ABSTRACT

The limitation of data representation in today's file systems is that data representation is bound only in a single way of hierarchically organizing files. A semantic file system provides addressing and querying based on the content rather than storage location. Semantic tagging is a new way to organize files by using tags in place of directories. In traditional file systems, symbolic links become non-existent when file paths are changed. Assigning multiple tags to each file ensures that the file is linked to several virtual directories based on its content. By providing semantic access to information, users can organize files in a more intuitive way. In this way, the same file can be accessed through more than one virtual directory. The metadata and linkages for tagging are stored in a relational database which is invisible to the user. This allows efficient searching based on context rather than keywords. The classification of files into various ontologies can be done by the user manually or through automated rules. For certain files types, tags can be suggested by analyzing the contents of files. The system would be modular in design to allow customization while retaining a flexible and stable structure.

Keywords : virtual file system, semantics, indexing, classification, database, tagging, information access, metadata

Contents

List of Figures

Chapter 1

PROBLEM DEFINITION

Due to the several drawbacks of traditional mono-hierarchical file system, organizing and retrieving information becomes difficult. Semantic approaches to solve this problem have addressed some issues but contain several drawbacks. Features are spread between different implementations and often focus on requiring the user to have advanced skills. The purpose of this project is to focus on performing efficient information retrieval based on context.

A semantic file system allows searching and storing of data based on context rather than file names and keywords. The advantage of creating a file system over keyword based search tools is that file systems allow system level management of data. Keyword based tools work by maintaining a database of metadata extracted from files. They do not provide any facilities to manage storage. These tools can only extract metadata from a pre-configured list of known file types. Plus, they do not allow metadata to be added by users. These kinds of tools do not provide a complete semantic experience that can be developed by semantic file systems.

Thus, the goal of this project is to develop a semantic file system that extracts metadata from files and allows storage and searching based on its context.

Chapter 2

LITERATURE SURVEY

Over the years, organizing and retrieving information accurately and efficiently has attracted lot of attention. While few have been succesful, a number of innovative implementations [1] have emerged. The idea of using a file's semantics as the means to categorize it has been around for quite some time. This section discusses the various implementations made in the field of semantic file system.

An efficient implementation of keyword based searching was brought to the desktop by Google's Desktop Search [2] and Apple's Spotlight [3]. Both allow efficient and quick file retrieval based on keywords. They support many file types and have a simple interface which attracts a large number of users. However, both of them are limited to returning search results without any way to organizing contents. In addition, they do not provide any provision to the user for classification of data. This limitation prevented the user from having a personalized way to retrieve data stored by them.

Semantic systems depend on data stored inside the files rather merely relying on an file's attributes. Most implementations use common methodologies like content recognition [4], tagging [5], extracting metadata, etc. to categorize files by using various algorithms.

"Semantic File System" [6], as developed by O'Toole and Gittord in 1992, provides access to file contents and metadata by extracting the attributes using special modules called "transducers". It was one of the very first attempts to classify files by semantics using metadata. Its biggest drawback was the need for file type specific transducers which were necessary to extract meta information and content from the file. Also, the user does not have any say in what kind of category the file is classified under. This drawback makes it an unattractive option to the general user. It was decided during designing Kwest, that it is necessary to involve the end-user in the tagging process. This allows each user to have their own personal way of classification and organization of files.

NHFS (Non Hierarchical File System) [7] was a project developed by Robert Freund in July 2007. It allows the user to place any file into any number of directories. Likewise, any directory can be placed into as many directories as required. NHFS therefore allows one to create a non-hierarchical structure with poly-hierarchically connected files. This allows for a powerful metaphor of finding a file in any of the category (directory) it could be stored under. Therefore, we decided to retain this feature by using tags in place of actual directories. Tags are associated with files and other tags as well. Thus, a tag may be placed under multiple tags allowing a relationship to be defined between them. This analogy is much more powerful than restricting files to actual directories. Using tags prevents duplication and redundancy, making it an efficient implementation.

A more recent implementation is Tagsistant [8], which is a semantic file system that also attempts to organize files using tags. It interacts with the Linux kernel using the FUSE module. Under Tagsistant, directories are considered to be equivalent to tags. As a consequence, creating a directory is creating a tag and putting a file inside a directory means tagging that file. After you have tagged your files, you can search all of them by using queries. Queries are just paths where each element is either a directory or logical operators. The entire system has a modular design and uses SQLite. However, it suffers from some speed issues and the lack of SQL indexes. Major flaws of this design were

high consumption of inodes on real file systems and high computational time which was required to fulfill each request. Most of the features of Tagsistant were decided to be included in Kwest. These were modular design, SQLite repository, tagged structure, etc. which enhance the semantics of a file system. However, care must be taken to prevent the occurrence of similar drawbacks.

Another implementation called Tagster [9], is a peer-to-peer tagging application for organizing desktop data. It is platform independent and is implemented in JAVA. Multiple files and also directories can be tagged through its interface. The selected directories are recursively examined and all files contained within them are tagged. The GUI for a Linux system consists of three main areas. Namely - ”Tag view”: which displays a list of tags, ”Resource view”: which lists resources that have the currently selected tags assigned and ”User view”: that displays a list of users that have tagged the currently selected resource with some selected tag. It also includes GUI support for Windows with some unresolved issues. However, it lacks auto classification of data due to which several common tags may be generated for each user increasing the database size.

For the development of Kwest, we mainly refered to the following papers in order to form a guideline for our project.

1. Chang. K, Ramadhana. B, Sethuraman. K, Le. T, Chachra. N, Tikale. S, Perdana. I, Jain. M, Kartasasmita. I, *Knowledge File System - A principled approach to personal information management*. 2010 IEEE International Conference on Data Mining Workshops, Page(s): 1037-1044.

Abstract: The Knowledge File System (KFS) is a smart virtual file system that sits between the operating system and the file system. Its primary functionality is to automatically organize files in a transparent and seamless manner so as to facilitate easy retrieval. Think of the KFS as a personal assistant, who can file every one of you documents into multiple appropriate folders, so that when it comes time for you to retrieve a file, you can easily find it among any of the folders that are likely to contain it. Technically, KFS analyzes each file and hard links (which are simply pointers to a physical file on POSIX file systems) it to multiple destination directories (categories). The actual classification can be based on a combination of file content analysis, file usage analysis, and manually configured rules. Since the KFS organizes files using the familiar file/folder metaphor, it enjoys 3 key advantages against desktop search based solutions such as Googles desktop Search, namely 1) usability, 2) portability, and 3) compatibility. The KFS has been prototyped using the FUSE (Filesystem in USErspace) framework on Linux. Apache Lucene was used to provide traditional desktop search capability in the KFS. A machine learning text classifier was used as the KFS content classifier, complimenting the customizable rule-based KFS classification framework. Lastly, an embedded database is used to log all file access to support file-usage classification.

Usefulness: This paper describes approach to personal information management through Knowledge File System. It is designed to help users organize information using Virtual File System to reduce the problem of manual information classification and retrieval. KFS provides functions so as to automatically classify the information based on the content similarity with respect to predefined ontologies or also give the option for manual classification of the information. The operations carried out on the KFS can also be monitored with the event logger feature. Searching of files can be carried out by keyword with the help of a text indexer. Furthermore the comparisons between Google desktop file system, beagle and KFS are given. Finally the details of the implementation of the KFS on the Linux platform using of FUSE are given.

2. Eck. 0, Schaefer. D, *A semantic file system for integrated product data management*. 2011 Advanced Engineering Informatics, Page(s): 177-184.

Abstract: We initially discuss a number of disadvantages of current file management systems. In the body of the paper our main contribution is presented. That is, a formal mathematical model of a new semantic file system, SIL (Semantics Instead of Location), that allows engineers to access data based on semantic information rather than storage location is proposed. A major difference between our approach and previous related work is that we do not aim at yet another point solution and, instead, propose an approach that may be employed by next generation engineering data processing systems on a larger scale. In addition, a corresponding programming interface along with a graphical user interface used as a file browser is presented and the benefits of utilizing the proposed semantic file system for product data management in the field of integrated design of mechatronic systems are discussed.

Usefulness: This paper describes a formal mathematical model that allows engineers to access data based on semantics rather than actual storage location. The main goal of this file system is to search files based on content of data. The browsing of the system is done based on file's metadata and attributes. Logical operator such as AND, OR, NOT are used to filter the results. The classification allows multiple tags to be created for files. Furthermore API's are written to create views and for the automation of notification updates.

3. Mohan. P, Venkateswaran. S, Raghuraman, Siromoney. A, *Semantic File Retrieval in File Systems using Virtual Directories*. Proc. Linux Expo Conference, Raleigh, NC, Page(s): 141-151, May 2007.

Abstract: Hard Disk capacity is no longer a problem. However, increasing disk capacity has brought with it a new problem, the problem of locating files. Retrieving a document from a myriad of files and directories is no easy task. Industry solutions are being created to address this short coming. We propose to create an extendable UNIX based File System which will integrate searching as a basic function of the file system. The File System will provide Virtual Directories which list the results of a query. The contents of the Virtual Directory are formed at runtime. Although, the Virtual Directory is used mainly to facilitate the searching of file, it can also be used by plugins to interpret other queries.

Usefulness: This paper describes the design of SemFS, which provides semantics based on the file's meta-data and attributes. It allows the usage of logical operators to filter query results. It is implemented as a user space file systemupon journaling storage. The architecture is server-client with support for API's to extend functionality. Features such as file tagging and versioning are also implemented.

Many tagging systems exist that allow efficient manual classification of information. Most implementations tend to be theoretical [12] demonstrations or complex implementations [13] existing for some very specific purpose. These suggest the possibility of using semantics [14] in operating systems in some future date. But a major problem is scalability with regard to related information. However, on a large multi-user file system, one can get tons of tags to shift through in each folder, increasing the load for users to search and maintain data. Our idea is to introduce a new concept of relating tags to overcome this situation. Implementing all the desired and necessary features from previous implementations, our design goal is to create an efficient Semantic File System which could be used by any class of users.

Chapter 3

SOFTWARE REQUIREMENT SPECIFICATION

3.1 Introduction

Traditional file systems are mono-hierarchical and implement directory trees to categorize and store files. In such systems, directories are the only means to access particular files. The path of a file contains directories, which refer to its context and categorization. As an example *"c:\photos\college\trip\museum*.jpg"* refers to all photos of a museum from a college trip. In this case, it is not possible to store that photo in another directory say *"c:\photos\museum*.jpg"* without copying the file. This severely limits the searching capabilities in a file system.

The user is faced with the dilemma of which directory best represents the context of current file. While storing, the file is identified by its file name alone, which serves as its identifier. For searching a particular file, the user has to accurately remember the path and file name. A file cannot be searched by any other information relating to its context. Creating the directory structure is based on the users organizational skills. Searching or browsing through someone elses data is tricky as the organization is different for every user.

Previous approaches [1] to such problems provided symbolic links and aliases as an incomplete answer. Symbolic links become redundant when the target file paths are changed. Similarly, aliases may become redundant or may not function properly with certain programs. Working with such solutions requires advanced skills on the users part. Keyword based searches which extract metadata from files were brought to fore by Apple's Spotlight [3] and Google's Desktop Search [2]. Both function only on limited file types and do not allow manual categorization.

This led to the development of semantic file systems, containing categorization of files based on context. It provides access to files by using categories formed from extracting metadata. It is similar to how music files can be searched by artist, genre, album etc. However, this presents a limitation on the amount and capabilities of what metadata can be extracted from a file. Virtual directories [11] are used to represent data from the file system. These directories do not have a permanent listing and the user has to explicitly query for data. There have been several implementations based on semantic file systems. However, they have several limitations in usability. Most of the projects are based only on a few key points, such as limitations over file types.

Our project thus is to create a semantic solution to the problems and shortcomings of traditional file systems while covering the limitations of other implemented projects.

3.1.1 Project Scope

This project is a virtual file system capable of storing semantics with which it facilitates the finding of relevant information.

1. Information is stored in tags, which are extracted from a files metadata. This information may be generated implicitly by the system or supplied explicitly by the user.

2. The validity of information is based on the users level of organizing things.

3. The system is currently designed to extract metadata from a limited set of popular file types for audio, video and images.
4. The modular architecture allows for plugins to be added which can add additional functionality, and recognition for more file types. This allows the project to be extended and modified according to the functionality required.
5. The level of awareness generated by the system is based on the frequency of access and input provided by the user.
6. The current implementation is based on the Linux kernel. Future implementations can be extended to other platforms and devices.
7. As the system is an virtual entity, it does not need extensive modifications to be ported to other file systems and operating systems.

3.1.2 User Classes and Characteristics

The system can be used by three types of users:

1. General user :
   Uses the system without any complex modifications in the system.
2. Advanced user :
   Understands the system and creates rules and automation's based on personal needs.
3. Developer :
   Uses the API provided and develops modules that extend the system.

Only advanced users utilize the semantic nature of underlying filesystem to the fullest. This does not create any blocks for the general user, who can also use Kwest satisfactorily. Developers are a different group of users who can extend Kwest through modules. These modules can modify or define additional behavior for the system for specific file types.

3.1.3 Operating Environment

- Kwest requires FUSE [15] version 2.8.6 and above.
- Kwest can run on any Linux installation which contains required versions of FUSE.
- Furthermore, since kernel version 2.6.14, FUSE has been merged into the mainstream Linux kernel tree. As a result Kwest can run on any Linux distribution created from Kernel version 2.6.14 or above.
- Kwest is a virtual filesystem mounted to a folder or a loop device.
- It is the responsibility of the user that mounting and unmounting of the system be done with standard rules and precaution.

3.1.4 Design and Implementation Constraints

Kwest has the following constraints:

1. Kwest uses FUSE to manage userspace filesystems. Hence, access is limited to the executing userspace for the program.

2. Since the entire application is executed in userspace only, there cannot be interaction with the kernel directly.
3. Although Kwest implements a virtual filesystem which is accessible to all entities, we have implemented the system with command line as the primary interface. Other file managers such as Nautilus can only browse but not tag files. This limitation can be addressed with plugins or additional modules built for the specific file manager.
4. The SQLite database is an integral part of Kwest and is contained in a single file. It is vital for the system that integrity of the database is maintained.

Design constraints:

1. Currently, the auto-tagging feature has been limited to common and popular file types such as audio (mp3, wav, etc.), images (jpeg, png, etc.) and video (mp4, avi, etc.). This functionality can be extended with modules or through special tools made specifically for this purpose.
2. The amount of information visible through common filesystem commands (e.g. ls - list contents) is a limitation for Kwest. We cannot show tagging information through these interfaces. Alternate methods for this can be implemented keeping the end user in mind.

3.1.5 Assumptions and Dependencies

Assumptions:

1. Users of this software are aware of how semantics are used to categorize information.
2. Users can recognize or identify appropriate tags in relation to files.
3. It is assumed that the user is well versed in organization information and uses Kwest as a tool rather than an assistant.
4. The user has the required privileges / rights to run Kwest and all its operations.

Dependencies:
Kwest uses several external libraries to extract metadata from popular formats for audio, images etc. TagLib, EXIF etc. which are required at compilation time. These enable the system to handle metadata extraction for popular file formats.

3.2 System Features

3.2.1 Tags

1. Manual Tagging
   Manual tagging is the basis of semantics in Kwest. The user can assign any tag to the files in Kwest. These tags are then stored internally in a database. The user can create new tags or use tags already defined by the system. Total freedom is given to the user to organize data.
2. Automatic Tagging
   Kwest also features automatic tagging of files. The user can define certain rules under which files will be assigned tags. The system will implement those rules for

all files satisfying the defined constraints. This would prevent repetitive tagging operations for the user.

3. Importing tags
   Certain popular file formats such as mp3, jpeg etc. have metadata embedded in them. Kwest supports such popular format and uses this metadata to automatically assign tags to the files. This feature enables the user to collectively classify and store the data under these tags.

3.2.2 Database

1. Consistency
   Kwest uses an internal database to store and manage data. It is vital that the database always remains consistent. Kwest uses logging mechanisms to ensure that operations on the database always reach an endpoint.

2. Access
   The database is included in the same directory as the Kwest executable. The files are not locked down or are access restricted. Other applications, modules or tools can access the database. However, this feature is made available with the understanding that the integrity of the database will be maintained always.

3.2.3 Relation with existing data

1. Importing semantics
   Users already have certain organizational structures in the way they store data in filesystems. Kwest imports these semantics by converting the storage hierarchy to tag-based hierarchy. This allows the entire filesystem to be imported into Kwest along with the users previous organization structure.

2. Reflecting changes to filesystem
   When users carry out certain changes in Kwest such as copying files, deleting files etc., these changes are virtual and do not affect actual filesystems. However, Kwest can enforce these operations on the real files in certain cases.

3.2.4 Exporting semantics

1. Export filesystem
   As the entire filesystem exists as a virtual entity, Kwest provides the export feature. Where the filesystem can be exported to another system where the data can be imported by another instance of Kwest.

2. Export tagged files
   It is also possible for the user to export data under certain tags to an external location. The semantic organization showed by tags is converted to actual directories and files are then copied to these directories. This way the user can export Kwest semantics and data to outside locations.

3.2.5 Modularity

1. Modules as plugins
   Kwest is an extendible system. It can use external modules to increase functionality

or to modify existing operations. Support for using modules is built into Kwest right from the design stage.

2. Support for developers
   Kwest provides support to developers by providing access to all internal features and database. The API layer allows developers to easily supplement internal operations with their modules.

3.3 External Interface Requirements

3.3.1 User Interfaces

The user can use the system through the command line. The system mounts a virtual file system which the user can use to navigate through. If the file explorer/browser supports virtual file system, the user can use that to navigate through the files.

3.3.2 Hardware Interfaces

No hardware interfaces used as the file system exists as a virtual entity. The system can run in any environment capable of running Linux. Recommended minimum configuration:

- Processor : Any architecture (Linux supported) 500MHZ
- Ram : 128MB
- HDD : 100MB free space for database

3.3.3 Software Interfaces

- FUSE [15]
  Kwest uses FUSE to run the file system code in user space without editing the kernel code. The FUSE module provides only a "bridge" to the actual kernel interfaces. Major file system operations are defined by FUSE and forwarded to Kwest for implementation.
- SQLite [16]
  Kwest uses SQLite database to store data. In contrast to other database management systems, SQLite is not a separate process but an integral part of Kwest. Database is accessed and modified for most of the operations performed by Kwest.

3.3.4 Communication Interfaces

Kwest can be accessed like any other file system via Command line or file managers like Nautilus, Thunar etc.

3.4 Nonfunctional Requirements

3.4.1 Performance Requirements

- Response Time
  The response time for any action on the filesystem or the database should be reasonable under normal operational circumstances.

- Capacity
  Kwest can be used by any user with sufficient permissions to initialize the filesystem. Subsequent operations like read, write, modify etc. are restricted by user permissions similar to normal filesystems.
- Scalability
  Kwest provides suggestions and includes automated tagging rules for Audio, Video and Images. It also allows manual tagging of files by the user. Various modules can be further added for recognizing and categorizing other file types.

3.4.2 Safety Requirements

- Power Failure
  The system maintains a log file for database. It prevents data corruption by committing data that has been fully written to the log. This helps in preventing data corruption.
- Data Loss
  If any file is accessed which is mentioned in database but deleted form the system then it is removed from the database and appropriate message is displayed to user.
- Access Rights
  The system checks if the file tagged is available to user for access. It does not allow files to be included which cannot be access by the user.

3.4.3 Security Requirements

Kwest can be used only by a single user with sufficient rights to execute the software. No other user will have permissions to modify tags and files in the system.

3.4.4 Software Quality Attributes

- Availability
  The System shall be available from mounting the filesystem until its unmounting.
- Updatability
  The system shall allow for addition or deletion of files under tags as well as tags.
- Reliability
  - The system shall save new tags created by active user.
  - The system shall save the file path of an active user to database whenever new files are added to a tag.
  - The system shall maintain a log file which records every operation on database.
  - The system shall modify database whenever tags or files are deleted.
  - The system shall remove file entries from database whenever it is unable to access them.
- Portability
  The system is implemented on Linux. The current implementation is on Ubuntu 12.04. It is compatible with various other Linux distributions like Fedora, Red Hat, etc.

- Testability
  New modules designed to be added to the system must be tested to check if they are compatible with the input-output format of the system.
- Usability
  The system does not have a large learning curve as the user deals with commands common to all file systems. Documentation provided for the system will include user manuals, developer reference and common FAQ.

3.5 Other Requirements

3.5.1 Database Requirements

The system uses a SQLite database which is contained in a single file to store and manage metadata. The SQLite database file is stored in the standard configuration folder *"$HOME/.config/"* of the program. The system does not require SQLite runtime installed on the system for the program to interact with SQLite databases.

The database contains multiple tables having information:

1. About Files: filenames, paths, attributes etc.
2. About Tags: relation between tags and files.

3.5.2 Legal Requirements

All the libraries, programs used in this project are open sourced under GPL. SQLite is a database which is free to use, distribute or modify. The FUSE kernel module is merged with the Linux Kernel, which is open sourced and freely available under GPL. There are no proprietary or closed source products, libraries or interfaces used in this program. The project Kwest and its subsequent implementations will be open sourced under the GPL upon completion.

3.6 Analysis Model

3.6.1 Data Flow diagram

Level 0

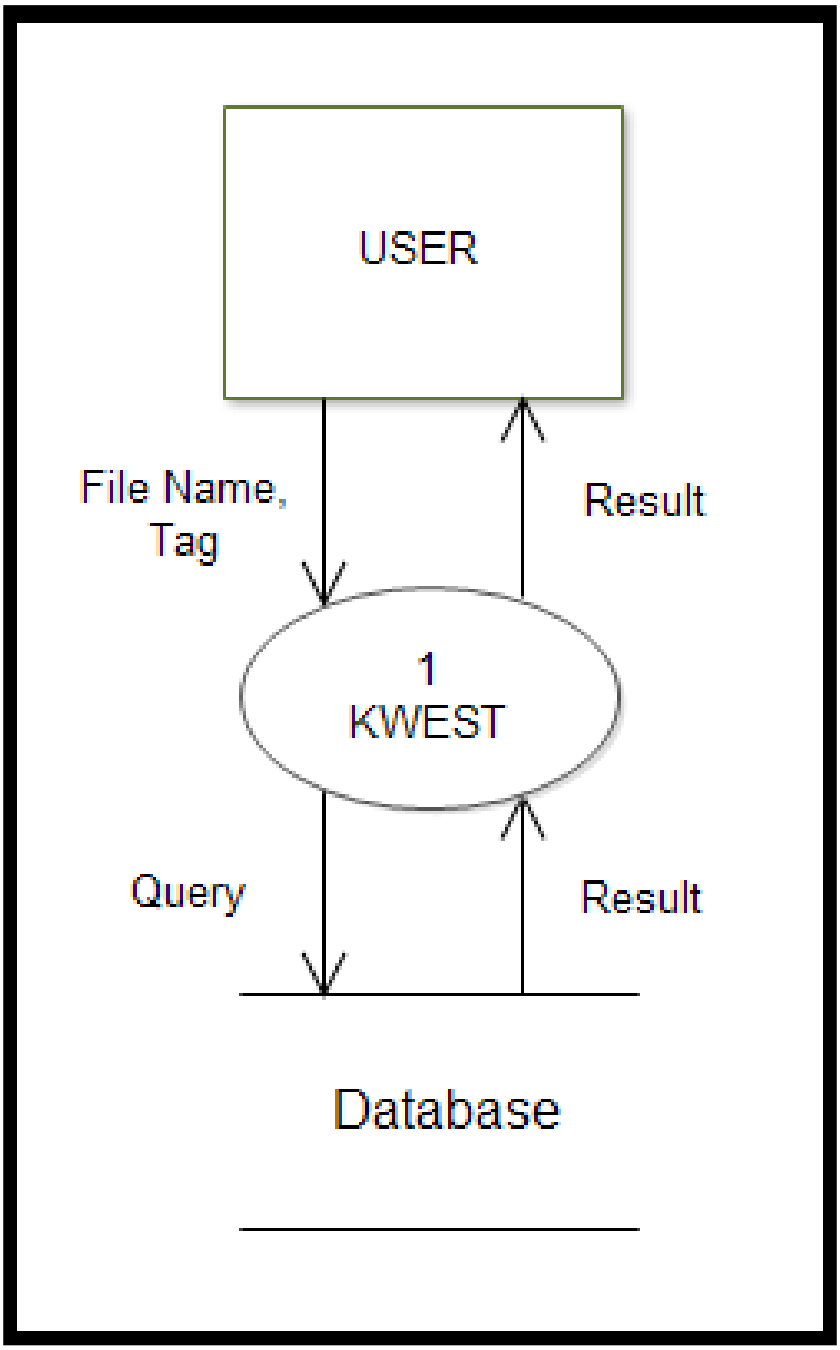

Figure 3.1: Data flow diagram - Level 0

Level 1

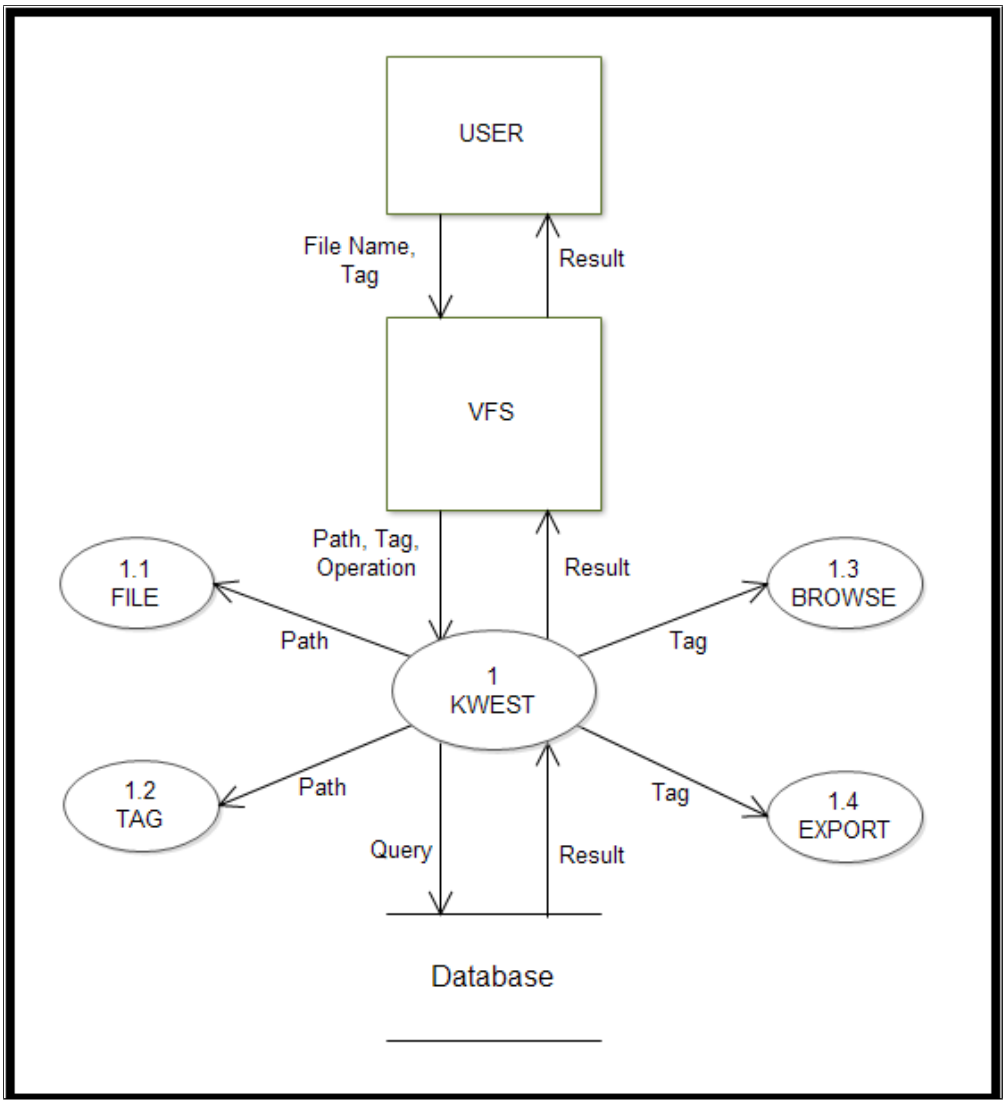

Figure 3.2: Data flow diagram - Level 1

Level 2

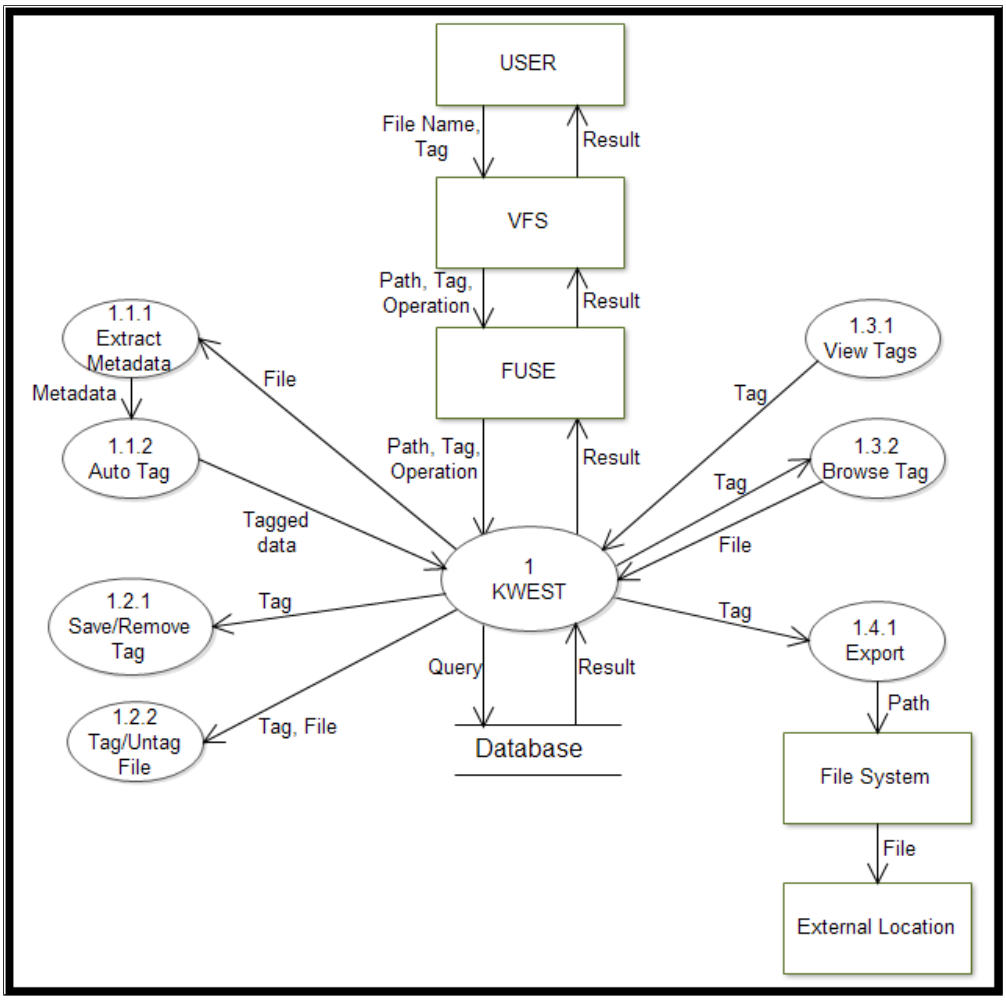

Figure 3.3: Data flow diagram - Level 2

3.6.2 Entity Relationship diagram

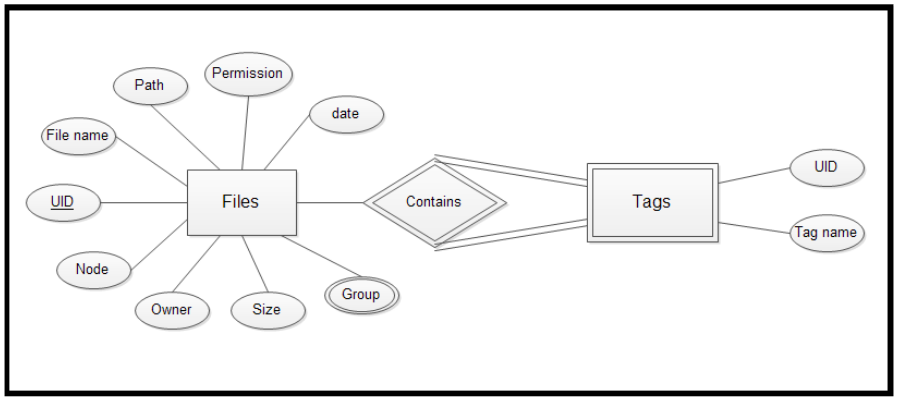

Figure 3.4: Entity Relationship diagram

3.7 System Implementation Plan

Task	Time Frame
Problem identification	1 week
Information gathering	1 week
Creating problem definition	1 week
Understanding underlying technology	2 week
Analyzing problem	1 week
Designing solution	2 week
Refining design	1 week
Creating design report	1 week
Building a stub implementation	2 week
Coding and testing cycle 1	2 week
Coding and testing cycle 2	2 week
Coding and testing cycle 3	2 week
Coding and testing cycle 4	2 week
Rigorous testing	2 week
Debugging and refining	4 week
System testing	2 week
Creating implementation report	2 week
System demonstration	2 week

Chapter 4

SYSTEM DESIGN

4.1 System Architecture

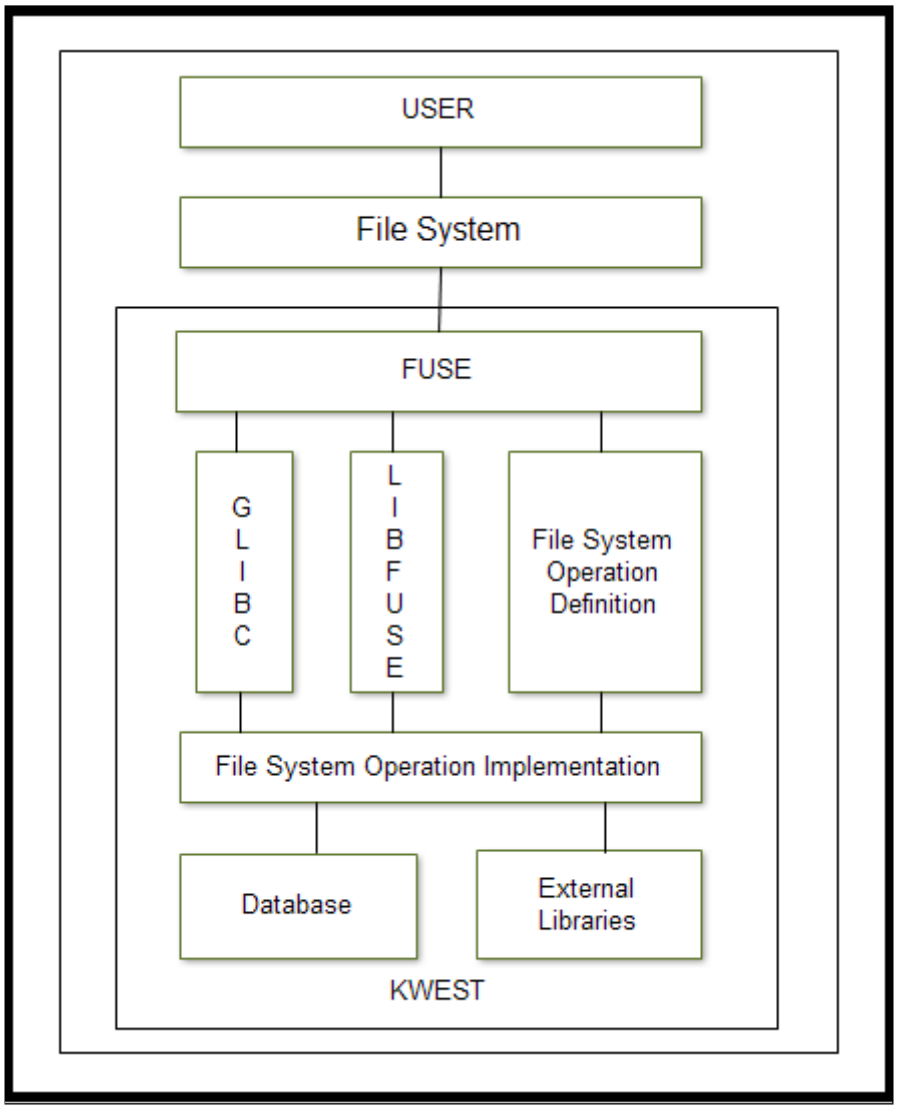

Figure 4.1: System Architecture

4.2 UML Diagrams

4.2.1 Use-case diagram

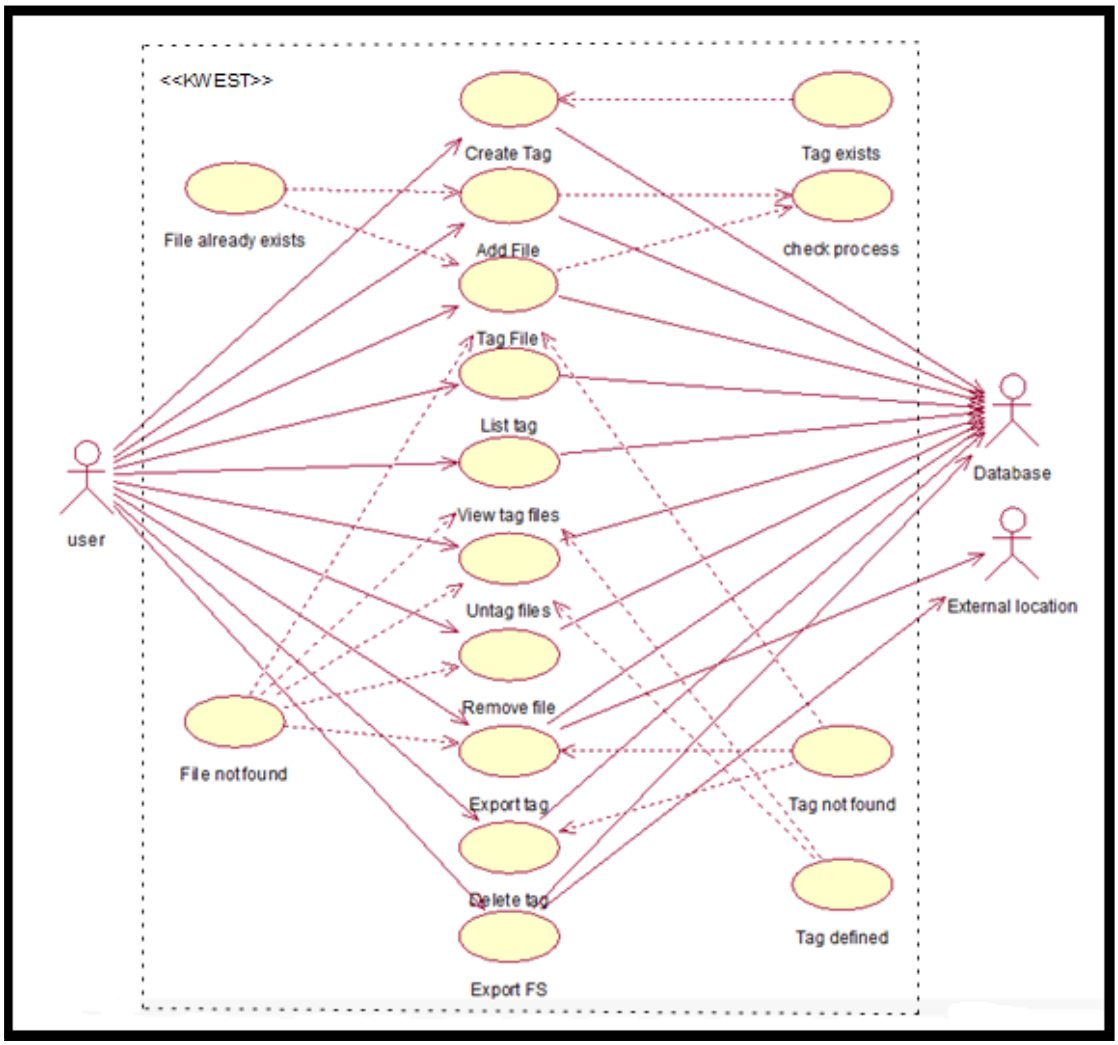

Figure 4.2: Use Case diagram

4.2.2 Component diagram

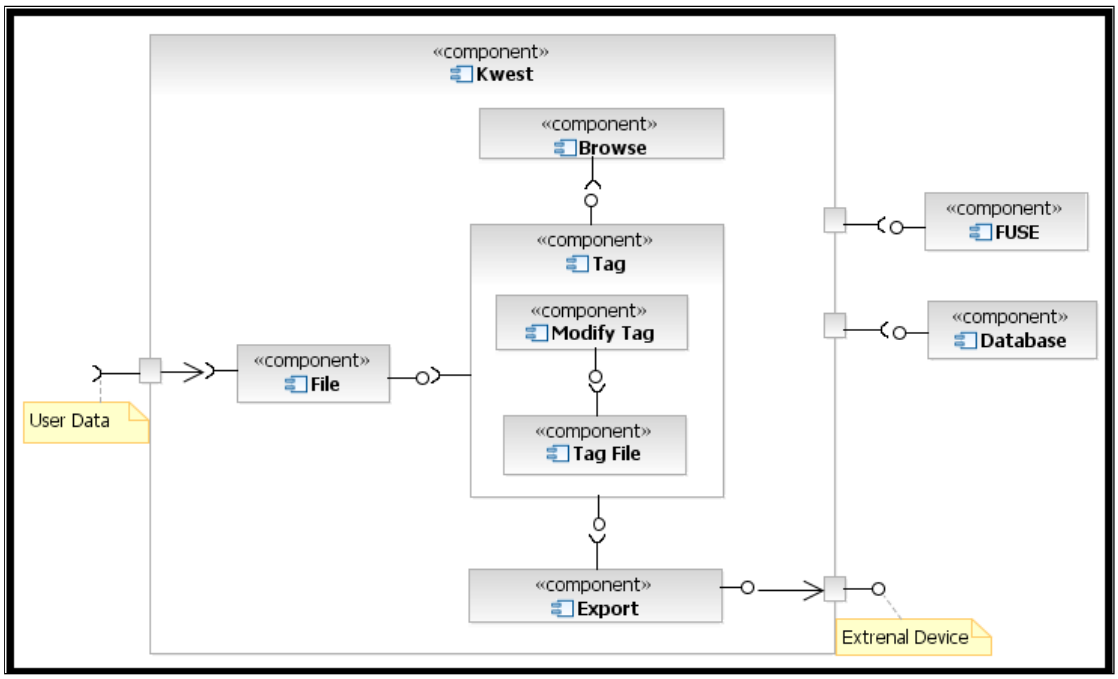

Figure 4.3: Component diagram

4.2.3 Deployment diagram

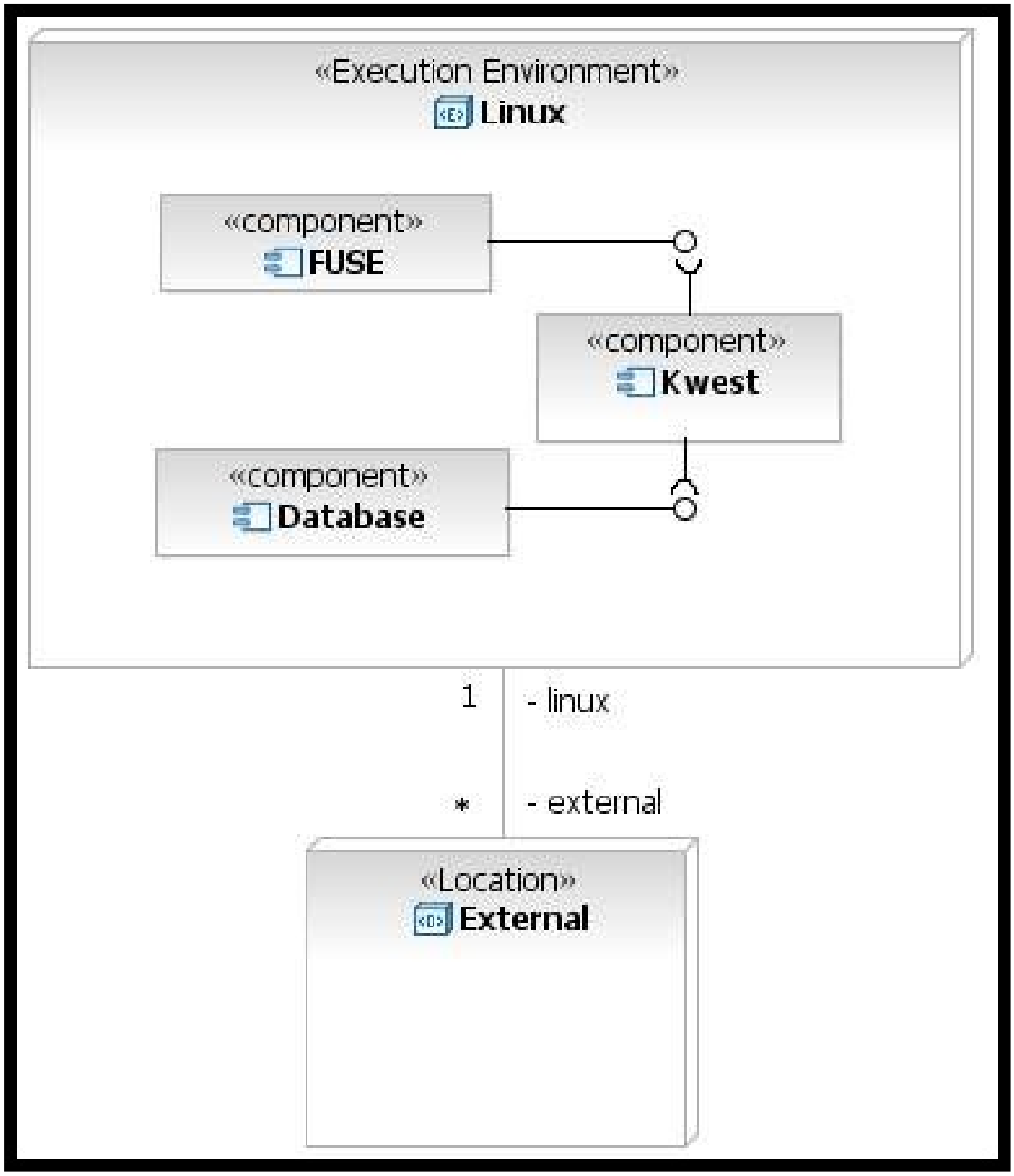

Figure 4.4: Deployment diagram

Chapter 5

TECHNICAL SPECIFICATIONS

Kwest is implemented as a virtual file system. It manages the organization of data while the storage is maintained by the underlying file system. It uses virtual directories to allow representation in the traditional model of directories and files. Tags are used to store metadata attributes of files. Searching is based on these tags, and the results are listed as virtual directories. The user can browse through the directories to search for relevant files. Kwest supports automated as well as manual tagging to allow user's choices for categorization of information. It also offers suggestions to the user in the form of related tags.

Kwest will use multiple tagging to categorize files based on its context. Its primary functionality is to automatically organize files in a transparent and seamless manner so as to facilitate easy retrieval. The virtual directories created in Kwest will be superficial and won't change or modify the location in which the data will be initially stored. It will only provide links and assign tags under which data can accessed.

Kwest will also provide a feature which will either automate the classification of the files or suggest possible tags to the user if the user wants to classify files manually.

Kwest can efficiently manage and navigate gigabytes of data. It can be used on a large scale to manage and access data in organizations where the need for effective and fast retrieval of relevant data is needed.

The Project is being implemented using loadable kernel module known as FUSE [15]. It is a kernel module in UNIX based operating systems that lets non-privileged user to create their own file system without editing the kernel code. Being open sourced, this module is available for public use. In the current versions of Linux, this module is included in the kernel itself.

The query processing and programing will be done using SQLite [16]. It is a relational database contained in a small C programming library. In contrast to other database management systems, SQLite is not a process that is accessed from the client application, but an integral part of it. This is also open source and is freely available.

Thus the cost for developing Kwest will be minimal and hence will be feasible enough without the need for large capital.

1. **Filesystem in Userspace (FUSE)** [15]
   FUSE is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces.
   Version used: 2.8.6

2. **SQLite** [16]
   SQLite is a relational database management system contained in a small C programming library. In contrast to other database management systems, SQLite is not a separate process that is accessed from the client application, but an integral part of it. It implements a self-contained, zero-configuration, transactional SQL database engine which can be embedded in applications.
   Version Used: 3.7.13

3. **Language for implementation: ANSI C**

4. **Compiler: GCC**
   The GNU Compiler Collection (GCC) is a compiler system which provides front ends for various languages including C. It provides optimizations, debugging and other features to help program development. GCC is often chosen for developing software that is required to be executed on a variety of platforms.
   Version used: 4.7.1

5.1 Advantages

- Kwest sorts files automatically while giving users the flexibility to organize files in their own way.
- Modular design provides scope to add functionalities later. This refines the systems portability and extendability.
- Kwest provides an export feature that allows the user to save contents of search results to an external location. Files are exported as a traditional hierarchy supported by generic file systems.
- It allows both, automated as well as manual classification of files based on metadata tags and file attributes.
- The user is in complete control over the operations carried out by the system.
- Metadata can be generated for all file types.
- Files are also categorized based on use, creation, size, etc.
- Virtual Directory Listings can be created at runtime to address searches.
- System can ported to any file system being a virtual file system.
- The user is more likely to find related information in fewer searches.
- The user does not have to remember path of file, only the context.
- System does not require advanced skills to use.
- A file can be found under any one its related tags.
- Virtual Directories enable files to be displayed under more than one directory.
- Virtual Directory listings provide a traditional layout for the user while maintaining the semantic tagging associated with files.
- Support for plugins allows users to add functionality to the project which can further be tagged under appropriate context.
- As it is a virtual file system, Kwest can be used with existing file systems already present on the machine.
- Querying for data is helpful when complex searches are to be implemented.

5.2 Disadvantages

1. Kwest depends on external modules for content recognition and advanced metadata extraction.
2. Since it is an userspace application, it is not always available across the system.
3. Filesystem operations are restricted to only those defined by FUSE (which are which most common filesystems contain)
4. Limitations in representing semantics (tags and relations) through current file systems.
5. The internal database does not contain any caching mechanism of its own.
6. Database size is restricted to 2TB in most cases.
7. Support for tag related operations is not available in GUI form (file managers) by default.

5.3 Applications

- Support for plugins allows users to add functionality to the project which can further be tagged under appropriate context.
- As it is a virtual file system, Kwest can be used with existing file systems already present on the machine.
- Kwest can be used by general as well as advanced user.
- Modular design provides scope to add functionalities later. This refines the systems portability and extendability.
- Kwest provides an export feature that allows the user to save contents of search results to an external location. Files are exported as a traditional hierarchy supported by generic file systems.

Chapter 6

APPENDIX

6.1 Appendix A : Mathematical Model

The relationship between files and tags can be represented by using Set theory. Set theory is the branch of mathematics that studies sets, which are collections of objects. The following mathematical model represents the working of this filesystem.

The following dynamic and variable sets are defined as,
F : Set of Files
T : Set of Tags
S : Set of Tags in query ($S \subseteq T$)

6.1.1 Relation between Files (F) and Tags (T)

$$R = \{(f,t) \mid f \; has \; tag \; t; f \in F, t \in T\}$$

Here R defines the relation between a file f and its tag t where $R \subseteq F \times T$. This relationship is *many-to-many*. That is a file can have many tags, and a tag can describe many files.

6.1.2 Association between Tags (T)

Using discovered associations, we can form various relations between tags. These *tag-to-tag* help in displaying related information. For any two tags there exists a distinct relation between them given by r. The function $X_r(A,B)$ returns the relation between two tags. Associations can be broadly categorized as:

- $A \sim B$: A and B are not directly related, but there *may* exist some indirect relation between them.
- $A \succ B$: A and B are directly related, where A always has a path leading to B. This relation is similar to $A \subset B$.
- $A \bowtie B$: A and B are not directly related, but B supplements additional information related to A.
- ϕ : This relation states that there does not exist any relation between the two tags.

6.1.3 Operations

$$g(f) = \{t : fRt\}$$

g is an operation which takes input as a file f and returns the set of tags ($t \in S$) related by R to that file.

$$h(t) = \{f : fRt\}$$

h is an operation which takes input as a tag t and returns the set of files ($f \in F_S$) related by R to that tag.

6.1.4 Storing Tags and Files

The relation R is stored as a set of ordered pairs (f,t), where $R \subseteq F \times T$. The operations g and h operate on these ordered pairs and return mapped or matched elements. A relation which has to be added must be represented in the form of of ordered pair (f,t). Storage of all relations is given by $F \times T$ where ordered pairs exist according to $R = \{f \in F, t \in T \mid fRt\}$.
For eg. we have the sets and their relations as:

$$F = \{f_1, f_2, f_3\}, T = \{t_1, t_2, t_3\}, R = \{f_1 R t_1, f_2 R t_2, f_3 R t_1, f_1 R t_3\}$$

Then we store this relation by its ordered pairs given by:

$$R = \{(f_1, t_1), (f_2, t_2), (f_3, t_1), (f_1, t_3)\}$$

6.1.5 Extraction of metadata

The extraction of metadata is defined by the function X_E which takes a file $f(f \in F)$ and returns a set of tags $(T_E \subseteq T)$ that form the relation $(fRt : t \in T_E)$.

$$X_E(f) = T_E \in 2^T$$

Addition of new information (metadata, tags) is done as:

$$if(t \notin T) then (T \leftarrow T \cup \{t\})$$

We then store this relation as a ordered pair $\{(f,t) \forall t \in T_E\}$. At the end of this operation $T_E \subseteq T$ will hold true.

6.1.6 Importing Semantics

The existing file-directory structure can be imported to the system and represented in the form of tags and files. We define:
F_H : Set of *Files* on hard-disk which are not represented in system.
D_H : Set of *Directories* on hard-disk which are not represented in system.

Then for every file stored within a directory d, the relation R is expressed as (fRd). When importing semantics we create the ordered pair (f,d) given by the relation $(fRd, \forall d \in D_d)$. Where D_d contains the directory the file is stored in, as well as every parent directory of that directory itself.
Directories also contain sub-directories which we store in the form of tag relationships. We represent them as: $\forall (d_1, d_2) \in D_H$, if $d_2 \subset d_1$ (d_2 is a sub-directory of d_1) store the relation as-

$$d_1 \rightarrow d_2$$

6.1.7 Queries

A query operation on a single tag is expressed as:

$$q(t) = F_S \; where \; \{f \in F_S \mid h(t) = f\}$$

The general form of a query is a string which contains tags and operators. For example, we have two tags (t_1, t_2) and operator σ. The query Q can be defined in terms of q as:

$$Q(t_1, \sigma, t_2) = q(t_1)\, \sigma\, q(t_2)$$

The operation σ can any one of Union $\cup$, Intersection $\cap$, Symmetric difference $\ominus$ etc. If no operation is explicitly mentioned, by default Intersection $\cap$ is performed. Therefore, the default form of query becomes

$$Q(t_1, t_2) = q(t_1) \cap q(t_2)$$

A default Query over a set of tags can be expressed in terms of $q(t)$ as:

$$Q(S) = \bigcap_{i=1}^{n} q(t_i)$$

A parser is used to parse the query string into tokens of the form (t_1, σ, t_2). It then calculates the results for each token as $Q(t_1, \sigma, t_2)$. The results for the entire query are calculated as:

$$Q(S) = \prod_{i,j=1}^{n,m} q(t_i \sigma t_j)$$

An example: Let A and B any two files having the following tags-
$tag(A) : files\{photo_1, photo_2, doc_1\}$
$tag(B) : files\{photo_1, doc_1, ppt_1\}$

1. Union
   The union of $\{photo_1, photo_2, doc_1\}$ and $\{doc_1, photo_1, ppt_1\}$ is the set $A \cup B = \{photo_1, photo_2, doc_1, ppt_1\}$.
   Use : For displaying all files in multiple tags.

2. Intersection
   The intersection of $\{photo_1, photo_2, doc_1\}$ and $\{doc_1, photo_1, ppt_1\}$ is the set $A \bigcap B = \{photo_1, doc_1\}$.
   Use : To get common files under multiple tags. When certain files under A also have the tag B, A and B are assumed to be related. In such conditions, browsing the directory containing tag A, there exists a set of files also tagged under B denoted by $A \cap B$. These files are made available in a sub-directory with the tag-name B.

3. Set Difference
   Set difference gives tags in a set which not common with another set.
   $A \setminus B = \{photo_1, photo_2, doc_1\} \setminus \{doc_1, photo_1, ppt_1\}$ is $\{photo_2\}$ is different from $B \setminus A = \{doc_1, photo_1, ppt_1\} \setminus \{photo_1, photo_2, doc_1\}$ is $\{ppt_1\}$.
   Use: For checking consistency of database and filtering results.

4. Symmetric difference
   For the sets $\{photo_1, photo_2, doc_1\}$ and $\{doc_1, photo_1, ppt_1\}$, the symmetric difference set is $\{photo_2, ppt_1\}$.
   Use: For checking consistency of database.

6.2 Appendix B : Testing of Data

Testing of the system will be done on the following points:

6.2.1 Storing the relation R:

The working of the system depends on correctly storing the relation R. These tests check whether the relations are stored and represented correctly.

1. For any file f associated with tag t there should exist an ordered pair (f,t).
2. For a file f associated with tags $S = \{t_1,t_2,t_3\}$, the operation $g(f)$ should return exacty S.
3. For a tag t containing files $F = \{f_1,f_2,f_3\}$, the operation $h(t)$ should return exactly F.

6.2.2 Relation between tags:

1. The relation r between two tags t_1,t_2 is given by function $X_r(t_1,t_2) = c$.
2. The relation r is distinct i.e. there exists only one relation between any two tags. If contradictions arise where more than one relation is present between two tags, then all those relations must be made void and $r = \phi$. The user can then explicitly specify which relation should be created between those two tags.
3. Extensive testing must be done on relations to determine which r needs to be set under certain conditions.

6.2.3 Extraction of Metadata:

Let M be the set of all metadata tags t for file f. The function X_E represents an algorithm to extract t from f. It returns a set T_E such that $\{t \in T_E \mid fRt\}$ and $(T_E \subseteq M)$. To test the efficiency of the function, or the effectiveness of it, we compare the cardinality of the generated set T_E with the set of Metadata M. The efficiency can be calculated by

$$\text{Efficiency}\,\varepsilon = \frac{\mid T_E \mid}{\mid M \mid}$$

Using ε we can compare algorithms and their efficiency. Appropriate algorithms can be chosen for various file types so that efficiency of the entire system remains high. For e.g.
$X_{E1} : \{\varepsilon = 0.7$ for audio, $\varepsilon = 0.4$ for images $\}$
$X_{E2} : \{\varepsilon = 0.6$ for audio, $\varepsilon = 0.6$ for images $\}$
Then we have the following options:

1. X_{E2} is a better choice as it yeilds a more consisten efficiency.
2. X_{E1} is used only for audio and X_{E2} is used only for image extractions.

6.2.4 Queries and their results:

Queries are parsed into tokens of the form (t_1,σ,t_2). We need to test whether σ returns the correct results for the query. A Query $Q(S)$ is said to be successfully executed when the expected result are shown.

- The Query $q(t)$ for a single tag t should return a set of files ($f \in F_S$) through the operation $h(t)$.
- In a query if no operation is given, Intersection should be performed.
- For Query $Q(S)$ the operations should be performed from left to right unless precedence is specified by paranthesis.

Example: Let t_1, t_2, t_3 be tags having the following files:
$t_1 : \{photo_1, photo_2, doc_1\}$
$t_2 : \{photo_1, doc_1, ppt_1\}$
$t_3 : \{photo_1, doc_3\}$

1. $q(t) = F_S$ where f should follow the relation $(f \in F_S \mid fRT)$. For example, q(t_1) returns $F_S = \{photo_1, photo_2, doc_1\}$.
2. Consider a query containing tags $S = \{t_1, t_2\}$. It generates results by performing the operations $Q(S) = q(t_1)\sigma q(t_2)$ where σ is an operator.
   a) Putting $\sigma = \cup$ for Union in query $Q(S) = q(t_1) \cup q(t_2)$; the result should be $F_S = \{photo_1, photo_2, doc_1, ppt_1\}$.
   b) Putting $\sigma = \cap$ for Intersection in query $Q(S) = q(t_1) \cap q(t_2)$; the result should be $F_S = \{photo_1, doc_1\}$.
   c) Putting $\sigma = \setminus$ for Set Difference in query $Q(S) = q(t_1) \setminus q(t_2)$; the result should be $F_S = \{photo_2\}$, whereas the query $Q(S) = q(t_2) \setminus q(t_1)$ will return the result as $F_S = \{ppt_1\}$.
   d) Putting $\sigma = \ominus$ for Symmetric Difference in query $Q(S) = q(t_1) \ominus q(t_2)$; the result should be $F_S = \{photo_2, ppt_1\}$.
3. Consider a query containing tags $S = \{t_1, t_2, t_3\}$. It performs operations as : $Q(S) = q(t_1)\sigma_1 q(t_2)\sigma_2 q(t_3)$ The default order for processing is from left to right unless precedence is specified through parenthesis.
   a) For the query $Q(S) = q(t_1)q(t_2)q(t_3)$ the result will be returned as files $\{photo_1\}$
   b) For the query $Q(S) = q(t_1) \cup q(t_2) \cap q(t_3)$ the result will be returned as files $\{photo_1, doc_1, doc_3\}$

After verifying that individual queries return correct results, we must check whether $Q(S)$ runs correctly as well. This is done by seperating the query into tokens, and calculating their results in turn. It must be verified that results are correct and have not been mis-intepreted through tokenization of the query.

6.3 Appendix C : Papers Published

6.3.1 Kwest : A Semantically Tagged Virtual Filesystem

Kwest

A Semantically Tagged Virtual File System

Aseem Gogte, Sahil Gupta, Harshvardhan Pandit, Rohit Sharma
Department of Computer Engineering, RSCOE, Pune University
Pune
aseem2691@gmail.com
euphoric.sg@gmail.com
hpandit86@gmail.com
rohitc.dude@gmail.com

***Abstract*— The limitation of data representation in today's file systems is that data representation is bound only in a single way of hierarchically organizing files. A semantic file system provides addressing and querying based on the content rather than storage location. Semantic tagging is a new way to organize files by using tags in place of directories. In traditional file systems, symbolic links become non-existent when file paths are changed. Assigning multiple tags to each file ensures that the file is linked to several virtual directories based on its content. By providing semantic access to information, users can organize files in a more intuitive way. In this way, the same file can be accessed through more than one virtual directory. The metadata and linkages for tagging are stored in a relational database which is invisible to the user. This allows efficient searching based on context rather than keywords. The classification of files into various ontologies can be done by the user manually or through automated rules. For certain files types, tags can be suggested by analyzing the contents of files. The system would be modular in design to allow customization while retaining a flexible and stable structure.**

Keywords*— *semantics, indexing, classification, database, tagging, virtual file system, information access, metadata

I. INTRODUCTION

Traditional file systems are mono-hierarchical and implement directory trees to categorize and store files. In such systems, directories are the only means to access particular files.

The path of a file contains directories, which refer to its context and categorization. As an example *"c:\photos\college\trip\museum*.jpg"* refers to all photos of a museum from a college trip. In this case, it is not possible to store that photo in another directory say *"c:\photos\museum*.jpg"* without copying the file. This severely limits the searching capabilities in a file system.

The user is faced with the dilemma of which directory best represents the context of current file. While storing, the file is identified by its file name alone, which serves as its identifier. For searching a particular file, the user has to accurately remember the path and file name. A file cannot be searched by any other information relating to its context. Creating the directory structure is based on the users organizational skills. Searching or browsing through someone else's data is tricky as the organization is different for every user.

Previous approaches [1] to such problems provided symbolic links and aliases as an incomplete answer. Symbolic links become redundant when the target file paths are changed. Similarly, aliases may become redundant or may not function properly with certain programs. Working with such solutions requires advanced skills on the user's part. Keyword based searches which extract metadata from files were brought to fore by Apple's Spotlight [3] and Google's Desktop Search [2]. Both function only on limited file types and do not allow manual categorization.

This led to the development of semantic file systems, containing categorization of files based on context. It provides access to files by using categories formed from extracting metadata. It is similar to how music files can be searched by artist, genre, album etc. However, this presents a limitation on the amount and capabilities of what metadata can be extracted from a file. Virtual directories [11] are used to represent data from the file system. These directories do not have a permanent listing and the user has to explicitly query for data. There have been several implementations based on semantic file systems.

However, they have several limitations in usability. Most of the systems are based only on a few key points, such as limitations over file types.

Our aim thus is to create a semantic solution to the problems and shortcomings of traditional file systems while covering the limitations of other implemented systems.

II. RELATED WORK

Over the years, organizing and retrieving information accurately and efficiently has attracted lot of attention. While few have been successful, a number of innovative implementations [1] have emerged. The idea of using a file's semantics as the means to categorize it has been around for quite some time. This section discusses the various implementations made in the field of semantic file system. An efficient implementation of keyword based searching was brought to the desktop by Google's Desktop Search [2] and Apple's Spotlight [3]. Both allow efficient and quick file retrieval based on keywords. They support many file types and have a simple interface which attracts a large number of users. However, both of them are limited to returning search results without any way to organizing contents. In addition, they do not provide any provision to the user for classification of data.

This limitation prevented the user from having a personalized way to retrieve data stored by them.

Semantic systems depend on data stored inside the files rather merely relying on an file's attributes. Most implementations use common methodologies like content recognition [4], tagging [5], extracting metadata, etc. to categorize files by using various algorithms.

"Semantic File System" [6], as developed by O'Toole and Gittord in 1992, provides access to file contents and metadata by extracting the attributes using special modules called "transducers". It was one of the very first attempts to classify files by semantics using metadata. Its biggest drawback was the need for file type specific transducers which were necessary to extract meta information and content from the file. Also, the user does not have any say in what kind of category the file is classified under. This drawback makes it an unattractive option to the general user. It was decided during designing Kwest, that it is necessary to involve the end-user in the tagging process. This allows each user to have their own personal way of classification and organization of files.

NHFS (Non Hierarchical File System) [7] was a system developed by Robert Freund in July 2007. It allows the user to place any file into any number of directories. Likewise, any directory can be placed into as many directories as required. NHFS therefore allows one to create a non-hierarchical structure with poly-hierarchically connected files. This allows for a powerful metaphor of finding a file in any of the category (directory) it could be stored under. Therefore, we decided to retain this feature by using tags in place of actual directories. Tags are associated with files and other tags as well. Thus, a tag may be placed under multiple tags allowing a relationship to be defined between them. This analogy is much more powerful than restricting files to actual directories. Using tags prevents duplication and redundancy, making it an efficient implementation.

A more recent implementation is Tagsistant [8], which is a semantic file system that also attempts to organize files using tags. It interacts with the Linux kernel using the FUSE module. Under Tagsistant, directories are considered to be equivalent to tags. As a consequence, creating a directory is creating a tag and putting a file inside a directory means tagging that file. After you have tagged your files, you can search all of them by using queries. Queries are just paths where each element is either a directory or logical operators. The entire system has a modular design and uses SQLite. However, it suffers from some speed issues and the lack of SQL indexes. Major flaws of this design were high consumption of inodes on real file systems and high computational time which was required to fulfill each request. Most of the features of Tagsistant were decided to be included in Kwest. These were modular design, SQLite repository, tagged structure, etc. which enhance the semantics of a file system. However, care must be taken to prevent the occurrence of similar drawbacks.

Another implementation called Tagster [9], is a peer-to-peer tagging application for organizing desktop data. It is platform independent and is implemented in JAVA. Multiple files and also directories can be tagged through its interface. The selected directories are recursively examined and all files contained within them are tagged. The GUI for a Linux system consists of three main areas. Namely - "Tag view": which displays a list of tags, "Resource view": which lists resources that have the currently selected tags assigned and "User view": that displays a list of users that have tagged the currently selected resource with some selected tag. It also includes GUI support for Windows with some unresolved issues. However, it lacks auto classification of data due to which several common tags may be generated for each user increasing the database size.

III. PROPOSED SYSTEM

Kwest is a virtual file system that is designed to help users organize information using the familiar hierarchical file/directory structure. It aims at providing a feasible solution towards efficient contextual storage and searching of information. It implements a semantic file system which structures data according to their context and intent. This allows the data to be addressed by their content and makes relevance in searching an efficient operation.

The system extracts metadata into tags and stores it in a relational database. These tags can be file attributes such as size, type, name etc. as well as extracted metadata such as author, content title, etc. Categorizing files by metadata allows linking a file in multiple ways while being able to search it using its context. This enables the users to find relevant information in as few searches as possible.

Assigning tags can be managed by automated rules and manual inputs. This makes the semantics mold according to the user's perspectives and helps make information relevant to the person managing it. The modular architecture of the system allows for plugins which can extend the functionality. For example a plugin to add more detection capabilities for certain file types will enhance the metadata extraction on those files. This makes the system highly customizable to power users. The automated rules help automate tasks and data categorization based on user inputs.

Virtual directories are used to display stored files in a semantic organization. Search results are displayed through dynamically created listings, which correspond to semantic segregation. The entire implementation is based on a virtual file system which manages only the data organization. The underlying file system takes care of storage. This allows it to be ported in future to any file system.

Finally, the system is implemented using open source technologies, which greatly reduce the cost and compromises associated with paid software. Thus the system aims to address the current shortcomings of relevant information access and storage by creating a virtual semantic file system which manages the data and provides search information based on semantics. The major design features are described in this section.

A. Tags

1) Manual Tagging:

Manual tagging is the basis of semantics in Kwest. The user can assign any tag to the files in Kwest. These tags are then

stored internally in a database. The user can create new tags or use tags already defined by the system. Total freedom is given to the user to organize data.

2) Automatic Tagging

Kwest also features automatic tagging of files. The user can define certain rules under which files will be assigned tags. The system will implement those rules for all files satisfying the defined constraints. This would prevent repetitive tagging operations for the user.

3) Importing tags

Certain popular file formats such as mp3, jpeg etc. have metadata embedded in them. Kwest supports such popular format and uses this metadata to automatically assign tags to the files. This feature enables the user to collectively classify and store the data under these tags.

B. Database

1) Consistency

Kwest uses an internal database to store and manage data. It is vital that the database always remains consistent. Kwest uses logging mechanisms to ensure that operations on the database always reach an endpoint.

2) Access

The database is included in the same directory as the Kwest executable. The files are not locked down or are access restricted. Other applications, modules or tools can access the database. However, this feature is made available with the understanding that the integrity of the database will be maintained always.

C. Relation with existing data

1) Importing semantics

Users already have certain organizational structures in the way they store data in file systems. Kwest imports these semantics by converting the storage hierarchy to tag-based hierarchy. This allows the entire file system to be imported into Kwest along with the users' previous organization structure.

2) Reflecting changes to filesystem

When users carry out certain changes in Kwest such as copying files, deleting files etc., these changes are virtual and do not affect actual file systems. However, Kwest can enforce these operations on the real files in certain cases.

D. Exporting semantics

1) Export filesystem

As the entire file system exists as a virtual entity, Kwest provides the export feature. Where the file system can be exported to another system where the data can be imported by another instance of Kwest.

2) Export tagged files

It is also possible for the user to export data under certain tags to an external location. The semantic organization showed by tags is converted to actual directories and files are then copied to these directories. This way the user can export Kwest semantics and data to outside locations.

E. Modularity

1) Modules as plugins

Kwest is an extendible system. It can use external modules to increase functionality or to modify existing operations. Support for using modules is built into Kwest right from the design stage.

2) Support for developers

Kwest provides support to developers by providing access to all internal features and database. The API layer allows developers to easily supplement internal operations with their modules.

IV. System Design

Kwest is implemented using loadable kernel module known as FUSE. User may interact with kwest like any other file system via Command line or file managers like Nautilus. Data is passed on to FUSE through the virtual file system. FUSE implements the operations of file system. FUSE uses Glibc and Libfuse for performing its operations. Glibc is the GNU Project's implementation of the C standard library. It provides functions for tasks like I/O processing, mathematical computation, memory allocation, etc. Libfuse contains functions internally used by fuse to create and manage virtual file system. SQLite will be used for storing all data relevant with the file system. To extract metadata, Kwest makes use of external libraries such as Taglib, EXIF.

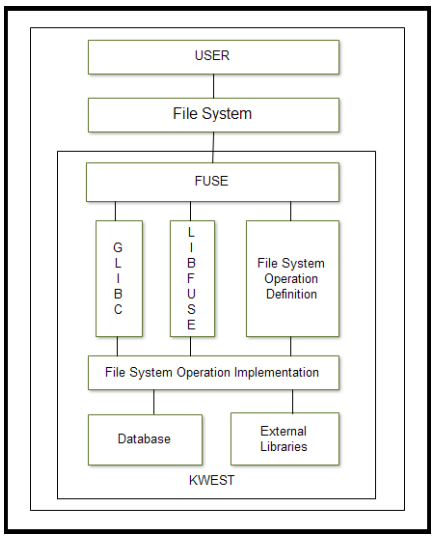

Figure 1: Design of Kewst

V. Mathematical Model

The relationship between files and tags can be represented by using Set theory. Set theory is the branch of mathematics that studies sets, which are collections of objects. The

following mathematical model represents the working of this file system.

The following dynamic and variable sets are defined as,

F: Set of Files

T: Set of Tags

S: Set of Tags in query ($S \subseteq T$)

A. Relation between Files (F) and Tags (T)

$$R = \{ (f, t) \mid f \text{ has tag } t;\ f \in F, t \in T \}$$

Here R defines the relation between a file f and its tag t where $R \subseteq F \times T$. This relationship is many-to-many. That is a file can have many tags, and a tag can describe many files.

B. Operations

$$g(f) = \{ t : f R t \}$$

g is an operation which takes input as files f and returns the set of tags ($t \in S$) related by R to that file.

$$h(t) = \{ f : f R t \}$$

h is an operation which takes input as tags t and returns the set of files ($f \in F_S$) related by R to that tag.

C. Storing Tags and Files

The relation R is stored as a set of ordered pairs (f, t), where $R \subseteq F \times T$. The operations g and h operate on these ordered pairs and return mapped or matched elements. A relation which has to be added must be represented in the form of ordered pair (f, t). Storage of all relations is given by $F \times T$ where ordered pairs exist according to

$$R = \{f \in F, t \in T \mid f R t\}.$$

For example, we have the sets and their relations as:

$$F = \{f_1, f_2, f_3\},\ T = \{t_1, t_2, t_3\},$$

$$R = \{f_1 R t_1, f_2 R t_2, f_3 R t_1, f_1 R t_3\}$$

Then we store this relation by its ordered pairs given by:

$$R = \{(f_1, t_1), (f_2, t_2), (f_3, t_1), (f_1, t_3)\}$$

D. Queries

A query operation on a single tag is expressed as:

$$q(t) = F_S \text{ where } \{ f \in F_S \mid h(t) = f \}$$

The general form of a query is a string which contains tags and operators. For example, we have two tags (t_1, t_2) and operator σ. The query Q can be defined in terms of q as:

$$Q(t_1, \sigma, t_2) = q(t_1)\ \sigma\ q(t_2)$$

The operation σ can be any one of Union $\cup$, Intersection $\cap$, Symmetric difference Θ etc. If no operation is explicitly mentioned, by default Intersection $\cap$ is performed.

VI. Conclusion And Future Work

In this paper we have proposed a system for organizing files using meta information by exploiting semantic information to provide efficient and scalable architecture. The system handles complex queries while enhancing functionality. Its novelty lies in the way it associates tags and derives rules that enables traversal based on semantics rather than path.

Currently, Kwest is in its initial stage of development. Its features are limited but its modular architecture allows plugins to be added which can add additional functionality, and recognition for more file types. This allows the system to be extended and modified according to the functionality required. The current implementation is based on the Linux kernel. Future implementations can be extended to other platforms and devices. As the system is a virtual entity, it does not need extensive modifications to be ported to other file systems and operating systems.

References

[1] Mangold. C, A survey and classification of semantic search approaches, *Int. J. Metadata, Semantics and Ontology*, Vol. 2, No. 1, 2007, Page(s): 23-34.

[2] Google Desktop Search, http://googledesktop.blogspot.in

[3] Apple Spotlight, http://developer.apple.com/macosx/spotlight.html

[4] Gopal. S, Yang. Y, Salomatin. K, Carbonell. J, Statistical Learning for File-Type Identification, *2011 10th International Conference on Machine Learning and Applications*, Page(s): 68-73.

[5] Bloehdorn. S, Grlitz. O, Schenk. S, Vlkel. M, TagFS - Tag Semantics for Hierarchical File Systems, *In Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06)*, Graz, Austria, September 6-8, 2006.

[6] Gifford. D, Jouvelot. P, Sheldon. M, OToole. J, Sematic File Systems, 13th ACM Symposium on Operating Systems Principles, *ACM Operating Systems Review*, Oct. 1991, Page(s): 16-25.

[7] Freund. R, File Systems and Usability the Missing Link, *Cognitive Science, University of Osnabruck July 2007*.

[8] Tagsistant, http://www.tagsistant.net

[9] Tagster, http://www.uni-koblenz.de

[10] Chang. K, Perdana. I, Jain. M, Kartasasmita. I, Ramadhana. B, Sethuraman. K, Le. T, Chachra. N, Tikale. S, Knowledge File System - A principled approach to personal information management, *2010 IEEE International Conference on Data Mining Workshops*, Page(s): 1037-1044.

[11] Mohan. P, Venkateswaran. S, Raghuraman, Dr. Siromoney. A, Semantic File Retrieval in File Systems using Virtual Directories. *Proc. Linux Expo Conference, Raleigh, NC*, May 2007, Page(s): 141-151.

[12] Hua. Y, Jiang. H, Zhu. Y, Feng. D, Tian. L, Semantic-Aware Metadata Organization Paradigm in Next-Generation File Systems, *IEEE Transactions on Parallel And Distributed Systems*, Vol. 23, No. 2, February 2012, Page(s): 337-344.

[13] Schroder. A, Fritzsche. R, Schmidt. S, Mitschick. A, Meiner. K, A Semantic Extension of a Hierarchical Storage Management System for Small and Medium-sized Enterprises, *Proceedings of the 1st International Workshop on Semantic Digital Archives (SDA 2011).*

[14] Eck. 0, Schaefer. D, A semantic file system for integrated product data management, *2011 Advanced Engineering Informatics*, Page(s): 177-184.

[15] File system in USERspace (FUSE) homepage and documentation, http://fuse.sourceforge.net

[16] SQLite database http://www.sqlite.org

6.4 Appendix D : Papers Referred

6.4.1 Knowledge File System

2010 IEEE International Conference on Data Mining Workshops

Knowledge File System

A principled approach to personal information management

Kuiyu Chang
School of Computer Engineering
Nanyang Technological University
Singapore 639798
e-mail: kuiyu.chang@pmail.ntu.edu.sg

I Wayan Tresna Perdana, Bramandia Ramadhana,
Kailash Sethuraman, Truc Viet Le, Neha Chachra
School of Computer Engineering
Nanyang Technological University
Singapore 639798

***Abstract*— The Knowledge File System (KFS) is a smart virtual file system that sits between the operating system and the file system. Its primary functionality is to automatically organize files in a transparent and seamless manner so as to facilitate easy retrieval. Think of the KFS as a personal assistant, who can file every one of you documents into multiple appropriate folders, so that when it comes time for you to retrieve a file, you can easily find it among any of the folders that are likely to contain it. Technically, KFS analyzes each file and hard links (which are simply pointers to a physical file on POSIX file systems) it to multiple destination directories (categories). The actual classification can be based on a combination of file content analysis, file usage analysis, and manually configured rules. Since the KFS organizes files using the familiar file/folder metaphor, it enjoys 3 key advantages against desktop search based solutions such as Google's Desktop Search, namely 1) usability, 2) portability, and 3) compatibility. The KFS has been prototyped using the FUSE (Filesystem in USErspace) framework on Linux. Apache Lucene was used to provide traditional desktop search capability in the KFS. A machine learning text classifier was used as the KFS content classifier, complimenting the customizable rule-based KFS classification framework. Lastly, an embedded database is used to log all file access to support file-usage classification.**

virtual file system; search engine; personal information management; indexing; classification

I. INTRODUCTION

Today, people are increasingly relying on their computers or mobile phones to manage their life, typically storing gigabytes of data that include personal emails, messages, documents, contacts, presentation slides, audios, videos, etc. However, due to the tremendous growth in the number of personal files, manually organizing these assets using the 40-year old folder/file metaphor is practically impossible. It is not surprising that many users nowadays simply can't find their files [1].

This shows that the existing manual mechanisms for manipulating files and directories are way out of touch with the explosive information growth personally faced by today's computer users. Keep in mind that the classical file/directory manipulation mechanisms were leftovers from systems of the past, where a disk was only a few kilobytes large that stored at most hundreds of mainly homogeneous text files. Anything more complicated has been traditionally stored in a database. We are thus at a critical junction in history where new solutions to this problem must be investigated.

Manually browsing through numerous directories is probably the simplest but yet most frustrating task when it comes to searching for a file. Technically savvy users may opt to install a desktop search engine such as Google Desktop Search, which to some extent reduces the severity of the problem but, on the other hand, it also brings in another set of problems.

First, searching is possible only if one knows what one is looking for, and is typically applicable to text content. There are times when a user is looking for a particular file that may contain too many generic words shared by other files, which leads to the second problem of returning too many hits. Going through the returned list of search results may be as frustrating as browsing though a set of hierarchical candidate directories. In fact, there are no simple strategies to rank a corpus of text documents without ready-available link/relationship information between documents; Google's Page Rank [2] strategy is powerless here. Third, a search simply finds the file, but does not help user organize it properly so that next time when he needs the same file he could navigate straight to the location instead of using a search engine all over again. Another consequence is that the information is forever tied to the search engine, i.e., no search engine, no organized information. This leads to the fourth problem; the non-portability of retrieved files, i.e., the search engine simply retrieves files but do nothing to organize them. If the corpus of documents is copied to a USB disk and moved to another system, the same problem will persist on that system unless it also has a desktop search engine installed. At the end of the day, using a search engine to find files may take equal or more time as going through the various hierarchically organized directories manually.

This last point above is the primary motivation for KFS, to "place" a file in as many appropriate directories as possible so that during manual navigation though the file system, a user is likely to encounter a hit at the very first few locations that come to his mind. To illustrate this, suppose a set of hierarchical directories have been painstakingly created by a user in the file system, and each

978-0-7695-4257-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDMW.2010.119

copy of some email is automatically placed into multiple appropriate directories by KFS as shown in the example of Figure 1. With the same email placed in four possible locations, his chance of finding this email (recall) is immediately increased 4 times.

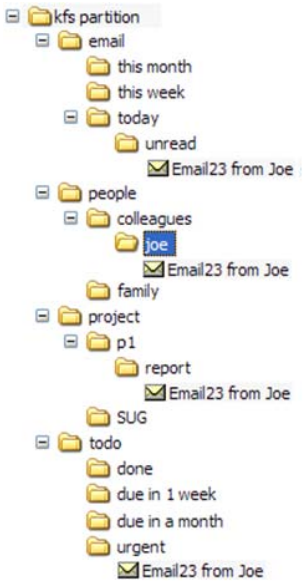

Figure 1. With "Email23 from Joe" automatically placed in 4 relevant folders, the user has a higher likelihood of locating this email later on.

II. Related Work

There exist many approaches to deal with the problem of organizing personal information. Here we review a few of the mainstream ideas.

Google Desktop Search [3] is a local desktop search engine made for various operating systems (Windows, Mac, and Linux). A similar desktop search tool offered by Apple is Spotlight [4], which works only under the OS X platform. Google Desktop Search and Apple Spotlight are two very efficient search engines that allow any file to be found fast and effectively. They support many types of files and their simple and intuitive interface is the main draw for most users. However, both of them are designed to return accurate search results without really organizing the files contained within. So all the limitations of the low level fix approach of search engines apply.

Gnome Storage [5] is an open-source effort to revolutionize the file system interface by storing everything in a relational database. It provides a virtual file system layer (GnomeVFS) for compatibility with existing applications. It is an ambitious effort that incorporates many advanced features like associations between objects, awareness of which application is opening an object, and revision tracking of objects.

Usenetfs is a stackable file system for large article directories [6]. It was developed to improve the file search on newsgroup servers. Similar to files on personal computers, files on servers typically reside in only a few commonly used directories over time, which makes searching inconvenient. Usenetfs changes the file structure by creating smaller directories containing fewer files, instead of the large existing flat file structures. This improves the processing rate of the articles on the server. Usenetfs is portable and imposes little overhead, thereby providing substantial performance improvements.

Microsoft has also put a lot of effort in this area. Microsoft's WinFS file system [7] is a commercial attempt to replace the file system with a relational database. Although well invested, it has been delayed many times since it was first announced in 2002. As of 2010, it has not been released commercially due to performance issues.

Another interesting research project carried out by Microsoft is MyLifeBits [8], which aims at recording everything in a person's life. With regard to computer use, every mouse click is recorded, every web page visited (not only the link) is stored, and every IM chat is logged. Moreover, a user wearing a sophisticated camera and GPS-tracking device as well as a set of sensors will have every event happening recorded and saved with time stamp and location information. In other words, MyLifeBits tries to create a "lifetime store of everything" [9]. While it seems very promising, the project is not yet available for public or commercial use.

A related but less ambitious approach is the "Stuff I've Seen" system by Dumais [10], which simply remembers all entities including files, web pages, emails, contacts, etc., that a user have come into contact on his computer. By revising this history, which is like a super-charged web browsing history, a user is able to find items guided mainly by his temporal recollection of the desired item..

MIT's haystack [11] is a different approach for information management. Haystack proposes a new concept of information management by organizing all types of information into one universal interface. Therefore, there is no need for separate applications to manage different types of data. By unifying the access to all types of information from within a single application, data from various applications can be easily associated and cross-referenced with one another. The main problem with this sort of half-revolutionary approach is that users have to rely on a single application, which might not be compatible with existing solutions.

NEPOMUK (Networked Environment for Personal Ontology-based Management of Unified Knowledge) [12] is a collaborative project between various European institutes (to name a few, German Research Center for AI, IBM Ireland, Thales SA – France, EDGE-IT – France, National University of Ireland, etc.). It ambitiously aims to turn the personal computer into a collaborative environment with state-of-the-art online collaboration and personal data management. The project seeks to augment the intellect of those involved by providing and organizing an enormous amount of information contributed by its

members from all over the world. In order to achieve this goal, the project introduces some new concepts and solutions, one of which is the Social Semantic Desktop. The Social Semantic Desktop is a new kind of desktop environment that enhances the traditional desktop by providing information with semantic meaning, thereby making information understandable by a computer. It also supports the interconnection and exchange of information with other desktops and users, i.e., it is social. Although the solution is original and praiseworthy, NEPOMUK's implementation is extremely complex and currently works only on the K Desktop Environment (KDE).

Rather than being a local personal information management of any sort, Freebase [13] is a vast public database that supports the categorization and sharing of information and knowledge. It attempts to create a global knowledge base that is structured, searchable, writable, and editable by a community of contributors. Freebase is free and open to anyone. It is powered by Metaweb, which allows enormous volumes of data to be collected, organized, connected, and modified. The data in Freebase are all linked together and can be collaboratively edited. Users of Freebase can contribute, structure, search, copy, and use data through either the Freebase.com web site or via the application program interface (API) for any commercial or non-commercial purposes. While the idea of building a public database for anyone to consume or contribute is laudable, it still has a long way to go before people can really benefit from this vast and organized source of information and knowledge.

The evolutionary approach adopted by KFS is different from all other approaches mentioned earlier. Its most related cousin is the GnomeVFS component of Gnome Storage. KFS makes changes at the file system level, not application level. Thus, the files are inherently organized on the storage media. The fact that KFS operates at file system level gives it more control over other tools that operate at the application level. This, however, does not restrict KFS to provide features commonly available in other desktop search tools. In fact, users can use existing desktop search or PIM tools on top of KFS. This kind of hybrid approach combines the powerful features of KFS with the features provided by any other PIMs or desktop search tools.

III. KNOWLEDGE FILE SYSTEM

The Knowledge File System (KFS) is a virtual file system that is designed to help users organize information using the familiar hierarchical file/directory metaphor. In other words, KFS offers a set of features that help alleviate the problem of manual information classification and retrieval. The main features of KFS include automatic classification of files, indexing, and logging of usage. Furthermore, KFS revamps the paradigm of hierarchical tree by explicitly allowing a file to reside in more than one directory. KFS is similar to Evolution's vFolders [14], but it is more flexible as it can classify any object that can be represented as a file.

KFS is designed as a stackable virtual file system (VFS) that works one layer above an underlying file system. The underlying file system does the actual work of storing and retrieving data. Being a virtual file system, we can view KFS as another file system like Ext4, except that it has many more special features. Typical VFS operations on the KFS like copy and move are simply passed on to the base file system. KFS comes with a bundle of features that are useful for users to organize their files. The major design features are described in this section.

A. Multiple Directories

A directory in the KFS partition can be considered as a category. A hierarchical directory structure mimics a hierarchical category or ontology. This is the dual of assigning multiple attributes to a file; we place it in multiple directories. In practice, a file can be semantically associated with more than one category, i.e., KFS allows a file to reside in one or more directory within a KFS partition without duplicating the data. This is a simple yet powerful mnemonic for information retrieval. For illustration, consider a video file of a 2006 basketball match between the San Antonio Spurs and Los Angeles Lakers, which could be logically placed into multiple subdirectories of a KFS partition, e.g., `/kfs/video`, `/kfs/sports`, `/kfs/nba`, `/kfs/lakers`, `/kfs/spurs`, `/kfs/2006/`. The multiple placements, facilitate effortless retrieval of the file at later times.

Clearly, multiple hard links can be created to achieve this, but hard links are difficult to manage and track. KFS provides a suite of user space tools to track and maintain such links via a central database. For instance, there are KFS tools that can list all hard links of a file, or delete a file including all its hard links at once. Internally, this is achieved by assigning a unique ID to every distinct file in KFS (which is similar to the inode concept in file systems). To facilitate efficient lookup, KFS stores the ID and filename association in an embedded database. To classify a file to a category, a user can simply link it under the directory/category. To de-classify a file from a category, user can delete the link from that directory. If a user would like to delete a file including all associated links from a KFS partition, a special tool called 'kfsremove' can be used. KFS will also physically delete a file from the KFS partition if there are no more links pointing to it, i.e., like dangling pointers.

B. Automatic Classification

KFS provides functions to perform automatic classification of files. Once a file is created, moved, or copied into a KFS partition, KFS automatically categorizes it into multiple pre-specified directories based on content similarity with respect to predefined classification ontology. This allows files to be automatically and logically organized in the partition, facilitating easy retrieval. Any suitable content or usage classifier can be used with the KFS. Out of the box, KFS comes with a built-in trained Support Vector Machines classifier from the libsvm library. The KFS can be reconfigured to use

other classifiers such as K-nearest neighbor or Naïve Bayes, but user must periodically retrain the classification models to ensure decent performance. Users can also write their own classification plug-ins and train a new classification model by supplying training documents for each category.

Currently, KFS supports automatic classification of text and HTML files via the content classifier and HTML KFS plug-ins. Support for other popular document formats like PDF and OpenOffice can be added by developing new KFS plug-ins. Similarly, classification of non-textual content such as images, sounds, can be added in the future. In addition to similarity based classification, users can create rule-based classification. Each rule is comprised of a set of regular expressions on the filename, location, and MIME type of the target file. A simple classification rule can be as follows: classify all PDF files with 'report' in its name to the 'report' directory. The following scenario illustrates the steps taken by Linux KFS to automatically classify a file:

1. File `sports.txt` is copied to `/kfs/data` directory.
2. KFS intercepts the copy operation and passes the file to the appropriate KFS plug-ins.
3. KFS creates a master copy of `sports.txt` in the backing store KFS directory, auto-renamed as some hashed valued filename `.store/0F0XA.txt`.
4. A hard link to `.store/0F0XA.txt` is created in the original user designated KFS target path `/kfs/data/sports.txt`.
5. KFS does the following in the background:

The KFS Classifier categorizes the file as a sports related file, and creates a hard link to `.store/0F0XA.txt` in `/kfs/category/sport/sports.txt`.

The KFS MIME Classifier determines that it is a text file and creates a hard link to `.store/0F0XA.txt` in `/kfs/.mime/text/sports.txt`.

C. *Custom Classifier*

The KFS classification framework allows user defined classifiers. Suppose a user added a classifier engine designed to classify a PDF document, then he can simply register the classifier with the KFS Classification framework. The user defined classifier will be called when a PDF file (as determined by the included Libmagic file type plug-in described below) needs classification. Before classifying a file, KFS must first determine its file type, whether it is plain text, HTML, image, binary, etc. This is important as to decide which classifier to invoke for classifying the file. Unlike Windows OS, there is no generally accepted concept of a file extension in POSIX systems. Moreover, file extensions are often unreliable for determining a file type. KFS employs the Libmagic library to guess the MIME type of a file.

D. *Text Indexer*

KFS comes with a text content indexer, which allows files to be searched by keywords. By default, KFS automatically indexes the entire KFS partition. Every file system change will be delegated to the KFS Indexing Engine for index updates. For example, when a file is copied to the KFS partition, the KFS Indexing Engine automatically indexes it. Likewise, the index is automatically updated whenever a file in the KFS partition is modified.

The KFS Indexing Engine is based on the CLucene [15] package, which is a high performance indexer implemented in C. KFS also supports incremental indexing, which means that files can be added and deleted without requiring a full re-index of all files in the system. In addition, the KFS Indexing Engine employs batch indexing to reduce the overall indexing time. KFS will index multiple files all at once instead of one by one. This is useful if user copies many files into the KFS partition at one go.

Before a file can be indexed, it must be pre-processed to create a pre-index document that contains the tokens to be indexed. For example, the initial processing stage will tokenize the data and get rid of some unimportant stopwords. The next step is to add the pre-index document to the index. The pre-processing step depends on the type of a file and its content language (for text). KFS provides a built-in mechanism (called a preprocessor) to pre-process a plain text file and produce a pre-index document that is ready to be indexed. A user-defined preprocessor can be attached to the KFS Indexing Framework to allow KFS to index other types of files. For example, users can create a preprocessor for PDF documents and register it with the KFS Indexing Framework. Then, whenever a PDF document needs indexing, the PDF preprocessor will be called to produce a pre-index text version of the file that is ready to be indexed. Once the pre-index text document is ready, the KFS Indexing Engine updates its index to include the new pre-index document. Using this modular framework, the KFS Indexing Engine can index any type of files provided the corresponding preprocessor is available.

E. *Event Logger*

Every operation on a KFS partition is monitored. Operations such as copying, moving, updating, and deleting of files are logged to the embedded database. Users can simply browse through the database records to retrieve the log information. Such information can be very useful for usage mining and auditing.

Related to the Event Logger, KFS allows users to register an external program (shared object library in practice) to be called whenever a specific KFS operation is logged. This comes in handy for situations where we wish to be notified whenever someone reads or writes to a common KFS sub-directory.

For example, suppose a user would like to automatically test and grade every programming assignment added to the `/submissions` directory. To do this, he can create a simple script to compile the submitted source files and run it with some predefined test cases and assign marks based on the testing results. After

that, the user can simply register it with the KFS Logging Framework.

F. Customizability of KFS

The KFS Indexer, KFS Classifier, and KFS Logger are all customizable. KFS provides a configuration file for each of the framework (namely Classification, Indexing, and Logging). On top of that, there is a master configuration file where the overall behavior of KFS is configured.

The classification configuration file lists the mappings between a set of rules to the corresponding classifier program. The set of rules comprises the filename, location, and MIME type of the file to be classified. Regular expressions can be used to refine each set of rules, thereby providing the user with a detailed level of customization of the KFS Classification Framework. For example, a user can create a rule targeting any file beginning with '`alaska`', whose MIME type is html, and which is located within `/kfs/website/pages` to be classified by a user defined classifier.

The Logging configuration file provides a similar function as the Classifier configuration file except that it defines mappings between a set of rules and external programs to be invoked when the corresponding rule is satisfied. The set of rules comprises the filename (there could be two files involved), location, MIME type of the file as well as the operation type. For instance, a user could log every link creation operation that points to any image file residing in `/kfs/data`.

The Indexer configuration file defines mappings between a set of rules and the corresponding preprocessor library to be called. The rule includes the filename, location, and MIME type. Users can specify which preprocessor library to be used for processing a particular type of file. For example, users can configure KFS to preprocess all Spanish HTML document prior to indexing.

G. Extensible Architecture

As described earlier, KFS can be easily extended by user-made plug-ins, which just needs to be attached to one of the KFS frameworks. Specifically, the KFS Classification Framework allows a user to add a new Classifier. This enables KFS to practically classify any type of file. The KFS Indexing Framework allows the user to add a new preprocessor, which enables KFS to index any type of file in any language. Lastly, the KFS Logging Framework allows user to monitor all accesses to the KFS partition by attaching a monitoring daemon.

IV. KFS VERSUS OTHER APPROACHES

Table I compares KFS with existing desktop search solutions. Clearly, neither Google Desktop nor Beagle supports file system organization via hard link management. This is expected as both of them work at the application level. On the other hand, both Google Desktop and Beagle clearly support a larger number of file types for indexing. Being extensible, additional KFS plug-ins can be written to increase the indexing capability of KFS. As for the automatic classification of files, only KFS provides this feature integrated with hard link management.

TABLE I. KFS VERSUS TWO DESKTOP SEARCH SOLUTIONS.

Features	KFS	Google Desktop	Beagle
File system organization	Automatic	No	No
Text Classification	English	No	No
Indexing	Text & HTML	Various	Various
Extensible	Yes	Yes	Yes

TABLE II. KFS VERSUS OTHER FILE SYSTEMS.

	KFS	UsenetFS	GnomeVFS
Virtual file system	Yes	Yes	Yes
Learning Curve	Low	Low	Low
Link management	Yes	-	Yes
Text Classification	English	No	No
Indexing	Text & HTML	Numbering based	Text, PDF, etc
Extensible	Yes	Yes	Yes
Maintains data consistency	Yes	Yes	Yes
Log user changes	Yes	No	Yes
Scalability	High	High	Moderate
File Browsing	Yes	flat	Yes
Stackable	Yes	Yes	No

TABLE III. KFS VERSUS TWO RADICAL APPROACHES.

	KFS	NEPOMUK	Freebase
Learning curve	Moderate	High	High
Application Compatibility	Yes	KDE only	Web & API
Portability	Yes	No	Yes
File System Compatibility	Yes	N/A	N/A
Personalizable	Yes	N/A	Yes
Requires Internet connection	No	Yes	Yes
Free	Yes	Yes	Yes
Open-source	Yes	Yes	Yes

Table II compares KFS with other file system approaches. According to Table II, all three systems are extensible meaning that users can add more functionality. Both KFS and Gnome Storage support indexing with Gnome Storage having a better support for file types. KFS is different from Gnome Storage both in philosophy and goals. Gnome Storage aims to be the revolutionary file system with lots of advanced features like arbitrary graph based associations between objects, and is based on a relational database. KFS, on the other hand, aims at maximizing the utility of the familiar file/directory hierarchical metaphor by providing tools to automatically manage hard links. The ultimate goal of KFS is to make the current file system metaphor manageable to end users without having them learn a whole new paradigm of

information management. We feel intuitively that the KFS is more acceptable to users of today, and should be much easier to use in the near term. Whether the KFS and file/directory metaphor will be viable in the long term remains to be seen.

Finally, Table III gives a rough comparison between KFS, NEPOMUK and Freebase (as mentioned earlier). Although they are of very different approaches, they all share the same goals and philosophy and are therefore worth comparing.

V. KFS FOR LINUX

KFS for Linux is implemented as 2 main parts that executes as two separate processes, KFS-FUSE as the virtual file system and KFS daemon, which contains the logger, indexer, and classifier. This approach follows the general rule of thumb to separate system (kernel) related part, which is the file system, and implement the remaining functionality in the other part (user space). The reasons to split the implementation are summarized as follows:

- Performance – the virtual file system component is very critical to the overall system performance because it relates to the operating system kernel. A clear separation will enable us to optimize the performance and minimize performance hit to the overall system. This still enables us to perform heavy processing such as classification in user space.
- Easy to debug and maintain – kernel programming and debugging is much more difficult than user space programming and debugging. The interface to the kernel is already provided by FUSE framework. Thus, we only need to take care of the file system implementation that resides in user space.
- More library support – by developing part of KFS in user space grants it access to a vast library of software unavailable in the kernel context. This greatly simplifies the development of KFS.
- Easily extensible – our approach allows third parties to easily write plug-ins for KFS in user space instead of the more complex kernel space.

A. *KFS – FUSE*

KFS is not a standalone file system, but instead a stackable file system. It sits on top of another underlying file system and utilizes it to store the actual data. This approach greatly reduces the amount of work needed to develop KFS as developing a high quality file system is a very complex process requiring many man years. Furthermore, this allows KFS to reap full benefits from years of work done in creating an efficient file system.

KFS resides between the user program and the underlying file system. This enables KFS to intercept any relevant file system calls and perform some tasks before or after delegating the call to the underlying file system.

For example, in the event that a file `data.txt` is updated in a KFS partition, the following string of events is expected:

- KFS FUSE is notified that a file is updated.
- KFS FUSE delegates the actual work to the underlying file system.
- KFS FUSE sends a message to user-space KFS that `data.txt` is being updated.

The KFS FUSE module is developed based on the FUSE (File System in User Space) [16] module. FUSE allows us to create our own file systems without editing the kernel code. This is achieved by running the file system implementation code in user space. FUSE provides a bridge to the actual kernel interfaces. FUSE was officially merged into the mainstream Linux kernel tree starting from version 2.6.14.

The KFS FUSE module resides both in kernel space and user space. It contains secure and efficient modules that reside in the kernel, as well as high performance communication interface between kernel and user space, and a set of APIs that can be used according to our needs. FUSE has been widely used to create file system drivers and interfaces. The FUSE kernel module and the FUSE library communicates via a special file descriptor that is obtained by opening `/dev/fuse`. This file can be opened multiple times, and the obtained file descriptor is passed to the mount syscall, to match up the descriptor with the mounted file system.

For most of file operations such as move, delete, create directory, and so on, there is an equivalent FUSE API call that will be executed when such an operation is performed. Thus KFS can intercept the operation by simply intercepting the corresponding API call.

KFS FUSE supports multiple virtual file system, which means that the same module can be used to mount different partitions at the same time, and each virtual file system runs as an independent process. Figure 2 shows the KFS FUSE design on top of an Ext3 file system.

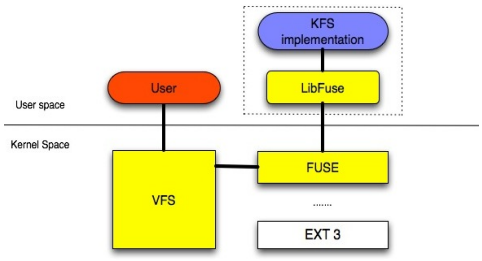

Figure 2. KFS FUSE.

B. *User-space KFS*

The user space component of KFS is responsible for the computationally intensive tasks including maintaining the KFS partition index, classifying files, updating of the KFS metadata database, and logging of user activities.

Being a complex framework, KFS does not rely on itself to perform all of its functionalities. Instead, it makes use of a number of existing technologies to build a complete and powerful file system. This minimizes the amount of code to be rewritten and avoids reinventing the wheel. For example, KFS relies on the CLucene framework for indexing and the libsvm for content classification. KFS also utilizes the Berkeley DB [17] embedded database. Berkeley DB was chosen because it is a lightweight database that runs in the same process as the calling application, with no context switching overhead.

The user space KFS daemon constantly listens to a predefined port for user space commands. This allows other process to communicate and alter the behaviors of KFS during runtime. For example, a user might wish to pause file indexing during peak periods.

The user space KFS is designed with scalability and extensibility in mind; it allows third party plug-ins to be attached to the system on the fly. In addition, some performance enhancement techniques such as batch indexing are also employed.

C. *Plug-ins and User-space Tools*

A number of user space tools were developed to fully utilize the KFS features are listed as follows.

- Mount, unmount, create, and delete KFS partition (multiple mounted KFS partitions possible).
- Search the Index for a keyword.
- Search for certain characters in filenames.
- Find all files that link to the same data.
- Delete a file and all of its references.
- Display usage log data.
- Pause or resume indexing.

In addition, a simple plug-in to strip HTML tags has been developed and attached to the KFS, which allows KFS to classify HTML files. For ordinary users, the set of command line tools provided above might be difficult to use, in which case they could use the included GUI java program to manage KFS partitions.

VI. Performance Evaluation of KFS

Although the KFS is designed to perform classification and indexing of files in batch mode, we want to evaluate how this will impact the actual use, i.e., whether it will cause a noticeable delay to the user. Experiments were simulated on an Intel Pentium 4 Processor 3.40 GHz desktop with 2048 MB of memory running Ubuntu Linux 7.10 [18] operating system with Linux kernel 2.6.22-14 and the following libraries:

- FUSE 2.7.0
- CLucene 0.9.20
- Libsvm2
- C++ Berkeley DB 4.3.28
- Boost C++ Library 1.34.1

Various quantities (100 to 1000) of text files, each of average size 2.6 KB were copied to a KFS partition. The time taken to copy as well as to classify and index the files was measured using the Linux time command. Measurements are also taken for other file systems and tools. GnomeVFS-copy is measured using a simple looping script to copy the files one by one.

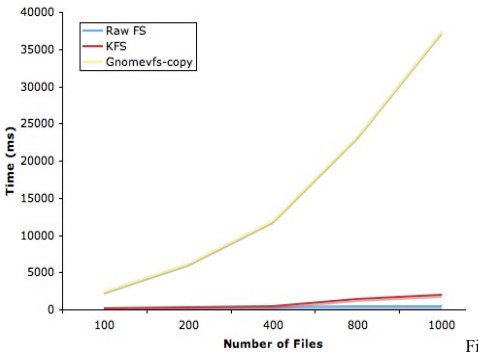

Figure 3. Time taken versus number of files for batch file copy.

From the results in Figure 3, the baseline raw file system performance is clearly the fastest among the three. Gnomevfs-copy was chosen because it implements the Gnome Virtual File System library [19]. Gnomevfs-copy is significantly slower than KFS. This is probably due to it been called by a looping script, since there is no way to specify a list of files for gnomevfs-copy.
KFS performs only marginally slower than the raw file system. The separation between the file system module and indexing/classification module enables the user to finish the file-copying process while KFS continue to index/classify in the background.

A. *Sample Classification*

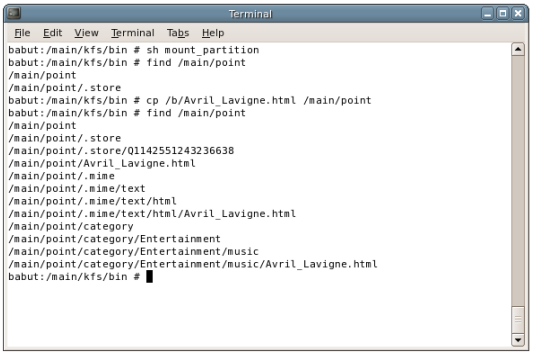

Figure 4. Sample KFS classification in console.

Figure 4 demonstrates the classification feature of KFS. When a file is copied to a KFS Partition, KFS automatically classifies the file and create hard links in the appropriate locations based on the classification result. In this example, the mounted KFS partition is `/main/point`, and `/main/point/.store` is where

the raw data file is located. We see that initially there is nothing in the KFS partition. However, after the file `Avril Lavigne.html` is copied to `/main/point`, it is indexed and classified by KFS, which results in the creation of two hard links to this new file, one in `.mime/text/html`, and the other in `category/Entertainment/music`.

VII. Conclusions

Currently, Linux KFS is still in its infancy. Its features are limited but could be greatly enhanced with its open and extensible architecture. Although we have only written a few plug-ins to improve the basic functionality of Linux KFS, in its current form Linux KFS could potentially be used to classify emails as individual files using the Mail Directory format of Courier's IMAP email system. The auto classification feature currently only works with English text and HTML files. This can be enhanced by providing classifier plug-ins that support non-English text content. In addition, an advanced translator plug-in is needed if we want to classify or index non-text files such as PDF or OpenOffice documents. Likewise, the indexing feature currently supports plain text only. This could be improved by writing a preprocessor library to preprocess a variety of non-text files. Linux KFS can help user organize their files without taking away any flexibility and life can go on as normal; as a file system, Linux KFS can seamlessly integrate with the vast majority of existing applications. Being extensible, Linux KFS can be built into a very sophisticated file system for managing personal information.

So far, we have not systematically evaluated user responses to the KFS. For future work, we need to test the KFS extensively on common day-to-day tasks like organizing emails and pictures, to truly measure its effectiveness in improving a user's recall when it comes to finding his file. Moreover, extensive end user testing is also very important to discover limitations and areas of improvement for the KFS approach to solving the information explosion phenomenon. We believe that the advent of huge portable storage in mobile wireless devices will make the KFS even more relevant as a unified paradigm to organize and retrieve information on the mobile devices.

Acknowledgements

This research was funded in part by NTU Startup Grant CE-SUG 11/03 and Singapore Ministry of Education's Academic Research Fund Tier 1 RG 30/09.

References

[1] C. A. N. Soules, G. R. Ganger, "Why can't I find my files? New methods for automating attribute assignment", Technical Report CMU-CS-03-116, School of Computer Science, Carnegie Mellon University, 2003.

[2] S. Brin, L. Page, "The Anatomy of a Large-Scale Web Search Engine", Proceedings of WWW, 1998.

[3] Google Desktop Search, http://desktop.google.com/en/, 2010.

[4] Apple Spotlight, www.apple.com/macosx/features/spotlight/, 2010.

[5] S. Nickell, "A Cognitive Defense of Associative Interfaces for Object Reference", Draft, http://people.gnome.org/~seth/storage/associative-interfaces.pdf, 2010.

[6] E. Zadok, I. Badulescu, "Usenetfs: A Stackable File System for Large Article Directories", Technical Report CUCS-022-98, Computer Science Department, Columbia University, 1998.

[7] R. Grimes, "Code Name WinFS: Revolutionary File Storage System Lets Users Search and Manage Files Based on Content", MSDN Magazine, Jan 2004.

[8] J. Gemmell, G. Bell, R. Lueder, "MyLifeBits: a personal database for everything", Communications of the ACM, vol. 49, Issue 1, pp. 88-95, Jan 2006.

[9] S. Cherry, "Total Recall", IEEE Spectrum, pp. 18-24, Nov 2000.

[10] Dumais, S.T., Cutrell, E., Cadiz, J., Jancke, G., Sarin, R., Robbins, D.C., "Stuff I've Seen: A system for personal information retrieval and re-use", ACM SIGIR, 2003.

[11] D.R. Karger, K. Bakshi, D. Huynh, D. Quan, V. Sinha, "Haystack: A customizable general-purpose information management tool for end users of semistructured data", in Proc. CIDR, 2005.

[12] N. Papailiou, C. Christidis, D. Apostolou, G. Mentzas, R. Gudjonsdottir, "Personal and Group Knowledge Management with the Social Semantic Desktop", in Proc. Collaboration and the Knowledge Economy: Issues, Applications and Case Studies, 2008.

[13] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, J. Taylor "Freebase: a collaboratively created graph database for structuring human knowledge", Proc. of SIGMOD, pp. 1247-1250, 2008.

[14] Gnome Evolution Project, http://projects.gnome.org/evolution

[15] CLucene: port of the popular Apache Lucene search engine, http://clucene.sourceforge.net/

[16] E. Zadok, I. Badulescu, "A Stackable File System Interface for Linux", in Proc. Linux Expo Conference, Raleigh, NC, pp. 141-151, May 1999.

[17] BerkeleyDB, http://www.sleepycat.com/

[18] Ubuntu Linux, http://www.ubuntu.com/

[19] Gnome File System library, http://library.gnome.org/devel/gnome-vfs-2.0/

6.4.2 Semantic File Systems

Advanced Engineering Informatics 25 (2011) 177–184

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei

A semantic file system for integrated product data management

Oliver Eck [a,*], Dirk Schaefer [b]

[a] Department of Computer Science, HTWG Konstanz, Germany
[b] Systems Realization Laboratory, Woodruff School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA, USA

ARTICLE INFO

Article history:
Received 31 October 2009
Received in revised form 10 June 2010
Accepted 17 August 2010
Available online 20 September 2010

Keywords:
Information retrieval
Semantics
Semantic file system
Engineering design
PDM
PLM
CAD
CAE

ABSTRACT

A mechatronic system is a synergistic integration of mechanical, electrical, electronic and software technologies into electromechanical systems. Unfortunately, mechanical, electrical, and software data are often handled in separate Product Data Management (PDM) systems with no automated sharing of data between them or links between their data. Presently, this is a significant drawback with regard to supporting the collaborative and integrated design of mechatronic systems. One underlying research question to be addressed is: "*How can such domain-specific PDM systems be integrated and what are the standards needed for this*?" Our starting point for addressing this question is to look into how engineering data is stored today. We believe that the more intelligent and sophisticated the mechanisms for file management are, the more Computer-aided Engineering systems will benefit from them in terms of making integrated cross-disciplinary product development more efficient. However, while Computer-Aided Design and Engineering (CAD/CAE) systems and related tools such as, for example, Product Data Management (PDM), Engineering Data Management (EDM), etc. have significantly matured over the past ten years, most of their associated database systems are still based on traditional file systems and hierarchical directory structures.

In light of this context, we will initially discuss a number of disadvantages of current file management systems. In the body of the paper our main contribution is presented. That is, a formal mathematical model of a new semantic file system, *SIL* (*Semantics Instead of Location*), that allows engineers to access data based on semantic information rather than storage location is proposed. A major difference between our approach and previous related work is that we do not aim at yet another point solution and, instead, propose an approach that may be employed by next generation engineering data processing systems on a larger scale. In addition, a corresponding programming interface along with a graphical user interface used as a file browser is presented and the benefits of utilizing the proposed semantic file system for product data management in the field of integrated design of mechatronic systems are discussed.

1. Introduction

1.1. Motivation and background

A mechatronic system is a synergistic integration of mechanical, electrical, electronic and software technologies into electromechanical systems. Mechatronic systems are excellent candidates for design process optimization due to the high complexity of mechatronic design, the high degree of integration of electrical, mechanical and information-processing components, the overlapping design disciplines and system behaviors, and the critical nature of optimizing the overall system. Unfortunately, mechanical, electrical, and software data are often handled in separate Product Data Management (PDM) systems with no automated sharing of data between the systems or links between the data. As a result, a significant drawback exists with regard to supporting the collaborative and integrated design of mechatronic systems [3]. In light of this, the key research question we address is: "*How can such domain-specific PDM systems be integrated and what are the standards needed for the integration*?" The work presented in this paper is a contribution towards answering the first part of the above-stated research question.

While PDM and other related Computer-Aided Design and Engineering (CAD/CAE) systems have significantly matured over the past ten years, their underlying database systems are still based on traditional file systems and hierarchical directory structures. However, in order to facilitate integrated cross-disciplinary computer-aided product creation, new sophisticated mechanisms for intelligent information retrieval and file management are required.

Most engineering information systems store and retrieve data in the form of files of different types. Especially in the computer-aided engineering and design field, the number of files associated with individual products, parts, components and their associated

* Corresponding author. Tel.: +49 7531 206 630; fax: +49 7531 206 559.
E-mail address: eck@htwg-konstanz.de (O. Eck).

1474-0346/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2010.08.005

CAD/CAE/PDM/PLM systems and system users has reached a level of vast complexity and little transparency [4]. This is due to the fact that current file systems are still based on traditional hierarchical directory structures. In such file systems, users are required to organize their data in hierarchical directory structures. In such directory structures, specific path and file names have to be determined and correctly entered in order to store and retrieve data. Unfortunately, these traditional hierarchies often do not scale to large data collections and to the fine-grained classifications required [5]. As the data classification hierarchy grows, files are increasingly likely to be assigned several different classifications that are all linked to only one associated file location. To locate and retrieve a particular file at a later time, system users must remember the one specific classification that was originally chosen.

Another significant issue with traditional file systems is that information, which identifies a document, must be captured in both the pathname and filename. Unfortunately, standards do not exist for describing the exact information to be represented in pathnames and filenames, respectively. Today, many companies define their own proprietary file and path naming conventions, including guidelines for the utilization of version numbers, status flags, file creation dates, filenames, and information regarding the person or system used to create a file. For example, in many of our collaborative industrial projects typical file names are of the following signature: 'CB_Spec_driveShaft_OE_draft_V9.doc', which represents the following information: the customer is CB, the document includes the draft specification of a drive shaft, the author initials are OE, and the document version number is 9. As mentioned before, such guidelines are proprietary and usually not strictly enforced. Presently, the only consistent aspect of file naming that represents at least a minimum of semantic information regarding file generating source and content is that of document type identifiers such as .dwg, .doc, .pdf, etc. For example, .dwg stands for a drawing created with AutoCAD, and .doc stands for a text file created with Microsoft Word.

Sharing traditional file systems among several users normally leads to a number of inherent problems. Users often have to guess or anticipate the correct words employed by other users to denote pathnames and filenames. For example, a particular *partA.dwg* used for a *machineTybeB* sold to *customerC* might be stored in the directory /customerC/machineX/partA.dwg or in the directory /machineX/customerC/partA.dwg, depending on the path and file naming conventions of the user who created and stored the file. As a result, users need to remember not only the correct keywords associated with a document but also the mapping of these keywords to a particular directory structure. If the subdirectories 'customerC' and 'machineX' appear at the root directory, users have to browse both subdirectories to locate the drawing file being sought. From a logical perspective, the file should appear in both directories, which simply is not possible in traditional file systems due to redundancy avoidance conventions. The definition of guidelines for organizing the path hierarchy and naming files resolves this problem only partially.

Recently, engineering tools have been developed that allow describing meta-information in files in order to improve the effort of document retrieval. For example, in the area of Model-Based Systems Engineering (MBSE) [2] and multi-view modeling [30], a number of standards and formats for describing meta-information of a system or a class of systems to be designed are used. Examples include model descriptions in SysML [7] or Modelica [8], to mention just a few.

All the before-mentioned meta-information-based efforts address the same important goal of improving file access and retrieval. However, the main problem with the existing meta-information-based approaches and their corresponding meta-information is the fact that not all files are described completely since the primary mandatory file description is still given by pathname and filename. Therefore, file information is separated into two areas that cannot be used together in a single efficient file retrieval method. Furthermore, efficient retrieval facilities require meta-data to be stored in a central database instead of distributed across thousands of files within the file system.

The utilization of semantically annotated ontology has become an established approach in numerous information retrieval tasks, and semantic annotation has emerged as an important research area. Semantic annotation can be described as both meta-data and the process of generating meta-data. Semantically annotating a document provides, as output, meta-data related to the document. On the other hand, ontology is a formal model describing some knowledge domain and consists of concepts, such as types and properties, and the relationships between concepts. Together, semantically annotated ontology within the scope of integrated computer-aided product design provides a formal methodology for the extraction of meta-data from diverse information sources such as domain-specific product design files coupled with their ontological relationships. Adopting the concept of semantically annotated ontology from Semantic Web applications to the field of computer-aided product design has shown to be very promising for design-related retrieval tasks. Therefore, research approaches for semantics-based engineering document analysis and retrieval as well as ontologies for Engineering Information Retrieval (EIR) are becoming more prominent [20,21].

1.2. Scope of work and contribution

In this paper, we propose a semantic file system, SIL (Semantics Instead of Location), which allows engineers to access data based on semantic information rather than storage location information thereby facilitating enhanced search and retrieval capabilities as needed for integrated, cross-disciplinary computer-aided product development processes.

In contrast to previous approaches such as [20,24], SIL does not attempt to realize yet another standard retrieval system with slightly more sophisticated retrieval mechanisms. Instead, the goal of SIL is to provide a semantic-enabled intelligent file system that provides the search mechanisms necessary for both effective and efficient retrieval on a software layer that may be used as a common platform for the enhancement of future integrated computer-aided product design systems and other related engineering data retrieval systems such as PDM, CAD, and CAE tools. In other words, instead of developing yet another point solution, our long-term goal is to create retrieval mechanisms based on a standard software architecture that may be embedded (or 'plugged') into future computer-aided product development applications. An example of a similar idea within the context of standard office software is the opportunity to install a single PDF software module that automatically enhances all installed office applications (Word, Excel, Access, PowerPoint, etc.) with the same additional functionalities and capabilities. Again, we would like to stress that our idea is aimed at providing a far-range solution of significant impact on an entire field rather than yet another stand-alone approach.

In order to represent file semantics and to provide associative access to a semantic file system, meta-data information is required. SIL represents property-based semantics and content-based semantics. The proposed file system paradigm is based on a set of keywords that describe the content-based semantics of data files. This information is based on two sources: user input and content analysis. Content analysis is realized by the extraction of meta-information contained within the data files. In the above outlined engineering context, semantic information of documents is almost always available. For example, semantic annotations for

product lifecycle management can be used as keywords as well as meta-data, which are automatically added through text processing when a data file is modified and saved. Extracting the meta-data from files and storing it in a central database enables efficient retrieval facilities to evaluate retrieval queries with a very short execution time.

In the proposed approach, such annotations are used to identify and locate files. In contrast to traditional tree-structured file systems, these keywords are not ordered and classified in a hierarchy. In contrast to other related approaches for semantic file systems, tree-structured file systems are completely replaced by a new structure of keywords specifically tailored to the requirements of collaborative design activities across technical disciplines. By adding ontology technologies, information search in the semantic file system is related to information search in the Semantic Web or other advanced information management systems, which can be integrated more easily than traditional file systems.

In this paper, we introduce a formal definition of a new semantic file system. In addition, a corresponding programming interface along with a graphical user interface used as a file browser is presented. Further, we present the benefits of utilizing such a semantic file system for product data management in the field of integrated design of mechatronic systems.

In summary, the presented semantic file system, SIL, provides the following features, which collectively are not available in any traditional file systems known to the authors. A detailed discussion of these features follows in Section 3.

- **Multi-path accessing**:
  The concept of multi-path accessing allows many paths leading to a file.
- **Enabling of efficient retrieval facilities**:
  A central database stores meta-data of the file and user specification collectively. Indexing of this database allows the evaluation of queries in a very fast execution time and makes possible efficient retrieval facilities.
- **Tag-based browsing**:
  Tag-based browsing provides the benefits of subdirectories of hierarchical file systems working with subset of files.
- **Formal model of semantic file systems**:
  The formal model of the semantic file system provides a precise definition of the file system behavior.
- **Automatic tag extraction**:
  User tags are completed by tags extracted automatically from the document itself. This rather specific information allows a more precise specification and improved retrieval facilities.

2. Related work

In general, the mechanism of tag-based retrieval is widely used in a variety of information systems that are required to process large quantities of data. Examples range from software applications for engineering data analysis and computer-aided product development to more general information systems that manage text, audio [17], video, photo [19], and mail files. In this section, we provide a brief overview of previously developed approaches for and elements of tag-based retrieval that are of relevance to the work presented in this paper.

Mendeley Desktop [25] is well known software for indexing and organizing Portable Document Format (PDF) documents and research papers into a personal digital bibliography. Document details are gathered from PDFs allowing one to effortlessly search and organize a bibliography. Similarly, indexing and tag-based technology is also used by many other retrieval systems for engineering-related documents [20,24]. Unfortunately, these approaches are isolated software solutions tailored to very specific and narrow contexts and application fields. Although many tag-based approaches share the central goal of efficient document retrieval, the approach presented in this paper addresses the problem from a different perspective. Instead of developing another isolated software system that would only lead to more compatibility issues, a new file system that may be embedded (or plugged) into engineering software applications via a software layer is proposed. Therefore, most of the following literature review focuses on approaches and extensions of traditional file systems.

In a number of approaches, associative access to documents is integrated into a tree-structured file system through the concept of a virtual directory ([10,26]). These virtual directories are interpreted as queries and provide flexible associative access to files and directories. In contrast to separating between real and virtual directories, in the approach presented in this paper tree-structured directories are replaced by a structure of keywords that do not distinguish between pathnames and filenames.

A file system that provides both simultaneous name and content-based access to files is presented in [11]. The concept of a semantic directory is introduced, associating every semantic directory with a query. Whenever a user creates a semantic directory, the file system automatically creates a set of pointers to the files in the file system satisfying the query associated with the directory.

In [27], the topic of file semantics is split into three classes: property-based semantics, content-based semantics and context-based semantics. Property-based semantics is defined as general content and context independent information related to files, such as owner, creation date, last modified date, and the type of file. In many approaches, these semantics are used for all files and shared for all applications and users. In the content-based semantics class, the internal organization of file content is represented. Roles are used to specify text and provide text-based searching. [27] defines context-based semantics as the information about the interrelated conditions in which a file exists. Therefore, context-based semantics express relationships of files with other subjects, such as other files, concepts, persons, etc.

The first archival file system that integrates semantic storage and retrieval capabilities is presented in the system Sedar [22]. Sedar introduces several novel file system features such semantic-hashing to reduce storage consumption, virtual snapshots of the namespace, and conceptual deletions of files and directories.

Connections [31] is a file system search tool that combines traditional content-based search with context information gathered from user activity. Connections can identify relationships between files by tracing file system calls and use them to expand traditional content search results.

Desktop search tools such as Google Desktop [12] and Glimpse [23] use hierarchical file systems to create an index of the stored files. The index provides a fast, associative text-based file search. In contrast to the approach presented here, support of these tools is limited to searching and does not support the creation and management of the file system taxonomy.

In [6], Semantic Web technologies are used to replace the hierarchical file system for file management. Semantic classification is used to support the transition of semantic file system interfaces. Additionally, a file system path construction is presented using semantic web ontologies and Resource Description Framework (RDF) data to build generic file system interfaces.

The proposed approach focuses on property-based and content-based semantics of files in order to share meta-information between several users and to use it in enhanced search facilities. Information relevant only to individuals should not be stored. In these regards, approaches for personal information management can be found under the topic of semantic desktop research [34].

Another well-known approach that uses file meta-data to organize and retrieve documents from relational database systems is

WinFS by Microsoft [13]. WinFS was planned to become a part of an upcoming version of Microsoft Windows, but has not yet been realized. In WinFS, a SQL database system that supports powerful queries is integrated into the operating system. Attributes are used to tag a data item's meta-data, and relationships are used to associate data items together. The semantic file system presented here focuses on meta-data and does not provide capabilities such as file relationships. However, the presented file system takes previously developed meta-data concepts a step further by providing automatic meta-data extraction, meta-data indexing or multi-path accessing, which leads to robust and efficient retrieval mechanisms with fast performance.

[5] develops and evaluates several new approaches to automatically generate file attributes based on context. Context-based attribute assignment is realized by introducing approaches from two categories: application assistance and user access patterns.

Other systems using semantics in conjunction with their data include the Semantic Web and the Semantic Desktop [27]. The World Wide Web Consortium (W3C) promotes Semantic Web technologies that enable people to share content beyond the boundaries of applications and websites [1]. The Semantic Web relies on formal ontologies that structure underlying data and specifies a representational vocabulary for a shared domain of discourse [28]. The core idea of Semantic Desktops is to bring Semantic Web technologies to the desktop. People are enabled to use their desktop computers like a personal Semantic Web where applications integrate and ideas are connected through ontologies [29].

3. A semantic file system for integrated product data management

In this section, the new semantic file system SIL (Semantics Instead of Location) is introduced. First, the approach is presented from the perspective of an engineer working with the system. The associated concept of tag-based browsing is then presented in Section 3.2, along with a demonstration of how the system can be used in file retrieval facilities. Second, a formal mathematical model that specifies the properties of the file system is defined in Section 3.3. Finally, the important issue of identifying document tags by extracting them from existing documents is addressed in Section 3.4.

3.1. The semantic file system – semantics instead of location (SIL)

The presented file system paradigm to store and search for files is based on a set of tags that describe the content-based semantics of the file. These keywords are used to identify and locate files. In this context, *tagging* means to *tag* files with additional information that describes the files.

According to the application of relations in other semantic representation structures, e.g. [32], semantics are represented by relationships. In conceptual graphs or ontologies, the concept of role types is used to describe interactions of instances. However, in contrast to these formalisms, roles do not connect classes or individuals in SIL. Instead, roles assign keywords that describe the data file to which they are attached.

A *tag* consists of two components:

- optional role name
- mandatory keyword

The role name is used to clarify the semantics of the tag and its corresponding keyword. Further, roles make the semantics of the associated keywords more precise and meaningful. When the optional role name is missing, the corresponding keyword is attached without additional description. The notation of *keyword* is used as a synonym to *attribute* of related approaches. The keyword defines the value of the tag. The type of value in regards to the role name and keyword is defined as a string.

The following information is relevant and mandatory for *all* documents and is, therefore, stored separately and not modeled by tags:

- Creation date
- Last modified date
- Version number
- Document size
- Document type

Mandatory information is defined as property-base semantics within the meta-model itself (see Section 4) and can be identified automatically by the system. All other optional information is modeled by tags, which provide a more flexible way of defining information relevant for single documents.

As an example, a component within a mechatronic system may include a large number of attributes needed for its identification. Examples include component id number, functional descriptor, machine or assembly in which the component is used, design date, and so forth as well as models or meta-models describing templates of system components or functions. In contrast to standards that may define predefined attributes for specific file types, SIL is not limited to specific file types and their attributes.An example of a CAD file of a drive shaft designed in 2008, used in a machine of type A, and sold to customer B is summarized in Fig. 1. This figure shows two tags using a role to describe the semantics of the corresponding keywords, and two keywords without a role showing a general characterization of the drawing file's content.

In contrast to tree-structured file systems storing the drawing, for example, in the file '/2008/DriveShaft/CustomerB/machine.dwg', these keywords are not ordered. When a user searches a drive shaft by giving the customer name, all drive shafts ever used in any machine sold to that customer are selected. Alternatively, when the user gives a design year, all drive shafts designed in the given design year are presented. In traditional directory structures, the drive shaft is stored in several subdirectories. This property of SIL is called *multi-path accessing* in [27].

3.2. Tag-based browsing

When presenting a semantic file system that does not provide hierarchical directory structure to engineers, we were often told that the concept of subdirectories is missing. Subdirectories provide the ability to group a huge amount of files in subgroups and allow work with files in a more manageable amount of documents. In this section, it is shown that SIL allows for utilizing subgroups in a much more flexible way. Using tag-based browsing actually provides a visualization of the semantic file system related to the tree-structure of traditional file systems.

Role	Keyword
Design Year	2008
Part Name	Drive Shaft
	Customer B
	Drive Shaft of Machine A

Fig. 1. A tagging example of a CAD drawing.

When searching for stored documents in file systems, generally two search facilities are possible:

- Searching by query
- Searching by browsing

First, users can create a query describing the documents for which they are searching. A search engine evaluates the submitted query and returns the set of files containing matching documents. This way of searching for documents demands that the user knows what he/she is asking for. When the user does not know the keywords of the wanted document, a browsing search method is necessary in presenting the user tags of documents and letting him/her choose these keywords that best describe the wanted document. SIL not only provides both search procedures separately, but also in combination. Therefore, it is possible to search by browsing on the results of a query or to ask a query on a selected subgroup selected by a browser. Since the first search facility can be realized by simple input fields of a user interface, in this paper we focus on the second, more interesting facility.

The presented approach to tag-based browsing allows users to browse files in a search tree similar to a traditional directory structure. Since tags in the SIL system show a structure similar to directory structures, the selected keywords can be presented in a search structure similar to directory names in traditional file systems. This leads to a new search procedure called *tag-based browsing*. A browsing tool can use this structure to give users the chance to browse documents and tags when searching for documents if the set of keywords are unknown. At the beginning of the search procedure, no keywords are selected and therefore all files with all their available keywords are presented. Therefore, at the root of the structure, all keywords are shown that are used to describe the stored files. Additionally, all documents that are tagged by all selected keywords can be shown.

In contrast to subdirectories in hierarchical file systems, the 'explorer tool' shows tags that are associated with the selected files and which are qualified to limit the number of selected documents by additional keywords. These tags are referred to as *option tags*, while the selected keywords are simply named *selected tags*. The set of files selected by the selected tags are called *selected files*.

When an additional tag is selected, the following happens:

- The set of selected tags is added by the new tag
- The set of selected files is reduced to the subset of files that are tagged with this keyword
- The set of option tags is reduced to the subset of tags appearing in the selected files

The property, that the number of optional tags and the number of selected files decreases when the user adds keywords to his set of selected keywords, ensures that the search process leads to the files being sought. In the next section, our formal model of tag-based browsing provides a formal specification of the system behavior and proves this important "conjunctive" property of selected file reduction.

In Fig. 2, a visualization of a browser realized in an application similar to current file explorers is shown. A ' + ' indicates that the diagram is expanded at this node, a '-' indicates that it is not expanded. A mouse click on an expanded node contracts the graph; a mouse click on a contracted node expands it. At the root of the browsing tree, all keywords that are used to tag files are presented and all files of the file systems are selected.

At first glance, tag-based browsing appears to be very similar to traditional tree browsing. Therefore, the differences of the presented approach need to be emphasized. Tag-based browsing provides multi-path access, which means that when a document

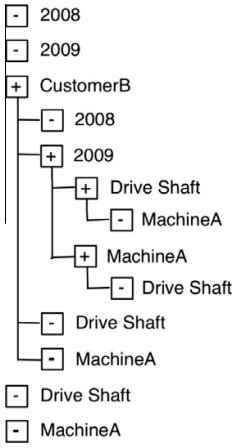

Fig. 2. Tag-based browsing.

appears in the browsing structure in A/B, it also appears in B/A where A and B are directories. In the example of Fig. 2, the drawing of the drive shaft is located in '2009/CustomerB/MachineA/DriveShaft' as well as in 'CustomerB/2009/MachineA/DriveShaft' or 'CustomerB/MachineA/DriveShaft/2009', meaning that the result of the query is independent from the order in which tags are selected. Additionally, we emphasize that no separation between filename and directory name exists even when the browsing diagram shows a graph similar to a directory tree.

Even this simple example clearly highlights some of the advantages of SIL compared to traditional file systems. Additionally, when a document is stored in 'DriveShaft/CustomerB/2009' in hierarchical file systems and a user searches in /2009, he will not find the document since he is searching in a false directory subtree. However, a document's location and its name are not relevant in SIL. With computer-aided engineering tools, the complexity and amount of data is significant and, therefore, the benefits of SIL have much more impact on the search process. Regardless of these differences in the underlying structure, the browser appears very familiar to users, and first tests of the prototype shown in Section 4 have shown that users are working easily with the browsing structure.

3.3. Formal model for semantic file systems

In this section, a formal mathematical model of the concept of tag-based browsing is provided. In order to make the formal model easier to understand, an informal textual specification of tag-based browsing was presented in the previous section. The formal model defines the behavior of the semantic file system, which prevents inconsistencies and incompatibilities between different implementations of the semantic file system's specification.

Let R be a set of roles, K a set of keywords, and F a set of files. The set of tags is defined as $T = R \times K$.

The mapping g is defined as g: $F \to \wp(T)$, where g maps a file to a set of tags that are associated to the corresponding file[1].

S is defined as the set of tags selected by a user in a search process (*selected tags*). We have: $S \subseteq T$. At the beginning of the search process, we assume $S = \varnothing$.

[1] In ontologies, conceptualization is a set of conceptual relations defined on a domain space [14,33]. Since objects are attached to keywords in the presented approach, a function is used to define roles here.

Let F_S be defined as the set of files that are conjunctively associated to all tags of the set S.

Definition : $F_S = \{f \in F \mid S \subseteq g(f)\}$ (1)

Proposition 1 : $S1 \subseteq S2 \Rightarrow F_{S2} \subseteq F_{S1}$ (2)

Proof: Let us assume the contrary, i.e.: $S1 \subseteq S2$ and $F_{S2} \not\subseteq F_{S1}$, i.e., $\exists f \in F_{S2}, f \notin F_{S1}$. According to (1) we have $S2 \subseteq g(f)$. $f \notin F_{S1} \Rightarrow \exists t \in S1 : t \notin g(f)$, i.e. we have $S1 \not\subseteq S2$ and a contradiction. Let $OS \subseteq T \setminus S$ be the set of *option tags* that allows a more precise selection on the selected files.

Definition : $O_S = \{t \in T \setminus S \mid \exists f \in F_S : t \in g(f)\}$ (3)

Proposition 2 : $F_{S2} \subseteq F_{S1} \Rightarrow O_{S2} \subseteq O_{S1}$ (4)

Proof: Let us assume the contrary, i.e.: $F_{S2} \subseteq F_{S1}$ and $O_{S2} \not\subseteq O_{S1}$, i.e. $\exists o \in O_{S2}, o \notin O_{S1}$. $o \notin O_{S1} \Rightarrow \neg\exists f1 \in F_{S1} : o \in g(f1)$. According to (3) we have $\exists f2 \in F_{S2} : o \in g(f2)$ and a contradiction to $F_{S2} \subseteq F_{S1}$.

Proposition 3 : $S1 \subseteq S2 \Rightarrow O_{S2} \subseteq O_{S1}$ (5)

Proof: Transitivity of (2) and (4). Proposition 3 verifies the decrease of selected files and option tags when additional tags are selected.

3.4. Extracting file content

Detailed and comprehensive semantic information are a premise for efficient retrieval facilities with complete retrieval results. The quality of the meta-information of files has a great impact on the effectiveness of the entire semantic file system. In general, tags that classify files based upon a set of keywords have to be entered manually. These keywords have to be selected carefully since they are used to identify files and are the only way to get the files back from the file system. Even if this activity is very time-consuming, the file creator has the best understanding of their files and is able to describe his/her file precisely. User input is often regarded as a drawback since users worry about locating their file again after storing it in SIL. However, it has to be remembered that when using traditional file systems, files are lost as well when the user doesn't remember the pathname and filename of his/her document.

While filenames and pathnames are mandatory in traditional file systems, the SIL system has to enforce that files are adequately specified. For this reason, we describe below how content analysis is realized by extracting tags automatically from the document.

Meta-data information can be used in search tools to provide enhanced search facilities. However, search tools using this meta-data are restricted to single standards and users have to switch between several tools when searching for files of different types. Extracting file meta-data to SIL makes this hidden information visible and provides a uniform search facility to all meta-data. Meta-data can be used in retrieval facilities with good performance when the meta-data is stored in a central database and not in thousands of files itself. In database systems, indexing allows the evaluation of queries on meta-data in a very fast execution time, which makes efficient retrieval possible at all.

Additional to one's manual specification of meta-data, the SIL approach considers the following automatic tagging techniques.

- Extraction of meta-data stored in the files themselves
- Extraction of meta-data from file content
- Reuse file or directory names of files stored in traditional file systems as tags

One technique to collect file content without manual input is to extract meta-information stored in the file itself. This meta-data provides higher level information about the files concerned. Values of this meta-data can be processed by many software systems. Each meta-data entity is called a property, and properties are grouped into administrative, descriptive and rights related properties.

In the architecture of a prototype of the presented approach, a meta-data extractor is introduced that extracts different meta-data into the semantic file system (Fig. 3). Since the meta-model of the file system is generic and not limited to certain attribute names, meta-information of all standards can be transformed into the system. A standard-specific filter has to be provided for each meta-data standard in order to integrate the meta-data into the file system.

The extraction of meta-data from file content requires intelligent mechanisms for analyzing the content of specific file types. An example of a system doing this extraction for PDF files is the system Mendeley [25], which extracts references of research PDF papers.

4. Methan – SIL prototype

The presented approach to realizing a new semantic file system, SIL, has been implemented as a prototype system called *Methan* in order to prove the concept, validate it and show its applicability. Methan is a Java-based application providing flexible associative access facilities by a programming interface, a shell command interface and a graphical explorer. The architecture of Methan is depicted in Fig. 4.

The graphical design of the Methan explorer, shown in Fig. 5, is similar to that of other explorers within current operating systems in order to facilitate a wider acceptance among users. The explorer consists of a browser that shows the structure shown in Fig. 1 and allows searching for documents by expanding and contracting nodes of the graph. Even if the structure presents no tree, the structure is called a *tag tree*. Additionally, an extended search is provided by text input. When keywords are typed into the input field, the structure is expanded automatically according to the searched keywords. This allows users to input known keywords and subsequently browse in the tag tree.

A functional filesystem is a userspace program that can be implemented easily using FUSE [9]. This module allows non-privileged users to create their own file systems without editing the kernel code and is available for several operating systems. The shell command interface of Methan expands the UNIX shell commands by instructions for operating with SIL. UNIX commands like *cd* or *ls* operating on hierarchical file systems are replaced by commands that operate on SIL. Here commands are added like *addTag* to add a tag, *remTag* to remove a tag, *lsfiles* to list files with the given keywords, or *lsopTag* to list option tags.

The extraction of ID3-meta-data of audio documents is realized in the Meta-data Extractor by the Jaudiotagger User API v1.0.9 [15]. IPTC-data of photos is extracted using Imagero 3.0 programming interface [18].

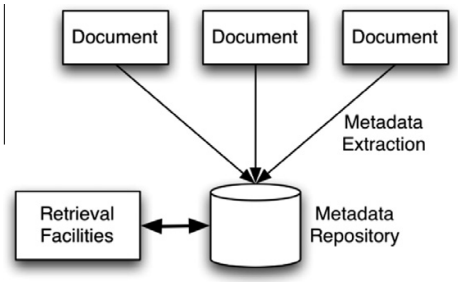

Fig. 3. Metadata Extraction.

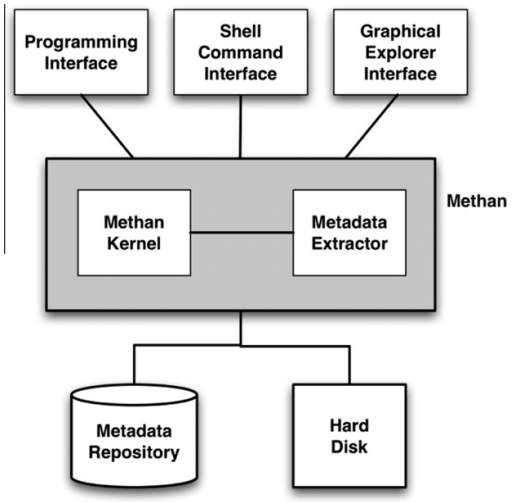

Fig. 4. Methan – a prototype of SIL.

Accessing documents from the hard disk is realized on top of the NTFS file system. This is of course a temporary solution since the goal of the file system is to replace current file systems. But as a first prototype, our implementation provides a feasible solution for the purpose of evaluating the benefits of our proposed semantic file system. Surprisingly, in spite of this rather inefficient implementation, the efficiency of the file system exceeded expectations even with a huge amount of data used. The Meta-data Repository is realized by the H2 database system [16]. H2 is a very fast, open source JDBC-API, which allows SQL-scripts to run in embedded mode. An evaluation of several database systems showed that the JDBC-API, a small but fast database, is more adequate to fulfill the requirements of Methan than fat client server database systems.

Fig. 6 shows the entity relationship model of the H2 database system representing the meta- model of SIL. Property-based semantics are represented by the attributes *CreateDate*, *LastModDate*, *MmeType*, *Size* and *VersionNr*. Documents are identified not only by their keywords, but also by a version number. This version number supports the integrated product data management working on several versions of documents. Each time a file is modified and closed, a new version of the file is produced. Similar to [22], meta-data is not versioned. Therefore, version information and tags are represented in different entites in the model. Attributes *PhysPath* and *PhysName* describe the location of files that are stored in the NTFS file system. When the semantic file system is not realized on top of a hierarchical file system, these attributes have to be replaced by attributes indicating the physical storage location on the hard disk. In order to speed up retrieval of data, attribute *Keyword* of entity *Tagvalue* and attribute *Rolename* of entity *Role* are indexed in the H2 database system. As first tests have indicated, these indexes allow for excellent retrieval performance and thus make the retrieval performance one of the main benefits of the Methan system.

5. Closing

The presented semantic file system SIL is applicable to all operating systems and all fields of application. The advantages of the file system become important when users are required to work with a large amount of complex documents that relate to significant amounts of meta-data. Therefore, fields such as computer-aided engineering and design, PDM, PLM, etc. are predestined for applying the semantic file system. However, in order to demonstrate the prototype system without having to gain access to a large number of CAE-related data files, Methan was evaluated and tested while managing approximately 5000 photos and audio files with great success. This demonstrates that SIL is highly transformative in nature and actually can be utilized in any application domain for which meta-information can be extracted from files. Feedback from alpha testers confirms that SIL is very easy to use and that a significant increase in efficiency with regard to search tasks can be achieved.

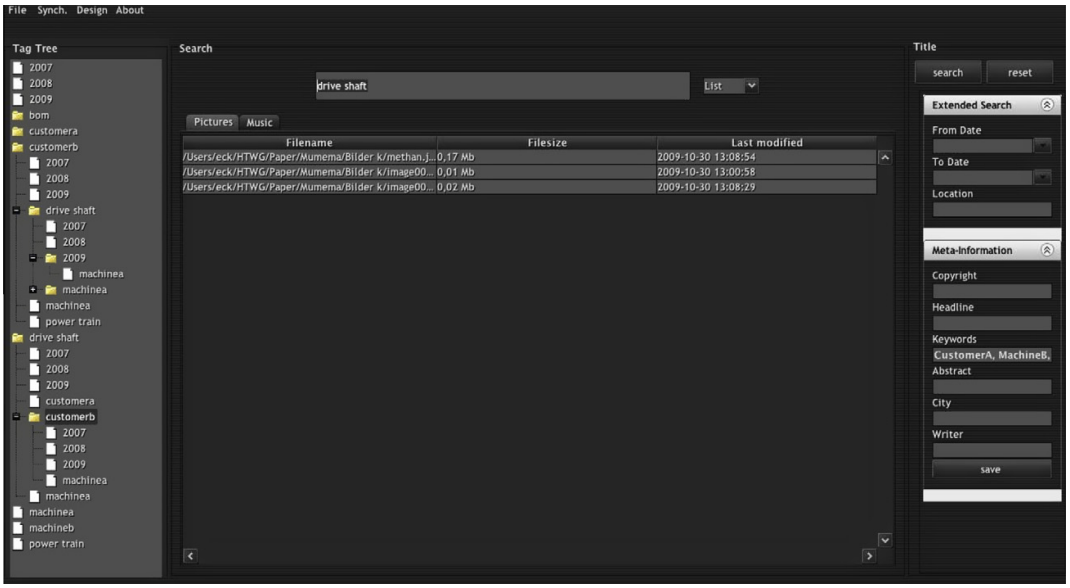

Fig. 5. The graphical explorer interface of Methan.

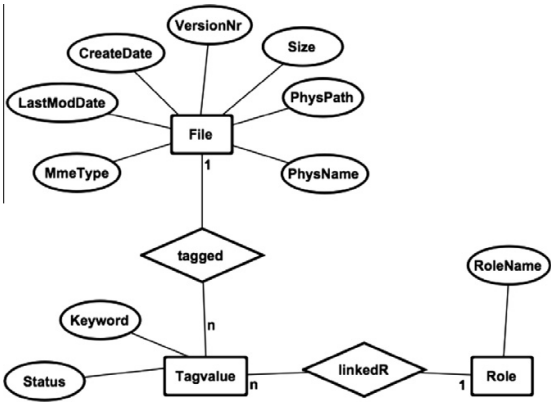

Fig. 6. The meta-model of the Meta-data Repository.

In order to get deeper semantics in describing digital objects and enable to share meanings of terms to overcome barriers between several engineering departments, it is planned to integrate ontologies into our semantic file system. Related works have shown great promise for integrating ontologies into semantic file systems [27]. The goal is to use ontologies to enhance knowledge sharing by specifying property-based semantics, content-based semantics, and context-based semantics.

An extension our proposed semantic file system is planned to integrate context-based semantics. Context-based semantics would allow representing the fact that two documents represent audio and visual information of the same object, which would be very helpful in search facilities.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, The Semantic Web, Scientific American, 2001.
[2] D.M. Buede: The Engineering Design of Systems: Models and Methods, Wiley, 2009, ISBN 0470164026.
[3] K. Chen, J. Bankston, J. Panchal, and D. Schaefer, A framework for the integrated design of mechatronic products", In: L. Wang and A.Y.C. Nee. (Eds.), Collaborative Design and Planning for Digital Manufacturing, Springer, ISBN 184822863, pp. 37–70.
[4] W. Cheung, and D. Schaefer, Product lifecycle management: state-of-the-art and future perspectives, In: M.M. Cruz-Cunha (Ed.), Enterprise Information Systems for Business Integration in SMEs: Technological, Organizational and Social Dimensions, (Advances in Information Management book series), IGI Global Publishing, ISBN 978-1-60556-892-5, pp. 37–55.
[5] A. Craig, N. Soules, Gregory R. Ganger, Toward automatic context-based attribute assignment for semantic file systems, Technical report CMU-PDL-04-105. June 2004.
[6] S. Faubel and C. Kuschel, Towards Semantic File System Interfaces, 7th International Semantic Web Conference ISWC2008, 2008.
[7] S. Friedenthal, A. Moore, R. Steiner, A Practical Guide to SysML, Morgan Kaufmann, 2009, ISBN 012378607X.
[8] P. Fritzson: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley, 2004, ISBN 0471471631.
[9] http://fuse.sourceforge.net/.
[10] D.K. Gifford, P. Jouvelot, J.J. O'Toole, and M.A. Sheldon, Semantic File Systems, ACM Operating Systems Review, Oct. 1991, pp 16–25.
[11] B. Gopal and U. Manber. Intergrating Content-based Access machanisms with Hierarchical File Systems. in: Usenix OSDI. 1999. New Orleans, Louisiana, USA.
[12] Website of Google Desktop: http://desktop.google.com.
[13] Richard Grimes: Revolutionary File Storage System Lets Users Search and Manage Files Based on Content, http://msdn.microsoft.com/en-us/magazine/cc164028.aspx.
[14] N. Guarino (Ed.), Formal Ontology in Information Systems, in: Proceedings of FOIS'98, Trento, Italy, 6-8 June 1998. Amsterdam, IOS Press, pp. 3–15.
[15] http://www.jthink.net/jaudiotagger.
[16] Website of H2 database system: http://www.h2database.com.
[17] Website of ID3: http://www.id3.org/.
[18] Website of Imagero: http://reader.imagero.com.
[19] Website of the International Press Telecommunications Council: http://www.iptc.org.
[20] Z. Li, V. Raskin, and K. Ramani, 2007. Developing ontologies for engineering information retrieval. in: Proceedings of the ASME 2007 IDETC/CIE 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Las Vegas, Nevada, USA, September 4–7, 2007. pp. 1–9.
[21] S.C. Lim, Y. Liu, and W.B. Lee, 2009. Faceted search and retrieval based on semantically annotated product family ontology. in: Proceedings of the WSDM '09 Workshop on Exploiting Semantic Annotations in information Retrieval, Barcelona, Spain, February 09-09, 2009). ESAIR '09. ACM, New York, NY, pp. 15–24.
[22] Mahalingam, C. Tang, Z. Xu, Towards a semantic, deep archival file system, in: The 9th International Trends of Distributed Computing Systems (FTDCS), 2002.
[23] U. Manber, S. Wu, Glimpse: a tool to search trough entire file systems. in: Proceedings of the USENIX Winter Conference, 1994, pp. 23–32.
[24] C. McMahon, A. Lowe, S. Culley, M. Corderoy, R. Crossland, T. Shah, D. Stewart, Waypoint: An integrated search and retrieval system for engineering documents, Journal of Computing and Information Science in Engineering 4 (4) (2004) 329–338.
[25] http://www.mendeley.com.
[26] P. Mohan, V.S. Raghuraman, A. Siromoney, Semantic File Retrieval in File Systems Using Virtual Directories, 13th Annual IEEE International Conference on High Performance Computing (HiPC), Bangalore, India, 2006.
[27] Hung Ba Ngo, C. Bac, F. Silber-Chaussumier, Thang Quyet Le, Towards Ontology-based Semantic File Systems, 2007 IEEE International Conference on Research, Innovation and Vision for the Future, RIVF 2007, 5–9 March 2007 pp. 8–13.
[28] J.M. Park, J.H. Nam, Q.P. Hu, H.W. Suh, Product Ontology Construction from Engineering Documents, International Conference on Smart Manufacturing Application, 2008. ICSMA 2008.
[29] L. Sauermann, A. Bernardi, and A. Dengel, "Overview and Outlook on the Semantic Desktop," Proceeding of Semantic Desktop Workshop, 2005.
[30] A. Shah, D. Schaefer, C. Paredis, Enabling Multi-View Modeling with SysML, International Conference on Product LIfecycle Management, July 6–8, University of Bath, UK, 2009.
[31] A. Craig, N. Soules, Gregory.R. Ganger, 'Connections: Using Context to Enhance File Search', Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, Brighton, UK, 2005 (pp. 119–132).
[32] J.F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, MA, 1984.
[33] C. Tziviskou1 and C. Maria Keet, A Meta-Model for Ontologies with ORM2, Springer Berlin/Heidelberg, 2009.
[34] H. Xiao and I.F. Cruz, A multi-ontology approach for personal information management, in: Proceeding of 1st Workshop on The Semantic Desktop, 2005.

6.4.3 Semantic File Retrieval in File Systems using Virtual Directories

1

Semantic File Retrieval in File Systems using Virtual Directories

Prashanth Mohan, Raghuraman, Venkateswaran S and Dr. Arul Siromoney
{*prashmohan, raaghum2222, wenkat.s*}*@gmail.com and asiro@vsnl.com*

***Abstract*— Hard Disk capacity is no longer a problem. However, increasing disk capacity has brought with it a new problem, the problem of locating files. Retrieving a document from a myriad of files and directories is no easy task. Industry solutions are being created to address this short coming.**

We propose to create an extendable UNIX based File System which will integrate searching as a basic function of the file system. The File System will provide Virtual Directories which list the results of a query. The contents of the Virtual Directory is formed at runtime. Although, the Virtual Directory is used mainly to facilitate the searching of file, It can also be used by plugins to interpret other queries.

***Index Terms*— Semantic File System, Virtual Directory, Meta Data**

I. Introduction

FILE and directory management is an essential and inevitable part of everyday computer usage. With hard disks growing in size by leaps, we are faced with the problem of locating files. Conventional file systems impose a hierarchical structure of storage on the user – a combination of its location and filename [2]. Features like 'symbolic links' allow a file to be accessed through more than one path. However, a strict enforcing of the path which does not necessarily depict the meaning of the file itself still exists for all files.

The world has come to realize that the need for efficient and effective file retrieval methods and thus the industry has responded with software like Google's Desktop Search[1], Apple's Spotlight[2], Beagle[3], Microsoft's proposed WinFS[4]. Locating a file on a large hard disk is tough unless we know exactly where the file is located.

A. Semantic Structure

The problem with the present heirarchial storage system is that the 'semantic' i.e. the meta data information of the file is not given adequate importance. The main semantic of the stored file is the directory in which it is stored in. To cite from *O. Gorter's* thesis [2], let us take a example of '`/home/user/docs/univ/project/file`'. Now, the property associated with the file is that of 'project' but not as much of univ or of documents. A listing of the `/home` directory does not list the file but a listing of the '`/home/user/docs/univ/project`' directory lists the file. The 'Database File System' [2], encounters this problem by listing recursively all the files within each directory. Thereby a funneling of the files is done by moving through sub-directories.

The objective of our project (henceforth reffered to as *SemFS*) is to give the user the choice between a traditional heirarchial mode of access and a query based mechanism which will be presented using virtual directories [1], [5]. These virtual directories do not exist on the disk as separate files but will be created in memory at runtime, as per the directory name. The name of the directory itself will be a query which the File System driver will parse and populate the virtual directory.

II. General Information

SemFS provides an intuitive way of browsing the file system. It lets one move within the file system based on the file's meta-data and attributes. The meta-data of files will not be common. All files will possess the attributes 'owner' and 'last modified date', but a JPEG file would also choose EXIF data like height and width as it's meta-data, while an MP3 file would choose id3 data like length, artist, album, etc as it's meta-data.

A. Features

SemFS provides:

- Searching as a basic function of the File System
- Browsing the File System based on the file's meta-data and attributes
- An easy and intuitive way to retrieve your required file (It is only natural that one remembers what the file is and not where the file is)
- Use of logical operators to filter results – ^ is AND, | is OR, ! is NOT
- An API to create 'views', which can be thought of as a persistent virtual directory. It will be updated automatically and the clients notified in case of any updates

B. Usage

A typical way of using SemFS would be:

1) Mount the File System using the driver
2) Chdir into the mounted directory
3) Do '`cd type:mp3^len>3m^artist:Mike`'

Dr. Arul Siromoney is with the College of Engineering, Guindy in India

[1] http://desktop.google.com/
[2] http://www.apple.com/macosx/tiger/
[3] http://beagle-project.org/Main_Page
[4] http://msdn.microsoft.com/data/ref/winfs/

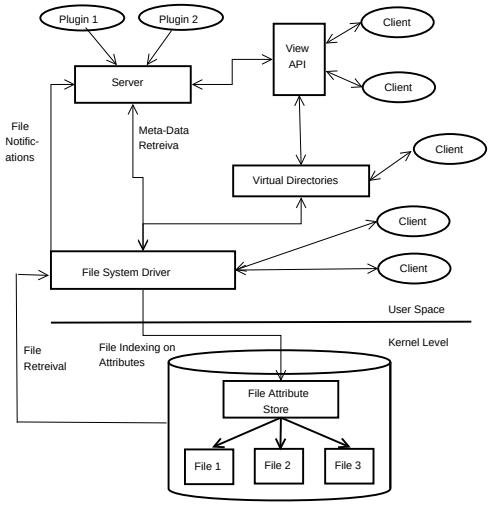

Fig. 1. Design of SemFS

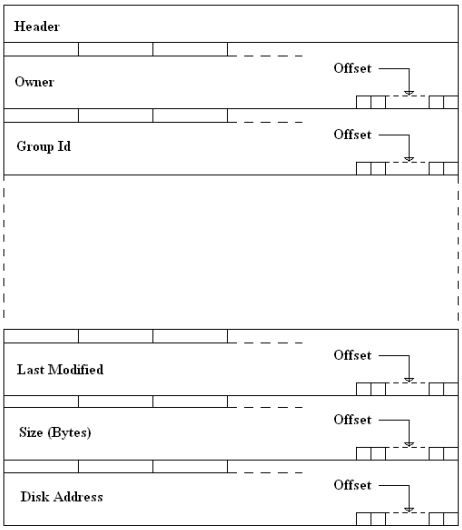

Fig. 2. Modified Inode Structure

4) The user is now chdired into a virtual directory which lists all MP3 files sung by 'Mike' of length greater than 3 minutes. (The queries will also support indexing based on file attributes like owner, file size, etc)

III. Design

The design of SemFS is meant to be easily 'extendable' or 'pluggable'. SemFS consists of 3 main components – The File System Driver, the SemFS daemon server and the SemFS API. SemFS works on the Server-Client architecture (Refer Figure 1).

A. File System Driver

SemFS will be a user space file system which can make use of the FUSE[5] or LUFS[6] libraries. We also considered the use of GNU/HURD translators [13], however, considering the wide use of FUSE and its active development, we decided to go ahead with FUSE. FUSE has bindings for a number of languages (including 4^{th} Generation languages like Python).

Apart from the user space daemon, SemFS also plans to change the storage of file attributes in the File System layout in order to optimize the indexing of files based on it's attributes (Refer §III-B.1). Hence, a kernel module will also be involved. However, the File System will also work without the kernel module, thereby keeping the project portable across most UNIX based Operating Systems. In the case that the kernel module is not installed, then the default ext3 based storage of files is used. In such a case, the indexing of file based on its attributes will not be optimized.

[5] http://fuse.sourceforge.net/
[6] http://directory.fsf.org/all/lufs.html

B. File Store

The files and directories itself will be stored in a partition or a special device file. The file system will support journaling (reusing ext3's data storage mechanism) and will be stored in a heirarchial fashion. Upon storing the file each time, it's meta-information is updated in the databases.

This meta information could either be stored in a in-kernel database [12], [14] (the KBDBFS project aims to maintain a Berkeley Data Base inside the kernel) or maintained in user space. We store some of the meta data in a user level database (a SQLite database should suffice). We store the File Attributes as usual in the inodes of the files. There will be no redundancy of file attributes, thereby doing away with a lot of race conditions. This will improve the look up time for such meta information.

1) File Attribute Storage: The internal representation of the file is given by an inode, which contains a description of the data layout of the file data. When a process refers to a file name, the kernel parses the file name – one component at a time, checks that the process has permission to search the directory or access the file, and eventually retrieves the inode for the file.

The problem in the current structure of the inode list is that when a query is executed wherein we index the files based on its attributes, we have to look into the irrelevant data of the inodes of all files. This naturally slows down querying since we would be wasting a lot of cycles due to unnecessary disk reads. Say, we want to query a set of files based on its file size, we would still need to read the other attributes like last modified date, time, etc although what we want to read is only the file size field. If however, we were able to read the size information of all the files in a single (or more) block access, the query speed would increase multi fold.

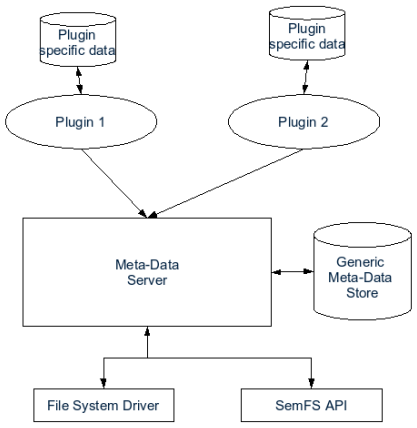

Fig. 3. Design of the SemFS Server

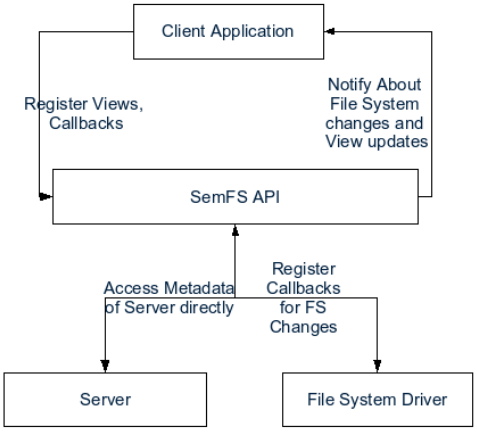

Fig. 4. Design of the SemFS API

In order to read the relevant data in fewer block access, the structure of an inode has been modified. Figure 2 shows the modified inode list. Searching based on a particular field of the inode is optimized since, the same field of all the files are grouped together in a minipage [15]. A minipage is a logical separation of similar attributes in the table. The rest of the file system structure including super block and directories remain the same as the UNIX file system layout. The advantages of using such a structure are:

- It maximizes inter-record spatial locality within each column in the page, thereby eliminating unnecessary requests to main memory without incurring space penalty
- Incurs a minimal record reconstruction cost
- It is orthogonal to other design decisions because it only affects the layout of data stored on a single page

The offset at the end of a minipage are actually binary bits denoting the availability of data in each record. If an item is present bit '1' will be present and '0' if it is deleted.

2) Data Storage: The existing storage mechanism of a typical UNIX file system like ext3 is reused. Most of the code base will be reused but for modifications where necessary, i.e. in places where inodes are referenced.

C. Server

The Server (Refer Figure 3) is the core of the Semantic File System. The server will translate the queries for the virtual directories into file listings. The server is also responsible for maintaining the 'Views'.

1) User Space Meta Data Store: As previously explained in §II-A, the meta-data information differs from file to file. The file specific meta-data is extracted and maintained in special databases by the Plugins for the server (Refer §III-C.2). We can store the meta-data in two ways:

- Inside the File System (Refer §III-B.1) – Within the inode List
- Outside the File System – In databases, which can be accessed by all applications

By storing the meta-data information outside of the file system, we will suffer a small performance hit. So, we will store the common file meta-data on disk in order to improve performance, while the file specific meta information will either be stored at the Server Database or databases maintained by the individual plug-ins.

2) Plugins: The SemFS daemon or server supports plugins which can define the logic for the query which is translated into the virtual directories. The plugins can also (usually) maintain their own database which will store the plugin specific meta-data. The work of the plugins include:

- Registering the plugins to the server
- Process and provide logic for the respective queries
- Register call backs for specific file modifications

D. SemFS API

The SemFS API will provide 'views' to the applications which are similar to Virtual Directories but they tend to be persistent in nature. The design of the SemFS API (Refer Figure 4) supports for:

- Support for Views
- Notify clients in case of view updates
- Shield the user from the database

E. Clients

Any application which makes use of the Semantic File System is a client. The use of the virtual directories is extended to all applications. In the case of the Database File System [2] and GLS Cube [5], the access to the meta data based searching is available only by recompiling the applications to use their custom APIs.

The SemFS API only makes the usability experience even better by providing features like automatic updating of 'Views'. All applications will still be able to access the Virtual Directories which offer a limited amount of Semantic information.

IV. Application of User Space Semantics

Hierarchical directory systems are very useful for organizing files, but they can only help to a certain point. SemFS provides the mechanism for easily accessing files based on its meta-data from all applications. Apart from these features, there can be certain extensible features that would highlight on the semantic structure storages.

A. File Tagging

Tagging is a feature that adds a user's logical perception about the file. Although tags do not necessarily define the semantics of the data, they are interpreted by the end-user as being related to a subset of his data, a subset that he logically creates. And contrary to the traditional directories, they are not monotonous. It also represents a relation between files. By this way the end-user can retrieve all documents related to a specific title.

The tags can be either added to a particular file or a group of files. The tag corresponding to a file are stored in the user space database along with SemFS server (which holds some of the file's meta-data).

B. File Versioning

Versioning is one of the feature that SemFS can support:

- Checkpoints and
- Versioning of files

Each time a file is modified, the server will be notified by the SemFS driver. The server checks if any of the plugins are waiting for the event that the file has been modified. If a call back exists, then the function corresponding to the plugin (i.e. the call back function) is initialized. This call back could retrieve the new file and store the diff between the two versions.

This can be used to restore the file to previous content after a certain period of time.

V. Related Work

There are a number of projects already working in similar areas. The need for creating a new architecture is to identify the problems in the current implementations and increase efficiency of retrieval.

A. GLS[3]

The GNU/Linux Semantic Storage System [5] is a solution designed to facilitate the management and retrieval of the data semantically. However, issues regarding consistency, stability and error-recovery exist. It does not offer any sort of error recovery. The inconsistency comes from the fact that GLScube is defined wholly in user-space, and thus, file system events may occur that GLScube does not record.

B. Leaftag

Leaftag[7] is a new project that facilitates the tagging of files. It does not use the file system's extended attributes. For example, when moving a tagged file, tagutils will index it again. This could potentially have side effects. It also lacks RDF features (the tag themselves cannot be nodes) and there is no way of expressing relations other than those in the tag.

C. Beaglefs

Beaglefs implements a file system representing a live Beagle query. The file system represents query hit results as symlinks to the targets. It provides constant time operation using extended attributes and supports many file operations.

References

[1] R. Pike, D. Presotto, and S. D. et al, "Plan 9 from bell labs," AT & T Bell Laboratories, Murray Hill, NJ, Tech. Rep., 1995.

[2] O. Gorter, "Database file system - an alternative to hierarchy based file systems," Master's thesis, University of Twente, August 2004.

[3] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O. Jr, "Semantic file systems," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, Oct. 1991, pp. 16–25.

[4] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis, "Towards a semantic-aware file store," in *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*. USENIX Association, May 2003, pp. 145–150.

[5] A. Salama, A. Samih, A. Ramadan, and K. M. Yousef, *GNU/Linux Semantic Storage System*, 2006.

[6] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, Third Edition*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2005.

[7] N. Murphy, M. Tonkelowitz, and M. Vernal, "The design and implementation of the database file system," Nov. 2001.

[8] N. H. Gehani, H. V. Jagadish, and W. D. Roome, "Odefs: A file system interface to an object-oriented database," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 249–260.

[9] D. Mazieres, "A toolkit for user-level file systems," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 261–274.

[10] J. Nielsen. (1996, Feb.) The death of file systems. [Online]. Available: www.useit.com/papers/filedeath.html

[11] H. Reiser. (2001, Jan.) Name spaces as tools for integrating the operating system rather than as ends in themselves. [Online]. Available: www.namesys.com/whitepaper.html

[12] A. Kashyap, "File system extensibility and reliability using an in-kernel database," Master's thesis, Stony Brook University, Dec. 2004.

[13] Debian. (2006) Translators. [Online]. Available: www.debian.org/ports/hurd/hurd-doc-translator

[14] C. P. Wright, "Extending acid semantics to the file system via ptrace," in *Proceedings of FAST 05: 4th USENIX Conference on File and Storage Technologies*. USENIX Association, Dec. 2005.

[15] J. Zhou and K. A. Ross, "A multi-resolution block storage model for database design," in *Database Engineering and Applications Symposium*, July 2003, pp. 22 – 31, 16 – 18.

[16] Y. Padioleau and O. Ridoux, "A logic file system," in *Proceedings of FAST 03: 2nd USENIX Conference on File and Storage Technologies*. USENIX Association, Mar. 2003.

[7] http://www.chipx86.com/wiki/Leaftag

6.5 Appendix E : Contribution of team members

Member 1: Aseem Gogte

- Mathematical Model
- Testing related to Database
- Extraction of Metadata from Images

Member 2: Sahil Gupta

- Tag relations and behavior
- Designing database queries
- Database Design

Member 3: Harshvardhan Pandit

- Interact with FUSE technology
- Design Program flow
- Component design

Member 4: Rohit Sharma

- Extraction of Metadata from Audio / Video
- Database Design
- Testing related to File operations

6.6 Appendix F : Glossary

Acronyms

1. FUSE - Filesystem in Userspace
2. GCC - GNU Compiler Collection
3. GPL - General Public License
4. API - Application programming interface
5. GUI - Graphical user interface
6. FAQ - Frequently asked questions
7. SFS - Semantic File System
8. VFS - Virtual File System
9. KFS - Knowledge File System

Term Definitions

1. Semantic File System
   Semantic File Systems are file systems used for information persistence which structure the data according to their semantics and intent, rather than the location as with current file systems. It allows the data to be addressed by their content (associative access) and querying for the data.
2. User Space
   User space is that portion of system memory in which user processes run. This contrasts with kernel space, which is that portion of memory in which the kernel executes and provides its services.
3. File System
   A file system is a means to organize data expected to be retained after a program terminates by providing procedures to store, retrieve and update data as well as manage the available space on the device(s) which contain it.
4. Virtual File system
   A virtual file system is an abstraction layer on top of a more concrete file system.
5. Virtual Directory
   A virtual directory is a directory created in IIS to host our applications and to hide the actual physical location from the application users. It may simply designate a folder which appears in a path but which is not actually a sub-folder of the preceding folder in the path.
6. Meta data
   Metadata describes how and when and by whom a particular set of data was collected, and how the data is formatted.
7. Symbolic Links and Aliases
   A symbolic link (also symlink or soft link) is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.

Chapter 7

BIBLIOGRAPHY

[1] Mangold, C. (2007), *A survey and classification of semantic search approaches*, Int. J. Metadata, Semantics and Ontology, Vol. 2, No. 1, Page(s): 23-34.

[2] Google Desktop Search, *http://googledesktop.blogspot.in*

[3] Apple Spotlight, *http://developer.apple.com/macosx/spotlight.html*

[4] Gopal.S, Yang.Y, Salomatin.K, Carbonell.J, *Statistical Learning for File-Type Identification*. 2011 10th International Conference on Machine Learning and Applications , Page(s): 68-73.

[5] Bloehdorn.S, Grlitz.O, Schenk.S, Vlkel.M, *TagFS - Tag Semantics for Hierarchical File Systems*. In Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06), Graz, Austria, September 6-8, 2006.

[6] Gifford.D, Jouvelot.P, Sheldon.M, and OToole.J, *Sematic File Systems*. 13th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Oct. 1991, Page(s): 16-25.

[7] Freund.R, *File Systems and Usability the Missing Link*. Cognitive Science, University of Osnabruck July 2007.

[8] Tagsistant, *http://www.tagsistant.net*

[9] Tagster, *http://www.uni-koblenz.de*

[10] Chang.K, Perdana.I, Jain.M, Kartasasmita.I, Ramadhana.B, Sethuraman.K, Le.T, Chachra.N, Tikale.S, *Knowledge File System-A principled approach to personal information management*. 2010 IEEE International Conference on Data Mining Workshops, Page(s): 1037-1044.

[11] Mohan.P, Venkateswaran.S, Raghuraman, Dr.Siromoney.A, *Semantic File Retrieval in File Systems using Virtual Directories*. Proc. Linux Expo Conference, Raleigh, NC, Page(s): 141-151, May 2007.

[12] Hua.Y, Jiang.H, Zhu.Y, Feng.D, Tian.L, *Semantic-Aware Metadata Organization Paradigm in Next-Generation File Systems*. IEEE Transactions On Parallel And Distributed Systems, Vol. 23, No. 2, February 2012, Page(s): 337-344.

[13] Schroder.A, Fritzsche.R, Schmidt.S, Mitschick.A, Meiner.K *A Semantic Extension of a Hierarchical Storage Management System for Small and Medium-sized Enterprises*. Proceedings of the 1st International Workshop on Semantic Digital Archives (SDA 2011).

[14] Eck.0, Schaefer.D, *A semantic file system for integrated product data management*. 2011 Advanced Engineering Informatics, Page(s): 177-184.

[15] File system in USERspace (FUSE) homepage and documentation, *http://fuse.sourceforge.net*

[16] SQLite database *http://www.sqlite.org*