ABSTRACT

The limitation of data representation in today’s file systems is that data representation is bound only in a single way of hierarchically organising files. A semantic file system provides addressing and querying based on the content rather than storage location. Semantic tagging is a new way to organise 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 organise 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 analysing the contents of files. The system would be modular in design to allow customisation while retaining a flexible and stable structure.

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

1 INTRODUCTION 1
   1.1 Overview .................................................. 1
   1.2 Brief Introduction ....................................... 2
   1.3 Problem Definition ...................................... 2
   1.4 Feasibility Study ......................................... 3
   1.5 Application of a Software Engineering Approach ......... 3

2 LITERATURE SURVEY 6

3 SOFTWARE REQUIREMENT SPECIFICATION 11
   3.1 Introduction ................................................. 11
   3.2 System Features ............................................ 13
   3.3 External Interface Requirements ......................... 15
   3.4 Nonfunctional Requirements ............................... 15
   3.5 Other Requirements ........................................ 17
   3.6 Analysis Model ............................................. 17
   3.7 System Implementation Plan ............................... 20

4 SYSTEM DESIGN 21
   4.1 System Architecture ....................................... 21
   4.2 UML Diagrams ................................................ 22

5 TECHNICAL SPECIFICATIONS 25
   5.1 Technology Details Used in the Project .................. 25
   5.2 Reference to Technology ................................... 26

6 PROJECT ESTIMATION, SCHEDULE AND TEAM STRUCTURE 29
   6.1 Project Estimate and Schedule ............................ 29
   6.2 Team Structure .............................................. 30

7 SOFTWARE IMPLEMENTATION 32
   7.1 Introduction ................................................ 32
   7.2 Databases ................................................... 32
   7.3 Important Modules and Algorithms ....................... 33
   7.4 Business Logic and Architecture ......................... 34

8 SOFTWARE TESTING 36
   8.1 Introduction ................................................ 36
   8.2 Test Cases .................................................. 36
   8.3 Snap Shots of the Test Cases and Test Plans .......... 37

9 RESULTS 51
   9.1 File System View after Mounting ......................... 51
   9.2 Usage and Performance ..................................... 53
   9.3 Displaying Suggested Files ............................... 53
   9.4 Exporting Files ............................................ 54
10 DEPLOYMENT AND MAINTAINENCE 55
10.1 Installation and Uninstallation ........................................ 55
10.2 User Help ...................................................................... 56
10.3 Browsing Files by Type .................................................. 56

11 CONCLUSION AND FUTURE SCOPE 60
11.1 Conclusion ................................................................... 60
11.2 Future Scope .................................................................. 60
11.3 Need for KWEST Tomorrow ............................................. 61

12 APPENDIX 62
Appendix A: Mathematical Model ......................................... 62
Appendix B: Testing of Data .................................................. 66
Appendix C: Papers Published ................................................. 71
Appendix D: Papers Referred .................................................. 81
Appendix E: Contribution of Team Members ......................... 118
Appendix F: Glossary ............................................................. 119

13 BIBLIOGRAPHY 120
List of Figures

1.1 Incremental model .............................................. 3
3.1 Data flow diagram - Level 0 .................................. 17
3.2 Data flow diagram - Level 1 .................................. 18
3.3 Data flow diagram - Level 2 .................................. 19
3.4 Entity relationship diagram .................................. 20
4.1 System architecture ........................................... 21
4.2 Use case diagram .............................................. 22
4.3 Component diagram ........................................... 23
4.4 Deployment diagram .......................................... 24
7.1 Business architecture .......................................... 35
8.1 Accessing non-existent entries on file system ................. 41
8.2 Error shown while copying suggestions ......................... 41
8.3 Error shown while copying an audio file to images .......... 42
8.4 Copying an image folder representing ‘date’ .................. 42
8.5 Creating a copy of the same file in KWEST .................... 43
8.6 Renaming an audio file in KWEST ............................ 43
8.7 Renaming a folder in KWEST .................................. 44
8.8 Removing a file in one tag without affecting other tags .... 44
8.9 Listing entries recursively to detect loops .................... 45
8.10 IOZone statistics for various operations ....................... 46
8.11 SPEW performing read-after-write on KWEST ............... 46
8.12 SPEW performing read-after-write on underlying file system 47
8.13 Read operation average time for 4.1MB text file ............. 48
8.14 Write operation average time on 4.1MB text file ............ 49
9.1 KWEST file system in terminal after mounting ............... 51
9.2 KWEST file system in PCManFM after mounting ............... 52
9.3 Chaotic organisation in normal file system vs KWEST auto-organisation 52
9.4 Performance test for KWEST with normal user task loads .... 53
9.5 KWEST showing suggestions for including files in the favourites tag 53
9.6 KWEST export function - Copy file to external location .... 54
10.1 Audio files being categorised by /Album and /Artist/Album views 56
10.2 Images being categorised by creator and creationdate .... 57
10.3 PDF documents being separated by author and title .......... 57
10.4 Browsing the KWEST video folder in a terminal ............. 58
10.5 User tags displayed in KWEST .................................. 58
10.6 KWEST offering suggestions for tag favourites ............... 59
12.1 Reviewers comments ......................................... 75
12.2 Reviewers comments ......................................... 80
List of Tables

3.1 Phase 1 implementation plan ........................................... 20
3.2 Phase 2 implementation plan ........................................... 20

6.1 Work breakdown structure .................................................. 29

8.1 File system operations ..................................................... 37
8.2 Risk management ........................................................... 40
8.3 Performance tests for mounting KWEST ............................... 45
8.4 Performance tests for using KWEST ..................................... 47
8.5 GDB debugging for KWEST ............................................... 50
Chapter 1

INTRODUCTION

1.1 Overview

Information organisation and management is an essential and inevitable part of everyday computer usage. Huge amounts of data are produced on a daily basis. With data growing in size, we are faced with the problem of locating related information. File system in use today, referred to as traditional file systems impose a hierarchical structure of storage on the user. These structures are mono-hierarchical and implement directory trees to categorise and store files. In such systems, directories are the only means to access any particular file.

In such systems, directories are the only means to access particular files. The path of a file contains directories it belongs to, which refer to its context and categorisation. As an example “c:\photos\collegetrip\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 choosing the directory that best represents the context of a particular 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 organisational skills. Searching or browsing through someone else’s data is tricky as the organisation is different for every user.

Previous approaches 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[14] and Google’s Desktop Search[15]. Both function only on limited file types and do not allow manual categorisation. This led to the development of semantic file systems, containing categorisation 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 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. The goal of this project is to create a semantic solution to the afore mentioned problems and shortcomings of traditional file systems while trying to cover the limitations of other implemented projects.
1.2 Brief Introduction

Organising and retrieving information accurately and efficiently has attracted a lot of attention. While few have been successful, a number of innovative implementations have emerged. KWEST is a virtual file system capable of storing semantics with which it facilitates the finding of relevant information. Information is stored in tags, which are extracted from a file's metadata. This information may be generated implicitly by the system or supplied explicitly by the user. Thus, the validity of information is based on the user's level of organising things.

The current system can extract metadata from a limited set of known file types. However, the modular architecture allows for plugins to be added which can add functionality for other file types. This allows for the project to be extended and modified according to the functionality required. The level of awareness generated by the system is based on the frequency of access and input provided by the user. The amount of relevance is determined by the tags generated and their associated files. This affects how the system categorises and searches files. Thus the actual outcome of the system which is the searching capabilities is totally dependent on these relationships. The current implementation is based on the Linux kernel. Future implementations can be extended to other platforms and devices. Furthermore, as the system is a virtual one, it needs only slight modifications to be ported to other file systems and operating systems.

1.3 Problem Definition

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. Such a system should overcome the drawbacks of traditional file systems while leveraging the limitations of other such similar implementations.

1.4 Feasibility Study

1.4.1 Technical Feasibility

The team has a fundamental knowledge of file system concepts. The Project is being implemented using a loadable kernel module known as FUSE. In the current versions of Linux distributions, this module is included in the kernel itself. The query processing and programming will be done using SQLite. 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 without the need for capital costs.

1.4.2 Economic Feasibility

Cost of Software: We have used open-source technologies for building our system, thus there were no software costs incurred.
Cost of Hardware: No hardware was required to be purchased, thus no cost was incurred in building the system.
1.5 Application of a Software Engineering Approach

We are using the incremental model of Software Development Life Cycle (SDLC). SDLC is a process used by a systems analyst to develop an information system, assert training, and identify user (stakeholder) ownership. The SDLC aims to produce a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance. The Software Development Life Cycle framework provides a sequence of activities for system designers and developers to follow. It consists of a set of steps or phases in which each phase of the SDLC uses the results of the previous one. A Software Development Life Cycle adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation.

![Incremental model](image)

Figure 1.1: Incremental model

1.5.1 Planning

During this stage, business opportunities and problems are identified, and information technology solutions are discussed. Multiple alternative projects may be suggested and their feasibility analysed.

Tasks proposed were

1. Analysing need of user to access related data.
2. Identifying drawbacks of existing file system.
3. The development of a project goals- identifying relations between data, file system operations, metadata storage, accessing related data.
4. The collection of project requirements.
5. The development of a project schedule.
1.5.2 System Analysis

The goal of system analysis is to determine where the problem is in an attempt to fix the system. This step involves breaking down the system in different pieces to analyse the situation, analysing project goals, breaking down what needs to be created and attempting to engage users so that definite requirements can be identified.

Tasks proposed were

1. Interface for implementing file system.
2. Analyse various database alternatives based on size and speed of operation.
3. Analysis of libraries to be used for metadata operations.
4. Generation of files and tags suggestions.
5. Analysing various methods to provide user with suggestions.

1.5.3 System Design

In systems design the design functions and operations are described in detail, including screen layouts, business rules, process diagrams and other documentation. The output of this stage will describe the new system as a collection of modules or subsystems.

Tasks proposed were

1. Separating system implementation into following modules.
3. Generating Tags.
4. Importing Semantics.
5. Exporting Semantics.
6. Database Consistency.

1.5.4 Coding

In coding we implement the system design, producing running software deployed as deliverables.

Tasks proposed were

1. Implementing modules of project using finalised programming language and libraries
2. Having modularity in code to allow for reusable modules.
3. Documenting the code using standard tools.
4. Testing project using various software testing approaches.
1.5.5 Testing

Testing is the process of evaluating a system or application, to check whether the application meets all requirements of the client and to detect the errors.

Tasks proposed were

1. Testing of executable code using robust testing tools.
2. Performing Regression Testing.
4. GUI Testing.
5. Stress Testing.
6. Integrity Testing.

1.5.6 Deployment

Deployment starts after the code is appropriately tested, approved for release, and sold or otherwise distributed into a production environment. This may involve installation, customisation (such as by setting parameters to the customer’s values), testing, and possibly an extended period of evaluation.

Tasks proposed were

1. The project will be deployed as a executable which will mount the KWEST file system.
2. Utilise available GUI tools in the form of file managers.
3. Installation of utility should follow standard Linux OS procedures.
Chapter 2

LITERATURE SURVEY

Over the years, organising and retrieving information accurately and efficiently has attracted a lot of attention. While few have been successful, a number of innovative implementations [12] have emerged. The idea of using a file’s semantics as the means to categorise 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 Apple’s Spotlight[14] and Google’s Desktop Search[15]. 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 organising contents. In addition, they do not provide any provision to the user for classification of data. This limitation prevented the user from having a personalised 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[2], tagging[8], extracting metadata, etc. to categorise files by using various algorithms.

“Semantic File System”[9], 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 organisation of files.

NHFS (Non Hierarchical File System)[13] 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[16], which is a semantic file system that also attempts to organise 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 fulfil 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[17], is a peer-to-peer tagging application for organising 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.

1. Tag view
   It displays a list of tags.

2. Resource view
   It lists resources that have the currently selected tags assigned.

3. User view
   It 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.
Papers referred for the development of KWEST.


**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 organise 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 your 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 analyses 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 organises 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 (File system 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 customisable 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 organise 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.


**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 utilising 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.


**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 run time. 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 system upon 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.


**Abstract:** We are given a large database of customer transactions. Each transaction consists of items purchased by a customer in a visit. We present an efficient algorithm that generates all significant association rules between items in the database. The algorithm incorporates buffer management and novel estimation and pruning techniques. We also present results of applying this algorithm to sales data obtained from a large retailing company, which shows the effectiveness of the algorithm.

**Usefulness:** For finding frequently occurring file and tags, creating association rules based on them give and providing suggestions to user.
Many tagging systems exist that allow efficient manual classification of information. Most implementations tend to be theoretical demonstrations or complex implementations [10] existing for some very specific purpose. These suggest the possibility of using semantics [3] 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

3.1.1 Purpose

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. The purpose of this document is to present a detailed description of KWEST. It will explain the purpose and features of the system, the interfaces of the system, what the system will do, the constraint under which it must operate and how the system will react to the users actions.

3.1.2 Intended Audience and Reading Suggestion

The intended audience of this document includes both developers and reviewers of the system.

3.1.3 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 file’s 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 organising things.

3. The system is currently designed to extract metadata from a limited set of popular file types for audio(.mp3, .ogg, .spx, .mpc, .ape, .flac, .wv, .tta, .wma, .m4a, .wav, .aif[f]), video(.flv, .real, .riff(.avi), .mpeg, .qt, .asf), images(.jpeg, .gif, .png, .tiff) and PDF documents.

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.

7. As the system is an virtual entity, it does not need extensive modifications to be ported to other file systems and operating systems.

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 automations based on personal needs.

3. Developer
   Uses the API provided and develops modules that extend the system. Only advanced users utilise the semantic nature of underlying file system 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 behaviour for the system for specific file types.

**Operating Environment**

1. KWEST requires FUSE [18] minimum version 2.8.6
2. KWEST can run on any Linux installation which contains required versions of FUSE.
3. 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 having a minimum Kernel version 2.6.14.
4. KWEST is a virtual file system mounted to a folder or a loop device.

**3.1.4 Design and Implementation Constraints**

**Implementation Constraints**

1. KWEST uses FUSE to manage userspace file systems. 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 file system 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. It is the responsibility of the user that mounting and unmounting of the system be done with standard rules and precaution.
5. 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, .ogg, .spx, .mpc, .ape, .flac, .wav, .tta, .wma, .m4a, .wav, .aiff), images(.jpeg, .gif, .png, .tiff), video(.flv, .real, .riff(.avi), .mpeg, .qt, .asf) and .pdf documents. This functionality can be extended with modules or through special tools made specifically for this purpose.
2. The amount of information visible through common file system 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 should be aware of how semantics are used to categorise information.
2. Users should recognise or identify appropriate tags in relation to files.
3. It is assumed that the user is well versed in organisation information and uses KWEST as a tool rather than an assistant.
4. The user should have the required privileges and rights to run KWEST and all its operations.

Dependencies

1. KWEST uses several external libraries to extract metadata from supported popular formats like TagLib for audio, Libextractor for images, video and Poppler for PDF documents. These libraries are required at compilation time. They enable the system to handle metadata extraction.

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 organise data. Multiple tags can be assigned to a single file thus allowing its access through multiple locations without duplication of 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 files used by KWEST are not locked down or access restricted. The KWEST API provides facilities that can be used to 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 organisational 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 user’s previous organisation structure.

2. Files are executable ready

The files can be directly executed through the virtual file system without making any modification to the files like audio, video files can be played through the virtual system, images can be viewed and documents can be opened and read.

3.2.4 Exporting Semantics

1. Export file system

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 organisation 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 extendable 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. Additional extraction libraries can be included using the plugin module.

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.

3.3.3 Software Interfaces

- FUSE[18]
  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[19]
  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 file system or the database should be reasonable under normal operational circumstances.

- Capacity
  KWEST can be used by any user with sufficient permissions to initialise the filesystem. Subsequent operations like read, write, modify etc. are restricted by user permissions similar to normal file systems.

- 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 recognising and categorising 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 should prevent most, if not all, data corruption.

- Data Loss
  If any file is accessed which is mentioned in database but deleted from 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 file system until its unmounted.

- Updatability
  The system shall allow for addition or deletion of files under 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. It is compatible with various other Linux distributions like Ubuntu, Fedora, Red Hat, etc.

- Testability
  New modules designed to be added to the system, to identify file types other than audio, video and images must be tested to check if they are compatible with 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 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

![Data Flow Diagram - Level 0](image-url)

Figure 3.1: Data flow diagram - Level 0
Level 1

![Diagram](image)

Figure 3.2: Data flow diagram - Level 1
Level 2

Figure 3.3: Data flow diagram - Level 2
3.6.2 Entity Relationship diagram

![Entity Relationship Diagram](image)

Figure 3.4: Entity relationship diagram

3.7 System Implementation Plan

**Phase 1: September 2012 - November 2012**

<table>
<thead>
<tr>
<th>Task</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem identification</td>
<td>01/09/12</td>
<td>07/09/12</td>
</tr>
<tr>
<td>Information gathering</td>
<td>08/09/12</td>
<td>14/09/12</td>
</tr>
<tr>
<td>Creating problem definition</td>
<td>15/09/12</td>
<td>21/09/12</td>
</tr>
<tr>
<td>Understanding underlying technology</td>
<td>22/09/12</td>
<td>05/10/12</td>
</tr>
<tr>
<td>Analysing problem</td>
<td>06/10/12</td>
<td>12/10/12</td>
</tr>
<tr>
<td>Designing solution</td>
<td>13/10/12</td>
<td>26/10/12</td>
</tr>
<tr>
<td>Refining design</td>
<td>27/10/12</td>
<td>02/11/12</td>
</tr>
<tr>
<td>Creating design report</td>
<td>03/11/12</td>
<td>09/11/12</td>
</tr>
</tbody>
</table>

Table 3.1: Phase 1 implementation plan

**Phase 2: December 2012 - March 2013**

<table>
<thead>
<tr>
<th>Task</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building a stub implementation</td>
<td>01/12/12</td>
<td>14/12/12</td>
</tr>
<tr>
<td>Implement file system operations</td>
<td>15/12/12</td>
<td>28/12/12</td>
</tr>
<tr>
<td>Extract Metadata from files</td>
<td>28/12/12</td>
<td>11/01/13</td>
</tr>
<tr>
<td>Implement modular plugins</td>
<td>12/01/13</td>
<td>25/01/13</td>
</tr>
<tr>
<td>Apply Apriori algorithm to database</td>
<td>26/01/13</td>
<td>08/02/13</td>
</tr>
<tr>
<td>Rigorous testing</td>
<td>09/02/13</td>
<td>22/02/13</td>
</tr>
<tr>
<td>Debugging and refining</td>
<td>23/02/13</td>
<td>08/03/13</td>
</tr>
<tr>
<td>System testing</td>
<td>09/03/13</td>
<td>15/03/13</td>
</tr>
<tr>
<td>Creating implementation report</td>
<td>16/03/13</td>
<td>31/03/13</td>
</tr>
</tbody>
</table>

Table 3.2: Phase 2 implementation plan
Chapter 4

SYSTEM DESIGN

4.1 System Architecture

![System Architecture Diagram]

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

5.1 Technology Details Used in the Project

5.1.1 Development

FUSE

For developing and managing the virtual file system we have used FUSE.

*Minimum required:* 2.8
*Version used:* 2.9.0

SQLite

We have used SQLite as the data repository for this project.

*Minimum required:* 3.6
*Version Used:* 3.7.13

Language for Implementation

We use ANSI C for implementing our project modules. Specifically, we follow the GNU C99 standards while compiling our code. GNU C99 is an extension of the C99 providing some extra features. *Note:* Some features are incompatible with other standards.

GNU C99

Compiler

For compiling our project modules we use the GNU Compiler Collection (GCC).

*Minimum required:* 4.6
*Version used:* 4.7.2 on Linux Mint 14

5.1.2 Operating Environment

Platform

The project is based on operating systems utilising the linux kernel. Any implementation which provides a POSIX compatible environment is sufficient.

*Minimum required:* 2.6.14
*Version used:* 3.5.0-generic on Linux Mint 14

Operating System

We have used various operating environments while creating and testing the project. Linux distributions were Ubuntu 12.04, Ubuntu 12.10, Fedora 18, Linux Mint 13, Linux Mint 14.
5.1.3 External Libraries

TagLib

Used for extracting Audio metadata from files.

Minimum required: TagLib 1.7.1
Version used: TagLib 1.8

LibExtractor

GNU Libextractor is a library used to extract meta data from files. It supports extraction from Image, PDF and Video file types.

Minimum required: 1.0.0
Version used: 1.0.1

Poppler

Poppler is a PDF rendering library based on the xpdf-3.0 code base.

Minimum required: 0.21
Version used: 0.22

5.1.4 Debugging

GDB: GNU Project Debugger

Used to debug program crashes and memory errors.

Version used: 7.5

Valgrind

Used as a profiling tool, for memory related issues and program crashes.

Version used: 3.7.0

5.2 Reference to Technology

5.2.1 Development

FUSE

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.

http://fuse.sourceforge.net/

SQLite

SQLite is a relational database management system contained in a small C programming library. It implements a self-contained, zero-configuration, transactional SQL database engine which can be embedded in applications.

http://www.sqlite.org/
Language for Implementation

We use GNU99 C standard for implementing our project modules.
http://gcc.gnu.org/c99status.html

Compiler

GCC is a compiler system which provides front ends for various languages including C. It provides optimisation’s, debugging and other features to help program development.
http://gcc.gnu.org/

5.2.2 Operating Environment

Platform

The project is based on operating systems utilising the linux kernel. Any implementation which provides a POSIX compatible environment is sufficient.
https://www.kernel.org/

Operating System

Linux distributions were:
http://www.ubuntu.com/
http://linuxmint.com/
http://fedoraproject.org/

5.2.3 External Libraries

TagLib

TagLib is a library for reading and editing the meta-data of several popular audio formats. Currently it supports both ID3v1 and ID3v2 for MP3 files, Ogg Vorbis comments and ID3 tags and Vorbis comments in FLAC, MPC, Speex, WavPack TrueAudio, WAV, AIFF, MP4 and ASF files.
http://taglib.github.com/

LibExtractor

GNU Libextractor is a library used to extract meta data from files. The goal is to provide developers of file-sharing networks, browsers or WWW-indexing bots with a universal library to obtain simple keywords and meta data to match against queries and to show to users instead of only relying on file names.
http://www.gnu.org/software/libextractor/

Poppler

Poppler is a PDF rendering library based on the xpdf-3.0 code base.
http://poppler.freedesktop.org/
5.2.4 Debugging

GDB: GNU Project Debugger

GDB, the GNU Project debugger, allows you to see what is going on ‘inside’ another program while it executes – or what another program was doing at the moment it crashed. 
http://www.gnu.org/software/gdb/

Valgrind

Valgrind is a GPL licensed programming tool for memory debugging, memory leak detection, and profiling. Valgrind was originally designed to be a free memory debugging tool for Linux on x86, but has since evolved to become a generic framework for creating dynamic analysis tools such as checkers and profilers. 
http://valgrind.org/
Chapter 6

PROJECT ESTIMATION, SCHEDULE AND TEAM STRUCTURE

6.1 Project Estimate and Schedule

We use the Work Breakdown Structure (WBS) for project estimation. The WBS is organised around the primary products of the project (or planned outcomes) instead of the work needed to produce the products (planned actions). The Work Breakdown Structure presented here represents all the work required to complete this project.

<table>
<thead>
<tr>
<th>1</th>
<th>Project Management</th>
<th>1 Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Project Initiation</td>
<td>1 Day</td>
</tr>
<tr>
<td>1.2</td>
<td>Project Planning</td>
<td>2 Days</td>
</tr>
<tr>
<td>1.3</td>
<td>Project Execution and Control</td>
<td>3 Days</td>
</tr>
<tr>
<td>1.4</td>
<td>Project Closeout</td>
<td>1 Day</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Definition</th>
<th>2 Weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Start-up and Orientation</td>
<td>2 Days</td>
</tr>
<tr>
<td>2.2</td>
<td>Determine Project Requirements</td>
<td>3 Days</td>
</tr>
<tr>
<td>2.3</td>
<td>Create future process model</td>
<td>2 Days</td>
</tr>
<tr>
<td>2.4</td>
<td>Reconcile Project Requirements</td>
<td>2 Days</td>
</tr>
<tr>
<td>2.5</td>
<td>Functional Specification</td>
<td>5 Days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>System Design</th>
<th>3 Weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Technical Architecture</td>
<td>5 Days</td>
</tr>
<tr>
<td>3.2</td>
<td>System Standards</td>
<td>2 Days</td>
</tr>
<tr>
<td>3.3</td>
<td>Physical Environment</td>
<td>2 Days</td>
</tr>
<tr>
<td>3.4</td>
<td>Technical Specification</td>
<td>3 Days</td>
</tr>
<tr>
<td>3.5</td>
<td>Creating Prototypes</td>
<td>6 Days</td>
</tr>
<tr>
<td>3.6</td>
<td>Test Plans</td>
<td>3 Days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>System Development</th>
<th>5 Weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Develop and Test Software Modules</td>
<td>10 Days</td>
</tr>
<tr>
<td>4.2</td>
<td>User Training Materials</td>
<td>2 Days</td>
</tr>
<tr>
<td>4.3</td>
<td>Technical Documentation</td>
<td>5 Days</td>
</tr>
<tr>
<td>4.4</td>
<td>Unit, Integration and System Test Results</td>
<td>10 Days</td>
</tr>
<tr>
<td>4.5</td>
<td>Performance testing</td>
<td>4 Days</td>
</tr>
<tr>
<td>4.6</td>
<td>Acceptance testing</td>
<td>4 Days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>System Implementation</th>
<th>3 Weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Acceptance Environment</td>
<td>5 Days</td>
</tr>
<tr>
<td>5.2</td>
<td>Data Initialisation and Conversion Test Results</td>
<td>8 Days</td>
</tr>
<tr>
<td>5.3</td>
<td>Acceptance test results</td>
<td>4 Days</td>
</tr>
<tr>
<td>5.4</td>
<td>Supporting Material</td>
<td>4 Days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>Transition to Ready State</th>
<th>1 Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Train Users</td>
<td>2 Days</td>
</tr>
<tr>
<td>6.2</td>
<td>Convert Data</td>
<td>2 Days</td>
</tr>
<tr>
<td>6.3</td>
<td>Deploy System</td>
<td>3 Days</td>
</tr>
</tbody>
</table>

Table 6.1: Work breakdown structure
6.2 Team Structure

<table>
<thead>
<tr>
<th>Member</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member 1</td>
<td>Aseem Gogte</td>
</tr>
<tr>
<td>Member 2</td>
<td>Sahil Gupta</td>
</tr>
<tr>
<td>Member 3</td>
<td>Harshvardhan Pandit</td>
</tr>
<tr>
<td>Member 4</td>
<td>Rohit Sharma</td>
</tr>
</tbody>
</table>

**Member 1: Aseem Gogte**

Prior Experience:

- Well versed in C, C++
- Visual Basic
- Java
- Databases - Oracle

New Technology Learnt:

- Version Control - GIT
- FUSE
- Profiling and Debugging Tools - Valgrind, GDB

**Member 2: Sahil Gupta**

Prior Experience:

- Well versed in C, C++
- Visual Basic
- Java
- Databases - Oracle

New Technology Learnt:

- Version Control - GIT
- FUSE
- Databases - SQLite

**Member 3: Harshvardhan Pandit**

Prior Experience:

- Well versed in C, C++
- Visual Basic, .NET
- Java
- System programming
New Technology Learnt:

- Version Control - GIT
- FUSE
- Profiling and Debugging Tools - Valgrind, GDB

Member 4: Rohit Sharma
Prior Experience:

- Well versed in C, C++
- Visual Basic
- Java
- Databases - Oracle

New Technology Learnt:

- Version Control - GIT
- FUSE
- Databases - SQLite
Chapter 7

SOFTWARE IMPLEMENTATION

7.1 Introduction

The implementation of the software was done in a modular manner using the incremental approach of SDLC. ANSI C was the language used to implement the project. An SQLite database was used for creating and managing the data repository. Various external libraries were used for the extraction of the metadata. The following modules constituted our project:

Module 1: Creation of a virtual file system using FUSE
Module 2: Interfacing a Data Repository using SQLite
Module 3: Adding automated Extraction of Metadata
Module 4: Importing Semantics in to the file system
Module 5: Exporting Semantics from the file system
Module 6: Association Rule learning using Apriori Algorithm

7.2 Databases

The project was to implement a semantic file system. This required a data repository to store all the information such as file name, physical location, attributes, etc. Also, the metadata extracted from the files was also stored in the database. This way the database formed a central information location for the file system. Therefore, for the file system implementation to be stable and efficient, the system required an in-place robust database which offers integrity and speed. From the various options available, the SQLite database was selected and used.

SQLite is a relational database management system contained in a small C programming library. Unlike client-server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. SQLite stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine. Features include:

1. Zero Configuration
2. Serverless
3. Single Database File
4. Stable Cross-platform database
5. Manifest typing
7.3 Important Modules and Algorithms

7.3.1 Creation of a Virtual File System using FUSE

The first module of implementation was to create a basic file system using FUSE. Using FUSE we created a virtual file system capable of doing all the operations that a normal file system does. The module consists of the following phases:

**Phase 1**: Implement FUSE to create a basic file system structure using ANSI C as the implementation language.

**Phase 2**: Connect the file system created to the data repository created using a SQLite database. Create tags and files view by querying the database through FUSE.

**Phase 3**: Implementing common file operations with respect to tags and files such as read, write, open, copy, move etc.

**Phase 4**: Extraction of metadata from the files using external libraries and organisation of the file system based on metadata.

**Phase 5**: Displaying suggestion based on associations derived using Apriori Algorithm.

7.3.2 Interfacing a Data Repository using SQLite

The database is an important module of the file system. All the data required to browse and navigate the file system is stored in the database. FUSE interacts with the data in the database by querying for particular data based on path accessed. It is vital for the proper functioning of the system that the database always remains consistent. Logging mechanisms ensure that operations on the database always reach an endpoint. This module is used to check, correct and maintain integrity of the database by checking for redundant entries. Also, if there are new files which have not been added to KWEST, this module can help the user add them. We implement this module in the following ways:

**Phase 1**: Create database tables for a file system.

**Phase 2**: The relation tables between tags, files are stored.

**Phase 3**: Store the extracted metadata in the database.

**Phase 4**: The association rules for the data are derived using the Apriori algorithm.

7.3.3 Adding Automated Extraction of Metadata

Metadata (meta content) is defined as data providing information about one or more aspects of the data. Metadata can be stored either internally, in the same file as the data, or externally, in a separate file. Metadata that is embedded with content is called embedded metadata. The metadata of the file is extracted by using external libraries. The data repository stores the extracted metadata in a predetermined format.

**Phase 1**: Test external libraries to determine which of them can be used.

**Phase 2**: Extract metadata using external libraries.

**Phase 3**: Store the extracted metadata in the database.

**Phase 4**: Form relations between metadata and files using association rules.
7.3.4 Importing Semantics into the File System

Users already have certain organisational structures in the way they store data in file systems. This module imports semantics by converting the storage hierarchy to tag-based hierarchy. This means the directory structure present in the file system will be used to form tags and the files listed under the directory are tagged under that tag.

- **Phase 1:** Parse the folder structure on local hard disk.
- **Phase 2:** Add entry for each file and folder to the database.
- **Phase 3:** Remove or ignore hidden and system files.
- **Phase 4:** Prune the database entries on every start up.

7.3.5 Exporting Semantics from the File System

This module can export the storage hierarchy to some external location. The semantic organisation of tags is converted to actual directories and the files are then copied to these directories. This is similar to copying contents from one file system to another.

- **Phase 1:** Copy virtual locations to external location.
- **Phase 2:** Perform physical copy of files.
- **Phase 3:** Create folders and sub folders based on tags.
- **Phase 4:** Copy suggestions using data repository.

7.3.6 Association Rule Learning using Apriori Algorithm

Association rules help in organising the file system data by providing suggestions while tagging files. These suggestions can be helpful when the user has either forgotten to tag the file, or is yet about to do it. This association rule learning approach uses the Apriori algorithm.

- **Phase 1:** Run Apriori over the KWEST database.
- **Phase 2:** Perform optimisation’s and prune steps.
- **Phase 3:** Store association rules in database.
- **Phase 4:** Integrate with KWEST to show suggestions.

7.4 Business Logic and Architecture

7.4.1 Business Logic

In computing, a file system (or filesystem) is a type of data store which can be used to store, retrieve and update a set of files. The term could refer to the abstract data structures used to define files, or to the actual software or firmware components that implement the abstract ideas.

Traditionally, file systems were always developed with performance parameters in mind. However, with the data-rich systems of today, the responsibilities of a file system have increased. The onus of organisation and retrieval of data is much more on the file system than the user. The file system structure defines the capability and capacity of searches performed on it. In such a scenario, a file system that facilitates retrieval by providing features that help automate organisation is a lucrative option.
7.4.2 Expenses and Legal Ramifications

The software has been open sourced under the Apache License. As such, there are no cost requirements that can be incurred by the adaptation of KWEST. All the utilised tools, libraries, platforms and algorithms are free to use. This encourages technology adaptation for other students, developers and organisations.

7.4.3 Novelty of idea

There have been no previous implementations of adapting data mining techniques to a file system. This suggests the novelty of the idea of KWEST and the possibilities for future work.

7.4.4 Business Architecture

![Business architecture diagram]

Figure 7.1: Business architecture
Chapter 8
SOFTWARE TESTING

8.1 Introduction

8.1.1 Purpose
Software testing can be stated as the process of validating and verifying that a computer program/application/product:

- Meets the requirements that guided its design and development.
- Works as expected.
- Can be implemented with the same characteristics.
- Satisfies the needs of stakeholders.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. Traditionally most of the test effort occurs after the requirements have been defined and the coding process has been completed, but in the Agile approaches most of the test effort is on-going. As such, the methodology of the test is governed by the chosen software development methodology.

8.1.2 Scope
The testing of the system was done manually and no testing tools were used. The test plan describes the unit, functional, performance, usability, regression tests that were performed. Only codes that were pushed as commits were considered as candidates for testing.

8.1.3 Intended Audience
The testing of this system is intended for 3 types of audiences:

1. **End Users**: The users who will be using the system will review the testing as a mark of stability and performance of the system.
2. **Developers**: Will view the testing for knowing existing limitations and bugs.
3. **Reviewers**: Will use these test results as a metric to evaluate the project.

8.2 Test Cases

8.2.1 Introduction
The purpose of this Test Case document is to specify and communicate the specific conditions which need to be validated to enable an assessment of the system. Test Cases are motivated by many things but will usually include a subset of Use Cases, performance characteristics and the risks the project is concerned with. A separate test case document is prepared for each testing phase (unit, integration, integrity, etc.) identified in the test plan. The test cases should be organised into related groups that are meaningful to the project i.e. test suites.
8.2.2 File System Operations

Testing the file system for implementations of required operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>getattr</td>
<td>YES</td>
<td>checks whether the given path exists</td>
</tr>
<tr>
<td>readdir</td>
<td>YES</td>
<td>lists the contents of the given tag</td>
</tr>
<tr>
<td>access</td>
<td>YES</td>
<td>checks for access to specified tag</td>
</tr>
<tr>
<td>truncate</td>
<td>YES</td>
<td>closes file after operation</td>
</tr>
<tr>
<td>destroy</td>
<td>YES</td>
<td>called on file system unmount</td>
</tr>
<tr>
<td>open</td>
<td>YES</td>
<td>opens for file for access</td>
</tr>
<tr>
<td>release</td>
<td>YES</td>
<td>releases file after access</td>
</tr>
<tr>
<td>mknod</td>
<td>YES</td>
<td>creates new file</td>
</tr>
<tr>
<td>rename</td>
<td>YES</td>
<td>renames files and folders</td>
</tr>
<tr>
<td>unlink</td>
<td>YES</td>
<td>removes file from system</td>
</tr>
<tr>
<td>read</td>
<td>YES</td>
<td>reads data from file</td>
</tr>
<tr>
<td>write</td>
<td>YES</td>
<td>writes data to file</td>
</tr>
<tr>
<td>chmod</td>
<td>YES</td>
<td>changes permissions</td>
</tr>
<tr>
<td>chown</td>
<td>YES</td>
<td>changes owner</td>
</tr>
<tr>
<td>mkdir</td>
<td>YES</td>
<td>creates new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>YES</td>
<td>removes directory</td>
</tr>
<tr>
<td>symlink</td>
<td>INVALID</td>
<td>not required in KWEST</td>
</tr>
<tr>
<td>readlink</td>
<td>INVALID</td>
<td>not required in KWEST</td>
</tr>
<tr>
<td>link</td>
<td>INVALID</td>
<td>not required in KWEST</td>
</tr>
<tr>
<td>utimens</td>
<td>NO</td>
<td>not implemented</td>
</tr>
<tr>
<td>statfs</td>
<td>NO</td>
<td>not implemented</td>
</tr>
<tr>
<td>fsync</td>
<td>INVALID</td>
<td>not required in KWEST</td>
</tr>
<tr>
<td>setxattr</td>
<td>NO</td>
<td>not implemented</td>
</tr>
<tr>
<td>getxattr</td>
<td>NO</td>
<td>not implemented</td>
</tr>
<tr>
<td>listxattr</td>
<td>NO</td>
<td>not implemented</td>
</tr>
<tr>
<td>removexattr</td>
<td>NO</td>
<td>not implemented</td>
</tr>
</tbody>
</table>

Table 8.1: File system operations

8.3 Snap Shots of the Test Cases and Test Plans

Test Plan

8.3.1 Target Items

The following have been identified as targets for testing:

1. Code and associated areas
2. File system operations
3. Databases: SQLite3
4. Operating Systems

8.3.2 Outline of Tests

Tests performed

1. Performance tests
2. Functional tests
3. Data Integrity tests
4. Regression tests
5. Usability tests

8.3.3 Test Approach

Any bugs found should be reported with related information, which should include who discovered it, how, a description of the bug, and who fixed it and when. Also, re-testing of the code done to make sure that defect has been fixed and there no new bugs produced due to change in code.

1. **Performance Testing:**
The focus of Performance testing is checking a software programs
- **Speed:** Determines whether the system responds quickly.
- **Scalability:** Determines maximum user load the software application can handle.
- **Stability:** Determines if the application is stable under varying loads.
**Tools required:** Software timers
**Success criteria:**
   a) Manual(user) perception does not notice any “lags”.
   b) Time to perform operations is within an acceptable range.

2. **Functional Testing:**
The prime objective of Functional testing is checking the functionalities of the software system. It mainly concentrates on -
- **Mainline functions:** Testing the main functions of an application.
- **Basic Usability:** It involves basic usability testing of the system. It checks whether an user can freely navigate through the screens without any difficulties.
- **Accessibility:** Checks the accessibility of the system for the user.
- **Error Conditions:** Usage of testing techniques to check for error conditions. It checks whether suitable error messages are displayed.
**Tools required:** None(manual testing)
**Success criteria:** All of the following are successfully tested:
   a) all key use-case scenarios.
   b) all key features.

3. **Data Integrity Testing:**
Data integrity refers to the quality of the data in databases and is the measurement by which users examine data quality, reliability and usefulness. Data integrity testing verifies that converted data is accurate and functions correctly within a given application. Testing data integrity involves:
- **Database:** Verifying that correct values are saved in databases.
- **Write-back:** Correct data is written to disk.
- **Read:** Correct data is read from disk.
- **File Integrity:** Operations do not break existing files.
Kwest: A Semantically Tagged Virtual File System

Tools required: File compare tools (manual testing)
Success criteria: All of the following are successfully tested:
   a) files are same in size, byte-blocks, permissions and parameters.
   b) data is not changed, modified or removed unless intended.

4. Regression Testing:
Regression Testing is required when there is a
   • Change in requirements and code is modified according to the requirement
   • New feature is added to the software
   • Defect fixing
   • Performance issue fix
Tools required: None (manual testing)
Success criteria: All of the following are successfully tested:
   a) all previous operations are successfully executed.
   b) previously solved bugs are not re-introduced.
   c) operations do not suffer from unwanted performance hits.

5. Usability Testing:
Goal of this testing is to satisfy users and it mainly concentrates on the following parameters of a system:
Effectiveness of the system
   • Is the system is easy to learn?
   • Is the system useful and adds value to the target audience?
   • Is Content, Colour, Icons, Images used are aesthetically pleasing?
Efficiency
   • Navigation required to reach desired screen/web page should be very less.
     Scroll bars shouldnt be used frequently.
   • Uniformity in the format of screen/pages in your application/website.
   • Provision to search within your software application or website
Accuracy
   • No outdated or incorrect data like contact information/address should be present.
   • No broken links should be present.
   • User Friendliness
   • Controls used should be self-explanatory and must not require training to operate
   • Help should be provided for the users to understand the application/website
   • Alignment with above goals helps in effective usability testing
Tools required: None (manual testing)
Success criteria: All of the following are successfully tested:
   a) operations are not changed drastically from a traditional file system.
   b) user can use the file system without any special tools.
8.3.4 Entry and Exit Criteria:

1. Test Plan
   A **Test Plan Entry Criteria:** Code is complete and has been pushed to the Git repository.
   B **Test Plan Exit Criteria:** All functional requirements have been verified.
   C **Suspension and Resumption Criteria:** Testing will be suspended on critical design flaws that will change in redesign of critical components. Testing will resume when the coding is complete and code is reviewed successfully.

2. Test Cycle
   A **Test Cycle Entry Criteria:** When a module has been completed.
   B **Test Cycle Exit Criteria:** All tests specified at the start of the testing have completed successfully.

8.3.5 Risks, Dependencies, Assumptions, Constraints

<table>
<thead>
<tr>
<th>Risk</th>
<th>Mitigation Strategy</th>
<th>Contingency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSE API changes</td>
<td>Use FUSE version numbers to run a static check while compiling for required version of FUSE.</td>
<td>Change operation code to new version.</td>
</tr>
<tr>
<td>External Library is no longer maintained</td>
<td>Try to use the latest version number of library available and keep a source ready for distribution.</td>
<td>Change to alternate library.</td>
</tr>
<tr>
<td>Performance has degraded</td>
<td>Code with performance in mind, using fast algorithms and approaches.</td>
<td>Use profiling tools to detect memory issues and static code analysers for code checking.</td>
</tr>
</tbody>
</table>

Table 8.2: Risk management

8.3.6 Problem Reporting, Escalation, and Issue Resolution

Each bug will be given a priority, which will determine when it is addressed in the current iteration. The bug priority may change due to other bugs, issues or re-evaluation of the bug by a peer review.

SNAP SHOTS OF TEST CASES

8.3.7 File system operation testing

There are no good tools for exhaustively testing file system operations for reliability, freedom from bugs, scalability, etc. Often file system developers will use some scripts to run various operations such as creating and deleting files. The best known tool for testing file systems is called SPEW, which will thoroughly test a small subset of file system operations concerning the mix of read/write operations, memory-mapped operations, and file truncation operations, all inside a single file. None of the existing solutions covers all file system operations.
Testing file system for ghost accesses

The above screenshot depicts access to a non-existant file \textit{TEST.txt} and directory \textit{Audio}. Both are not present in the current folder. The bash terminal error displayed is \textit{No such file or directory}.

Copying suggestions

Suggestions are virtual entities which cannot be copied to another folder. However, the file represented by it can be opened and used as any normal file. The error shown was displayed in the \textit{Nemo} file manager.

Copying files within different mime types

Files can only be moved within their own mime types and within user tags. Putting an audio file in an image folder is not allowed. The error shown above was displayed in the \textit{Nemo} file manager.
Copying system generated folders

The folders are generated by the system depending on the meta-data of the files displayed under them. They cannot be moved anywhere other than their correct category. E.g. Audio folders can be moved within artists and albums, but not anywhere else. The error displayed above was taken from the Nemo file manager.

Copying files within KWEST

Copying a file within KWEST is an unsupported operation. This is due to the duplicity of metadata generated by two copies of the same file. In the screenshot, the user tries to create a copy using the cp command. KWEST restricts the operation and bash displays the error Cannot create regular file: input/output error.

Renaming files and folders within KWEST

Files cannot be renamed in KWEST. This is in part due to files being displayed by their metadata contents, and also due to the duplication of metadata in case of renamed files. In the screenshot above a file and a folder are being renamed using the mv command. In this case bash returns the error Operation not allowed signifying that KWEST does not allow this operation. The mv command can still be used for moving files.

Deleting files in KWEST

A user file can be removed from the current tag without it being deleted from the other tags. The screenshot shows the file Doctor Pressure.m4a being removed from the night
tag while the file is still present in other tags.

**Loops in browsing the file system**

A virtual file system having references to multiple parents can have loops where the user can traverse the same set of folders while going *deeper* into the file system hierarchy. The simple terminal test `ls -R` can show whether such loops exist. The end of operation denotes that the terminal was able to traverse all paths and return, signifying that there were no loops in the file system.
8.3.8 Performance of file system operations

Comparison of KWEST file system against underlying file system.

Test Bench:

- Operating System: Linux Mint 14 3.5.0-25-generic
- Original file system: ext4 500GB disk with partition size 150GB
- RAM: 4GB
- swap: 8GB on disk
- CPU utilisation: average 4

Contents of Music folder imported into KWEST:

- Audio: 17 files totalling 102.9MB
- Images: 81 files totalling 196MB
- PDF: 11 files totalling 17.9MB
- Video: 4 files totalling 1GB
Others: 7 files totalling 7MB

Total: 120 files of size 1.3GB

<table>
<thead>
<tr>
<th>Test</th>
<th>Time taken</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>all files</td>
<td>120sec</td>
<td>total file size imported was 1.3GB</td>
</tr>
<tr>
<td>videos</td>
<td>40sec</td>
<td>extracting metadata from videos is more expensive compared to other file types</td>
</tr>
<tr>
<td>images</td>
<td>35sec</td>
<td>images having metadata take longer than those without</td>
</tr>
<tr>
<td>audio</td>
<td>4sec</td>
<td>audio files are the fastest to parse and load</td>
</tr>
<tr>
<td>PDF</td>
<td>2sec</td>
<td>PDF files are parsed quickly as compared to other document types</td>
</tr>
<tr>
<td>forming associations</td>
<td>2sec</td>
<td>time is proportional to number of common files in user tags</td>
</tr>
</tbody>
</table>

Table 8.3: Performance tests for mounting KWEST

IOZone statistics

IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations. IOzone has been ported to many machines and runs under many operating systems. IOzone is useful for performing a broad filesystem analysis of a vendor’s computer platform. The benchmark tests file I/O performance for the following operations: Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants. Homepage: [http://www.iozone.org/](http://www.iozone.org/)

SPEW performance tests on KWEST

Spew is used to measure I/O performance of character devices, block devices, and regular files. It can also be used to generate high I/O loads to stress systems while verifying data integrity. Spew is easy to use and is flexible. No configuration files or complicated client/server configurations are needed. Spew also generates its own data patterns that are designed to make it easy to find and debug data integrity problems. Homepage: [http://spew.berlios.de](http://spew.berlios.de)
Figure 8.10: IOZone statistics for various operations

Figure 8.11: SPEW performing read-after-write on KWEST

The screenshot depicts the spew read-after-write test which writes random bytes to a file and then performs sequential reads on it. The statistics show that KWEST has a good write data rate which is comparable to regular file systems. However, read rate sees a significant drop. In spite of this, usability tests show that for a regular user, the file system offers satisfactory performance.
Test scripts

The following is a simple test script used to evaluate file system operations. The script works by calculating the difference in time before and after the execution of operations.

```bash
#store current time
let DA=('date +%s')

#perform file system operation
ls -R kwest/src/mnt

#store new time
let DB=('date +%s')

#calculate the difference
let DC=$DB-$DA

#output the time taken
echo $DC
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>ext4</th>
<th>KWEST</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>list directory</td>
<td>500ms</td>
<td>550ms</td>
<td>there is no noticeable difference</td>
</tr>
<tr>
<td>read file</td>
<td>700ms</td>
<td>850ms</td>
<td>some extra time is taken to read a file depending on the amount of data being read. In general, there is no noticeable difference.</td>
</tr>
<tr>
<td>write file</td>
<td>1200ms</td>
<td>2200ms</td>
<td>(for 5MB text file) writing takes slightly more time, but the difference is within acceptable range.</td>
</tr>
<tr>
<td>read and write</td>
<td>1010ms</td>
<td>2400ms</td>
<td>(for 5MB text file) reading and writing simultaneously does not produce any performance degradation.</td>
</tr>
</tbody>
</table>

Table 8.4: Performance tests for using KWEST
READ operation TEST

Read operation test included reading from a 4.1MB file and outputting the contents on terminal. The file was accessed on the KWEST file system.

```bash
GNU nano 2.2.6       File: ./test.sh
echo "READ_test"
echo "cat ./mnt/Files/Music/t2.txt"
let kT=0
for i in 1 2 3 4 5
do
echo "Run#" $i
#store current time
let kS=('date +%s')
#perform filesystem operation
let kE=('date +%s')
#store new time
let kO=$kE-$kS
#calculate the difference
let kT=kT+kO
#output the time taken
echo "kwest_operation_time=" $kO " sec"
done
let kT=kT/5
echo "average_time_taken=" $kT
```

Figure 8.13: Read operation average time for 4.1MB text file
WRITE operation test

Write operation test included reading from a 4.1MB file and writing the contents to another file. Both the files were accessed from the KWEST file system.

```
GNU nano 2.2.6   File: ./test.sh
echo "READ_test"
echo "cat ./mnt/Files/Music/t2.txt > ./mnt/Files/Music/test.txt"
let kT=0
for i in 1 2 3 4 5
do
echo "Run#" $i
#store current time
let kS=('date +%s')
#perform file system operation
cat ./mnt/Files/Music/t2.txt > ./mnt/Files/Music/test.txt
#store new time
let kE=('date +%s')
#calculate the difference
let kO=$kE-$kS
let kT=kT+kO
#output the time taken
echo "kwest operation time = " $kO " sec"
done
let kT=kT/5
echo "average time taken = " $kT
```

Figure 8.14: Write operation average time on 4.1MB text file
8.3.9 Profiling Code

Code can be profiled using Manual methods, or using specific tools such as Valgrind, GDB, Splint etc. For testing KWEST, we have used the following profiling tools:

**GDB**

GDB can be used to debug the file system and check for memory leaks, errors and irregular operations. The sample output given below shows a clean mount and unmount of the KWEST file system.

```
$ gdb ./kwest
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
Reading symbols from kwest/src/kwest....
(gdb) run -s -d -f mnt
Starting program: kwest/src/kwest -s -d -f mnt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
KWEST - A Semantically Tagged Virtual File System
...
[Inferior 1 (process 20863) exited normally]
(gdb) bt
No stack.
```

Using GDB, we can test the file system and record the various errors occurring under testing. The following table depicts each file system operation and the possible errors occurring on it -

<table>
<thead>
<tr>
<th>Test</th>
<th>Status</th>
<th>Possible Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>list directory</td>
<td>PASS</td>
<td>I/O error, illegal operation, transport endpoint not connection, connection abort</td>
</tr>
<tr>
<td>read file</td>
<td>PASS</td>
<td>I/O error, illegal operation, access denied, database error</td>
</tr>
<tr>
<td>write file</td>
<td>PASS</td>
<td>I/O error, illegal operation, access denied, file system busy</td>
</tr>
<tr>
<td>mknod</td>
<td>PASS</td>
<td>Operation not permitted, I/O error, database error</td>
</tr>
<tr>
<td>unlink</td>
<td>PASS</td>
<td>Device busy, Operation not permitted, I/O error, database error</td>
</tr>
<tr>
<td>mkdir</td>
<td>PASS</td>
<td>Operation not permitted, I/O error, database error</td>
</tr>
<tr>
<td>rmdir</td>
<td>PASS</td>
<td>Device busy, Operation not permitted, I/O error, database error</td>
</tr>
<tr>
<td>chmod</td>
<td>PASS</td>
<td>Access denied, I/O error, device busy, database error</td>
</tr>
<tr>
<td>associations</td>
<td>PASS</td>
<td>memory error, segmentation fault, unconditional jump, I/O error</td>
</tr>
<tr>
<td>fuse main</td>
<td>PASS</td>
<td>incompatible version</td>
</tr>
</tbody>
</table>

Table 8.5: GDB debugging for KWEST
Chapter 9

RESULTS

9.1 File System View after Mounting

The KWEST file system needs to be mounted by running a program and specifying a mount point. Once mounted, the file system can be used through any tool or application.

9.1.1 Terminal View

![Screenshot of KWEST file system in terminal after mounting]

Figure 9.1: KWEST file system in terminal after mounting

9.1.2 File Manager View

A file manager or file browser is a computer program that provides a user interface to work with file systems. Files are typically displayed in a hierarchy. This is the default and only way traditional file systems can work. For semantic file systems like KWEST, virtual entries act like folders and files. These allow the file system to specify what entries to display without adhering to any strict hierarchy.

File managers used and tested with KWEST include:

- Nautilus - The default file manager on Gnome distributions.
- Nemo - The default file manager for Cinnamon desktop.
- PCManFM - Lightweight file manager found in LXDE.
- XFE - File manager for the X Window system.
- Dolphin - The default file manager for KDE distributions.
- Thunar - Fast and extendable file manager which supports plugins.
9.1.3 Organisation by File Types

The KWEST file system extracts metadata from files while it is being imported. Depending on the extracted metadata, files are categorised and put into folders corresponding to their types. Currently, KWEST supports the following file types:

1. Audio files
2. Images
3. PDF Documents
4. Videos

The separation of files into individual folders based on their types makes it easy to find a particular file. This is because each file type has its own metadata, which is used to categorise that file. An automated and organised view of files helps avoid clutter and provides efficient searching facilities.
9.2 Usage and Performance

Although KWEST is a virtual file system, it can be used for normal daily tasks such as listening to music, reading documents, watching videos, etc. In the screenshot below, the applications each run or open a file on the KWEST file system simultaneously. No perceptible lag or performance degradation was noticed while playing music or watching the video. The file system performed sufficiently well in all four application operations.

9.3 Displaying Suggested Files

Suggestions are files which are not actually present in that tag, but can be tagged as per the system’s recommendation. There are two types of suggestions:

- **Probably Related**: The files probably belong to the tag, but the system is not sure about it.
- **Related**: The file definitely belongs to the tag, and the user can tag it.

Figure 9.4: Performance test for KWEST with normal user task loads

Figure 9.5: KWEST showing suggestions for including files in the favourites tag
9.4 Exporting Files

The entries, files, tags in KWEST are all virtual entries. They do not exist anywhere on a physical disk. In order to “export” this organisation to another location outside of KWEST, there is the Export feature. Practically there is no difference between a normal copy and export. Export copies the virtual entries as real files by physically accessing the location where they are stored, and then copying them. To the end user, there is no increase in steps or time for the copying to complete.
Chapter 10

DEPLOYMENT AND MAINTAINENCE

10.1 Installation and Uninstallaion

10.1.1 Installing

- Get the latest copy of KWEST by downloading from the project hosting site: [https://code.google.com/p/kwest/downloads](https://code.google.com/p/kwest/downloads)
- After extracting the contents, open up a terminal and type in the following commands:

```
make kwest_libs
export LD_LIBRARY_PATH=../lib:LD_LIBRARY_PATH
make
```

10.1.2 Mounting and Unmounting

- To mount a KWEST file system, the user needs to run the `mount` script which will mount the file system in the specified folder.

```
./kwest mount-point
```

- To unmount a KWEST file system, the user needs to run the `fusermount -u` command with the argument `path of KWEST mount point`. A successful unmount operation does not return any message.

```
fusermount -u mount-point
```

10.1.3 Uninstallation and Removal of Data

1. KWEST “installs” files in the users local directory.
2. To clean all the installation data including the database and log files, the user should use the `make clean` or `make ob` commands.
3. These commands are present in the Makefile, which comes with the project source.
4. In case of manual installation, there is a `KWEST` folder in the `.config` directory.
5. Incomplete removal of files may cause haphazard execution of the program or corruption of files.
10.2 User Help

KWEST is a virtual file system. This means that the folders and files represented by it are a part of its virtual organisation. Each file in KWEST represents an actual file stored somewhere on the underlying file system. The main focus of using KWEST is organisation.

10.2.1 Importing Files and Folders

- By default KWEST imports from the user’s HOME folder. It recursively scans all the sub folders and imports the folder hierarchy into KWEST. Hidden files are ignored by KWEST.

- The user can also explicitly use the `import` tool to import files and folders into KWEST. The import tool accepts the folder to import as argument and imports all files and folders within it. It can work regardless of whether the KWEST file system is mounted or not.

- While importing, the metadata of the file is used to organise each file. For e.g.: A music file contains the song’s artist, which is used to categorise that file.

10.3 Browsing Files by Type

10.3.1 Audio

This folder contains all the files recognised by the system as being of type audio. Upon accessing the folder, each Audio file is further categorised by `Album`, `Artist`, `Genre`. If a particular metadata is absent for the audio file, KWEST categorises it in the Unknown folder.

For e.g. If the audio file does not have any artist associated with it, it can be found in `UnknownArtist`.

![Figure 10.1: Audio files being categorised by /Album and /Artist/Album views](image)
10.3.2 Image

This folder contains all the image files recognised by the system. Inside, the images are organised by ImageCreator and ImageDate. The ImageCreator sub folder is based on *Creators* like software - Adobe Photoshop, or hardware - Camera Models. Each ImageCreator tag is further organised by ImageDate. The ImageDate folder contains images sorted by *Month-Year* of creation. E.g.: A picture taken on “2nd March 1992” will appear under “1992Mar”. As with Audio, files with metadata missing will appear under appropriate *Unknown* sub folders.

![Figure 10.2: Images being categorised by creator and creation date](image)

10.3.3 PDF

All PDF documents are tagged with the PDF tag. Each PDF document is organised by its *Author, Publisher, Subject* and *Title*. Files with metadata missing are appropriately tagged under *Unknown* tags.

![Figure 10.3: PDF documents being separated by author and title](image)
10.3.4 Video

Currently, the system organises video based only on Length, with the categories being Short, Medium and Long. A short video is anything with less than 1800s of playtime. Videos with playtime equal to or greater than 5400s are considered Long, and anything between them is considered Medium. The Video tag does not contain any categories as most of the videos stored by the user do not have any metadata present.

```
$ ls -l
-rw-r--r-- 1 harsh harsh 733853696 Apr 17 2000 Cloverfield.avi
-rw-r--r-- 1 harsh harsh 32511138 Jun 27 2010 contraption.avi
-rwxrwxr-x 1 harsh harsh 1785475 Oct 31 2008 everything.m4a
```

Figure 10.4: Browsing the KWEST video folder in a terminal

10.3.5 USER

The tag ‘USER’ refers to the username of the current user. The user can create and manage his own personal tags in this folder.

Figure 10.5: User tags displayed in KWEST
10.3.6 Using Suggestions to tag files

KWEST helps the user with organisation by providing suggestions for tagging files. These suggestions are provided as files prefixed with the word - “SUGGESTED”. The user may make use of that suggestion by tagging that file in the current tag. For example, in tag Zodiac, there are 3 suggestions: ARIES, GEMINI, CANCER each shown as a file with SUGGESTED prefixed in their names. To make use of the suggestion on ARIES, the user tags the file ARIES under the tag Zodiac. The file is now seen in the tag without the suggested prefix. The other suggestions are still present and may be used further or deleted.

![Figure 10.6: KWEST offering suggestions for tag favourites](image)

Figure 10.6: KWEST offering suggestions for tag favourites
CONCLUSION AND FUTURE SCOPE

11.1 Conclusion

Considering data organisation

A file system, considering that it stores data, is created with stability and performance in mind. An end-user is more concerned with how they can store and access their data efficiently. Using a KWEST file system, the user can organise their data efficiently. All files are stored and accessed by their content or context rather than just a bunch of string-names. This allows the user to think and remember the file in terms of what it represents, rather than a path-name which may not be related to the data it contains. This semantic approach is helpful to the user, as it becomes easier to manage for them to search for and manage their data.

Automation of Tasks

KWEST automatically uses the metadata embedded in a file to apply tags and categorise it. This automation helps the user by providing access to files according to their respective contexts. By doing this, the user gets all Audio files under the Audio folder, all Image files under the Image folder and so on. Further, each folder is organised by meta-type belonging to that mime type. E.g. the Audio folder is sorted by Album, Artist, Genre.

Providing Suggestions

It may happen that the user inadvertently misses out tagging some file, which may result in an incomplete organisation. The user may later search for that particular file in the tag, but will not find it. In a KWEST file system, based on the occurrences of files in various user tags, the system provides suggestions that help the user tag a file in appropriate places. This helps avoid missing out on important files, and allows a faster method of organisation as the files to be tagged are available as suggestions.

Performance and Stability

Although a KWEST file system is created and operates as a VFS, there is no perceptible lag or performance hit on the operations the user performs. Common operations like listening to music, watching a video, writing to a document can be carried out smoothly.

Also, since KWEST is based on an always stable implementation of FUSE, the file system itself is stable. Strict coding standards and rigorous tests against memory allow the file system to remain stable even under moderate usage.

11.2 Future Scope

The project, with its novel concept of Applying association rule learning in a semantic file system; is the first implementation of a semantic file system to actively help the user categorise and organise their data.
Support for more Standard File Types

Right now the import feature can extract metadata only from a fixed set of file types. More file types can be handled which increase the feature and usefulness of the file system. Every operating system or file manager has a certain knowledge of what kind of metadata each file type can contain. Using this knowledge in the KWEST file system will allow the user to be able to browse a file system completely based on its semantics.

More Relations and Associations

Currently, the system creates associations based on the common occurrences of files between various tags. This is done using the apriori algorithm. There are a lot of other interesting approaches which can be utilised. Like algorithms to create various different associations. Or changing the way apriori handles files and tags. File accesses, frequency of usage, explicit user choices can also be utilised for forming results.

Efficiency and Performance

Although the system is both efficient and performant, it can be vastly improved to provide a high-quality file system. Operations can be threaded to reduce the wait time. Simultaneous access to files can be used to provide a fluid experience. Algorithm throughput can be raised to get more accurate results. These are just a few performance and efficiency related things we can do with KWEST. The ultimate approach is to integrate this file system at the kernel level. This will allow performance and stability similar those of traditional file systems.

Collect Data from Various Locations

The KWEST file system imports data only from the user’s HOME folder. The data stored there might not be the only location a user wants to use. In today’s world, each person has a multitude of devices ranging from laptops, PCs, tablets and phones to Cloud services like Dropbox, Google Drive. Each of them have data which the KWEST file system can utilise to form associations and display using virtual suggestions. It will result in a unified view of all the user’s data categorised and organised, which is spread and available across all of their devices.

11.3 Need for KWEST Tomorrow

With data usage almost doubling with each passing year, people are bound to focus on organising it. A tool like KWEST, with its semantic roots, automation and suggestions will be immensely helpful when a large data storage has to be properly catalogued, organised and accessed. Thus, we have tried to implement a research based project with its usefulness reflected in the problems of tomorrow.
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 (\( S \subseteq T \))

1. 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 \textit{many-to-many}. That is a file can have many tags, and a tag can describe many files.

2. Association between Tags (\( T \))

Using discovered associations, we can form various relations between tags. These \textit{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 \( A \).
- \( \phi \): This relation states that there does not exist any relation between the two tags.

3. Operations

\[
g(f) = \{ t : f R t \}
\]

\( g \) 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.

\[
h(t) = \{ f : f R t \}
\]

\( h \) 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.
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 an 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:

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

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 $(f R t : t \in T_E)$.

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

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 an ordered pair $\{(f, t) \forall t \in T_E\}$. At the end of this operation $T_E \subseteq T$ will hold true.

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 $(f R d)$. When importing semantics we create the ordered pair $(f, d)$ given by the relation $(f R d, \forall d \in D_d)$. 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 $d_1 \rightarrow d_2$

7. Apriori Algorithm

The apriori algorithm[7] is a classic algorithm for learning association rules. The algorithm is designed to operate on databases containing transactions[11]. As is common in association rule mining, given a set of itemsets (for instance, sets of retail transactions, each listing individual items purchased), the algorithm attempts to find subsets which are common to at least a minimum number $C$ of the itemsets. Apriori uses a ”bottom up” approach, where frequent subsets are extended one item at a time (a step known as
candidate generation), and groups of candidates are tested against the data. The algorithm terminates when no further successful extensions are found.

The purpose of the apriori algorithm is to find associations between different sets of data. Each set of data has a number of items and is called a transaction. The output of apriori is sets of rules that tell us how often items are contained in sets of data.

**Itemset**

A collection of one or more items.

Example: A, B, C

**k-itemset**

An itemset that contains k items.

**Support count (S)**

Number of transactions containing an itemset.

Example: $S(A, B) = 2$

**Support (supp)**

The support $supp(X)$ of an itemset $X$ is defined as the proportion of transactions in the data set which contain the itemset. Suppose $minsup$ is the minimum support threshold.

Example: $supp(A, B) = 2/5$

**Frequent Itemset (L)**

An itemset satisfies minimum support if the occurrence frequency of the itemset is greater or equal to a threshold. If an itemset satisfies minimum support, then it is a frequent itemset. Thus an itemset whose support is greater or equal to $minsup$ is a frequent itemset.

**Confidence (conf)**

The confidence of a rule is defined as,

$$conf(A \rightarrow B) = supp(A \rightarrow B)/supp(A)$$  \hspace{1cm} (12.1)

Suppose $minconf$ is the minimum confidence threshold.

**Rule Generation**

Given a set of transactions $T$, the goal of association rule mining is to find all rules having

$$support \geq minsup \text{ threshold}$$  \hspace{1cm} (12.2)

$$confidence \geq minconf \text{ threshold}$$  \hspace{1cm} (12.3)

Given a frequent itemset $L$, find all non-empty subsets $f \subset L$ such that $f \rightarrow L - f$ satisfies the minimum confidence requirement.

Example: If $A, B, C$ is a frequent itemset, then the following candidate rules are formed

$$AB \rightarrow C, AC \rightarrow B, BC \rightarrow A, A \rightarrow BC, B \rightarrow AC, C \rightarrow AB$$
If $|L| = k$, then there are $(2^k - 2)$ candidate association rules (ignoring $L \to \Phi$ and $\Phi \to L$)

**Apriori principle**

The principle states that if an itemset is frequent, then all of its subsets must also be frequent. Apriori principle holds due to the following property of the support measure:

$$\forall X, Y : (X \subseteq Y) \to s(X) \geq s(Y) \quad (12.4)$$

Support of an itemset never exceeds the support of its subsets. This is known as the anti-monotone property of support.

**Algorithm**

Input
- T - Database of transactions
- I - Items
- L - Itemset
- s - support
- c - confidence

Output
- R - Association rules satisfying s and c

**Algorithm to Generate Frequent Itemsets**

Apriori(T, s)

1. $L_1 \leftarrow \{\text{Large 1-Itemset}\}$
2. $k \leftarrow 2$
3. while $L_{k-1} \neq \Phi$
   1. $C_k \leftarrow \text{Generate}(L_{k-1})$
   2. for transactions $t \in T$
      1. $C_t \leftarrow \text{Subset}(C_k, t)$
      2. for candidates $c \in C_t$
         1. $\text{count}[c] \leftarrow \text{count}[c] + 1$
      3. $L_k \leftarrow \{c | c \in C_k \land \text{count}[c] \geq s\}$
   4. $k \leftarrow k + 1$
   5. return $\cup L_k$

**Algorithm to Generate Association Rules**

GenRule

1. $R = \Phi$;
2. for each $l \in L$
   1. for each $x \subset l$ such that $x \neq \Phi$ and $x \neq l$
      1. if $(\text{supp}(l) / \text{supp}(x)) \geq c$ then
      2. $R = R \cup (x \to (l - x))$;
APPENDIX B: TESTING OF DATA

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

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 exactly \( S \).
   3. For a tag \( t \) containing files \( F = \{f_1, f_2, f_3\} \), the operation \( h(t) \) should return exactly \( F \).

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.

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 \( \{t \in T_E \mid f \in T\} \) 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

\[
\text{Efficiency} \, \varepsilon = \frac{|T_E|}{|M|}
\]

Using \( \varepsilon \) 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.

\( X_{E1} : \{\varepsilon = 0.7 \text{ for audio, } \varepsilon = 0.4 \text{ for images}\} \)

\( X_{E2} : \{\varepsilon = 0.6 \text{ for audio, } \varepsilon = 0.6 \text{ for images}\} \)

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.
4. Queries and their results

Queries are parsed into tokens of the form \((t_1, \sigma, t_2)\). We need to test whether \(\sigma\) 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:
\begin{align*}
  t_1 & : \{\text{photo1, photo2, doc1}\} \\
  t_2 & : \{\text{photo1, doc1, ppt}\} \\
  t_3 & : \{\text{photo1, doc3}\}
\end{align*}

1. \(q(t) = F_S\) where \(f\) should follow the relation \((f \in F_S \mid f \text{RT})\). For example, \(q(t_1)\) returns \(F_S = \{\text{photo1, photo2, doc1}\}\).

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 \(\sigma\) is an operator.
   a) Putting \(\sigma = \cup\) for Union in query \(Q(S) = q(t_1) \cup q(t_2)\); the result should be \(F_S = \{\text{photo1, photo2, doc1, ppt}\}\).
   b) Putting \(\sigma = \cap\) for Intersection in query \(Q(S) = q(t_1) \cap q(t_2)\); the result should be \(F_S = \{\text{photo1, doc1}\}\).
   c) Putting \(\sigma = \setminus\) for Set Difference in query \(Q(S) = q(t_1) \setminus q(t_2)\); the result should be \(F_S = \{\text{photo2}\}\), whereas the query \(Q(S) = q(t_2) \setminus q(t_1)\) will return the result as \(F_S = \{\text{ppt}\}\).
   d) Putting \(\sigma = \oplus\) for Symmetric Difference in query \(Q(S) = q(t_1) \oplus q(t_2)\); the result should be \(F_S = \{\text{photo2, ppt}\}\).

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 \(\{\text{photo1}\}\)
   b) For the query \(Q(S) = q(t_1) \cup q(t_2) \cap q(t_3)\) the result will be returned as files \(\{\text{photo1, doc1, doc3}\}\)

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-interpreted through tokenization of the query.

5. Forming Associations

Consider a database, \(D\), consisting of 9 transactions. Suppose minimum support count required is 2 (i.e. \(\text{minsup} = 2/9 = 22\%\) ). Let minimum confidence required is \(\text{minconf} = 70\%\). We have to first find out the frequent itemset using apriori algorithm. Then, association rules will be generated using minsup and minconf.
Step 1: Generating initial Candidate Itemset
In the first iteration of the algorithm, each item is a member of the set of candidate.

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{F1}</td>
<td>6</td>
</tr>
<tr>
<td>{F2}</td>
<td>7</td>
</tr>
<tr>
<td>{F3}</td>
<td>6</td>
</tr>
<tr>
<td>{F4}</td>
<td>2</td>
</tr>
<tr>
<td>{F5}</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 2: Generating 1-itemset Frequent Pattern
The set of frequent 1-itemsets, L1, consists of the candidate 1-itemsets satisfying minimum support.

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{F1}</td>
<td>6</td>
</tr>
<tr>
<td>{F2}</td>
<td>7</td>
</tr>
<tr>
<td>{F3}</td>
<td>6</td>
</tr>
<tr>
<td>{F4}</td>
<td>2</td>
</tr>
<tr>
<td>{F5}</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 3: Generating 2-itemset Frequent Pattern
To discover the set of frequent 2-itemsets, L2, the algorithm uses L1 Join L1 to generate a candidate set of 2-itemsets, C2. Next, the transactions in D are scanned and the support count for each candidate itemset in C2 is accumulated. The set of frequent 2-itemsets, L2, is then determined, consisting of those candidate 2-itemsets in C2 having minimum support.

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{F1, F2}</td>
<td>4</td>
</tr>
<tr>
<td>{F1, F3}</td>
<td>4</td>
</tr>
<tr>
<td>{F1, F5}</td>
<td>2</td>
</tr>
<tr>
<td>{F2, F3}</td>
<td>4</td>
</tr>
<tr>
<td>{F2, F4}</td>
<td>2</td>
</tr>
<tr>
<td>{F2, F5}</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 4: Generating 3-itemset Frequent Pattern
The generation of the set of candidate 3-itemsets, C3, involves use of the Apriori Property. First, we generate C3 using L2 join L2.
C3 = \{\{F1, F2, F3\}, \{F1, F2, F5\}, \{F1, F3, F5\}, \\
\{F2, F3, F4\}, \{F2, F3, F5\}, \{F2, F4, F5\}\}

Now we will apply Apriori property to determine which candidate itemsets are frequent.

The 2-item subsets of \{F1, F2, F3\} are \{F1, F2\}, \{F1, F3\} and \{F2, F3\}. Since all 2-item subsets of \{F1, F2, F3\} are members of L2, we will keep \{F1, F2, F3\} in C3.

The 2-item subsets of \{F2, F3, F5\} are \{F2, F3\}, \{F2, F5\} and \{F3, F5\}. But, \{F3, F5\} is not a member of L2 and hence it is violating Apriori Property. Thus we will remove \{F2, F3, F5\} from C3. Therefore,

C3 = \{\{F1, F2, F3\}, \{F1, F2, F5\}\}

Now, the transactions in D are scanned in order to determine L3, consisting of those candidates 3-itemsets in C3 having minimum support.

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{F1, F2, F3}</td>
<td>2</td>
</tr>
<tr>
<td>{F1, F2, F5}</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 5: Generating 4-itemset Frequent Pattern

The algorithm uses L3 Join L3 to generate a candidate set of 4-itemsets, C4. Although the join results in \{\{F1, F2, F3, F5\}\}, this itemset is removed since its subset \{\{F2, F3, F5\}\} is not frequent. Thus, \(C4 = \Phi\), and algorithm terminates, having found all of the frequent items. This completes our apriori algorithm.

Step 6: Generating Association Rules from Frequent Itemsets

For each frequent itemset ‘l’, generate all nonempty subsets of l. For every nonempty subset \(s\) of \(l\), output the rule \(s \rightarrow (l - s)\) if \(\text{supp}(l) / \text{supp}(s) \geq \text{minconf}\)

We had \(L = \{\{F1\}, \{F2\}, \{F3\}, \{F4\}, \{F5\}, \{F1, F2\}, \{F1, F3\}, \{F1, F5\}, \{F2, F3\}, \{F2, F4\}, \{F2, F5\}, \{F1, F2, F3\}, \{F1, F2, F5\}\}.

Consider \(l = \{F1, F2, F5\}\). Its all nonempty subsets are \{F1, F2\}, \{F1, F5\}, \{F2, F5\}, \{F1\}, \{F2\}, \{F5\}.

The association rules are shown below, each listed with its confidence.

1. \(R1 : F1, F2 \rightarrow F5\)
   \(\text{Confidence} = \frac{\text{supp}\{F1, F2, F5\}}{\text{supp}\{F1, F2\}} = 2/4 = 50\%\)
   \(R1\) is Rejected.

2. \(R2 : F1, F5 \rightarrow F2\)
   \(\text{Confidence} = \frac{\text{supp}\{F1, F2, F5\}}{\text{supp}\{F1, F5\}} = 2/2 = 100\%\)
   \(R2\) is Selected.

3. \(R3 : F2, F5 \rightarrow F1\)
   \(\text{Confidence} = \frac{\text{supp}\{F1, F2, F5\}}{\text{supp}\{F2, F5\}} = 2/2 = 100\%\)
   \(R3\) is Selected.

4. \(R4 : F1 \rightarrow F2, F5\)
   \(\text{Confidence} = \frac{\text{supp}\{F1, F2, F5\}}{\text{supp}\{F1\}} = 2/6 = 33\%\)
   \(R4\) is Rejected.

5. \(R5 : F2 \rightarrow F1, F5\)
   \(\text{Confidence} = \frac{\text{supp}\{F1, F2, F5\}}{\text{supp}\{F2\}} = 2/7 = 29\%\)
   \(R5\) is Rejected.
6. \( R6 : F5 \rightarrow F1,F2 \)
   
   \[ \text{Confidence} = \frac{\text{supp}\{F1,F2,F5\}}{\text{supp}\{F5\}} = \frac{2}{2} = 100\% \]
   
   R6 is Selected.

In this way, we have found the following three strong association rules.

1. \( F1,F5 \rightarrow F2 \)
2. \( F2,F5 \rightarrow F1 \)
3. \( F5 \rightarrow F1,F2 \)
### APPENDIX C: PAPERS PUBLISHED

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Paper Name</th>
</tr>
</thead>
</table>


**USING ASSOCIATION RULE LEARNING IN A SEMANTIC FILE SYSTEM**

**HARSHIVARDHAN PANDIT, ASEM GOGTE, SAHIL GUPTA, ROHIT SHARMA**

Department of Computer Engineering, VSSCET, Talewadi, Pune University, Pune
Email: harshivandan18@gmail.com, aseem2699@gmail.com, sahilag@gmail.com, rohitr.ck@gmail.com

**Abstract** — The main concern in information-rich systems is to efficiently navigate and access desired information. Traversing a file system using long pathways is cumbersome and requires the user to accurately remember where each file can be found. Semantic file systems help in finding a file under various contexts (called as tags) which act like directories. However, the user still has to traverse these paths to reach the files. It is left up to the user to view and manage an efficient system of tags. Since all data items and relations are stored in a database structure, association rule learning can be used to form useful relations between various tags and files. Using various algorithms, we can create associations (links) between data sets that can help the user traverse related data, without the burden of long pathways. Popular data mining algorithms can be readily adapted to a semantic file system’s database. Thus utilizing associations, a semantic file system can offer a more efficient and contextual way to search, store and organize data.

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

**I. INTRODUCTION**

The boom in information has created a situation where it becomes difficult to categorize and search relevant information. Compared to a file system, the web has highly active services and algorithms for data navigation. Tools such as Google, DuckDuckGo [1], and Apple’s Siri etc. allow the user to search using keywords and show relevant information by utilizing data mining concepts. Developers creating web services think up of innovative ways to provide easy access to data. However, file system developers are still mostly focusing on stability and performance. While this becomes a necessity in a company server, the home user is more concerned with efficient navigation of data. Semantic file systems address this concern by providing access based on context. However, there is still scope for a richer and rewarding experience of navigating data in a file system.

This paper introduces the concept of using association rule learning in a semantic file system. Association rule learning is a popular and well-researched method for discovering interesting relations between variables in databases. By utilizing this, it becomes easy to understand the relativity between different data sets. Since most semantic file systems utilize a database to store and manage meta-information, algorithms such as 'Apriori' can be easily adapted for such applications.

**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 [2] 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 [3] and Apple's Spotlight [4]. 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 than merely relying on a file's attributes. Most implementations use common methodologies like content recognition [5], tagging [6], extracting metadata, etc. to categorize files by using various algorithms. Each of these has the drawback that although they make organizing data easier, the task of searching relevant data is not tackled. Although Semantic file systems allow the user to browse data by context, there is simply no provision to recognize the relativity between various contexts.

Using data mining in the semantic-aware file system, one can easily come up with relations that can help the user get to related data quickly and efficiently.

**III. SEMANTIC FILE SYSTEM**

A semantic file system [7] is a virtual file system that uses a database to store and manage file metadata. It is capable of performing its own interpretation of common file system functionality. The database is utilized to store tags which are virtual...
directory entries used to categorize files. Files can be tagged with any number of contexts as long as no file system rule is violated. 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.

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.

IV. APRIORI ALGORITHM

In data mining, association rule learning is a popular and well researched method for discovering interesting relations between variables in large databases. It is intended to identify strong rules discovered in databases using different measures of interestingness [8]. Based on the concept of strong rules, Rakesh Agrawal [9] introduced association rules for discovering regularities between products in large-scale transaction data recorded by point-of-sale (POS) systems in supermarkets. For example, the rule found in the sales data of a supermarket would indicate that if a customer buys onions and potatoes together, he or she is likely to also buy hamburger meat. Such information can be used as the basis for decisions about marketing activities such as, e.g., promotional pricing or product placements. In addition to the above example from market basket analysis association rules are employed today in many application areas including Web usage mining, intrusion detection and bioinformatics [10]. As opposed to sequence mining, association rule learning typically does not consider the order of items within a transaction or across transactions.

Apriori [11] is a classic algorithm for frequent item set mining and association rule learning over transactional databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets as long as those item sets appear sufficiently often in the database. The frequent item sets determined by apriori can be used to determine association rules which highlight general trends in the database. This has applications in domains such as market basket analysis.

Apriori is designed to operate on databases containing transactions (for example, collections of items bought by customers, or details of a website frequented).

where the algorithm attempts to find subsets which are common to at least a minimum number of the item sets.

Adapting the same for a semantic database, we can utilize apriori to show us files which are common to various tags (or directories). Having a cut-off percentage, the apriori results can determine whether two tags are related by looking at the files common to them. This way, the user can be shown related information through links to tags obtained from the apriori algorithm.

This example demonstrates a music player like capability where the user is suggested similar music artists based on his organization of files. This is similar to a music player application or on-line service which uses data collected from hundreds of users to show similar tags. The in-built metadata in a music file is also utilized to show similar artists by genre.

The user has various audio files tagged according to their metadata which results in the following setup.

The user then creates new tags to form playlists based on these artists. The user manually (or through some process) adds artists under the new tags created to form the following playlists.

Running an instance of the apriori algorithm on these item sets, we can form the following association rules –

1. Common occurrence

Whenever the tag of files tagged as 'Coldplay' occurs, the tag 'Green Day' is also in the same item set. Therefore, Coldplay and Green Day can be safely assume to be related. Thus, whenever the user wants to create a new playlist, and selects one of them, the system can suggest the other to be included as well.
2. Deducing Related tags

The tags ‘Evening’ and ‘Rock’ have some common artists. So are they related to each other? It may be possible, since Rock and Evening both can play some common files. Therefore while browsing, a link to this related information can be shown. For example use the tag ‘Related’ to show related tags.

When the user browses the tags Evening and Rock under Audio, related tags (Evening – Rock) can be shown as linked. This allows the user to play all related files from one location.

V. EXTENDING MODEL FOR ALL FILE-TYPES

This example demonstrated how association rule learning can be useful and helpful to the user willing to play related files. Similarly it can be easily demonstrated that the model can be extended to work with almost any file-type as long as it has been tagged properly. For the a priori algorithm, the file-type is of no special significance. As long as the semantic database keeps meta-information about the file, the a priori can form association rules by scanning the database.

The association rules thus obtained can be utilized to show related files and directories to the user while navigating the semantic file system. Extending the example to include all files, the a priori algorithm can be utilized to include the following: 1. Give suggestions when the user is tagging a file. For example a user tagging a certain document as ‘Project’ can be suggested to also tag it under ‘Confidential’ as most files under ‘Project’ are tagged as ‘Confidential’.

2. Show related files and tags while browsing. Consider a user is browsing through all pictures tagged under ‘Monuments’. Suggested pictures can include cities where the pictures were taken or documents tagged describing those Monuments.

CONCLUSION AND FUTURE WORK

Combined with the semantic file system’s ability to store data by context, associations rules help improve the user experience by providing a richer experience in browsing data. This is achieved by providing links to directories which are determined to be related using algorithms such as the a priori. Users can navigate to related directories using these links. This allows the users to browse through related data without the burden of long or incomprehensible pathnames.

REFERENCES

Figure 12.1: Reviewers comments
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
rohit.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 file 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 a 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.

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⊂T)

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

\[ R = \{ (f, t) | f \text{ has tag } t; f \in F, t \in T \} \]

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 (\( t \subseteq S \)) 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 (\( f \subseteq F \)) 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 \( R = \{ (f \in F, t \in T) | f R t \} \).

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

\[ F = \{ t_1, t_2, t_3 \}, T = \{ t_1, t_2, t_3 \}, R = \{ (f_1, t_1), (f_2, t_2), (f_3, t_3) \} \]

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

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

D. Queries

A query operation on a single tag is expressed as:

\[ q(t) = F_3 \text{ where } \{ f \in F_3 | h(t) = f \} \]

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 \( \sigma \). 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 \( \sigma \) can be any one of Union \( \cup \), Intersection \( \cap \), Symmetric difference \( \Delta \) 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


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

![Figure 12.2: Reviewers comments](image-url)

Figure 12.2: Reviewers comments
## APPENDIX D: PAPERS REFERRED

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Paper Name</th>
</tr>
</thead>
</table>
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 your 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.

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, audio, 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
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 a 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 on the K Desktop Environment (KDE).

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/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/OF0XA.txt`.
4. A hard link to `store/OF0XA.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/OF0XA.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/OF0XA.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 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 file system management.
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.](image)

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.

**Figure 2. KFS FUSE.**
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 processes 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 as 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.

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 being 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 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

Semantic File Systems
David K. Gifford, Pierre Jouvelot¹, Mark A. Sheldon, James W. O’Toole, Jr.

Programming Systems Research Group
MIT Laboratory for Computer Science

Abstract
A semantic file system is an information storage system that provides flexible associative access to the system’s contents by automatically extracting attributes from files with file type specific transducers. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Compatibility with existing file system protocols is provided by introducing the concept of a virtual directory. Virtual directory names are interpreted as queries, and thus provide flexible associative access to files and directories in a manner compatible with existing software. Rapid attribute-based access to file system contents is implemented by automatic extraction and indexing of key properties of file system objects. The automatic indexing of files and directories is called “semantic” because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Experimental results from a semantic file system implementation support the thesis that semantic file systems present a more effective storage abstraction than do traditional tree-structured file systems for information sharing and command level programming.

1 Introduction
We would like to develop an approach for information storage that both permits users to share information more effectively, and provides reductions in programming effort and program complexity. To be effective this new approach must be used, and thus an approach that provides a transition path from existing file systems is desirable.

In this paper we explore the thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. A semantic file system is an information storage system that provides flexible associative access to the system’s contents by automatically extracting attributes from files with file type specific transducers. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Automatic indexing is performed when files or directories are created or updated.

The automatic indexing of files and directories is called “semantic” because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Through the use of specialized transducers, a semantic file system “understands” the documents, programs, object code, mail, images, name service databases, bibliographies, and other files contained by the system. For example, the transducer for a C program could extract the names of the procedures that the program exports or imports, procedure types, and the files included by the program. A semantic file system can be extended easily by users through the addition of specialized transducers.

Associative access is designed to make it easier for users to share information by helping them discover and locate programs, documents, and other relevant objects. For example, files can be located based upon transducer generated attributes such as author, exported or imported procedures, words contained, type, and title.

A semantic file system provides both a user interface and an application programming interface to its associative access facilities. User interfaces based upon browsers [In90, Ver90] have proven to be effective for query based access to information, and we expect browsers to be offered by most semantic file system implementations. Application programming interfaces that permit remote access include specialized protocols for information retrieval [NIS91], and remote procedure call based interfaces [GC87].

It is also possible to export the facilities of a semantic file system without introducing any new interfaces. This can be accomplished by extending the naming semantics of files and directories to support associative access. A benefit of this approach is that all existing applications, including user interfaces, immediately inherit the benefits of associative access.

A semantic file system integrates associative access into a tree structured file system through the concept of a

¹ Also with CRI. École des Mines de Paris, France.
virtual directory. Virtual directory names are interpreted as queries and thus provide flexible associative access to files and directories in a manner compatible with existing software.

For example, in the following session with a semantic file system we first locate within a library all of the files that export the procedure lookuup fault, and then further restrict this set of files to those that have the extension .c:

```bash
% cd /sfs/exports:/lookup_fault
% Is -F
% cd /sfs/exports:/lookup_fault
% Is -F
% virtdir_query.c0
% cd ext:/c
% Is -F
% virtdir_query.c0
%```

Semantic file systems can provide associative access to a group of file servers in a distributed system. This distributed search capability provides a simplified mechanism for locating information in large nationwide file systems.

Semantic file systems should be of use to both individuals and groups. Individuals can use the query facility of a semantic file system to locate files and to provide alternative views of data. Groups of users should find semantic file systems an effective way to learn about shared files and to keep themselves up to date about the status of group projects. As workgroups increasingly use file servers as shared library resources we expect that semantic file system technology will become even more useful.

Because semantic file systems are compatible with existing tree structured file systems, implementations of semantic file systems can be fully compatible with existing network file system protocols such as NFS [SGK+85, Sun88] and AFS [Kaz88]. NFS compatibility permits existing client machines to use the indexing and associative access features of a semantic file system without modification. Files stored in a semantic file system via NFS will be automatically indexed, and query result sets will appear as virtual directories in the NFS name space. This approach directly addresses the "dusty data" problem of existing UNIX file systems by allowing existing UNIX file servers to be converted transparently to semantic file systems.

We have built a prototype semantic file system and run a series of experiments to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. We tried to locate various documents and programs in the file system using unmodified NFS clients. The results of these experiments suggest that semantic file systems can be used to find information more quickly than is possible using ordinary file systems, and add expressive power to command level programming languages.

In the remainder of the paper we discuss previous research (Section 2), introduce the interface and a semantics for a semantic file system (Section 3), review the design and implementation of a semantic file system (Section 4), present our experimental results (Section 5) and conclude with observations on other applications of virtual directories (Section 6).

2 Previous Work

Associative access to on-line information was pioneered in early bibliographic retrieval systems to locate information in large databases [Sal83]. The utility of associative access motivated its subsequent application to file and document management. The previous research we build upon includes work on personal computer indexing systems, information retrieval systems, distributed file systems, new naming models for file systems, and wide-area naming systems:

- Personal computer indexing systems such as On Location [Tec90], Magellan [Cor], and the Digital Librarian [NC89b, NC89a] provide window-based file system browsers that permit word-based associative access to file system contents. Magellan and the Digital Librarian permit searches based upon boolean combinations of words, while On Location is limited to conjunctions of words. All three systems rank matching files using a relevance score. These systems all create indexes to reduce search time. On Location automatically indexes files in the background, while Magellan and the Digital Librarian require users to explicitly create indexes. Both On Location and the Digital Librarian permit users to add appropriate keyword generation programs [Tec90, NC89b] to index new types of files. However, Magellan, On Location, and the Digital Librarian are limited to a list of words for file description.

- Information retrieval systems such as Basis [Inf90], Verity [Ver90], and Boss DMS [Log91] extend the semantics of personal computer indexing systems by adding field specific queries. Fields that can be queried include document category, author, type, title, identifier, status, date, and text contents. Many of these document relationships and attributes can be stored in relational database systems that provide a general query language and support application program access. The WAIS system permits information at remote sites to be queried, but relies upon the user to choose an appropriate remote host from a directory of services [KM91, Ste91]. Distributed information retrieval systems [GCS87, DAN091] perform query routing based upon database content labels to ensure that all relevant hosts are contacted in response to a query.

- Distributed file systems [Sun89, Kaz88] provide remote access to files with tree structured names. These systems have enabled file sharing among groups of people and over wide geographic areas. Existing UNIX tools such as grep and find [Gro86] are often used to perform associative searches in distributed file systems.
New naming models for file systems include the Portable Common Tool Environment (PCTE) [GMT86], the Property List DIRectory system (PLDIR) [Mog86], Virtual Systems [Neu90] and Sun’s Network Software Environment (NSE) [SC88]. PCTE provides an entity-relationship database that models the attributes of objects including files. PCTE has been implemented as a compatible extension to UNIX. However, PCTE users must use specialized tools to query the PCTE database, and thus do not receive the benefits of associative access via a file system interface. The Property List DIRectory system implements a file system model designed around file properties and offers a Unix front-end user interface. Similarly, Virtual Systems permit users to hand-craft customized views of services, files, and directories. However, neither system provides automatic attribute extraction (although [Mog86] alludes to it as a possible extension) or attribute-based access to their contents. NSE is a network transparent software development tool that allows different views of a file system hierarchy called environments to be defined. Unlike virtual directories, these views must be explicitly created before being accessed.

Wide-area naming systems such as X.500 [CCI88], Profile [Pet88], and the Networked Resource Discovery Project [Sch89] provide attribute-based access to a wide variety of objects, but they are not integrated into a file system nor do they provide automatic attribute-based access to the contents of a file system.

Key advances offered by the present work include:

- Virtual directories integrate associative access into existing tree structured file systems in a manner that is compatible with existing applications.
- Virtual directories permit unmodified remote hosts to access the facilities of a semantic file system with existing network file system protocols.
- Transducers can be programmed by users to perform arbitrary interpretation of file and directory contents in order to produce a desired set of field-value pairs for later retrieval. The use of fields allows transducers to describe many aspects of a file, and thus permits subsequent sophisticated associative access to computed properties. In addition, transducers can identify entities within files as independent objects for retrieval. For example, individual mail messages within a mail file can be treated as independent entities.

Previous research supports our view that overloading file system semantics can improve system uniformity and utility when compared with the alternative of creating a new interface that is incompatible with existing applications. Examples of this approach include:...
A transducer is a filter that takes as input the contents of a file, and outputs the file’s entities and their corresponding attributes. A simple transducer could treat an input file as a single entity, and use the file’s unique words as attributes. A complex transducer might perform type reconstruction on an input file, identify each procedure as an independent entity and use attributes to record their reconstructed types. Figure 1 shows examples of an object file transducer, a mail file transducer, and a TeX file transducer.

The semantics of a semantic file system can be readily constructed types. Figure 1 shows examples of an object file transducer, a mail file transducer, and a TeX file transducer. The semantics of a semantic file system can be readily extended because users can write new transducers. Transducers are free to use new field names to describe special attributes. For example, a CAD file transducer could introduce a drawing: field to describe a drawing identifier.

The associative access interface to a semantic file system is based upon queries that describe desired attributes of entities. A query is a description of desired attributes that permits a high degree of selectivity in locating entities of interest. The result of a query is a set of files and/or directories that contain the entities described. Queries are boolean combinations of attributes, where each attribute describes the desired value of a field. A query request can be extended to ask for all of the values of a given field in a query result set. The values of a field can be useful when narrowing a query to eliminate entities that are not of interest.

A semantic file system is query consistent when it guarantees query results that correspond to its current contents. If updates cease to the contents of a semantic file system it will eventually be query consistent. This property is known as convergent consistency. The rate at which a given implementation converges is administratively determined by balancing the user benefits of fast convergence when compared with the higher processing cost of indexing rapidly changing entities multiple times. It is of course possible to guarantee that a semantic file system is always query consistent with appropriate use of atomic actions.

In the remainder of this section we will explore how conjunctive queries can be mapped into tree-structured path names. As we mentioned earlier, this is only one of the possible interfaces to the query capabilities of a semantic file system. It is also possible to map disjunctive and negation into tree-structured names, but they have not been implemented in our prototype and we will not discuss them.

Queries are performed in a semantic file system through use of virtual directories to describe a desired view of file system contents. A virtual directory is computed on demand by a semantic file system. From the point of view of a client program, a virtual directory is indistinguishable from an ordinary directory. However, unlike ordinary directories, virtual directories do not have to be explicitly created to be accessed.

The query facilities of a semantic file system appear as virtual directories at each level of the directory tree. A field virtual directory is named by a field, and has one entry for each possible value of its corresponding field. Thus in /sfs, the virtual directory /sfs/owner: corresponds to the owner: field. The field virtual directory /sfs/owner: would have one entry for each owner that has written a file in /sfs. For example:

```
% ls -F /sfs/owner:
jones/ root/ smith/
```

The entries in a field virtual directory are value virtual directories. A value virtual directory has one entry for each entity described by a field-value pair. Thus the value virtual directory /sfs/owner:smith contains entries for files in /sfs that are owned by Smith. Each entry is a symbolic link to the file. For example:

```
% ls -F /sfs/owner:smith
bio.txt@ paper.tex@ prop.tex@
```

When an entity is smaller than an entire file, a view of the file can be presented by extending file naming semantics to include view specifications. To permit the conjunction of attributes in a query, value virtual directories contain field virtual directories. For example:

```
% ls -F /sfs/owner:smith/text:/resume
bio.txt@
```

A pleasant property of virtual directories is their synergistic interaction with existing file system facilities. For example, when a symbolic link names a virtual directory the link describes a computed view of a file system. It is also possible to use file save programs, such as tar, on virtual directories to save a computed subset of a file system. It would be possible also to generalize virtual directories to present views of file systems with respect to a certain time in the past.

A semantic file system can be overlaid on top of an ordinary file system, allowing all file system operations to go through the SFS server. The overlaid approach has the advantage that it provides the power of a semantic file system to a user at all times without the need to refer to a distinguished directory for query processing. It also allows the server to do indexing in response to file system mutation...
operations. Alternatively, a semantic file system may create virtual directories that contain links to the files in the underlying file system. This means that subsequent client operations bypass the semantic file system server.

When an overlaid approach is used field virtual directories must be invisible to preserve the proper operation of tree traversal applications. A directory is invisible when it is not returned by directory enumeration requests, but can be accessed via explicit lookup. If field virtual directories were visible, the set of trees under /sfs in our above example would be infinite. Unfortunately making directories invisible causes the UNIX command pwd to fail when the current path includes an invisible directory. It is possible to fix this through inclusion of unusual . . . entries in invisible directories.

The distinguished field: virtual directory makes field virtual directories visible. This permits users to enumerate possible search fields. The field: directory is itself invisible. For example:

```
% ls -F /sfs/field:
author:/ exports:/ owner:/ text:/
category:/ ext:/ priority:/ title:
dir:/ name:
% ls -F
/sfs/field:/text:/semantic/owner:/jones
mail.txt@ paper.tex@ prop.tex@
```

The syntax of semantic file system path names is:

```
<sfspath> ::= /<pn> | <pn>
<pn> ::= <name> | <attribute>
        <field-name>/<value>
<attribute> ::= field: | <field-name><value>
<field-name>::= <string>
<value>::= <string>
```

The semantics of semantic file system path names is:

- The universe of entities is defined by the path name prefix before the first virtual directory name.
- The contents of a field virtual directory is a set of value virtual directories, one for each value that the field describes in the universe.
- The contents of a value virtual directory is a set of entries, one for each entity in the universe that has the attribute described by the name of the value virtual directory and its parent field virtual directory. The contents of a value virtual directory defines the universe of entities for its subdirectories. In the absence of name conflicts, the name of an entry in a value virtual directory is its original entry name. Entry name conflicts are resolved by assigning nonce names to entries.
- The contents of a field: virtual directory is the set of fields in use.

4 Semantic File System Implementation

We have built a semantic file system that implements the NFS [SGK+85, Sun89] protocol as its external interface. To use the search facilities of our semantic file system, an Internet client can simply mount our file system at a desired point and begin using virtual directory names. Our NFS server computes the contents of virtual directories as necessary in response to NFS lookup and readdir requests.

A block diagram of our implementation is shown in Figure 2. The dashed lines in the figure describe process boundaries. The major processes are:

- The client process is responsible for generating file system requests using normal NFS style path names.
- The file server process is responsible for creating virtual directories in response to path name based queries. The SFS Server module implements a user level NFS server and is responsible for implementing the NFS interface to the system. The SFS Server uses directory faults to request computation of needed entries by the Virtual Directory module. A faulting mechanism is used because the SFS Server caches virtual directory results, and will only fault when needed information is requested the first time or is no longer cached. The Virtual Directory module in turn calls the Query Processing module to actually compute the contents of a virtual directory.
The file server process records file system modification events in a write-behind log. The modification log eliminates duplicate modification events.

- The indexing process is responsible for keeping the index of file system contents up-to-date. The Index Master module examines the modification log generated by the file server process every two minutes. The indexing process responds to a file system modification event by choosing an appropriate transducer for the modified object. An appropriate transducer is selected by determination of the type of the object (e.g. C source file, object file, directory). If no special transducer is found a default transducer is used. The output of the transducer is fed to the Indexer module that inserts the computed attributes into the index. Indexing and retrieval are based upon Peter Weinberger’s BTree package [Wei] and an adapted version of the refer [Les] software to maintain the mappings between attributes and objects.

- The mount daemon is contacted to determine the root file handle of the underlying UNIX file system. The file server process exports its NFS service using the same root file handle on a distinct port number.

- The kernel implements a standard file system that is used to store the shared index. The file server process could be integrated into the kernel by a VFS based implementation [Kle86] of an semantic file system. We chose to implement our prototype using a user level NFS server to simplify development.

Instead of computing all of the virtual directories that are present in a path name, our implementation only computes a virtual directory if it is enumerated by a client readdir request or a lookup is performed on one of its entries. This optimization allows the SFS Server to postpone query processing in the hope that further attribute specifications will reduce the amount of work necessary for computation of the result set. This optimization is implemented as follows:

- The SFS Server responds to a lookup request on a virtual directory with a lookup not found fault to the Virtual Directory module. The Virtual Directory module checks to make sure that the virtual directory name is syntactically well formed according to the grammar in Section 3. If the name is well formed, the directory fault is immediately satisfied by calling the create_dir procedure in the SFS Server. This procedure creates a placeholder directory that is used to satisfy the client’s original lookup request.

- The SFS Server responds to a readdir request on a virtual directory or a lookup on one of its entries with a fill directory fault to the Virtual Directory module. The Virtual Directory module collects all of the attribute specifications in the virtual directory path name and passes them to the Query Processing module. The Query Processing module uses simple heuristics to reorder the processing of attributes to optimize query performance. The matching entries are then materialized in the placeholder directory by the Virtual Directory module that calls the create_Link procedure in the SFS Server for each matching file or directory.

The transducers that are presently supported by our semantic file system implementation include:

- A transducer that describes object files with exports: and imports: attributes for procedures and global variables.
- A transducer that describes C, Pascal, and Scheme source files with exports: and imports: attributes for procedures.
- A transducer that describes mail files with from: to: subject: and text: attributes.
- A transducer that describes text files with text: attributes. The text file transducer is the default transducer for ASCII files.

In addition to the specialized attributes listed above, all files and directories are further described by owner, group, dir, name, and ext: attributes.

At present, we only index publicly readable files. We are investigating indexing protected files as well, and limiting query results to entities that can be read by the requester. We are in the process of making a number of improvements to our prototype implementation. These enhancements include 1) full support for multi-host queries using query routing, 2) an enhanced query language, 3) better support for file deletion and renaming, and 4) integration of views for entities smaller than files. Our present implementation deals with deletions by keeping a table of deleted entities and removing them from the results of query processing. Entities are permanently removed from the database when a full reindexing of the system is performed. We are investigating performing file and directory renames without reindexing the underlying files.

5 Results

We ran a series of experiments using our semantic file system implementation to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. All of the experimental data we report are from our research group’s file server using a semantic file system. The server is a Microvax-3 running UNIX version 4.3bsd. The server indexes all of its publicly readable files and directories.
To compact the indexes our prototype system reconstructs a full index of the file system contents every week. On 23 July 1991, full indexing of our user file system processed 68 MBytes in 7,771 files (Table 1).\(^2\) Indexing the resulting 1 million attributes took 1 hour and 36 minutes (Table 2). This works out to an indexing rate of 712 KBytes/minute.

File system mutation operations trigger incremental indexing. In update tests simulating typical user editing and compiling, incremental indexing is normally completed in less than 5 minutes. In these tests, only 2 megabytes of modified file data were reindexed. Incremental indexing is slower than full indexing in the prototype system because the incremental indexer does not make good use of real memory for caching. The full indexer uses 10 megabytes of real memory for caching; the incremental indexer uses less than 1 megabyte.

The indexing operations of our prototype are I/O bound. The CPU is 60% idle during indexing. Our measurements show that transducers generate approximately 30 disk transfers per second, thereby saturating the disk. Indexing the resulting attributes also saturates the disk. Although the transducers and the indexer use different disk drives, the transducer-indexer pipeline does not allow I/O operations to proceed in parallel on the two disks. Thus, we feel that we could double the throughput by improving the pipeline’s structure.

We expect our indexing strategy to scale to larger file systems because indexing is limited by the update rate to a file system rather than its total storage capacity. Incremental processing of updates will require additional read bandwidth approximately equal to the write traffic that actually occurs. Past studies of Unix file system activity [OCH'85] indicate that update rates are low, and that most new data is deleted or overwritten quickly; thus, delaying slightly the processing of updates might reduce the additional bandwidth required by indexing.

To determine the increased latency of overlaid NFS operations introduced by interposing our SFS server between the client and the native file system, we used the

\(^2\) The 162 MBytes in publicly readable files that were not processed were in files for which transducers have not yet been written: executable files, PostScript files, DVI files, tar files, image data, etc.

\(^3\) in parallel with \texttt{Transduce}
nhfsstone benchmark [Leg89] at low loads. The delays observed from an unmodified client machine were smaller than the variation in latencies of the native NFS operations. Preliminary measurements show that lookup operations are delayed by 2 ms on average, and operations that generate update notifications incur a larger delay.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for information sharing:

- The typical response time for the first ls command on a virtual directory is approximately 2 seconds. This response time reflects a substantial time savings over linear search through our entire file system with existing tools. In addition, subsequent ls commands respond immediately with cached results.

We ran a series of experiments to test how the number of attributes in a virtual directory name altered the observed performance of the Is command on a virtual directory. Attributes were added one at a time to arrive at the final path name:

```
/sfs/text:/virtual/ text:/semantic/ ext:/tex/ owner:/gifford
```

The two properties of a query that affect its response time are the number of attributes in the query and the number of objects in the result set. The effect of an increase in either of these factors is additional disk accesses. Figure 3 illustrates the interplay of these factors. Each point on the response time graph is the average of three experiments. In a separate experiment we measured an average response time of 5.4 seconds when the result set grew to 545 entities.

- We began to use the semantic file system as soon as it was operable to help coordinate the production of this paper and for a variety of other everyday tasks. We have found the virtual directory interface to be easy to use. (We were immediately able to use the GNU Emacs directory editor DIRED [Sta87] to submit queries and browse the results. No code modification was required.) At least two users in our group reexamined their file protections in view of the ease with which other users could locate interesting files in the system.

- Users outside our research group have successfully used the query interface to locate information, including newspaper articles, in our file system.

- Users outside our research group have failed to find files for which no transducer had yet been installed. We are developing new transducers in response to these failed queries.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for command level programming:

- The UNIX shell pathname expansion facilities integrate well with virtual directories. For example, it is possible to query the file system for all dvi files owned by a particular user, and to print those whose names begin with a certain sequence of characters.

- Symbolic links have proven to be an effective way to describe file system views. The result of using such a symbolic link as a directory is a dynamically computed set of files.

6 Conclusions

We have described how a semantic file system can provide associative attribute-based access to the contents of an information storage system with the help of file type specific transducers. We have also discussed how this access can be integrated into the file system itself with virtual directories. Virtual directories are directories that are computed upon demand.

The results to date are consistent with our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. We plan to conduct further experiments to explore this thesis in further detail. We plan also to examine how virtual directories can directly benefit application programmers.

Our experimental system has tested one semantics for virtual directories, but there are many other possibilities. For example:

- The virtual directory syntax can be extended to support a richer query language. Disjunctive queries would permit users to use “or” in their queries, and would also offer the ability to search on multiple network semantic file systems concurrently.
• Users could assign attributes to file system entities in addition to the attributes that are automatically assigned by transducers.
• Transducers could be created for audio and video files. In principle this would permit access by time, frame number, or content [Nee91].
• The data model underlying a semantic file system could be enhanced. For example, an entity-relationship model [Cat83] would provide more expressive power than simple attribute based retrieval.
• The entities indexed by a semantic file system could include a wide variety of object types, including I/O devices and file servers. Wide-area naming systems such as X.500 [CCI88] could be presented in terms of virtual directories.
• A confederation of semantic file systems, possibly numbering in the thousands, can be organized into an *semantic library system*. A semantic library system exports the same interface as an individual semantic file system, and thus a semantic library system permits associative access to the contents of its constituent servers with existing file system protocols as well as with protocols that are designed specifically for content based access. A semantic library system is implemented by servers that use content based routing [GLB85] to direct a single user request to one or more relevant semantic file systems.
• We have already completed the implementation of an NFS compatible query processing system that forwards requests to multiple hosts and combines the results.
• Virtual directories can be used as an interface to other systems, such as information retrieval systems and programming environment support systems, such as PCTE. We are exploring also how existing applications could access object repositories via a virtual directory interface. It is possible to extend the semantics of a semantic file system to include access to individual entities in a manner suitable for an object repository [CG91].
• Relevance feedback and query results could be added by introducing new virtual directories.

The implementation of real-time indexing may require a substantial amount of computing power at a semantic file server. We are investigating how to optimize the task of real-time indexing in order to minimize this load. Another area of research is exploring how massive parallelism [SK86] might replace indexing.

An interesting limiting case of our design is a system that makes an underlying tree structured naming system superfluous. In such a system all directories would be computed upon demand, including directories that correspond to traditional tree structured file names. Such a system might help us share information more effectively by encouraging query based access that would lead to the discovery of unexpected but useful information.

Acknowledgments

We would like to thank Doug Grundman, Andrew Myers, and Raymie Stata, for their various contributions to the paper and the implementation. The referees provided valuable feedback and concrete suggestions that we have endeavored to incorporate into the paper. In particular, we very much appreciate the useful and detailed comments provided by Mike Burrows.

References


Semantic File Retrieval in File Systems using Virtual Directories
Prashanth Mohan, Raghuraman, Venkateswaran S and Dr. Arul Siromoney
{prashmohan, raaghun2222, wenkat.a}@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[2], Apple’s Spotlight[2], Beagle[1], 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
• 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:mp3ˆlen>3mˆartist:Mike’

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

1http://desktop.google.com/
2http://www.apple.com/macosx/tiger/
3http://beagle-project.org/MainPage
4http://msdn.microsoft.com/data/ref/winfs/
4) The user is now chdirred 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 4th 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/luufs.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 heirarchical 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 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.
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

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 GLScribe is defined wholly in user-space, and thus, file system events may occur that GLScribe does not record.

B. Leaftag

Leaftag 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 symlink to the targets. It provides constant time operation using extended attributes and supports many file operations.

REFERENCES

Fast Algorithms for Mining Association Rules

Rakesh Agrawal Ramakrishnan Srikant*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We consider the problem of discovering association rules between items in a large database of sales transactions. We present two new algorithms for solving this problem that are fundamentally different from the known algorithms. Empirical evaluation shows that these algorithms outperform the known algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We also show how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid. Scale-up experiments show that AprioriHybrid scales linearly with the number of transactions. AprioriHybrid also has excellent scale-up properties with respect to the transaction size and the number of items in the database.

1 Introduction

Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the basket data. A record in such data typically consists of the transaction date and the items bought in the transaction. Successful organizations view such databases as important pieces of the marketing infrastructure. They are interested in instituting information-driven marketing processes, managed by database technology, that enable marketers to develop and implement customized marketing programs and strategies [8].

The problem of mining association rules over basket data was introduced in [4]. An example of such a rule might be that 98% of customers that purchase tires and auto accessories also get automotive services done. Finding all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalag design, add-on sales, store layout, and customer segmentation based on buying patterns. The databases involved in these applications are very large. It is imperative, therefore, to have fast algorithms for this task.

The following is a formal statement of the problem [4]: Let \( I = \{i_1, i_2, \ldots, i_m\} \) be a set of literals, called items. Let \( D \) be a set of transactions, where each transaction \( T \) is a set of items such that \( T \subseteq I \). Associated with each transaction is a unique identifier, called its TID. We say that a transaction \( T \) contains \( X \), a set of some items in \( I \), if \( X \subseteq T \).

An association rule is an implication of the form

\[
X \Rightarrow Y, \quad \text{where } X \subseteq I, \ Y \subseteq I, \text{ and } X \cap Y = \emptyset.
\]

The rule \( X \Rightarrow Y \) holds in the transaction set \( D \) with confidence \( c \) if \( \% \) of transactions in \( D \) that contain \( X \) also contain \( Y \). The rule \( X \Rightarrow Y \) has support \( \text{supp} \) in the transaction set \( D \) if \( \% \) of transactions in \( D \) contain \( X \cup Y \). Our rules are somewhat more general than in [4] in that we allow a consequent to have more than one item.

Given a set of transactions \( D \), the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called \( \text{minsup} \)) and minimum confidence (called \( \text{minconf} \)) respectively. Our discussion is neutral with respect to the representation of \( D \). For example, \( D \) could be a data file, a relational table, or the result of a relational expression.

An algorithm for finding all association rules, henceforth referred to as the \( \text{MIS} \) algorithm, was presented in [4]. Another algorithm for this task, called the \( \text{SET} \) algorithm, has been proposed in [3]. In this paper, we present two new algorithms, \textit{Apriori} and \textit{AprioriTid}, that differ fundamentally from these algorithms. We present experimental results showing

---

"Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the \#s of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment."
that the proposed algorithms always outperform the earlier algorithms. The performance gap is shown to increase with problem size, and ranges from a factor of three for small problems to more than an order of magnitude for large problems. We then discuss how the best features of Apriori and AprioriTid can be combined into a hybrid algorithm, called AprioriHybrid. Experiments show that the AprioriHybrid has excellent scale-up properties, opening up the feasibility of mining association rules over very large databases.

The problem of finding association rules falls within the purview of database mining [2] [12], also called knowledge discovery in databases [2]. Related, but not directly applicable, work includes the induction of classification rules [9] [11] [22], discovery of causal rules [9], learning of logical definitions [18], fitting of functions to data [11], and clustering [9] [10]. The closest work in the machine learning literature is the KID3 algorithm presented in [20]. If used for finding all association rules, this algorithm will make as many passes over the data as the number of combinations of items in the antecedent, which is exponentially large. Related work in the database literature is the work on inferring functional dependencies from data [16]. Functional dependencies are rules requiring strict satisfaction. Consequently, having determined a dependence \( X \rightarrow A \), the algorithms in [16] consider any other dependency of the form \( X + Y \rightarrow A \) redundant and do not generate it. The association rules we consider are probabilistic in nature. The presence of a rule \( X \rightarrow A \) does not necessarily mean that \( X + Y \rightarrow A \) also holds because the latter may not have minimum support. Similarly, the presence of rules \( X \rightarrow Y \) and \( Y \rightarrow Z \) do not necessarily mean that \( X \rightarrow Z \) holds because the latter may not have minimum confidence.

There has been work on quantifying the "usefulness" or "interestingness" of a rule [20]. What is useful or interesting is often application-dependent. The need for a human in the loop and providing tools to allow human guidance of the rule discovery process has been articulated, for example, in [2] [14]. We do not discuss these issues in this paper, except to point out that these are necessary features of a rule discovery system that may use our algorithms as the engine of the discovery process.

1.1 Problem Decomposition and Paper Organization

The problem of discovering all association rules can be decomposed into two subproblems [4]:

1. Find all sets of items (itemsets) that have transaction support above minimum support. The support for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called large itemsets, and all others small itemsets. In Section 2, we give new algorithms, Apriori and AprioriTid, for solving this problem.

2. Use the large itemsets to generate the desired rules. Here is a straightforward algorithm for this task.

For every large itemset \( l \), find all non-empty subsets of \( l \). For every such subset \( a \), output a rule of the form \( a \Rightarrow (l - a) \) if the ratio of support for \( l \) to support for \( a \) is at least \( minconf \). We need to consider all subsets of \( l \) to generate rules with multiple consequents. Due to lack of space, we do not discuss this subproblem further, but refer the reader to [5] for a fast algorithm.

In Section 3, we show the relative performance of the proposed Apriori and AprioriTid algorithms against the AIS [4] and SETM [13] algorithms. To make the paper self-contained, we include an overview of the AIS and SETM algorithms in this section. We also describe how the Apriori and AprioriTid algorithms can be combined into a hybrid algorithm, AprioriHybrid, and demonstrate the scale-up properties of this algorithm. We conclude by pointing out some related open problems in Section 4.

2 Discovering Large Itemsets

Algorithms for discovering large itemsets make multiple passes over the data. In the first pass, we count the support of individual items and determine which of them are large, i.e., have minimum support. In each subsequent pass, we start with a seed set of itemsets found to be large in the previous pass. We use this seed set for generating new potentially large itemsets, called candidate itemsets, and count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, we determine which of the candidate itemsets are actually large, and they become the seed for the next pass. This process continues until no new large itemsets are found.

The Apriori and AprioriTid algorithms we propose differ fundamentally from the AIS [4] and SETM [13] algorithms in terms of which candidate itemsets are counted in a pass and in the way that those candidates are generated. In both the AIS and SETM algorithms, candidate itemsets are generated on-the-fly during the pass as data is being read. Specifically, after reading a transaction, it is determined which of the itemsets found large in the previous pass are present in the transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. However, as we will see, the disadvantage
is that this results in unnecessarily generating and counting too many candidate itemsets that turn out to be small.

The Apriori and AprioriTid algorithms generate the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass—without considering the transactions in the database. The basic intuition is that any subset of a large itemset must be large. Therefore, the candidate itemsets having \( k \) items can be generated by joining large itemsets having \( k-1 \) items, and deleting those that contain any subset that is not large. This procedure results in generation of a much smaller number of candidate itemsets.

The AprioriTid algorithm has the additional property that the database is not used at all for counting the support of candidate itemsets after the first pass. Rather, an encoding of the candidate itemsets used in the previous pass is employed for this purpose. In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort. We will explain these points in more detail when we describe the algorithms.

**Notation** We assume that items in each transaction are kept sorted in their lexicographic order. It is straightforward to adapt these algorithms to the case where the database \( D \) is kept normalized and each database record is a \(<TID, item> \) pair, where \( TID \) is the identifier of the corresponding transaction.

We call the number of items in an itemset its **size**, and call an itemset of size \( k \) a \( k \)-itemset. Itemsets within an itemset are kept in lexicographic order. We use the notation \( c[1] \cdot c[2] \cdots c[k] \) to represent a \( k \)-itemset \( c \) consisting of items \( c[1], c[2], \ldots, c[k] \), where \( c[1] < c[2] < \ldots < c[k] \). If \( c = X \cdot Y \) and \( Y \) is an \( m \)-itemset, we also call \( Y \) an \( m \)-extension of \( X \). Associated with each itemset is a count field to store the support for this itemset. The count field is initialized to zero when the itemset is first created.

We summarize in Table 1 the notation used in the algorithms. The set \( C_k \) is used by AprioriTid and will be further discussed when we describe this algorithm.

### 2.1 Algorithm Apriori

Figure 1 gives the Apriori algorithm. The first pass of the algorithm simply counts item occurrences to determine the large \( k \)-itemsets. A subsequent pass, say pass \( k \), consists of two phases. First, the large itemsets \( L_{k-1} \) found in the \((k-1)\)th pass are used to generate the candidate itemsets \( C_k \), using the apriori-gen function described in Section 2.1.1. Next, the database is scanned and the support of candidates in \( C_k \) is counted. For fast counting, we need to efficiently determine the candidates in \( C_k \) that are contained in a given transaction \( f \). Section 2.1.2 describes the subset function used for this purpose. See [5] for a discussion of buffer management.

1. \( L_1 = \{ \text{large 1-itemsets} \} \)
2. for ( \( k = 2; L_{k-1} \neq \emptyset; k+1 \) ) do begin
3. \( C_k = \text{apriori-gen}(L_{k-1}) \); // New candidates
4. forall transactions \( t \in D \) do begin
5. \( C_t = \text{subset}(C_k, t) \); // Candidates contained in \( t \)
6. forall candidates \( c \in C_t \) do
7. \( c.\text{count}++ \\
8. end
9. \( L_k = \{ c \in C_k \mid c.\text{count} \geq \text{minsup} \} \\
10. end
11. Answer = \( \cup L_k \);

Figure 1: Algorithm Apriori

#### 2.1.1 Apriori Candidate Generation

The apriori-gen function takes as argument \( L_{k-1} \), the set of all large \([k-1]\)-itemsets. It returns a superset of the set of all \( k \)-itemsets. The function works as follows. First, in the **join** step, we join \( L_{k-1} \) with \( L_{k-1} \):

- **insert into** \( C_k \)
  - select \( p.\text{item}_1, p.\text{item}_2, \ldots, p.\text{item}_{k-1} \)
  - from \( L_{k-1} \) p, \( L_{k-1} \) q
  - where \( p.\text{item}_1 = q.\text{item}_1, \ldots, p.\text{item}_{k-2} = q.\text{item}_{k-2}, \)
  - p.\text{item}_{k-1} < q.\text{item}_{k-1} \)

Next, in the **prune** step, we delete all itemsets \( c \in C_k \) such that some \([k-1]\)-subset of \( c \) is not in \( L_{k-1} \):

1. Concurrent to our work, the following two-step candidate generation procedure has been proposed in [17]:
2. \( C_k = \{ X \cup X' \mid X \in L_{k-1}, X' \in L_{k-1} - \{ X \} : |X \cup X'| = k \} \\
3. \{ X \in C_k \mid X \text{ contains } k \text{ members of } L_{k-1} \}

These two steps are similar to our join and prune steps respectively. However, in general, step 1 would produce a superset of the candidates produced by our join step.
forall itemsets \( c \in C_k \) do
forall \((k-1)\)-subsets \( s \) of \( c \) do
if \( s \notin L_{k-1} \) then
    delete \( c \) from \( C_k \);

Example: Let \( L_2 \) be \{ \{1 2 3\}, \{1 2 4\}, \{1 3 4\}, \{1 3 5\}, \{2 3 4\}\}. After the join step, \( C_4 \) will be \{ \{1 2 3 4\}, \{1 3 4 5\} \}. The prune step will delete the itemset \{1 3 4 5\} because the itemset \{1 4 5\} is not in \( L_2 \).

We will then be left with only \{1 2 3 4\} in \( C_4 \).

Contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass \( k \) of these algorithms, a database transaction \( t \) is read and it is determined which of the large itemsets in \( L_{k-1} \) are present in \( t \). Each of these large itemsets \( l \) is then extended with all those large items that are present in \( t \) and occur later in the lexicographic ordering than any of the items in \( l \). Continuing with the previous example, consider a transaction \{1 2 3 4 5\}. In the fourth pass, AIS and SETM will generate two candidates, \{1 2 3 4\} and \{1 2 3 5\}, by extending the large itemset \{1 2 3\}. Similarly, an additional three candidate itemsets will be generated by extending the large itemset \{1 2 4 5\}.

Starting from the root node, the subset function finds all the candidates contained in a transaction \( t \) as follows. If we are at a leaf, we find which of the itemsets in the leaf are contained in \( t \) and add references to them to the answer set. If we are at an interior node and have reached it by hashing the item \( i \), we hash on each item that comes after \( i \) and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in \( t \).

To see why the subset function returns the desired set of references, consider what happens at the root node. For any itemset \( c \) contained in transaction \( t \), the first item of \( c \) must be in \( t \). At the root, by hashing on every item in \( t \), we ensure that we only ignore itemsets that start with an item not in \( t \). Similar arguments apply at lower depths. The only additional factor is that, since the items in any itemset are ordered, if we reach the current node by hashing the item \( i \), we only need to consider the items in \( t \) that occur after \( i \).

2.2 Algorithm AprioriTid

The AprioriTid algorithm, shown in Figure 2, also uses the aprior-gen function (given in Section 2.1.1) to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database \( D \) is not used for counting support after the first pass. Rather, the set \( C_k \) is used for this purpose. Each member of the set \( C_k \) is of the form \( <TID, \{X_k\}> \), where each \( X_k \) is a potentially large \( k \)-itemset present in the transaction with identifier \( TID \). For \( k = 1 \), \( C_1 \) corresponds to the database \( D \), although conceptually each item \( i \) is replaced by the itemset \{\} for \( k > 1 \), \( C_k \) is generated by the algorithm (step 10). The member
of \( C_k \) corresponding to transaction \( t \) is \( \prec t.TID \), \( \{ c \in C_k \mid c \text{ contained in } t \} \). If a transaction does not contain any candidate \( k \)-itemset, then \( C_k \) will not have an entry for this transaction. Thus, the number of entries in \( C_k \) may be smaller than the number of transactions in the database, especially for large values of \( k \). In addition, for large values of \( k \), each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, for small values for \( k \), each entry may be larger than the corresponding transaction because an entry in \( C_k \) includes all candidate \( k \)-itemsets contained in the transaction.

In Section 2.2.1, we give the data structures used to implement the algorithm. See [5] for a proof of correctness and a discussion of buffer management.

1. \( L_1 = \text{[large itemsets]} \);  
2. \( C_k = \text{database } D \);  
3. for \(( k = 2; L_{k-1} \neq \emptyset; k++) \) do begin  
4. \( C_k = \text{apriori-gen}(L_{k-1}) \); // New candidates  
5. \( C_k = \emptyset \);  
6. for all entries \( t \in C_{k-1} \) do begin  
7. \( \text{// determine candidate itemsets in } C_k \text{ contained in } t \text{.} \)  
8. \( C_k = \{ c \in C_k \mid \{ c - c[k] \} \in \text{set-of-itemsets} \land \{ c - c[k-1] \} \in \text{set-of-itemsets} \} \);  
9. \( \text{forall candidates } c \in C_k \text{ do} \)  
10. \( \text{c.count}++ \);  
11. if \( C_k \neq \emptyset \) then \( C_k = \prec t.TID, C_k \rangle \);  
12. \( L_k = \{ c \in C_k \mid \text{c.count } \geq \text{mnsup} \} \);  
13. end  
14. end

Figure 2: Algorithm AprioriTid

Example: Consider the database in Figure 3 and assume that minimum support is 2 transactions. Calling apriori-gen with \( L_1 \) at step 4 gives the candidate itemsets \( C_2 \). In steps 6 through 10, we count the support of candidates in \( C_2 \) by iterating over the entries in \( C_1 \) and generate \( C_3 \). The first entry in \( C_1 \) is \( \{ \{1,3\} \} \) corresponding to transaction 100. The \( C_3 \) at step 7 corresponding to this entry \( t \) is \( \{ \{1,3\} \} \), because \( \{1,3\} \) is a member of \( C_3 \) and both \( \{1,3\} \land \{1\} \) and \( \{1,3\} \land \{3\} \) are members of \( t \text{-set-of-itemsets} \).

Calling apriori-gen with \( L_2 \) at step 5 gives \( C_3 \). Making a pass over the data with \( TID_1 \) and \( C_2 \) generates \( C_3 \). Note that there is no entry in \( C_3 \) for the transactions with \( TID_1 \) 100 and 400, since they do not contain any of the itemsets in \( C_3 \). The candidate \( \{2,3,5\} \) in \( C_3 \) turns out to be large and is the only member of \( L_2 \). When we generate \( C_3 \) using \( L_2 \), it turns out to be empty, and we terminate.

### Table 1

<table>
<thead>
<tr>
<th>TID</th>
<th>Itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1 3 4</td>
</tr>
<tr>
<td>200</td>
<td>2 3 5</td>
</tr>
<tr>
<td>300</td>
<td>1 2 3 5</td>
</tr>
<tr>
<td>400</td>
<td>2 5</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>TID</th>
<th>Set-of-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{ {1}, {3}, {4}}</td>
</tr>
<tr>
<td>200</td>
<td>{ {2}, {3}, {5}}</td>
</tr>
<tr>
<td>300</td>
<td>{ {1}, {2}, {3}, {5}}</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>TID</th>
<th>Itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{ {1}}</td>
</tr>
<tr>
<td>200</td>
<td>{ {2}}</td>
</tr>
<tr>
<td>300</td>
<td>{ {2}, {5}}</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>TID</th>
<th>Itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{ {1}}</td>
</tr>
<tr>
<td>200</td>
<td>{ {2}}</td>
</tr>
<tr>
<td>300</td>
<td>{ {2}, {5}}</td>
</tr>
</tbody>
</table>

Figure 3: Example

#### 2.2.1 Data Structures

We assign each candidate itemset a unique number, called its ID. Each set of candidate itemsets \( C_k \) is kept in an array indexed by the IDs of the itemsets in \( C_k \). A member of \( C_k \) is now of the form \( \prec t.TID, \{ID\} \rangle \). Each \( C_k \) is stored in a sequential structure.

The apriori-gen function generates a candidate \( k \)-itemset \( c_k \) by joining two large \( (k-1) \)-itemsets. We maintain two additional fields for each candidate itemset: i) generators and ii) extensions. The generators field of a candidate itemset \( c_k \) stores the IDs of the two large \( (k-1) \)-itemsets whose join generated \( c_k \). The extensions field of an itemset \( c_k \) stores the IDs of all the \( (k-1) \)-candidates that are extensions of \( c_k \). Thus, when a candidate \( c_k \) is generated by joining \( c_{k-1} \) and \( c_{k-1} \), we save the IDs of \( c_{k-1} \) and \( c_{k-1} \) in the generators field for \( c_k \). At the same time, the ID of \( c_k \) is added to the extensions field of \( c_{k-1} \).
We now describe how Step 7 of Figure 2 is implemented using the above data structures. Recall that the setofitemsets field of an entry \( t \) in \( T_{i+1} \) gives the IDs of all \((k-1)\) candidates contained in transaction \( t \). For each such candidate \( c_{k-1} \), the extensions field gives \( T_k \), the set of IDs of all the candidate \( k \) itemsets that are extensions of \( c_{k-1} \). For each \( c_k \) in \( T_k \), the generators field gives the IDs of the two itemsets that generated \( c_k \). If these itemsets are present in the entry for \( c_{k-1} \), we can conclude that \( c_k \) is present in transaction \( t \). Thus, we can create a set of candidates \( C_k \) from those candidates \( c_k \) that we have already counted.

3 Performance

To assess the relative performance of the algorithms for discovering large sets, we performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, 64 MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

We first give an overview of the AIS [4] and SETM [13] algorithms against which we compare the performance of the Apriori and AprioriTid algorithms. We then describe the synthetic datasets used in the performance evaluation and show the performance results. Finally, we describe how the best performance features of Apriori and AprioriTid can be combined into an AprioriHybrid algorithm and demonstrate its scalability properties.

3.1 The AIS Algorithm

Candidate itemsets are generated and counted on-the-fly as the database is scanned. After reading a transaction, it is determined which of the itemsets that were found to be large in the previous pass are contained in this transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. A large itemset \( l \) is extended with only those items that are large and occur later in the lexicographic ordering of items than any of the items in \( l \). The candidates generated from a transaction are added to the set of candidate itemsets maintained for the pass, or the counts of the corresponding entries are increased if they were created by an earlier transaction. See [4] for further details of the AIS algorithm.

3.2 The SETM Algorithm

The SETM algorithm [13] was motivated by the desire to use SQL to compute large itemsets. Like AIS, the SETM algorithm also generates candidates on-the-fly based on transactions read from the database. It thus generates and counts every candidate itemset that the AIS algorithm generates. However, to use the standard SQL join operation for candidate generation, SETM separates candidate generation from counting. It saves a copy of the candidate itemset together with the TID of the generating transaction in a sequential structure. At the end of the pass, the support count of candidate itemsets is determined by sorting and aggregating this sequential structure.

SETM remembers the TIDs of the generating transactions with the candidate itemsets. To avoid needing a subset operation, it uses this information to determine the large itemsets contained in the transaction read. \( T_k \subseteq C_k \) and is obtained by deleting those candidates that do not have minimum support. Assuming that the database is sorted in TID order, SETM can easily find the large itemsets contained in a transaction in the next pass by sorting \( T_k \) on TID. In fact, it needs to visit every member of \( T_k \) only once in the TID order, and the candidate generation can be performed using the relational merge-join operation [13].

The disadvantage of this approach is mainly due to the size of candidate sets \( C_k \). For each candidate itemset, the candidate set size has as many entries as the number of transactions in which the candidate itemset is present. Moreover, when we are ready to count the support for candidate itemsets at the end of the pass, \( C_k \) is in the wrong order and needs to be sorted on itemsets. After counting and pruning out small candidate itemsets that do not have minimum support, the resulting set \( T_k \) needs another sort on TID before it can be used for generating candidates in the next pass.

3.3 Generation of Synthetic Data

We generated synthetic transactions to evaluate the performance of the algorithms over a large range of data characteristics. These transactions mimic the transactions in the retailing environment. Our model of the "real" world is that people tend to buy sets of items together. Each such set is potentially a maximal large itemset. An example of such a set might be sheets, pillow case, comforter, and ruffles. However, some people may buy only some of the items from such a set. For instance, some people might buy only sheets and pillow case, and some only sheets. A transaction may contain more than one large itemset. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form another large itemset. Transaction sizes are typically clustered around a mean and a few transactions have many items. Typical sizes of large itemsets are also
clustered around a mean, with a few large itemsets having a large number of items.

To create a dataset, our synthetic data generation program takes the parameters shown in Table 2.

Table 2: Parameters

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of transactions</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Average size of the transactions</td>
<td>100K</td>
</tr>
<tr>
<td>3</td>
<td>Average size of the maximal potentially large itemsets</td>
<td>100K</td>
</tr>
<tr>
<td>4</td>
<td>Number of maximal potentially large itemsets</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Number of items</td>
<td>6</td>
</tr>
</tbody>
</table>

We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean $\mu$ equal to $|I|$. Note that if each item is chosen with the same probability $p$, and there are $N$ items, the expected number of items in a transaction is given by a binomial distribution with parameters $N$ and $p$, and is approximated by a Poisson distribution with mean $NP$.

We then assign items to the transaction. Each transaction is assigned a series of potentially large itemsets. If the large itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases, and the itemset is moved to the next transaction the rest of the cases.

Large itemsets are chosen from a set $T$ of such itemsets. The number of itemsets in $T$ is set to 2. There is an inverse relationship between $|I|$ and the average support for potentially large itemsets. An itemset in $T$ is generated by first picking the size of the itemset from a Poisson distribution with mean $\mu$ equal to $|I|$. Items in the first itemset are chosen randomly. To model the phenomenon that large itemsets often have common items, some fraction of items in subsequent itemsets are chosen from the previous itemset generated. We use an exponentially distributed random variable with mean equal to the correlation level $c$ to decide this fraction for each itemset. The remaining items are picked at random. In the datasets used in the experiments, the correlation level was set to 0.5. We ran some experiments with the correlation level set to 0.25 and 0.75 but did not find much difference in the nature of our performance results.

Each itemset in $T$ has a weight associated with it, which corresponds to the probability that this itemset will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets in $T$ is 1. The next itemset to be put in the transaction is chosen from $T$ by tossing an [2]-sided weighted coin, where the weight for a side is the probability of picking the associated itemset.

To model the phenomenon that all the items in a large itemset are not always bought together, we assign each itemset in $T$ a corruption level $c$. When adding an itemset to a transaction, we keep dropping an item from the itemset as long as a uniformly distributed random number between 0 and 1 is less than $c$. Thus for an itemset of size $l$, we will add $l$ items to the transaction $1 - c$ of the time, $l - 1$ items $c(1 - c)$ of the time, $l - 2$ items $c^2(1 - c)$ of the time, etc. The corruption level for an itemset is fixed and is obtained from a normal distribution with mean 0.5 and variance 0.1.

We generated datasets by setting $N = 10^8$ and $|I| = 2000$. We chose 3 values for $|T|$: 5, 10, and 20. We also chose 3 values for $|I|$: 2, 4, and 6. The number of transactions was set to 180,000 because, as we will see in Section 3.4, SETM could not be run for larger values. However, for our scale-up experiments, we generated datasets with up to 10 million transactions (838MB for T200). Table 3 summarizes the dataset parameter settings. For the same $|T|$ and $|I|$ values, the size of datasets in megabytes were roughly equal for the different values of $|I|$.

Table 3: Parameter settings

| Name                | $|T|$ | $|I|$ | Size in Megabytes |
|---------------------|-----|-----|------------------|
| T5.12_D100K         | 5   | 2   | 100K             |
| T10.12_D100K        | 10  | 2   | 100K             |
| T10.14_D100K        | 10  | 4   | 100K             |
| T20.12_D100K        | 20  | 2   | 100K             |
| T20.14_D100K        | 20  | 4   | 100K             |
| T20.16_D100K        | 20  | 6   | 100K             |

3.4 Relative Performance

Figure 4 shows the execution times for the six synthetic datasets given in Table 3 for decreasing values of minimum support. As the minimum support decreases, the execution times of all the algorithms increase because of increases in the total number of candidate and large itemsets.

For SETM, we have only plotted the execution times for the dataset T5.12_D100K in Figure 4. The execution times for SETM for the two datasets with an average transaction size of 10 are given in Table 4. We did not plot the execution times in Table 4 on the corresponding graphs because they are too large compared to the execution times of the other algorithms. For the three datasets with transaction sizes of 20, SETM took too long to execute and we aborted those runs as the trends were clear. Clearly, Apriori beats SETM by more than an order of magnitude for large datasets.
Figure 4: Execution times
Table 4: Execution times in seconds for SETM

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Minimum Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0%</td>
</tr>
<tr>
<td>SETM</td>
<td>74</td>
</tr>
<tr>
<td>Apriori</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Dataset T10H.D100K

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Minimum Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.1</td>
</tr>
<tr>
<td>Apriori</td>
<td>3.8</td>
</tr>
</tbody>
</table>

3.5 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 5 the sizes of the large and candidate sets in different passes for the T10H.D100K dataset for the minimum support of 0.75%. Note that the Y-axis in this graph has a log scale.

The fundamental problem with the SETM algorithm is the size of its \( C_k \) sets. Recall that the size of the set \( C_k \) is given by

\[
\sum_{\text{candidate itemsets } c} \text{support-count}(c).
\]

Thus, the sets \( C_k \) are roughly \( S \) times larger than the corresponding \( C_j \) sets, where \( S \) is the average support count of the candidate itemsets. Unless the problem size is very small, the \( C_k \) sets have to be written to disk, and externally sorted twice, causing the SETM algorithm to perform poorly. This explains the jump in time for SETM in Table 4 when going from 1.5% support to 1.0% support for datasets with transaction size 10. The largest dataset in the scale-up experiments for SETM in [13] was still small enough that \( C_4 \) could fit in memory; hence, they did not encounter this jump in execution time. Note that for the same minimum support, the support count for candidate itemsets increases linearly with the number of transactions. Thus, as we increase the number of transactions for the same values of \( |T| \) and \( |I| \), though the size of \( C_4 \) does not change, the size of \( C_5 \) goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other algorithms will become even larger.

The problem with AIS is that it generates too many candidates that later turn out to be small, causing it to waste too much effort. Apriori also counts too many small sets in the second pass (recall that \( C_3 \) is really a cross-product of \( L_1 \) with \( L_1 \)). However, this wastage decreases dramatically from the third pass onward. Note that for the example in Figure 5, after pass 3, almost every candidate itemset counted by Apriori turns out to be a large set.

AprioriTid also has the problem of SETM that \( C_4 \) tends to be large. However, the apriori candidate generation used by AprioriTid generates significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the \( C_4 \) of AprioriTid has fewer entries than that of SETM. AprioriTid is also able to use a single word (ID) to store a candidate rather than requiring as many words as the number of items in the candidate. In addition, unlike SETM, AprioriTid does not have to sort \( C_4 \). Thus, AprioriTid does not suffer as much as SETM from maintaining \( C_4 \).

AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set \( C_k \). Hence, AprioriTid is very effective in later passes when the size of \( C_k \) becomes small compared to the
size of the database. Thus, we find that AprioriTid beats Apriori when its $C_k$ sets can fit in memory and the distribution of the large itemsets has a long tail. When $C_k$ doesn’t fit in memory, there is a jump in the execution time for AprioriTid, such as when going from 0.75% to 0.5% for datasets with transaction size 10 in Figure 4. In this region, Apriori starts beating AprioriTid.

3.6 Algorithm AprioriHybrid

It is not necessary to use the same algorithm in all the passes over data. Figure 6 shows the execution times for Apriori and AprioriTid for different passes over the dataset T10.14.D100K. In the earlier passes, Apriori does better than AprioriTid. However, AprioriTid beats Apriori in later passes. We observed similar relative behavior for the other datasets, the reason for which is as follows. Apriori and AprioriTid use the same candidate generation procedure and therefore count the same itemsets. In the later passes, the number of candidate itemsets reduces (see the size of $C_k$ for Apriori and AprioriTid in Figure 5). However, Apriori still examines every transaction in the database. On the other hand, rather than scanning the database, AprioriTid scans $C_k$ for obtaining support counts, and the size of $C_k$ has become smaller than the size of the database. When the $C_k$ sets can fit in memory, we do not even incur the cost of writing them to disk.

![Figure 6: Per pass execution times of Apriori and AprioriTid (T10.14.D100K, minsup = 0.75%)](image)

Based on these observations, we can design a hybrid algorithm, which we call AprioriHybrid, that uses Apriori in the initial passes and switches to AprioriTid when it expects that the set $C_k$ at the end of the pass will fit in memory. We use the following heuristic to estimate if $C_k$ would fit in memory in the next pass. At the end of the current pass, we have the counts of the candidates in $C_k$. From this, we estimate what the size of $C_k$ would have been if it had been generated. This size, in words, is $|\sum_{c \in C_k} \text{support}(c) + \text{number of transactions}|$. If $C_k$ in this pass was small enough to fit in memory, and there were fewer large candidates in the current pass than the previous pass, we switch to AprioriTid. The latter condition is added to avoid switching when $C_k$ in the current pass fits in memory but $C_k$ in the next pass may not.

Switching from Apriori to AprioriTid does involve a cost. Assume that we decide to switch from Apriori to AprioriTid at the end of the $k$th pass. In the $(k+1)$th pass, after finding the candidate itemsets contained in a transaction, we will also have to add their IDs to $C_{k+1}$ (see the description of AprioriTid in Section 2.2). Thus, there is an extra cost incurred in this pass relative to just running Apriori. It is only in the $(k+2)$th pass that we actually start running AprioriTid. Thus, if there are no large $(k+1)$-itemsets, or no $(k+2)$-candidates, we will incur the cost of switching without getting any of the savings of using AprioriTid.

Figure 7 shows the performance of AprioriHybrid relative to Apriori and AprioriTid for three datasets. AprioriHybrid performs better than Apriori in almost all cases. For T10.12.D100K with 1.5% support, AprioriHybrid does a little worse than Apriori since the pass in which the switch occurred was the last pass; AprioriHybrid thus incurred the cost of switching without realizing the benefits. In general, the advantage of AprioriHybrid over Apriori depends on how the size of the $C_k$ set declines in the later passes. If $C_k$ remains large until nearly the end and then has an abrupt drop, we will not gain much by using AprioriHybrid since we can use AprioriTid only for a short period of time after the switch. This is what happened with the T20.16.D100K dataset. On the other hand, if there is a gradual decline in the size of $C_k$, AprioriTid can be used for a while after the switch, and a significant improvement can be obtained in the execution time.

3.7 Scale-up Experiment

Figure 8 shows how AprioriHybrid scales up as the number of transactions is increased from 100,000 to 10 million transactions. We used the combinations (T5.12), (T10.14), and (T20.16) for the average sizes of transactions and itemsets respectively. All other parameters were the same as for the data in Table 3. The sizes of these datasets for 10 million transactions were 239MB, 439MB and 838MB respectively. The minimum support level was set to 0.75%. The execution times are normalized with respect to the times for the 100,000 transaction datasets in the first
Next, we examined how AprioriHybrid scaled up with the number of items. We increased the number of items from 1000 to 10,000 for the three parameter settings T10.12.D100K, T10.14.D100K and T20.16.D100K. All other parameters were the same as for the data in Table 3. We ran experiments for a minimum support at 0.75%, and obtained the results shown in Figure 9. The execution times decreased a little since the average support for an item decreased as we increased the number of items. This resulted in fewer large itemsets and, hence, faster execution times.

Finally, we investigated the scale-up as we increased the average transaction size. The aim of this experiment was to see how our data structures scaled with the transaction size, independent of other factors like the physical database size and the number of large itemsets. We kept the physical size of the
The number of items in the database was roughly constant by keeping the product of the average transaction size and the number of transactions constant. The number of transactions ranged from 200,000 for the database with an average transaction size of 5 to 20,000 for the database with an average transaction size 50. Fixing the minimum support as a percentage would have led to large increases in the number of large itemsets as the transaction size increased, since the probability of an itemset being present in a transaction is roughly proportional to the transaction size. We therefore fixed the minimum support level in terms of the number of transactions. The results are shown in Figure 10. The numbers in the key (e.g., 500) refer to this minimum support. As shown, the execution times increase with the transaction size, but only gradually. The main reason for the increase was that in spite of setting the minimum support in terms of the number of transactions, the number of large itemsets increased with increasing transaction length. A secondary reason was that finding the candidates present in a transaction took a little longer time.

4 Conclusions and Future Work

We presented two new algorithms, Apriori and AprioriTid, for discovering all significant association rules between items in a large database of transactions. We compared these algorithms to the previously known algorithms, the AIS [4] and SETM [13] algorithms. We presented experimental results, showing that the proposed algorithms always outperform AIS and SETM. The performance gap increased with the problem size, and ranged from a factor of three for small problems to more than an order of magnitude for large problems.

We showed how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid, which then becomes the algorithm of choice for this problem. Scale-up experiments showed that AprioriHybrid scales linearly with the number of transactions. In addition, the execution time decreases a little as the number of items in the database increases. As the average transaction size increases (while keeping the database size constant), the execution time increases only gradually. These experiments demonstrate the feasibility of using AprioriHybrid in real applications involving very large databases.

The algorithms presented in this paper have been implemented on several data repositories, including the AIX file system, DB2/MVS, and DB2/6000. We have also tested these algorithms against real customer data, the details of which can be found in [5]. In the future, we plan to extend this work along the following dimensions:

- Multiple taxonomies (is-a hierarchies) over items are often available. An example of such a hierarchy is that a dish washer is a kitchen appliance is a heavy electric appliance, etc. We would like to be able to find association rules that use such hierarchies.

- We did not consider the quantities of the items bought in a transaction, which are useful for some applications. Finding such rules needs further work.

The work reported in this paper has been done in the context of the Quest project at the IBM Almaden Research Center. In Quest, we are exploring the various aspects of the database mining problem. Besides the problem of discovering association rules, some other problems that we have looked into include...
the enhancement of the database capability with classification queries [2] and similarity queries over time sequences [1]. We believe that database mining is an important new application area for databases, combining commercial interest with intriguing research questions.

Acknowledgment We wish to thank Mike Carey for his insightful comments and suggestions.

References


APPENDIX E : CONTRIBUTION OF TEAM MEMBERS

**Member 1: Aseem Gogte**
- **Module 3**: Extraction of metadata (for audio using TagLib)
- Literature Survey
- Mathematical model
- Testing related to File operations
- Source code formatting and documentation

**Member 2: Sahil Gupta**
- **Module 2**: Interface a Data repository using SQLite
- **Module 4**: Importing semantics into the file system
- **Module 5**: Exporting semantics from the file system
- **Module 6**: Association rule learning using Apriori Algorithm
- Designing database queries
- Database design
- Tag associations
- Integrating database with FUSE
- Data mining, association rule learning alternatives

**Member 3: Harshvardhan Pandit**
- **Module 1**: Creation of a virtual file system using FUSE
- **Module 3**: Extraction of metadata (for images and videos using Libextrator)
- Design program flow
- Component design
- Integrating FUSE with database
- Plugin architecture for metadata extraction
- API for developers, dynamic libraries
- Logging mechanism for for debugging code
- Debugging using GDB, valgrind

**Member 4: Rohit Sharma**
- **Module 2**: Interface a Data repository using SQLite
- **Module 3**: Extraction of metadata (for PDF using Poppler)
- Database design
- Managing database consistency
- Integrating modules
- Testing related to database
### APPENDIX F: GLOSSARY

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDLC</td>
<td>Software Development Life Cycle</td>
</tr>
<tr>
<td>FUSE</td>
<td>File system in Userspace</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>FAQ</td>
<td>Frequently asked questions</td>
</tr>
<tr>
<td>SFS</td>
<td>Semantic File System</td>
</tr>
<tr>
<td>VFS</td>
<td>Virtual File System</td>
</tr>
<tr>
<td>KFS</td>
<td>Knowledge File System</td>
</tr>
</tbody>
</table>
Chapter 13

BIBLIOGRAPHY


