7 08:30	Mar 6 02:05	8930
project.java	197386	33188	Mar 12 07:28	Mar 17 09:10	Mar 12 07:28	2738
query.sql	197387	33152	Jan 6 01:09	Jan 6 09:23	Jan 6 03:19	1939
assign.java	197388	33188	Oct 24 10:06	Dec 20 08:30	Oct 30 09:27	12099
tutorial.pdf	5940	33152	Feb 02 11:03	Feb 02 13:30	Feb 02 11:50	4913
comparison.txt	5940	33152	Feb 02 10:29	Feb 02 13:10	Feb 02 10:50	3878
test.xml	197389	33152	Jan 01 09:06	Jan 17 08:28	Jan 03 09:27	1520
security.doc	197390	33152	Oct 20 10:06	Mar 01 14:48	Oct 20 10:06	2645

Uid	Gid	Links	Inode	Desp	Categories
1201	1	1	5956		<6|C++>
1201	1	1	5950		<6|C++>
1201	1	1	5960		<7|JAVA>
1201	1	1	2450		<7|JAVA>
1201	1	1	14250		<8|SQL>
1201	1	1	5610		<9|embeddedSQL>
1201	1	1	2568		<2|research>
1201	1	1	2590		<2|research>
1201	1	1	3100		<3|XML>
1201	1	1	2530		<4|database>

Figure 5.12: FileTBL after `load`

lscat [option] arg_1 arg_2 ... arg_n

The arg_1,arg_2...arg_n are categories or files, whose information will be listed. To list the properties of files, option '-f' should be used.

Examples:

1. List all file names and category names, which are under the current working category *research*;

```
CASH-research>lscat
XML/     database/    comparison.txt    tutorial.pdf
```

From the result, we can know that *comparison.txt* and *tutorial.pdf* are classified to category *research*, and, *XML* and *database* are child categories of *research*.

2. Lists all information under category *sample*, including all sub-categories and files;

```
CASH-sample>lscat -R
/sample:
code/     research/
/sample/code:
C++/     JAVA/     SQL/
/sample/code/C++:
insert.CPP  insert.CPP
/sample/code/JAVA:
gui.java   project.java
/sample/code/SQL:
embeddedSQL/  query.sql
/sample/code/SQL/embeddedSQL:
assign.java
/research:
XML/     database/    comparison.txt    tutorial.pdf
/research/XML:
test.xml
```

```
/research/database:
security.doc
```

3. Lists all files, which satisfy the following conditions:
   1. The files are under catgory *JAVA* and *research*;
   2. The file names match the given pattern 's?t*';

```
CASH-sample>lscat JAVA,research -n=s?t*
```

`lscat` covers the function of `find`.

5.5.2 Commands for Building a Category Hierarchy

5.5.2.1 Renaming an Existing Category

The **`rename`** utility is used to rename a category. We designed two approaches for this command. One is only to rename a category and to retain all of its aliases. The other is to rename all of its aliases when a category is renamed. A **`rename`** command line specifies an existing category and a new category name. The format is as follows:

`rename` *existing-cat new-name*

Examples:

1. Rename category *code* to *programming*;

```
CASH-sample>rename code programming
CASH-sample>lscat
programming/     research/
```

After the execution of this command, we will get Figure 5.13 and Figure 5.14.

5.5.2.2 Creating a New Category

The **`mkcat`** utility is used to create new categories. It can create a category with a single parent or multiple parent categories.

CID	Cname
0	/
1	sample
2	research
3	XML
4	database
5	**programming**
6	C++
7	JAVA
8	SQL
9	embeddedSQL

Figure 5.13: CategoryTBL after `rename`

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	<2|research><**5|programming**>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>		
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<**5|programming**>		
7	<**5|programming**>		
8	<**5|programming**>	<9|embeddedSQL>	
9	<8|SQL>		

Figure 5.14: HierarchyTBL after `rename`

- To create categories with a single parent, the format is as follows:

  mkcat [option] cat_1 cat_2 ... cat_n

  The cat_1,cat_2,...cat_n are the categories that will be created.

- To create a new category with multiple parents, the format is follows:

  mkcat [cat_1,cat_2,...,cat_n]/*new-cat*

  The *new-cat* is the category, which will be newly created. And, the cat_1,cat_2,...cat_n are the parent categories of *new-cat.*

Examples:

1. Create a category named *XSLT* and *DOM* under *XML.*

```
CASH-XML>mkcat XSLT DOM
```

2. Create a category *embeddedSQL* with multiple parent categories: *SQL* and *C++.*

```
CASH-programming>mkcat  [SQL,C++]/embeddedSQL
```

After the execution of these two commands, we will get Figure 5.15 and Figure 5.16. There are two categories named *embeddedSQL,* but with different parent category lists.

5.5.2.3 Create Aliases for a Category

The **lncat** utility is used to create one or multiple aliases for a category. This function helps users to find a potential category easily. The format is as follows:

lncat *cat* n_1 n_2 ... n_n

cat is an existing category, which will have aliases. The n_1 ... $alias_n$ are aliases for *cat.*

Example:

1. To create an alias named *DB* for an existing category *database*;

```
CASH-programming>lncat ../reseach/database DB
```

Only the information in the *category* table will be updated. Figure 5.17 shows the result.

CID	Cname
0	/
1	sample
2	research
3	XML
4	database
5	programming
6	C++
7	JAVA
8	SQL
9	embeddedSQL
10	**XSLT**
11	**DOM**
12	**embeddedSQL**

Figure 5.15: CategoryTBL after `mkcat`

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	<2|research><5|programming>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>	**<11|DOM><10|XSLT>**	
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|programming>	**<12|embeddedSQL>**	
7	<5|programming>		
8	<5|programming>	<9|embeddedSQL>**<12|embeddedSQL>**	
9	<8|SQL>		
10	**<3|XML>**		
11	**<3|XML>**		
12	**<6|C++><8|SQL>**		

Figure 5.16: HierarchyTBL after `mkcat`

CID	CName
0	/
1	sample
2	research
3	XML
4	database
...	...
12	embeddedSQL
4	**DB**

Figure 5.17: CategoryTBL after `lncat`

5.5.2.4 Creating Parent-Child Relationship between Categories

The **supcat** utility creates parent-child relationships between one or multiple parent categories and one child, while the whole category hierarchy remains acyclic. The format is as follows:

supcat [option] p_1,p_2,...,p_n *child*

child is a category name, which will be a child category of the existing categories named p_1,p_2,...,p_n.

Example:

1. Make *JAVA* to be one parent category of *DOM*;

```
CASH-XML>supcat  /sample/programming/JAVA  DOM
CASH-XML>lscat -lz DOM
JAVA, XML/DOM
Child-Categories:
Description:
```

Then we will get Figure 5.18 from Figure 5.16.

5.5.2.5 Deleting Parent-Child Relationship between Categories

The **desup** utility deletes existing parent-child relationships between one or multiple parent categories and a child category. The format is as follows:

desup p_1,p_2,...,p_n *child*

child is a category, which will be removed from the child list of the categories named p_1,...p_n.

Example:

1. Remove the parent-child relationship between *SQL* and *embeddedSQL*;

```
CASH-programming>lscat /
sample/
```

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	<2|research><5|programming>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>	<11|DOM><10|XSLT>	
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|programming>	<12|embeddedSQL>	
7	<5|programming>	**<11|DOM>**	
8	<5|programming>	<9|embeddedSQL><12|embeddedSQL>	
9	<8|SQL>		
10	<3|XML>		
11	<3|XML>**<7|JAVA>**		
12	<6|C++><8|SQL>		

Figure 5.18: HierarchyTBL after `supcat`

```
CASH-programming>desup  SQL embeddedSQL
<0> SQL/embeddedSQL
Description:
<1> C++,SQL/embeddedSQL
Description:
Please choose a number, or 'Q' to quit! [0]
CASH-programming>lscat /
embeddedSQL/   sample/
```

Since there are two *embeddedSQL* under *SQL*, users have to choose which one will be involved in this command. In addition, the selected *embeddedSQL* had only one parent category *SQL*, so after the removal of this parent-child relationship, its parent category becomes `'/'` by default. Then Figure 5.18 will be changed to Figure 5.19.

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	<2|research><5|programming>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>	<11|DOM><10|XSLT>	
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|programming>	<12|embeddedSQL>	
7	<5|programming>	<11|DOM>	
8	<5|programming>	**<12|embeddedSQL>**	
9	**<0|/>**		
10	<3|XML>		
11	<3|XML><7|JAVA>		
12	<6|C++><8|SQL>		

Figure 5.19: HierarchyTBL after `desup`

5.5.2.6 Removing Categories or Files

The **rmcat** utility is used to remove categories. There are several different cases considered for this command:

(1). Removing files given in a category path format.

(2). Removing a category, but retaining its aliases in the system (similar with *hard link* deletion).

(3). Removing a category with all of its aliases.

(4). Removing a category with all of its files, which are associated to it.

(5). Removing a category with all of its sub-categories encountered.

If an internal node is removed, all of its child categories could inherit its parent categories or not. The format of **rmcat** is as follows:

rmcat [option] *cat*

cat is a category name or filename, which will be deleted. Figure 5.20 show the algorithm for removing a category.

Examples:

1. Remove category *DB*, but keep its alias and files;

```
CASH-research>rmcat DB
Alias: database
Retain aliases? or 'q' to quit!  [y]
```

Then, we will get Figure 5.15 again from Figure 5.17.

2. Remove category *embeddedSQL* and all of its files;

```
CASH-research>rmcat -D /embeddedSQL
/export/home/lyang/sample/code/SQL/embeddedSQL/assign.jave: is
deleted!
```

Then, besides the record of *embeddedSQL*(CID=9) being deleted from Figure 5.15 and Figure 5.16, the record of *assign.java* will also be removed from Figure 5.12.

Function: To remove a category *cat* with the option *opt*

```
1.  IF category cat category exists THEN
2.      IF cat has aliases THEN
3.          prints an option: "cat: remove it with its aliases?"
4.          IF 'y' THEN
5.              CALL delete_category with the cat, opt;      //Figure 5.21
6.              return;
7.          ELSE
8.              removes cat from Category_Names table;
9.              removes cat from all parent category lists and child category lists in Category Hierarchy
                table;
10.             return;
11.         ENDIF
12.     ELSE
13.         prints an option: "cat: the last copy of the category. remove?"
14.         IF 'y' THEN
15.             CALL delete_category with the cat, opt
16.         END IF
17.     END IF
18. ELSE
19.     prints an error message;
20. END IF
21. return;
```

Figure 5.20: Algorithm for `rmcat`

```
PROCEDURE delete_category {cat, option}
1.  IF the FORCE bit in option is not set THEN    //FORCE bit determines if idle-check is needed;
        //if a category has only one parent category and no child category, and there is no file
        //classified to it, then we say it is idle.
2.      IF cat has multiple parent categories or has any child category THEN
3.          return;
4.      END IF
5.  END IF
6.  deletes cat with all of its aliases in Category_Names table;
7.  deletes the record of cat in Category_Hierarchy table;
8.  IF the RECURSIVE bit in option is set THEN        //RECURSIVE bit determines if its child
                                                      //categories should be deleted.
9.      deletes cat with all of its aliases from all child category lists in
        Category_Hierarchy table;
10.     FOR each category sub in the child category list of cat
11.         CALL delete_category with sub, option
12.     END FOR
13. ELSE
14.     delete cat with all of its aliases from all parent category lists and child category lists in
        Category_Hierarchy table;
15.     FOR each category c in the parent category list of cat
16.         appends the child category list of cat to the child category list of c;
17.     END FOR
18.     FOR each category c in the child category list of cat
19.         appends the parent category list of cat to the parent category list of c;
20.     END FOR
21. END IF
END PROCEDURE
```

Figure 5.21: Algorithm for category deletion.

5.5.2.7 Category Copy

The **cpcat** utility makes a copy of a category. A **cpcat** command line specifies source and destination categories. The format is as follows:

cpcat [option] *source-cat destination-cat*

source-cat is the name of the category that **cpcat** is going to copy. The *destination-cat* is the name that **cpcat** assigns to the resulting copy of the category.

There are two kinds of copy functions that are considered: Shadow Copy and Deep Copy.

Shadow Copy: Creates a copy of *source category*, and this new category also inherits its child category list. Figure 5.22 shows the algorithm for shadow copy.

Function: Shadow copy *a* to *b*.

1. IF both category *a* and category *b* exist THEN
2. CALL new_category with *b*, *a*, child list of the original *a*, 1; //Figure 5.8
3. ELSE
4. prompts error message;
5. END IF

Figure 5.22: Algorithm for shadow copy

Deep Copy: Creates a copy of *source category* with all of its sub-categories. Figure 5.23 shows the algorithm for deep copy. Examples:

1. Shadow copies *XML* to *sample*;

```
CASH-sample>cpcat research/XML .
CASH-sample>lscat .
XML/    programming/   research/
CASH-sample>lscat XML
DOM/    XSLT/
```

```
Function: Deep copy a to b.
1.  IF both category a and category b exist THEN
2.      CALL new_category with b, a, child list of the original a, 0;
3.      FOR each category c under the original a
4.          CALL new_category with the parent list of c, c, child list of c, 0;   //Figure 5.8
5.      END FOR
6.      FOR each newly created category n
7.          reads n's child category list and parent category list;
8.          FOR each category d in the parent category list
9.              IF d is duplicated in this operation THEN
10.                 replaces d with its newly duplicated category;
11.             ELSE
12.                 appends n to d's child category list;
13.             END IF
14.         END FOR
15.         FOR each category in the child category list
16.             replaces it with the newly duplicated one;
17.         END FOR
18.     END FOR
19. ELSE
20.     prompts an error message;
21. END IF
22. return;
```

Figure 5.23: Algorithm for deep copy

Figure 5.24 and Figure 5.25 show the current information in database after the execution of the above command.

CID	Cname
0	/
1	sample
2	research
3	XML
4	database
5	programming
6	C++
7	JAVA
8	SQL
10	XSLT
11	DOM
12	embeddedSQL
13	XML

Figure 5.24: CategoryTBL after `shadow copy`

Figure 5.26 displays the current category hierarchy. 2. Deep copy *research/XML* to *sample*;

```
CASH-research>rmcat /sample/XML
CASH-research>lscat ../sample
programming/  research/
CASH-research>cpcat -C XML ../sample
CASH-research>lscat sample
XML/  programming/  research/
```

Then, we will get Figure 5.27 and Figure 5.28.

Figure 5.29 visualizes the current category hierarchy.

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	**<13|XML>**<2|research><5|programming>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>	<11|DOM><10|XSLT>	
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|programming>	<12|embeddedSQL>	
7	<5|programming>	<11|DOM>	
8	<5|programming>		
10	<3|XML>**<13|XML>**		
11	<7|JAVA><3|XML>**<13|XML>**		
12	<8|SQL>		
13	**<1|sample>**	**<11|DOM><10|XSLT>**	

Figure 5.25: HierarchyTBL after `shadow copy`

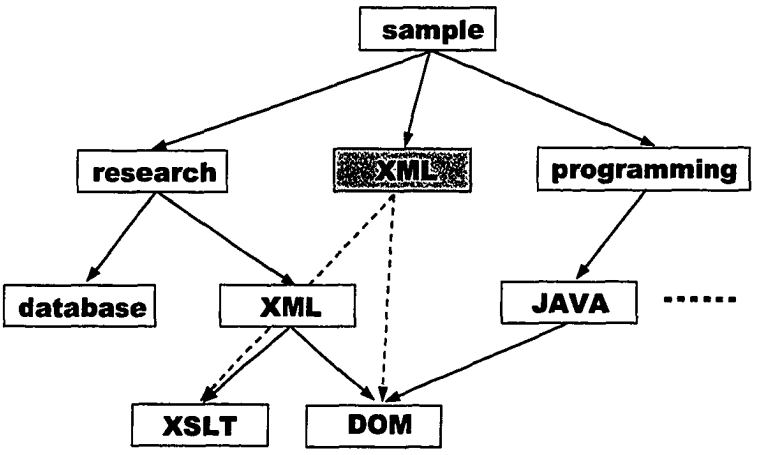

Figure 5.26: Sample Shadow Copy

CID	Cname
0	/
1	sample
2	research
3	XML
4	database
5	programming
6	C++
7	JAVA
8	SQL
10	XSLT
11	DOM
12	embeddedSQL
14	**XML**
15	**XSLT**
16	**DOM**

Figure 5.27: CategoryTBL after `deep copy`

CID	Parents	Children	Desc
0		<1|sample>	
1	<0|/>	**<14|XML>** <2|research><5|programming>	
2	<1|sample>	<3|XML><4|database>	
3	<2|research>	<11|DOM><10|XSLT>	
4	<2|research>		
5	<1|sample>	<6|C++><7|JAVA><8|SQL>	
6	<5|programming>		
7	<5|programming>	<11|DOM>**<16|DOM>**	
8	<5|programming>	<12|embeddedSQL>	
10	<3|XML>		
11	<7|JAVA><3|XML>		
12	<8|SQL>		
14	**<1|sample>**	**<16|DOM><15|XSLT>**	
15	**<14|XML>**		
16	**<7|JAVA><14|XML>**		

Figure 5.28: HierarchyTBL after `deep copy`

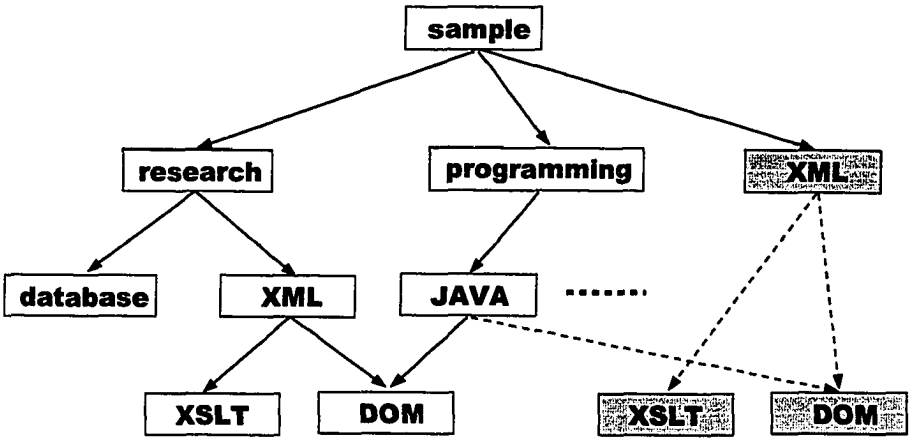

Figure 5.29: Sample Deep Copy

5.5.2.8 Category/File Record Editing

The **edcat** utility edits the description of a file or a category. The format is as follows:

edcat [option] *source desc*

source could be a filename or a category name. It is determined by the *option. desc* is the description for it.

Examples:

1. Edit the description of the category *sample*;

```
CASH-/>edcat  sample  "a sample category for testing"
CASH-/>lscat -l
sample/
parent-category: /
child-category: XML, programming, research
alias:
description: a sample category for testing
```

2. Edit the description of file *sort.CPP*;

```
CASH-/>edcat -f sample/programming/C++/sort.CPP  "a bubble-sorting"
```

```
CASH-/>lscat -l
sort.CPP
Description: a bubble-sorting
```

5.5.3 Commands for Classification

5.5.3.1 Classify Files to Categories

The **classify** utility classifies files to one or multiple categories. The format is as follows:

classify [option] *source* cat_1,...,cat_n

Two sets of path formats are considered in this command as mentioned above. So, the *source* could be a directory, a file in the directory path format, a category, or a file in the category path format. cat_1,...,cat_n are existing categories which will be classified to *source*. Examples:

1. Classify */research/tutorial.pdf* to *C++* and *JAVA*;

```
CASH-research>lscat -lf tutorial.pdf
research/tutorial.pdf
-rw-r--r--   lyang  other     4913   Feb 20 08:21 /export/home/
lyang/sample/research/tutorial/pdf
CASH-research>classify -f tutorial.pdf C++,JAVA
C++: No such a category, auto-search? [y]
JAVA: No such a category, auto-search? [y]
CASH-research>lscat -lf tutorial.pdf
C++, JAVA, research/tutorial.pdf
-rw-r--r--   lyang  other     4913   Feb 20 08:21 /export/home/
lyang/sample/research/tutorial/pdf
```

After the execution of this command, we will get Figure 5.30 from the result above.

FName	DID	Mode	Ctime	Atime	Mtime	Size
insert.CPP	6	33152	Jan 6 11:45	Jan 7 12:15	Jan 6 11:45	74590
sort.CPP	6	33152	Jan 6 12:50	Jan 7 12:45	Jan 6 12:50	6415
gui.java	7	33152	Mar 6 02:05	Mar 7 08:30	Mar 6 02:05	8930
project.java	7	33188	Mar 12 07:28	Mar 17 09:10	Mar 12 07:28	2738
query.sql	8	33152	Jan 6 01:09	Jan 6 09:23	Jan 6 03:19	1939
tutorial.pdf	2	33152	Feb 02 11:03	Feb 02 13:30	Feb 02 11:50	4913
comparison.txt	2	33152	Feb 02 10:29	Feb 02 13:10	Feb 02 10:50	3878
test.xml	3	33152	Jan 01 09:06	Jan 17 08:28	Jan 03 09:27	1520
security.doc	4	33152	Oct 20 10:06	Mar 01 14:48	Oct 20 10:06	2645

Uid	Gid	Links	Inode	Desp	Categories
1201	1	1	5956		<6|C++>
1201	1	1	5950	a bubble sorting	<6|C++>
1201	1	1	5960		<7|JAVA>
1201	1	1	2450		<7|JAVA>
1201	1	1	14250		<8|SQL>
1201	1	1	2568		**<6|C++><8|SQL>**<2|research>
1201	1	1	2590		<2|research>
1201	1	1	3100		<3|XML>
1201	1	1	2530		<4|database>

Figure 5.30: FileTBL after `classify`

5.5.3.2 De-Classify Files from Categories

The **declassify** utility de-classifies files from one or multiple categories. The format is as follows:

declassify [option] *source* cat_1,...,cat_n

Unlike **classify**, **declassify** only considers the category path format. *source* could be a category, or a file in the category path format. cat_1,...,cat_n are existing categories, which will be de-classified to *source*.

Example:

1. De-classify */research/tutorial.pdf* from category *research*;

```
CASH-research>declassify -f tutorial.pdf research
CASH-research>lscat -lf tutorial.pdf
C++, JAVA/tutorial.pdf
-rw-r--r--   lyang  other     4913   Feb 20 08:21 /export/home
/lyang/sample/research/tutorial/pdf
```

After the execution of this command, Figure 5.31 shows the result of this command.

5.5.3.3 Classification Copy

The **catcp** utility is designed to copy a categorization of a file to that of another file. The format is as follows:

catcp [option] *source-file destination-files*

source-file is the name of the file whose categorization is going to be copied, and the *destination-file* is the name of the file that is assigned the categorization.

Example:

1. Copy the categorization of *tutorial.pdf* to *comparison.txt*

```
CASH-research>lscat -lf comparison.txt
research/comparison.txt
```

FName	DID	Mode	Ctime	Atime	Mtime	Size
insert.CPP	6	33152	Jan 6 11:45	Jan 7 12:15	Jan 6 11:45	74590
sort.CPP	6	33152	Jan 6 12:50	Jan 7 12:45	Jan 6 12:50	6415
gui.java	7	33152	Mar 6 02:05	Mar 7 08:30	Mar 6 02:05	8930
project.java	7	33188	Mar 12 07:28	Mar 17 09:10	Mar 12 07:28	2738
query.sql	8	33152	Jan 6 01:09	Jan 6 09:23	Jan 6 03:19	1939
tutorial.pdf	2	33152	Feb 02 11:03	Feb 02 13:30	Feb 02 11:50	4913
comparison.txt	2	33152	Feb 02 10:29	Feb 02 13:10	Feb 02 10:50	3878
test.xml	3	33152	Jan 01 09:06	Jan 17 08:28	Jan 03 09:27	1520
security.doc	4	33152	Oct 20 10:06	Mar 01 14:48	Oct 20 10:06	2645

Uid	Gid	Links	Inode	Desp	Categories
1201	1	1	5956		<6|C++>
1201	1	1	5950	a bubble sorting	<6|C++>
1201	1	1	5960		<7|JAVA>
1201	1	1	2450		<7|JAVA>
1201	1	1	14250		<8|SQL>
1201	1	1	2568		**<6|C++><8|SQL>**
1201	1	1	2590		<2|research>
1201	1	1	3100		<3|XML>
1201	1	1	2530		<4|database>

Figure 5.31: FileTBL after `declassify`

```
-rw-r--r--   lyang  other     3878   Feb 02 10:50 /export/home
/lyang/sample/research/comparison.txt
CASH-research>catcp turotial.pdf comparison.txt
CASH-research>lscat -lf comparison.txt
C++, JAVA/comparison.txt
-rw-r--r--   lyang  other     3878   Feb 02 10:50 /export/home
/lyang/sample/research/comparison.txt
```

Then , we will get Figure 5.32.

FName	DID	Mode	Ctime	Atime	Mtime	Size
insert.CPP	6	33152	Jan 6 11:45	Jan 7 12:15	Jan 6 11:45	74590
sort.CPP	6	33152	Jan 6 12:50	Jan 7 12:45	Jan 6 12:50	6415
gui.java	7	33152	Mar 6 02:05	Mar 7 08:30	Mar 6 02:05	8930
project.java	7	33188	Mar 12 07:28	Mar 17 09:10	Mar 12 07:28	2738
query.sql	8	33152	Jan 6 01:09	Jan 6 09:23	Jan 6 03:19	1939
tutorial.pdf	2	33152	Feb 02 11:03	Feb 02 13:30	Feb 02 11:50	4913
comparison.txt	2	33152	Feb 02 10:29	Feb 02 13:10	Feb 02 10:50	3878
test.xml	3	33152	Jan 01 09:06	Jan 17 08:28	Jan 03 09:27	1520
security.doc	4	33152	Oct 20 10:06	Mar 01 14:48	Oct 20 10:06	2645

Uid	Gid	Links	Inode	Desp	Categories
1201	1	1	5956		<6|C++>
1201	1	1	5950	a bubble sorting	<6|C++>
1201	1	1	5960		<7|JAVA>
1201	1	1	2450		<7|JAVA>
1201	1	1	14250		<8|SQL>
1201	1	1	2568		<6|C++><8|SQL>
1201	1	1	2590		**<6|C++><8|SQL>**
1201	1	1	3100		<3|XML>
1201	1	1	2530		<4|database>

Figure 5.32: FileTBL after `catcp`

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this paper, we proposed a category-based file management system, CASH, which is implemented in Unix/Linux systems and is syntactically compatible with the traditional Unix/Linux hierarchical file system. In conventional file management systems, users' data are organized by files, and files are organized by directories in a tree hierarchy. A file belongs to one directory, and a directory has only one parent directory. Since a file name or a directory name is not enough to describe all knowledge aspects for a file or a directory, computer users sometimes feel that they can't find an appropriate name for a file, and also, they don't know where to properly place them. Right placement is the prerequisite for effective file searching. To solve these problems, we adopt a *category* to organize groups of files, instead of a *directory*, and use *category hierarchy* instead of *tree hierarchy* to manage all categories. The main features of CASH are: A file can be classified to multiple categories; A category could have multiple parent categories. These features greatly improve the flexibility for file placing and file searching because of multiple file access and multiple category access. Moreover, benefiting from existing database technologies, path entry is

not necessary for a file or a category because CASH provides the automatic searching function for both files and categories. So, CASH is an effective and efficient file management system, which helps computer users organize, manage, maintain and search for files much more easily and with more flexibility.

6.2 Future Work

In CASH, we implement category auto-searching, but didn't develop automatic category completion as the automatic completion mechanism for file names, directory names in TCSH, or some other Shells. To best help users, we will develop this function in the future.

In addition, to keep the consistency of data in the database, we extended some existing Shell commands: `rm`, `cp`, `mv`. But there are some other commands that can be developed, such as `vi` and `emacs`, because they can affect the information of files.

On the other hand, MySQL is a relational database with client/server architecture. In such a scenario, user interface and application programs run on the client side, while queries and transactions run on the sever side. This results in unnecessary overhead. Therefore, we are considering to use a simple embedded database engine: Berkeley DB, instead of Mysql, to provide the database facilities in our system.

Chapter 7

Appendixes. Command details

7.1 Commands for Displaying

7.1.1 Prompts

1. Switch the current prompt to category prompt;

Command: **`category`**

2. Switch the current prompt back to directory prompt;

Command: **`directory`**

7.1.2 Loading a Directory

Command: **`load`** *directory*

7.1.3 Changing the Current Working Category

Command: **`cdcat`** *category*

In this command, *auto-searching* function is disabled.

7.1.4 Displaying the Current Working Category

Command: **pwdcat**

7.1.5 Listing Information of Files or Categories

Command: **lscat** [option] arg_1 arg_2 ... arg_n

Option:

none: Lists the contents of given categories, including files that are classified to them and all sub-categories.

-a: Shows the information of all categories, which are named as the given category without path.

-D: Shows information of files, which are classified to the given categories.

-C: Shows all sub-directories of the given categories.

-z: Shows the given categories' own information, not their sub-categories or files.

-f: The given arguments are considered to be files' names, and listing their category location.

-l: Lists the long format, giving categorization, directory location, mode, number of links, owner, group, size in bytes, and time of last modification for each file. If the time of last modification is greater than six months ago, it is shown in the format 'month-date-year', otherwise in the format 'month-date-time'.

-R: Recursively lists sub-categories encountered.

7.2 Commands for Building a Category Hierarchy

7.2.1 Renaming an Existing Category

Command: **rename** *originalname newname*

7.2.2 Creating a New Category

Command: **mkcat** [option] $category_1$ $category_2$ ... $category_n$

Option:

-p: Creates category by creating all the non-existing parent categories first.(same as mkdir -p)

7.2.3 Creating Parent-Child Relationship between Categories

Command: **supcat** [option] $parent_1, parent_2, \ldots, parent_n$ *child*

Option:

none: Adds more parent categories to a category.

-e: Removes all current parent-child relationship, and add new parent categories to a category.

7.2.4 Deleting Parent-Child Relationship between Categories

Command: **desup** $parent_1, parent_2, \ldots, parent_n$ *child*

7.2.5 Removing Categories or Files

Command: **rmcat** [option] arg_1 ... arg_n

Option:

none: Removes the given categories, but remain the files that are classified to them.

-D: Removes the given categories with all files that are classified to them.

-f: Removes the given files.

-r: Removes the given categories with all sub-categories encountered.

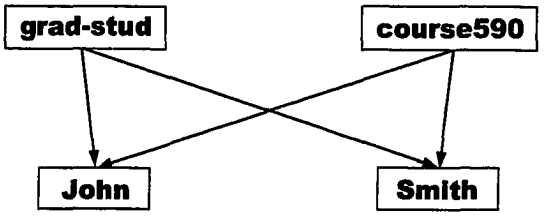

Figure 7.1: Sample Parent Copy

7.2.6 Category Copy

Command: **cpcat** [option] *source destination*

Option:

none: Shadow copies *source category* to *destination category.*

-c: Copies the child categories list of *source category* to that of *destination category.*

-D: Deep copies *source category* with all of its child categories to *destination category.*

-p: Copies the parent categories list of *source category* to that of *destination category.*

Figure 7.1 shows a sample result for the execution of the following command:

cpcat -p *John Smith*

7.2.7 Category/File Record Editing

Command: **edcat** [option] *source text*

Option:

none: *source* is a category, *text* is used to edit its description.

-f: *source* is a file in category format; *text* is used as its abstract.

7.3 Commands for Classification

7.3.1 Classify Files to Categories

Command: **classify** [option] *source* cat_1,...,cat_n

Option:

none: *source* is a category.

-A: cat_1...cat_n will not replace but be appended to original categorization.

-f: *source* is a file in category format.

-z: *source* is a file/directory in directory format.

7.3.2 De-Classify Files from Categories

Command: **declassify** [option] *source* cat_1,...,cat_n

Option:

none: *source* is a category.

-f: *source* is a file in category format.

-d: *source* is a file/directory in directory format.

7.3.3 Classification Copy

Command: **catcp** [option] *source* *dest*

Option:

none: *dest* is a category.

-f: *dest* is a file in category format.

-d: *dest* is a file/directory in directory format.

Bibliography

[1] BARREAU, D. AND NARDI, B. A. *Finding and reminding: File organization from the desktop.* SIGCHI Bull 39-43 July 1995.

[2] Scott Fertig, Eric Freeman, David Gelernter *"Finding and Reminding" Reconsidered.* ACM SIGCHI Bulletin, Volume 28 Issue 1 January 1996

[3] Paul Dourish, W. Keith Edwards, Anthony Lamarca. *Extending Document Management Systems with User-Specific Active Properties.* ACM Transactions on Information Systems. vol. 18, No. 2 April 2000.

[4] Anne Kao, Lesley Ouach, Steve Poteet,Steve Woods. *User assisted text classification and knowledge management.* Proceedings of the twelfth international conference on Information and knowledge management.November2003.

[5] R. Mack, Y. Ravin, and R.J. Byrd. *Knowledge portals and the emerging digital knowledge workplace.* IBM Systems Journal,40(4) p925-p955,2001.

[6] David K.Gifford, Pierre Jouvelet. *A semantic file system.* Proceedings of the thirteenth ACM symposium on Operating systems principles,1991.

[7] Mark G. Sobell. *A practical guide to Unix system V, Third edition.*

[8] http://dev.mysql.com/doc/mysql/en/mysql-indexes.html

[9] http://www.mysql.com/it-resources/benchmarks

[10] http://en.wikipedia.org/wiki/Filesystem

[11] http://databases.about.com/od/shareware/a/mysql.htm

[12] http://perl.about.com/cs/mysql

[13] John R. Doucer, William J. Bolosky *A large-scale study of file-system contents.* ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Atlanta, GA, 1999). Published as ACM SIGMETRICS Performance Evaluation Review, 27(1): 50-70. ACM Press 1999.

[14] Alvarado, Jaime Teevan, Mark S. Ackerman, and David Karger. *Surviving the information explosion: how people find their electronic information.* Technical report AIM-2003-006. Massachusetts Institute of Technology, April 2003.

[15] Merriam-Webster OnLine, http://www.m-w.com/.

[16] Craig A. N. Soules, Gregory R. Ganger, *Towards automatic content-based attribute assignment for semantic file systems".* Parallel Data Laboratory, Carnegie Mellon University.

[17] http://desktop.yahoo.com

[18] http://desktop.google.com

[19] http://www.trafficfile.com/internet_marketing/search_engine/introduction.htm