GNU/Linux Semantic Storage System

Ahmed Salama, Ahmed Samih
Amr Ramadan, Karim M. Yousef

Contents

List of Figures

List of Tables

Preface

As the amount of information stored on and accessed through computers has increased over the past twenty years, the tools available for organizing and retrieving such information have become outdated. The GNU/Linux Semantic Storage System is an information store that represents data based on their attributes, contents, and relationships. The system provides access to the data through advanced organization mechanisms and fast data searching, and it also maintains compatibility with existing applications.

Objective

With the disk drive capacity growing at a rate faster than Moore's law - a doubling of capacity every year, the increase in the amount of data that can be stored on the computer has led to a similar growth in the complexity in its retrieval.

However, the current tools and solutions have not kept pace with this information explosion, and the absence of such vision when they were designed does not make them easily extensible. The conventional file systems, for instance, impose a hierarchical structure on the user and force him to create strict organization schemes, and provide a monotonic constraint for document retrieval; a combination of its location and name.

GNU/Linux Semantic Storage System addresses these issues. It presents the user with a "semantic" interface to his data, and is designed to pull the user away from thinking of where the data is, and encourage him to think of what the data contains. The semantic attributes of a file are automatically extracted using developer-programmable importers and are indexed for efficient retrieval against a user's search queries.

Our goal is to make the transition to the new system almost transparent to the less experienced computer user, and to offer the more experienced ones a rich system that would greatly enhance how they organize and retrieve data. Also, we aim to provide application developers with enough tools to provide similar organization and searching functionality in their applications at minimum overhead.

Licensing

The license used for this Document grants free access to its content. The license permits the content to be copied, modified, and redistributed so long as the new version grants the same freedoms to others and acknowledges the original authors of the GNU/Linux Semantic Storage System. This principle is known as copyleft.

To fulfill the above goals, the text contained in this Document is licensed to the public under the GNU Free Documentation License (GFDL). The full text of this license is included in the Appendix.

Who Should Read this Documentation?

This Documentation is an overview of the design of the GNU/Linux Semantic Storage System. It is not meant to be a complete reference, but rather, a starting point for those interested in understanding how the GNU/Linux Semantic Storage System works. This Documentation is targeted to experienced Developers. An introductory-level understanding of the Linux kernel and object-oriented programming is required.

For additional information, refer to the Code Documentation, a documentation automatically generated from latest stable source code checkpoint. End-Users may refer to the User Guide for a non-technical introduction to GNU/Linux Semantic Storage System, and how to use it.

Part I

Introduction

Chapter 1

Introduction

With the disk drive capacity growing at a rate faster than Moore's law - a doubling of capacity every year [13, 23], the increase in the amount of data that can be stored on the computer has led to a similar growth in the complexity in its retrieval. Moreover, a user's personal computer is no longer his personal data store; as his data can be physically spread over the entire globe, from one server storing his e-mail messages to a corporate file server from which he retrieves daily work snapshots. Even though the files are physically disturbed, their close relationship to the user makes him regard them as "his" files.

However, the current tools and solutions have not kept pace with this information explosion, and the absence of such vision when they were designed does not make them easily extensible. These currently available file systems, for instance, impose a hierarchical structure on the user and force him to create strict organization schemes [6], and provide a monotonic constraint for document retrieval, which is a combination of its location and name.

The GNU/Linux Semantic Storage System, hereafter refered to as GLScube, is a Semantic File System that addresses these issues. It presents the user with a rich interface to his data, and is designed to pull the user away from thinking of Where the data is, and encourage him to think of What the data contains. GLScube has been designed from the ground up with the users in mind, so that the less-experienced computer users would be able to transition to the new system as easy as possible, and in the mean time, offer the more experienced users a rich system that would greatly enhance how they organize and retrieve data. Furthermore, GLScube provides application developers with capabilities that would allow them to provide rich organization and retrieval functionality in their programs at minimum overhead.

The problem with the currently available file systems [1], the so-called hierarchical file systems, is that they offer no scalable mechanisms for data organization. Their key entity for organization is the directories, or folders, which were designed in analogy to the filing cabinets used for centuries, and consequently, inherited many of it problems among which is the inability to file documents in more than one category [20]. And even though some remedies have been later added, like shortcuts or links, their adoption did not take off. Thus, it was the case that a user could only put his file in a specific place; this place is represented as a node in a (huge) tree; there was only one way to reach this place, one constraint, the path name; the file system offered no restriction on the placement or meaning of the organization, a video file can be put under a directory called “my text files”; even more, these file systems did not offer any inherent mechanism for searching, in order to search for the text file containing the word “beatles”, a program must itterate over all files, linearly scanning each one’s contents.

1.1 Previous Work

As the amount of information and data stored in the computer increased, researchers approached various techniques and solutions for organization and retrieval of documents. We build upon the finding and results of some of these systems, which you can refer to in the Reference, and we follow this with an introduction to the features of some of them:

- One of the earliest storage systems that approached file system access and organization from a content-based view, rather than a hierarchical one, was the Semantic File System [8]. The Semantic File System was an extension to the tree-structured file system that allowed associative access to the system’s

[1] From here on, “currently available file systems,” “traditional file systems” and “hierarchical file systems” are synonymous, with which we refer to the most popular file systems currently in use such as FAT32, NTFS and Extended 3 (ext3).

content. Virtual directories, whose names were interpreted as queries, were introduced as a compatibility layer for legacy applications. Extensibility was maintained by allowing for user programmable transducers that could extract key properties from files.

- Dourish et al backed up the concept of the failure of the hierarchical file system to meet the users' demands, and presented the Placeless Documents [6]. Similar to the Semantic File System, document properties were the primary tool for document management and interaction. However, the document properties were expressed relative to the consumer of the document, not its producer. In other words, document properties are user-defined, and the system is not supposed to extract them from the files. Additionally, their implementation supported Active Properties; properties carrying code that can be invoked to control or augment document functionality. Placeless distinguished between each user's properties on the same system.

- The Be File System [7] integrated meta-data support in the file system. Attribute values are stored along with their files, and each attribute is associated with its own index which was stored as a regular file in a hidden index directory. The file system supported only three indices: name, size and last modification date. When an index is created for some new attribute, the BFS leaves the responsibility to iterate over all the files on the system to the application developer. Possibly one of the advantages of such integrated design approach is inherent transactions support, where both the creation and the deletion of files were atomic.

- One of the key points of the design of the Inversion File System was to leverage the power of database management systems by building the file system on top of a low-level DBMS service [19]. This approach took advantage of the transaction protection, fine-grained time travel, and fast crash recovery for user files and file system Metadata in the underlying DBMS. However, Olson's goals were primarily focused on simplifying the file system design and not improving the usability of data access and organization.

- The Database File System [17] was a user-level file system built on top of the Berkley DB, an embedded database management system, which is built on top of the Berkley Fast File System. One of their goals of such design was to serve the needs of an evolving world wide web, which quickly started to embrace dynamic content derived from database servers, and they suggested that such unification could widen the spread of dynamic websites due to easier deployment.

- Desktop search tools emphasized on data mining all the information that is available on the user's PC, and storing the extracted meta-data and text in an index to be used later for efficient retrieval. These tools were designed to allow the user to retrieve search results quickly, and in general, were not concerned with the document organization. Examples of desktop search tools are Google Desktop Search, Microsoft MSN Search and Beagle. A more advanced search tool was Apple's Spotlight [5], which allowed the creation of Smart Folders - the technical equivalence of Gifford's Virtual directories - and provided an API that allowed developers to add Spotlight search capabilities to their own programs in their own proprietary data file formats. Among the functionalities presented by Spotlight are substring search, keyword substitution and support for separate user accounts.

1.2 What is the GNU/Linux Semantic Storage System?

As the amount of information stored on a user's hard drive increases, organization and fast access to documents becomes an even more important necessity. After all, the purpose of filing a document is to make it easier to retrieve later. One of the major difficulties introduced by current systems is their single classification approach, where a file is allowed to reside in only one folder. And although modern operating systems provided mechanisms for placing documents in more than one folder (like "Symbolic Links" in GNU/Linux), people made very little use of such features [6]. Our design permits the user to overcome the static organization issues posed by the hierarchical file systems, by introducing methods to retrieve information in a way not dependant from how they were initially organized. This lifts from the user the load of having to remember or guess were the desired documents could be.

Although in the past, an application could have had similar features built-in, developers had to implement them in their own ways, which could be too complex for many small, or even large projects. Thus, besides enhancing the user's experience in information access, GLScube provides application developers with the required tools to extend their applications with the same advanced functionality we provide to end-users.

Next, we describe the key elements that allow GLScube to solve the previously described problems in the traditional file systems.

1.2.1 Rich Information

One of the key issues with the traditional file systems is their lake of type support. The unit of data storage in these file systems is a File, which is merely a collection of bytes. These file system make no distinction between an audio file or a video file, for to them, they are all a collection of bytes. They are responsible for storing them the first time they are created, modifying them when necessary and replacing or deleting them when they are no longer needed. It is up to the operating system, application developer or end-user to interpret their meaning to them.

GLScube introduces a rich description of the user's data. It "understands" the Type of a file, and thus distinguishes between an MP3 file and a JPEG file. It extracts relevant information from each file, depending on its Type, and allows users and developers to access this information. For example, it would extract that some MP3 file is 3 minutes, 22 seconds long, its birate is 128Kbps, the name of the singers, the name of the band and more information stored in the file (in this case, in the ID3 tag). For some JPEG image, it would extract the width and height in pixels, the number of colors, and more. But of course, it is not just about extracting information, but rather, how to make this information available to the users. After all, the singer of some song has always been stored in most MP3 files, but it was not possible to do a search for the songs of a specific singer.

GLScube extracts the attributes (for example, the Creation Date or the File Size), the Metadata (for example, the duration and bitrate of an MP3 file), and the textual content (in case of text files), and uses these information as identifiers for the user's data. A user no longer has merely a file's name to look for, but rather, can use any of the information associated with the file to reach it. He can look for the MP3 files longer than 3 minutes or the JPEG images less than 600 pixels wide.

Furthermore, GLScube allows users to Tag their Documents. A Tag is a keyword which acts like a subject or category for some Document. 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. For example, a college student may Tag a set of Documents as "sheets", which he, in the scope of his usage, would interpret as solutions to the assignments required from him in college, while another user, a musician, may use the same tag, "sheets", to refer to the set of musical sheets he wrote. With Tags, the naming of a Tag is not necessarily semanticlly correct in resepct to the underlying data, and it is all up to the user to select the appropriate Tags. This may sound like reverting back to the hierarchical structure we previously criticized, which did not impose any restriction on the relation between how the user names his data, and what his data really is.

However, Tagging, as will soon unfold, is not the only method for organization. And contrary to the traditional directories, they are not monotonous. This means that an end-user could stick a "sheets" tag on one file, and then stick to it more tags, say "economy" and "export." This way, a user can have a multitude of options to reach his data. He can at any time, look for the Sheets of the Economy subject that discuss Export, or just look for all the Economy Documents, which could as well cover e-Books he tagged as related to Economy.

Moreover, GLScube provides the ability to assign relations between Documents. When working on a project, the end-user may assign a relation to two Documents he has been working on, so that later, he could retrieve all Documents related to a specific one.

1.2.2 Organization

As described, the traditional directories impose too many hurdles in organization due to their monotonous constrain, which is the combination of the path and the file name. In GLScube, we introduce Virtual Collections, which are analogous to the traditional directories with the exception that their contents are retrieved dynamically, when the user access them, and represent the state of the system at the time of access. To the user, one Virtual Collection is different from another not merely by its name, as what it is the case with directories, but rather, with a Query associated with it. A Query of one Virtual Collection may be set by the user to list all images larger than 600 pixels, all files that contain the word sheets, or all the audio files shorter than 30 seconds and are tagged as “samples”. The user does not have to specifically add any file to one, or more, Virtual Collections as it is the case with directories. When the user access a Virtual Collection, let us say that associated with the search query for the previously described subset of audio files, then at this specific moment, GLScube looks all those files he stored that match these criteria, all the audio files that are in fact shorter than 30 seconds in duration and have a tag ”samples” assigned to them.

Virtual Collections are nested into a hierarchy. This gives the users greater power and flexibility in organizing their data. For example, a Virtual Collection could be made to list all text Documents, and inside it, two Virtual Collections would be made, one to list Documents tagged as ”sheets” and the other to list Documents with 500 words or more. When any of those last two Virtual Collections are opened, the combined query of the child Virtual Collection and its parent is executed. Thus, the parent Virtual Collection will show all text files, and its two child Virtual Collections. The first child would list all text Documents that are tagged as “sheets,” and the second child Virtual Collection would list all text Documents with 500 words or more.

This kind of dynamic organization lifts the user from having to think of where he should store his data, and instead, focus on what his data means to him. If he works as a sound editor for example, he might be more interested in the duration of sample sounds and their category than their creator, and thus create virtual collections that describe this semantic view of data, for example, an example organization for this scenario would be:

```
VC [Audio Samples] : Documents tagged as ‘‘samples" of Type ‘‘Audio’’
   VC [Animals] : Documents tagged as ‘‘animals’’
   VC [Industrial] : Documents tagged as ‘‘industrial’’
   VC [Cars] : Documents tagged as "cars" or ‘‘transportation’’
   ...
```

1.2.3 Searching

It should be now clear that GLScube relies on searching as the underlying mechanism for organization; this type of Dynamicity in organization is pulled of by performing search queries against the user's data at run-time; at the time the user demands to retrieve information.

GLScube generalizes searching to not just searching for a file name, but rather, by search almost every relevant bit of information associated with each file. Doing an unconstrained search with the keyword "hello" will search for every Document whose file name, Metadata, tags, or contents contain the word "hello".

1.2.4 Developer Support

GLScube presents Developers with a Plugin-based Type System they can extend to support additional file formats they create. A Developer may easily create an XML file that describes the metadata and attributes of his file format, create an Importer that extracts the relevant information from his file format, and distribute the XML and Importer to be used by end-users.

Besides Type extensibility, Developers are provided by an API they can use to add GLScube-like functionality in their applications. Developers can use Shared Stores to organize the user's data so that they would be read in a uniform manner by other GLScube-aware applications, or use the searching functionalities in their applications. Developers can also use the Web Interface to retrieve information in XML, or use the Pseudo File System to interact with the Stores through the traditional hierarchical APIs.

1.3 Roadmap

The following is a list of the features that are yet to be implemented, or complete, in GLScube:

- Transaction Processing: Encapsulating requests in transactions and rolling back the failed ones.
- Considering migrating to a lighter Database Management System. This issue is described in Section 2.7.
- Additional Importers: Currently, the latest build of GLScube contains Importers for the following file formats: OpenOffice Calc, OpenOffice Impress, OpenOffice Writer, PDF, JPEG, WAV, MP3 and AVI. The next collection of Importers we plan to develop are for the following file formats: Microsoft Word, Microsoft Powerpoint, Microsoft Excel, BMP, PNG, QuickTime, RealAudio, RealVideo and OGG.
- Composite Documents: Composite Documents are Documents that preserve the hierarchical organization of a group of tightly-related Documents, for example, the source code of some application.
- Keyword Substitution: Maintaince of a user-editable dictionary of keywords and their possible subistitutions. For example, a search for "UN" would yield results containing "United Nations", and a search for "cv" would yield results containing "resume".
- Active Queries: A client application using the GLScube API or Web Interface can register a Live Query, and when any changes occur to the stored information that match this query, the client is notified. For example, if a client application is showing search results, and a Document in these results is deleted, the application would be notified about this change so that it would remove this specific entry from the displayed result.
- Metadata Boosting: Allow Developers to specify boosting values to specific metadata.

Chapter 2

Architecture

GLScube is implemented as a userspace file system, that augments the features of the underlying file system. This means that GLScube is not concerned with the actual storage of the bits of data on the physical disk drive, of error recovery of sectors, or the various other details of implementing file systems. GLScube is concerned however with enhancing the usability of file systems, by providing a semantic representation of the data, while not putting much concern on how the data would be stored on the disk drives.

2.1 Client-Server Model

GLScube is designed as a Client-Server model. The server is a userspace daemon that is always running, responding to requests from client applications and performing actions due to events occurring on the underlying file system.

2.1.1 Daemon

The GLScube daemon process runs when the operating system starts up and is responsible for performing all the augmented features of GLScube. It is a userspace process responsible for monitoring the changes in the files, acting according to the changes, and serving requests. All the GLScube requests are handled and controlled by the daemon. When you need to search for all the documents that start with “a”, for example, the daemon would perform all the steps to handle this request and then it sends the result to the calling process.

The GLScube daemon has two main tasks, Monitoring the underlying file system and interacting with the client processes. Monitoring the underlying file system is done with the aid of the inotify kernel module (see Chapter 4). The next sections describe the different means to communicate with the daemon. The details of the operation of the daemon and the services it provide will be clear in the next chapters. Figure 2.1 shows a general overview of the architecture of GLScube.

2.1.2 Web Interface

The Web Interface is an XML-based interface to the Documents stored in the user’s Stores. It allows for local clients to communicate with the daemon through Internet Sockets. A request is passed to the GLScube daemon in the form of XML, and after the daemon retrieves the result, it returns the response in the form of XML.

The daemon listens on a predefines port number, and accepts only local requests coming to that port. This interface is used to power the semantic view of the browser (see Section 2.5). Requests are served through a compact implementation of the HTTP protocol.

2.1.3 API

The Application Programmer Interface (API), named libglscube can be used by Developers to bring the functionality of GLScube to their applications. The API communicates with the GLScube daemon through UNIX domain sockets, serializing objects to and from XML.

We have to address why we implemented the Web Interface using UNIX Internet Sockets, and implemented the API using UNIX Domain Sockets. First of all, it is not possible to implement the Web Interface with UNIX Domain Sockets, because these require using a file descriptor. It would have been possible to implement the API using Internet Sockets, however, the performance of Internet Sockets is slower compared to Domain Sockets. Because Domain Sockets are only available on the same machine and not between networked computers, they do not

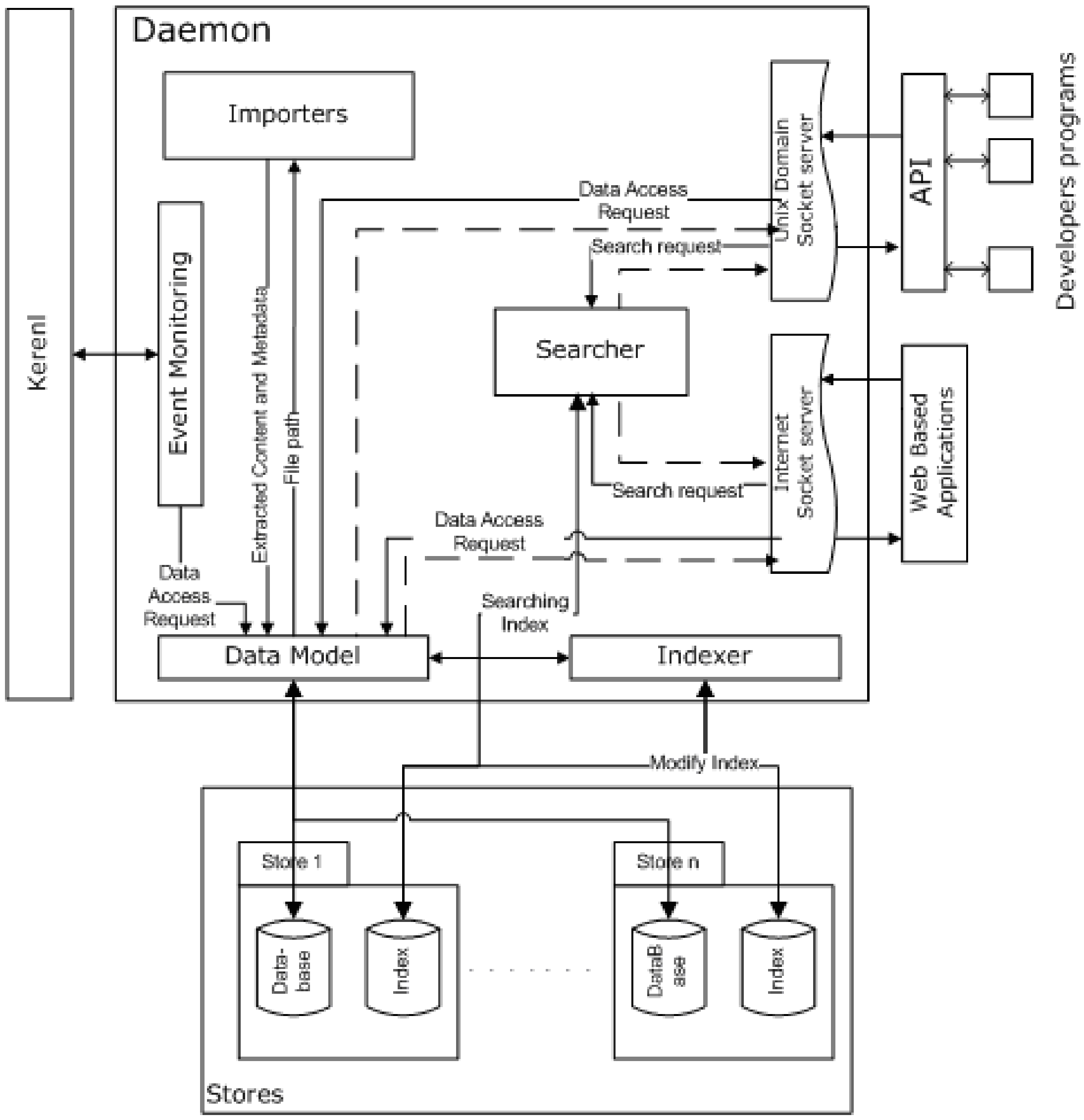

Figure 2.1: Architecture of GLScube

require checksums, addition of headers or routing calculations, which is required in case of Internet Sockets.

2.1.4 Pseudo File System

The Pesudo File System is a compatibility layer to allow applications to make use of the features of GLScube without requiring any changes to their code or even a recompilation.

Initially, the first solution approached to provide applications with the ability to browse through the semantic representation was to change the behaviour and features of the Open and Save File Dialogues in desktop environments like KDE and Gnome. However, when evaluating this approach for KDE, we found it would require patching and recompiling the kdelibs package, and then recompiling each and every KDE application so that it would link against the newly built binaries. Inconvenience does not stop here however, because unfortunately, not all applications use KDE's Open and Save dialogues.

The other solution we undertook is to create a pseudo file system that interacts with the GLScube API to represent a semantic representation of the data. In order to communicate with the userspace daemon through the API, the file system was implemented in userspace with FUSE.

FUSE consists of a kernel module and a library that communicates with the it via a special file descriptor. Using this API, file system code can be run in userspace. FUSE was officially merged into the mainstream Linux kernel tree since version 2.6.14.

Queries to the userspace file system are in the form

`/StoreName[:search]/path_or_query` or

`/:search/query`.

The initial "/" is the root of the mount point of the GLScubeFS. The former query is used to browse or search through a specific Store (see Section 3.1.2), while the later query is used to search through all Stores. Browsing a Store is like browsing through traditional file system directories, and the same tools and applications can be used against a pseudo path.

Figure 2.2 shows the path of the `ls` filesystem call on the FUSE GLScubeFS.

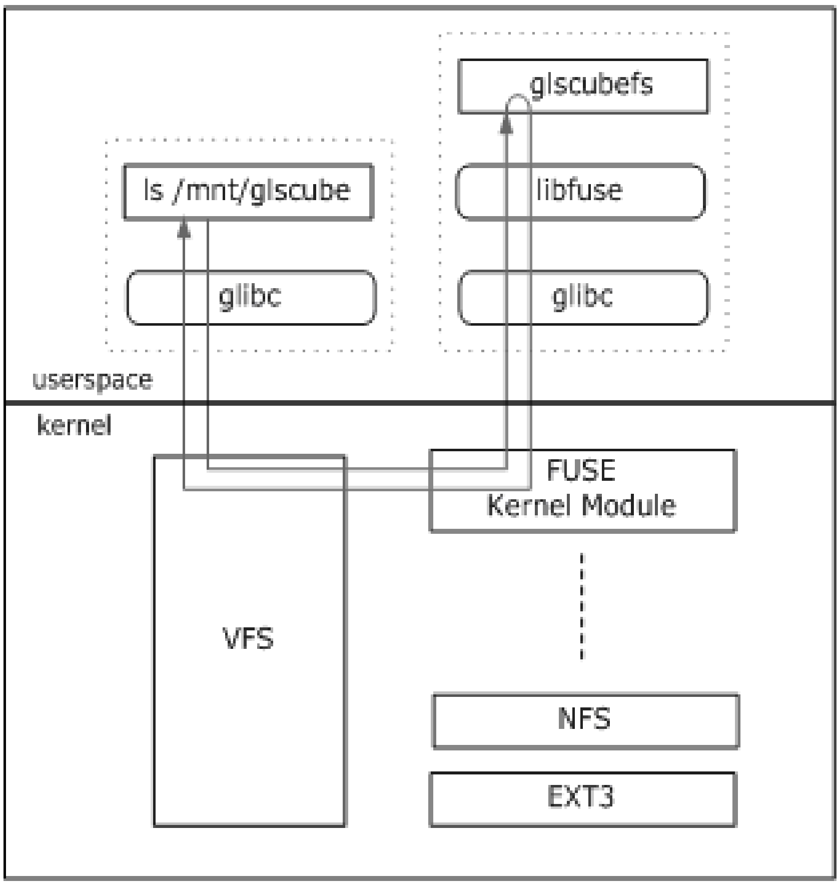

Figure 2.2: Path of a System Call on the FUSE glscubeFS

2.2 Event Monitoring

The Event Monitoring module (see Chapter 4) is the module responsible for monitoring changes occurring in the user's files and folders due to modification, creation and deletion. It is also responsible for specifying the proper action to be taken in order to maintain consistency between the actual file data and stored information about these files. For example, when a user copies a new file to his hard drive, the Event Monitoring module detects that a new file has been created, and then invokes the appropriate actions in other modules that would then extract information from the new file, and make it available to other modules.

The Event Monitoring module encapsulates the inotify kernel module, a module that can be used to notify applications with changes to watched files and directories in the mounted file systems. The inotify kernel modules allows developers to register watches for specific directories. Once registered, we refer to this directory as a watched directory, as inotify would send notifications about changes to this directory. For example, when a file is deleted from a watched directory, inotify sends a notification about this change.

It comprises of two submodules. The first submodule, Event Watcher, is responsible for reading and storing the events occurring on the underlying file system. The second submodule, Action Executor is responsible for performing the appropriate actions on the captured events.

2.3 Request Handling

The GLScube daemon is responsible for handling request arriving from applications using either either the API or Web Interface. Eitherway, requests are eventually represented as a DataAccessRequest object, which is then queued up, and later a thread is allocated to serve it.

Figure 2.3 shows how a DataAccessRequest object propagates through the system.

2.4 Type System

Each file format represents its data in its own way, and it is up to the application developer to provide an Importer to any file format he defines. Otherwise, The file is assumed to be of a Generic Type, and only basic information is extracted from the file like the Creation Date. Thus, it is up to the developer to enhance the user experience by extracting interesting data embedded within file formats he creates.

A Type is not a meaningful object on its own. It is only an abstract representation of what a Type is. The Type class is derived to a Content Type and Empty Type classes. A Content Type is the definition which describes a Content Document of some Type. It adds to the base Type class the path to the Importer library object, the extensions that this Type uses, and the Category of the Type. An Empty Type on the other hand defines a group of Empty Documents that may

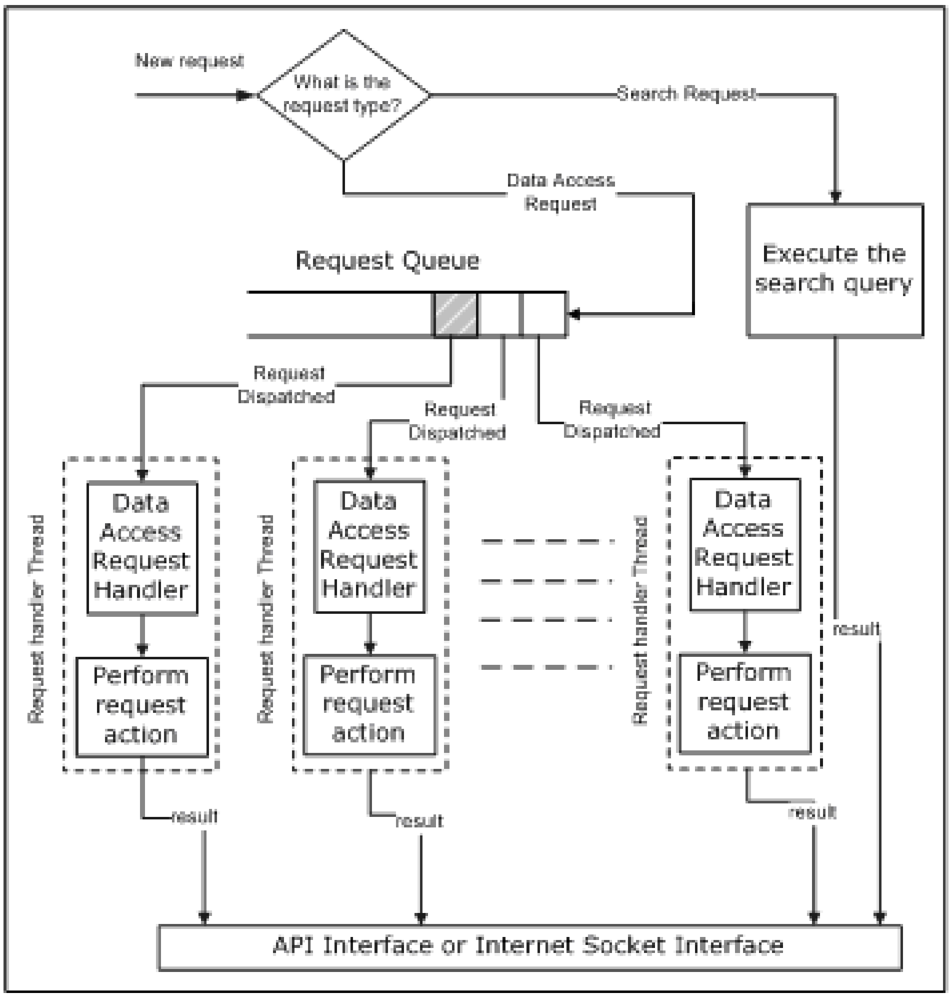

Figure 2.3: Propagation of a DataAccessRequest

be used by a Developer to store his applications' data.

The most basic unit of information in GLScube is a Document. A Document is a generalization of some type of information, it is a representation of information that has permissions for those that can access it, what other Documents it are related to it and what are its defining metadata and attributes. Documents can be either Content Documents, which are always associated with a stream of data, and are analogous to the traditional file systems' files; Documents can be Empty Documents, where each is a record of information in some domain, and are analogous to a tuple in a database table; and finally, Documents can be Virtual

Collections, which are Collections of Content Documents and Empty Documents, they are analogous to the traditional directories, but contrary to them, their contents are retrieved at the time they are opened, based on a Search Query associated with it.

Documents are stored in Stores. A Store is the logical representation of files stored in a disk's mount point. GLScube is transparent to the mount point of some partition, because all information concerning the physical storage of some file is stored relative to the root of the partition, and not the full path in some system. This comes to an advantage so that when a disk drive is unmounted from one computer and mounted in another, the stored information for the contents of this partition can be automatically integrated without needing to recreate it.

2.5 Browser

The GLScube browser is an application provided for end users to browse their Documents. It is designed with a stress on usability and how it would be easier for users to find their data. Using the browser, users can either search for their Documents and get instant results, or, they can browse through Virtual Collections to get a live view of their Documents based on their semantics.

The browser is written in HTML, CSS and Javascript. It heavily uses XML-HttpRequests (commonly known as AJAX, or Asynchronous Javascript and XML), to update the viewport without needing to refresh the entire page. A wrapper KDE application was created for the browser, which uses KHTML to render it.

For more information on the features of the browser, and how to use it, refer to the User Guide.

2.6 Unicode Support

On Unix, Linux and POSIX-type platforms, the locale environment is the set of parameters that describe the user's language, how the time is displayed, and other language and cultural rules. A specific locale specifies preferences like how a character is converted to uppercase (using the `toupper` function), how `mblen` should count a multi-byte string, whether to use a point or a decimal point or decimal comma in numbers and various other conventions.

Each of the possible preferences is mapped to an environment variable; for example, the character encoding is defined by the LC_CTYPE environment variable and the format of time and date is defined by LC_TIME. The environment variable LC_ALL, if set, acts as a subistitution for all the other variables, and only if it is not set that the other variables are looked up.

When a C/C++ program starts up, it initially uses the "C" locale by default.

As of today, many GNU/Linux distributions have switched their default locales to UTF-8, including, but not limited to, Red Hat Linux 8.0 (and higher), SUSE Linux 9.1 (and higher) and Ubuntu Linux.

GLScube uses multi-byte strings for storage of text in all of its submodules except the Indexing and Searching module, which uses wide characters due to the nature of the CLucene implementation.

Additional information on Unicode support in GNU/Linux is provided in Appendix B.

2.7 Implementation Details

GLScube is implemented in Standard C++, with much reliance on the Standard Template Library (STL). It has been compiled with the GNU Compiler Collection (GCC), and built with Automake, Autoconf and Libtools [26].

C++ does not have any prebuild support for threading. Although we could have used pthreads, we decided on using Zthread, a mult-threading C++ library that provides abstraction for the native POSIX threads implementation used in GNU/Linux.

Apache's Lucene (see Chapter 5) was used for Information Retrieval (IR). Lucene is a widely recognized IR library used in the implementation of internet search engines and local, single-site searching. After evaluating several IR libraries, we selected Lucene for its leading performance results, the detailed documentation available and its wide adoption, and thus, active development.

We used PostgreSQL as the Database Management System (DBMS) for storage of Metadata about Documents and Types. In our design phase, the Type System (see Section 3.1) was modeled as a hierarchy of inheritance. In this design, a Document type would be inherited by an Audio type, which would define such metadata as Duration and Sample Rate, and that Audio type would be inherited by an MP3 type, which would add Metadata like Album Name. The initial plan was to make use of PostgreSQL's Object Relational capabilities to model this design, however, later on this approach was dropped out. PostgreSQL was then selected for its Object Relational capabilities, which we ended up not making use of. Consequently, we are currently considering and evaluating migrating to a lighter DBMS, specifically, SQLite.

As described above, the core of GLScube is an always-running daemon process, that communicates with other client processes through either an API or an Internet Socket Interface. Information is sent to the Socket Interface as XML, and requests are received through it also through XML, hence comes the need to parse XML documents in the GLScube daemon. Additionally, To pass objects between the GLScube daemon and the API, objects are serialized to XML. Thus, the need arises again for parsing XML both at the daemon side and the client side. libxml2 was used for parsing XML documents.

Additional dependencies may be brought up by Importers. For example, in the Importers currently implemented in GLScube the PDF Importer depends on xpdf.

GLScube does not depend on any libraries specific to a desktop environment. Nearly most of the code base is independant from any desktop environment, with the only exception being the Browser. Even for that, the prototype browser was implemented in HTML, CSS and JavaScript, along with a small KDE container application that uses KHTML to render the browser files.

Part II

Design

Chapter 3

Data Model

The Data Model is the representation of the data structures and their relationships to each other. It defines what a unit of a storage is, how this unit is organized into the system and how it is related to other units in the system. The Data Model distinguishes between possible types of information, and allows for type extensibility by using a Plugin-based Type System architecture.

The Data Model supports type extensibility by the use of schema files. A new File Type is represented by two XML files: `schema.xml` defines the metadata attributes and the associated Importer, and, `text.xml` defines the textual description of each metadata attribute.

The Document is the most basic unit in GLScube. A Document is an abstract representation; it is the definition of what an entity in the system is. In an GLScube Store, each Document has a unique identifier, that together with the Store identifier, uniquely identifies the Document in the entire system.

GLScube specializes the Document structure into a Content Document, a Composite Document, an Empty Document and a Virtual Collection. These documents serve as the foundation of organizing the data in the system. Content Documents are documents that are associated with a data stream–they are the semantic representation of a file in the hierarichal file system. An Empty Document is a document that has no data source, and is only associated with a set of metadata attributes. Virtual Collections are collections of Documents whose actual contents are retrieved at run time–they are the semantic representation of the hierarchal file system's directory.

3.1 Type System

Each file format represents its data in its own way, and it is up to the application developer to provide an Importer to any file format he defines. Otherwise, The file is assumed to be of a Generic Type, and only basic information is extracted from the file like the Creation Date. Thus, it is up to the developer to enhance the user experience by extracting interesting data embedded within file formats he creates.

Each importer can be used for one more file formats, and each is associated with a schema file, named schema.xml. This file describes the attributes that the Importer fills and their types. A schema file can use from any system wide attribute, or define its own.

Custom attributes can be of a String, Integer, Float, Boolean or Date. It can be multivalued, which means that the Importer will not return one object, but an array of objects from its type. Finally, each custom attribute can be prevented from being searched for in wildcard searches, and only be searched for if it was specifically declared as a target metadata attribute in the search string, by setting the "nosearch" property to true. If omitted in the type definition, it is assumed false.

When parsing a schema's declared attributes, GLScube first checks to see if it is a predefined attribute. If not, its definition is looked for in the schema file. Thus, the precedence is for predefined types. The ordering of the declared attributes in the schema file is relevant. Applications that display file metadata will not necessarily show all the attributes, and a convention to regard the priority of attributes to decrease from top to bottom of their definition is in effect. Duplicate attribute names are not allowed.

3.1.1 Types

A Type has a name, which uniquely identifies it in the GLScube Type System, it has a Display Name, which could used by client applications to show a more readable name for the Type. It has a Description, which could be filled with

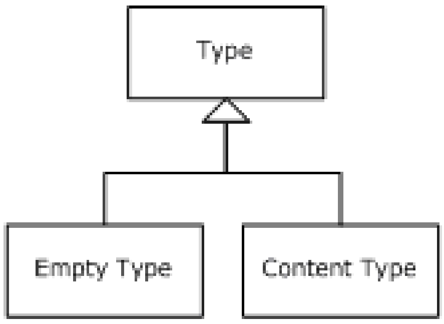

Figure 3.1: Inheritance in the Type System

information about this Type and it has icons.

A Type is not a meaningful object on its own. It is only an abstract representation of what a Type is. The Type class is derived to a Content Type and Empty Type classes, as shown in Figure 3.1, details of which will be provided next.

Content Types

A Content Type is the definition which describes a Content Document of some Type. It adds to the base Type class the path to the Importer library object, the extensions that this Type uses, and the Category of the Type.

The Category is a logical grouping of Content Types. Categories are predefined by GLScube and cannot be changed by users or developers. The provided Categories are “Audio”, “Text”, “Video”, “Images” and “Others”. A Content Type should specifify its category, and if none of the predefined categories fits its description, it should use the “Others” Category.

Empty Types

An Empty Type defines a group of Empty Documents that may be used by a Developer to store his applications’ data. Empty Types share most of the same attributes with Content Types, with the exception of not being associated with an Importer or Category.

An example Empty Type would be one that represents an Address Book. In this type, possible metadata could be “Name”, “Phone” and “E-Mail”. An Empty

Type could be regarded by Developers exactly as a database table, a table that contains records that can, or cannot be shared between applications and UNIX users on the same system, based on the permissions associated with each record, or more accurately, each Empty Document added to that Type.

3.1.2 Stores

A Store is the logical representation of files stored in a disk's mount point. GLScube is transparent to the mount point of some partition, because all information concerning the physical storage of some file is stored relative to the root of the partition, and not the full path in some system. This comes to an advantage so that when a disk drive is unmounted from one computer and mounted in another, the stored information for the contents of this partition can be automatically integrated without needing to recreate it.

3.1.3 Documents

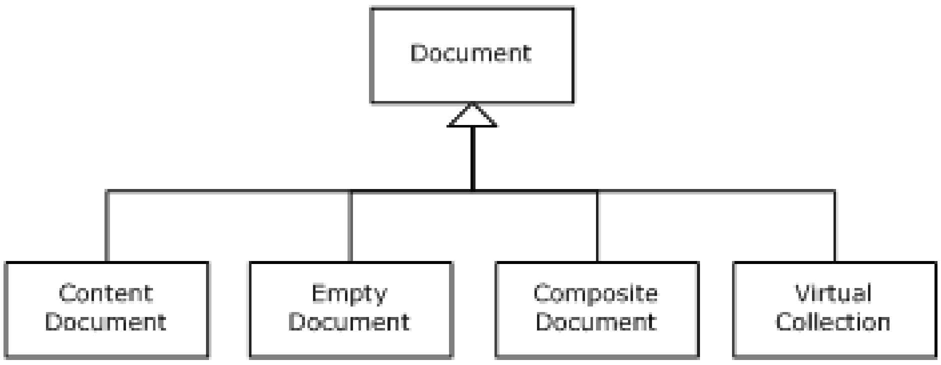

Figure 3.2: Specializations of Documents

The most basic unit of information in GLScube is a Document. A Document is a generalization of some type of information (see Figure 3.2), it is a representation of information that has permissions for those that can access it, what other Documents it are related to it and what are its defining metadata and attributes. Documents can be either Content Documents, which are always associated with a stream of data, and are analogous to the traditional file systems' files; Documents can be Empty Documents, where each is a record of information in some domain, and are analogous to a tuple in a database table; and finally, Documents can be Virtual Collections, which are Collections of Content Documents and Empty Documents, they are analogous to the traditional directories, but contrary to them, their contents are retrieved at the time they are opened, based on a Search Query associated with it.

Content Documents

Each Content Document has a Type. This type is internally used to decide which Importer to use when the Content Document has been created or updated. In either case, the appropriate Importer, if any, is selected for the given Content Document and is run to extract from it all the relevant MetaData; Tags, if any; and Content, if any. This extracted information is then stored, and indexed in the persistant storage, for later retrieval. Besides the Importers bundled with GLScube, an application developer can created custom Importers for their file formats and distribute it with their application, then they can be seamlessly registered with GLScube. For more information on Importers, how they work, and how they can be written, refer to Section 3.1.4.

Content Documents can be created, modified, and deleted either by the direct request of a client application using the API, as described in Section 2.1.3, or the Web Interface, as described in Section 2.1.2; or, through the Event Monitoring module.

Empty Documents

An Empty Document is a document that has no data source, and is only associated with a set of metadata attributes. Unlike Content Documents, Empty Documents are empty from a content stream. An Empty Document is the equivalent of a tuple in a database table.

By defining an Empty Type and registering it in GLScube, Developers can use a unified storage and interface for the users' data, and make use of GLScube's searching and organization capabilities without having to implement complicated functionality in their applications.

Table 3.1: An example Empty Document for an “AddressBook” Empty Type

	Field	Value
Attributes	User	5000
	Document ID	125987
	Store ID	2
	Group	10
	Permissions	rwx-r–r–
	Creation Date	June 12, 2006, 11:22:33
Metadata	Name	Santa Clause
	Phone	200-555-1000
	E-Mail	santa@northpole.com

An example Empty Document for a fictitious “AddressBook” Empty Type is shown in Table 3.1.

Virtual Collections

A Virtual Collection is analogous to the traditional directories, but rather than users having to explicitly organize their files into specific directories, the contents of a Virtual Collection are automatically retrieved when they are opened according to user-defined query associated with them.

Virtual Collections can be nested into a hierarchy, just like traditional directories. However, their hierarchical organization forms a semantic dependence between them. Consider a Virtual Collection made to list all text files, if a Virtual Collection was added as a child to the former one, and associated with a query to list Documents tagged as “vacation”, then this later Virtual Collection would actually list all text files marked as “vacation”.

In other words, the query of some Virtual Collection as formed as the combination of all queries of its parent Virtual Collections up to the root of the Store.

Each Virtual Collection has associated with it a set of Tags, and a search query, and at least one of the two must be set. A Virtual Collection with some Tags restricts its content to the Documents that these specific Tags are applied to. A Virtual Collection with a query will only list the Documents that match this

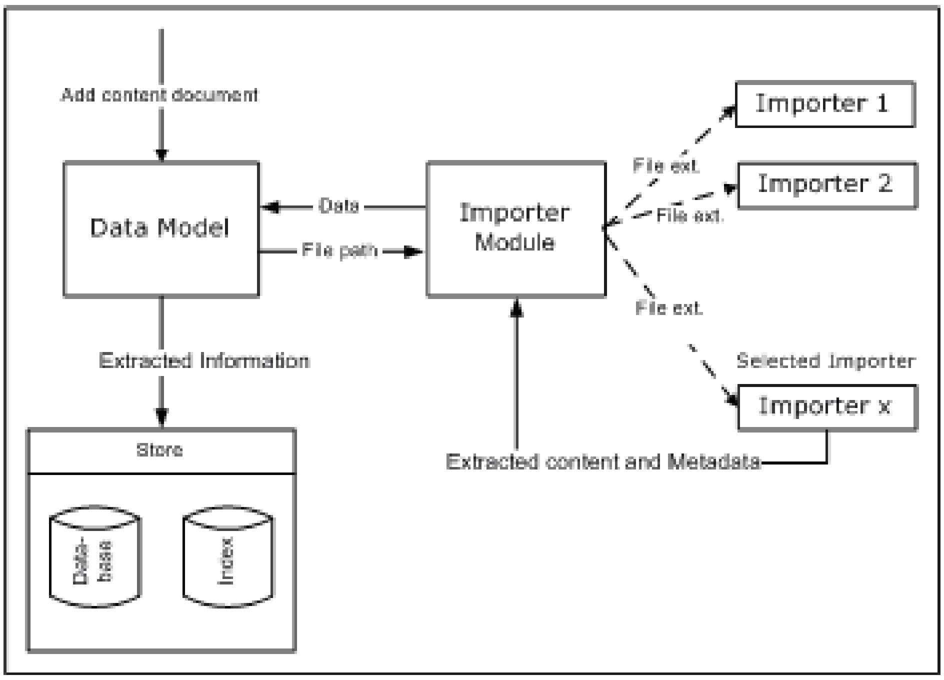

Figure 3.3: Design diagram of the Importers submodule

query.

Composite Documents

One problem not yet approached in the previous discussion of the Type System is how to collect together a group of Documents tightly related, like the source code of some application. In this case, it is necessary to maintain the hierarchical organization, because the location of each Document in this hierarchy is part of its semantics. Composite Documents preserve the hierarchical organization of a group of Documents.

Composite Documents have not yet been fully implemented in the first prototype.

3.1.4 Importers

To GLScube, a Content Document is of a specific Type, namely a Content Type. A Content Type defines such properties as what its unique name is, what its

Table 3.2: Information Extracted by an Importer

Information	Description
Metadata	An Importer should extract from a file any relevant metadata that could be latter used to identify it. As an example, an Importer for MP3 audio files should extract, among other metadata entries, the duration of the audio file in seconds.
Tags	Although users are encouraged to Tag their Documents, it is also possible that Importers would tag extract Tags from a file. This kind of flexibility is allowed because some file formats can store Tags inside their headers.
Content	If the file format contains textual data, for example, as is the case with plain text or PDF files, the Importer should extract such text. However, as with Tags, this optional and can always be left as an empty string by the Importer.

metadata is, its tags, and other attributes. Each Content Type also has a m-1 mapping to a file format Importer, which is provided with the Type when it is registered with the runtime system. An Importer is a plugin provided by the file format developer to extract from any file its defining attributes and metadata. This information is then sent back to the GLScube runtime for storage and indexing. By separating file format support from the core of GLScube, the system becomes more easily extensible to additional file formats. Table 3.2 shows the types of data extracted by an Importer from a file.

An Importer is a compiled C++ shared object, with a predefined Interface as outlined in Importer.hpp. A developer that desires adding support for a new file format is required to create a new Importer object and an XML file describing it. The GLScube daemon loads all the registered Importers at initialization, and can further load any new Importer dynamically while it is running. This adds the convenience of extensibility while not requiring a restart. Similarly, Importers can be unregistered from the system while running, with no side effects on the running processes or queued and unhandled requests.

GLScube does not treat any file in any way different from the other, and does not inherently understand or interpret any file format, but rather employs the described plugin architecture which allows the Type System to be expanded to understand any file format.

When a Type is registered with the GLScube runtime, a copy of the object file and the XML file of that Importer are copied to the GLScube installation directory, and additional copies are made, one for each mounted Store. This redundancy is necessary, so that when later, a disk drive with a GLScube Store is unmounted and added to another system with GLScube installed, all the foreign types to that system could be imported to it, if desired.

An important issue related to Importing is how a file's format is identified. The approach taken by GLScube is to select the appropriate Importer based on the file's extension. However, this is not perfectly reliable, and a mechanism to select the Importer after interpreting the header would be more accurate, however, this would incur a huge performance cost.

To create an Importer, a developer has to derive the Importer class, defined in Importer.hpp, and provide a getInstance C function that returns a new object of his derived class. getInstance is the first function looked for by the GLScube runtime in the object file when an Importer is loaded.

For additional information on how to create an Importer for a file type that is not already supported by GLScube, refer to Importer.hpp and the source code of the accompanying Importers.

3.2 Persistent Storage

GLScube is designed to allow seamless addition and removal of Stores. All the persistant data saved by GLScube are stored on a Store-by-Store basis. There are no shared data and all the data saved on a Store are not dependant on where the disk partition is mounted. This is advantegous for two reasons: first, it provides a degree of fault tolerance because an errornous Store will not render the other Stores inaccessible, and second, it allows for removal of a disk drive from one computer and using it to in any other computer with GLScube installed, where it will be instantly identified.

To achieve this seamlessness, all data associated with one Store is saved on the Store's physical mount point. This data includes all the configuration files; information about the Types, Attributes, and Watched directories; object files of Importers; a database and a search index.

It would have been quite simpler if we assumed that all Stores shared on one computer can only be accessible to that computer. But then this would have been utterly inconvenient. To avoid this, we have to maintain consistency between the redundant files in one system. That is, maintain that not any of the redundant files may at any time be different, so that they are always identical. Furthermore, when a disk drive with a GLScube Store on it is unplugged from one computer and connected to another, it must operate correctly. This requires that all the Types and Attributes defined on the foreign installation be imported to the new installation; all the Importers be installed; and that all references to file paths be relative to the root of the disk drive, not where it is mounted.

The last point requires some elaboration. Consider for example an ext2 partition of a disk drive that is mounted on /home/user1/, and then a Content Document has been created with today's ToDo list. For this Content Document, a file is automatically created by GLScube and stored in /home/user1/120000.txt. If the reference to this file was saved as /home/user1/120000.txt in the database, this means that if this partition was later mounted under /home/user2/ it would not be accessible. For this reason, only the relative path to a file is saved, in this case /120000.txt, so that the full path can always be reconstructed wherever the parent partition is mounted.

3.3 Concurrency

A queuing model which allows for one request to be fulfilled at any time is easier to design, however, with some operations prone to long execution times (for example, Importing large text file from a very slow disk drive), the system would have starved completing one task while much more requests get queued. Thus, GLScube is designed to allow concurrent execution of operations. Using the C++ Multithreading library Zthreads, several of the GLScube modules use multiple threads to service many requests simultaneously.

Yet, concurrency imposes many problems, mainly due to the shared data structures. Consider two threads that acquire connections from the Connection Pool, and that the Pool currently has only one connection. Assume one thread starts the acquiring process before the other, and that the later thread enters execution exactly after the former thread has verified there are still enough connections in the pool. Now, the later thread may find there are still enough connections in the Pool, because the early thread has not yet updated it, and thus, we have two threads aware that they can acquire connections, even though there is only one connection available.

To solve this problem, we can either stuff the code with error checking that may still be suspect to failure, or we could use a Mutual Exclusion locking mechanism. With locking, we allow a shared resource to be accessed only once at the same time. This works fine in the Pool example, but other resources are more complex to manage.

Another example is the types data structure, which holds the Types registered with the system. Assume there is a thread handling a request to add a new Content Document, and after this thread verifies that the Type of the new Content Document exists, another thread instantanously removes this Type, then, the behaviour of the procedure adding the Content Document is undefined, since it is assuming that a removed Type still exists.

Chapter 4

Event Monitoring

GLScube is implemented as an augmented file system that stores and indexes information about the users files - information like the attributes, metadata or contents of text files[1]. This stored information must be kept consistent with the actual contents of the files on the hard disk; for example, when the user modifies a file, this change should trigger an equivalent change in the stored information about this specific file.

Various approaches for detecting changes in the user's files and folders were discussed during the design stage. The first approach was to build a custom file polling scheme that regularly iterates over all files and folders the user has and compare the current contents and metadata of each file or folder with the last stored contents and metadata for this file or folder, and then update the stored information if a change was detected. This approach will maintain consistency between the actual data, and the stored information about the files. However, its efficiency is the worst possible, especially with the increase in the number of files.

The second approach is a simple modification for the first approach by restricting the test during the iteration over all files to only for files whose size changed, based on the fact that changes in the contents of the file will probably cause an increase or decrease in the file size. Although this solution decreases the overhead presented in the first approach, it doesn't guarantee the required consistency. For

[1]From here on, we would refer to information that represents the semantics of a file as Stored Information. It is irrelevant to the Event Monitoring module how this information is represented, the specifics of this are discussed in other parts of this Documentation)

example, consider that a user replaces the word ”meat” by the word ”beat” in one of his files. Even though this is a change in the contents of the file, it doesn’t cause any change in the file size since no more characters were added or removed, and thus, this approach will not detect such change.

Another approach is to work with an operating system whose kernel can generate notifications due to changes in files and folder. New versions of the Linux kernel support this feature. This approach is the most applicable among the three discussed approaches since it eliminates the overhead of iterating over all files, and in the same time it guarantees that a notification will be generated for every change. We chose this approach for its high reliability and applicability.

The Event Monitoring module is the module responsible for monitoring changes occurring in the user’s files and folders due to modification, creation and deletion. It is also responsible for specifying the proper action to be taken in order to maintain consistency between the actual file data and stored information about these files. For example, when a user copies a new file to his hard drive, the Event Monitoring module detects that a new file has been created, and then invokes the appropriate actions in other modules that would then extract information from the new file, and make it available to other modules.

4.1 Kernel Notifications

A solution for monitoring ordinary file system events called dnotify was introduced in 2002 by Oskar Liljeblad, a simple program that makes it possible to execute a command every time the contents of a specific directory changes. dnotify was written entirely in C, and uses the Linux kernel directory notification feature to monitor directories. Because of this, dnotify does not need to poll files every few seconds; all time is spent sleeping until an event occurs. However, dnotify suffers from several shortcomings such as the requirement for opening a file descriptor for each watched directory, which may possibly exceed the maximum allowed file descriptors per process; and that it is directory-based, which means that one only learns about changes to directories. Although a change to a file in a directory triggers a notification on its parent directory, if watched, this means one is forced to keep a cache of directory contents in order to distinguish which file exactly changed. Additionally, dnotify file descriptors pin the directories, thus disallowing the backing device to be unmounted, which causes problems in scenarios involving removable media.

A newer solution to monitoring file system events is inotify, an inode-based solution for monitoring ordinary file system events which does not require opening millions of file descriptors. With inotify, file descriptors are opened for device nodes, where each device could be used to watch up to 8000 directories or files. inotify solves the previously mentioned issues of dnotify. inotify has an event that is triggered if one is watching a file or directory on a file system that is unmounted, and, when watching a directory, inotify sends events to its child directories and files without having to register watches for them individually.

4.2 Methodology of Monitoring

The Event Monitoring module encapsulates inotify. It represents an inotify device as a DeviceDescriptor object, which describes an internal index number, the device file descriptor used by inotify and the number of watches assigned to this device. The Event Monitoring module keeps track of the initialized devices in the system under a restriction that the maximum number of devices is 256, which comes from a restriction in inotify on the maximum number of device nodes.

4.2.1 Watches

The inotify kernel modules allows developers to register watches for specific directories. Once registered, we refer to this directory as a watched directory, as inotify would send notifications about changes to this directory. For example, when a file is deleted from a watched directory, inotify sends a notification about this change. It should be noted however that inotify watches have a depth of one level in a directory hierarchy, which means that it detects events on a watched directory, and events on the children of this directory, but not on deeper children.

As an example, consider we have a watched directory called ”parent”. This directory contains a subdirectory called ”child” which contains a file called ”leaf”. Any event on the file ”leaf” will not be detected by the watch on the directory

Table 4.1: A Typical Sequence of inotify Events

OPEN (file) FILENAME = test
ACCESS (file) FILENAME = test
CLOSE (file) FILENAME = test
DELETE (file) FILENAME = test
OPEN (file) FILENAME = test
CREATE (file) FILENAME = est
OPEN (file) FILENAME = test
ACCESS (file) FILENAME = test
MODIFY (file) FILENAME = test
CLOSE (dir) 0x40000010
MODIFY (file) FILENAME = test
OPEN (file) FILENAME = test
MODIFY (file) FILENAME = test
CLOSE (file) FILENAME = test
OPEN (file) FILENAME = test
ACCESS (file) FILENAME = test
CLOSE (file) FILENAME = test
OPEN (dir) 0x40000020
OPEN (dir) 0x40000020
CLOSE (dir) 0x40000010

"parent". To solve this problem, we need to add a watch on the subfolder "child", or more generally, we need to add a watch for all directories and subdirectories. To eliminate the overhead of reading the hierarchy of all directories in the disk at initialization, we store absolute paths of all directories to be watched in an XML file that will be loaded at initialization.

4.2.2 Devices

When a new request arrives to create a watch, whether that request is at initialization or when a new directory is created, the Event Monitoring module selects the latest used Device, and adds the new watch to this device unless the maximum number of watches assigned to this device has been reached, and in such a case, a non-full device is searched. If no devices were found with empty slots for new watches, a new Device is created. Note that a device may not be full due to the release of a previously created watch descriptor, as in when a directory is deleted. By this methodology we guarantee that the minimum number of devices is in use.

4.2.3 File System Events

The Event Monitoring module can detect, by encapsulating the inotify kernel module, various file system events like creation, modification, deletion, and opening of files, creation and deletion of directories, and several other file system events. A typical sequence of events due to editing the contents of a file is shown in Listing 4.1.

From the listing in Table 4.1 we can notice that a simple action like writing a few words to a file and saving it can result in the generation of many events by i-notify. As we described above, inotify can only queue up to 256 events per device. Unless these events are read from the FIFO fast enough, events may be lost due to newer ones replacing older ones, and thus, inconsistency will be inevitable.

A solution to this problem will be discussed later.

4.3 Design

The Event Monitoring module comprises of two submodules. The first submodule, Event Watcher, is responsible for reading and storing the events reported by inotify. The second submodule, Action Executor is responsible for performing the appropriate actions on the captured events. Figure 4.1 shows an overview of the design of the Event Monitoring module.

4.3.1 Event Watcher Thread(s)

As shown in Figure 4.1, Event Watcher is a thread that is responsible for continuous reading and flushing of the FIFOs used in the communication between inotify and the kernel, and store the read events in an Action Container (see Section 4.3.3). Events from inotify arrive containing the filename, ID of the inotify device node and the ID of the inotify watch.

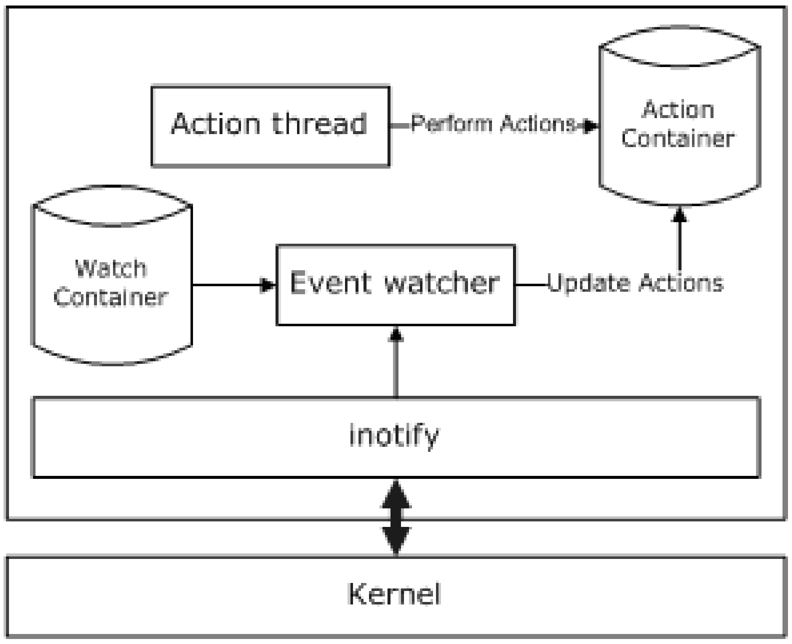

Figure 4.1: Design of the Event Monitoring module

However, and as shown is Listing 4.1, numerous events may need to be processed for a single file, and information about earlier events must be stored. In other words, the behavior that should be performed based on the arrival of some event cannot be totally decided without knowledge of previous events.

In order to decrease storage requirements, any new event arriving for some file or directory is merged with a previous event recorded for that same file or directory, if any. Such merger is based on Significance Levels assigned for file system events. The level of significance to each event was assigned based on the fact that actions due to an event may be overridden by an action due to another event. As an example, suppose we have a modification event on a file leaf shortly followed by a delete event on the same file. This means that, optimally, we do not have to take any action to the modification event, because whether or not we do, the file would be deleted anyway and this later event is all that matters. Significance Levels of various events were assigned according to Table 4.2.

Each event is associated with a predefined delay value that indicates the amount of time that must be waited before the corresponding action for this event would be invoked. Less significant events were assigned larger delay values, during there

Table 4.2: Significance Levels for inotify events

File System Event	Level
Create Directory	7
Delete Directory	6
Delete File or Move From	5
Create File or Move To	4
Close a File After Writing	3
Modify File Attributes	2
Modify a File	1

is a high probability that a higher significant event will occur and replace the less significant one.

The Action Container is a container used for storage of information describing events on files and directories, and when a new event is added, updates the current ones, replaces them or removes them. The specific action taken by Action Container depends on the Significance Levels scheme, which will be discussed in Section 4.3.1.

As we described above, an inotify event contains information about the name of the file or directory, the ID of the watch, and the ID of the device holding that watch. It is important to note that inotify passes the name of the file or directory, and not the full path. To augment this lack of information, the Watch Container is used to store the full paths to directories, indexed by their device descriptors and watch descriptors. Watch Container will be later described in Section 4.3.3.

Multithreaded Design

As we shortly described above, and which will be detailed in Section 4.3.1, processing inotify events as soon as they are queued would cause a huge performance overhead, because simply, many of them could be cancelled out or ignored. Thus, we need to implement a queuing model where inotify are continuously read and queued, and regularly, these stored events are analyzed and filtered. Hence comes the need for Action Executor to be a standalone thread separate from Event Watchers.

In the Event Monitoring module, each inotify device node is read by a separate thread. A straightforward approach would be to allocate one Event Watcher thread that would sequentially iterate over all used device nodes, and read their buffers.

To keep the number of devices, and hence threads, to a minimum, each time we create a new watch we try to find a device among the created devices that the maximum number of assignable watches have not been met for it. Otherwise, if we already assigned the maximum number of watches for all devices, we create a new device and assign the new watch to it.

Levels of Significance

Creation of a new Directory must be immediately handled, without any imposed delays, in order to achieve maximum possible consistency, and thus is given the highest Significance Level. However, there is time duration between the actual creation of a directory and the Event Monitoring modules creation of an inotify watch for this directory. This means that during this duration, one or more files may be created, and when the watch is created, there would no way to tell that these files exist. Thus, once the watch for the directory is created, its contents are retrieved to make sure that any events that might have occurred in the mentioned time span were missed.

Similarly, deletion of a directory is more significant than any other file system events, since if we delete a directory, there is no need to keep any information about the previous events occurring on its contents. In other words, the directory was deleted, so there is no need to perform queued actions on files subdirectories residing in it.

Deleting a file is the most significant file related event since deleting a modified file or a newly created file should not require adding or updating the store information about this file. This case is common with temporary files.

A file Creation event cannot be replaced by any event other than Delete, because it is irrelevant what other operation arrived if the file has not yet been processed and added to a GLScube Store. If however a Create event arrived for a file that had a Delete event queued, what might happen in case of temporary files created by applications, then this Creation event is replaced with an Update event.

A Close on Write events signals the end of modification to a file, and hence it is required to update the stored information for this file. A Close on Write also overrides an Attrib event, which signals a change in the attributes of a file like its access permissions.

The Modify event is sent by inotify each time a block of data is written to the disk drive, whose frequency of occurrence is very high. Consequently, it would be better not to monitor the Modify events, resorting only to Close on Write events to detect a change in a file's contents. However, when a file is replaced, inotify sends a sequence by Modify events followed by not a Close on Write event, but rather, Close on No Write event, which is used elsewhere to signal closing a file without making changes to its contents. Hence, monitoring the Modify event is required, so that if it was followed by a Close on No Write event, the corresponding GLScube Content Document would be updated.

4.3.2 Action Executor Thread

As mentioned above, every event has a time delay after which it must be executed. The value of this time delay varies according to the Significance Level of the event based on the significance scheme mentioned above. The Action Executor Thread is the second standalone thread in the submodule in the Event Monitoring module, and is responsible for filtering the actions stored in the Action Container to keep only the relevant events. For example, if a file is opened and no modification occurred to the contents or the attributes of the file, there is no need to care about the resulting sequence of events, because it does not affect the system consistency. In other words, no change has been made to the data in a way that should cause an equivalent change in the stored information.

Besides filtering events and keeping and keeping only the relevant ones, the Action Executor dispatches requests to the Data Model (see Chapter 3) for the events whose timeout value has passed and there are no pending events to cancel them out.

Table 4.3 shows file system events and the corresponding actions executed by the Action Executor Thread. There are two events that are handled internally

Table 4.3: File system events and the corresponding actions executed by the Action Executor Thread

File System Event	Level
Delete file or Move From	Delete the corresponding file attributes stored in the database and modify the index files
Create file or Move To	Indexing the contents of the file and add its attributes in the database
Close a file after writing	Re-indexing the contents of the file
Changing a file Attribute	Update the file attributes in the database
Modifying a file	Wait till a close on write event occurred

without dispatching any requests to the Data Model. These are the creation of directories, in which case a watch is created for the new directory; and deletion of a directory, in which case its associated watch is deleted. In both cases, corresponding XML files that store information about the currently watched directories are updated.

4.3.3 Data Structures

The two main data containers used in the Event Monitoring module are the Watch Container and the Action container. We described the role of each one of them briefly without any indication to the internal structure of each container. In this section we are going to describe the internal structure of each container.

Watch Container

As described above, the role of the Watch Container is to keep track of the absolute paths of the watched directories in order to solve the problem of file identification. For example, suppose we have two watches on directories X and Y. Under each of these two directories there is a file called Z. Now, suppose inotify reported that a delete event occurred on the Z file in the X directory, the received information can be represented as:

<Device Descriptor, Watch Descriptor 1, "/Z", Delete>

Where the watch descriptor is an integer used by inotify to distinguish between various watches. The issue here is how one could identify whether the event occurred on the file represented by the absolute path "/X/Z" or by the absolute path "/Y/Z". In order to solve this problem, Watch Container stores the absolute path of each watched directory, indexed by the device descriptor and the watch descriptor.

Action Container

The Action Container is used to store a single event for each file, for which some event has been invoked by the user. The event stored by the Action Container is the most significant event (see Section 4.3.1) that occurred on this file during the current window, which started right after the last most significant event was executed for this file, if any.

When any of the Event Watcher threads reads an event from inotify, it inserts it in the singleton Action Container, and the Action Container's insertion procedure determines which event will be store, according to the one already stored, if any.

Chapter 5

Indexing and Searching

In this chapter, we will present the available approaches to implementing searching, what are the design constraints and an overview of our design. Next, we will outline the possible approaches for searching the metadata and full-text of files:

The first, most straightforward approach to implementing searching would be in a way like how it has long been implemented for traditional file systems, in most operating system linearly iterating over all files and searching each of them at the time of request, with no notion of memory between different searches; each search is completely independent on the previous and future searches. This approach is quite slow, specifically in case of full-text search.

The second approach would be to pre-collect the semantics of all files, Store them in a database, and searching against this information. The problem with this approach is that Database Management Systems (DBMS) are not designed for searching, are not designed for answering a question like "where is Alexandria," but rather, "what are the cities of Egypt." Hence, using a database would require writing a layer on top of the DBMS to provide the required searching interface.

The third approach would be using an Information Retrieval Library for indexing, and searching the data.

Our approach of choice was using both a DBMS and an Information Retrieval Library. We use a database for storage of the semantics about Documents, Types and Stores, and only use an Information Retrieval Library for indexing and searching a subset of these semantics.

In our search for an Information Retrieval library, we selected the following as our required minimum of features:

- Support for full-text search,
- boolean queries,
- addition, deletion and update capabilities,
- high performance and scalability for large data,
- small storage space compared to actual, non-indexed data,
- Keyword Substitution, for example, UN is expanded to United Nations,
- and Unicode support.

Also, as additional overhead would be added to the system, the following constraints must not be broken:

- Search is consistent with real system data,
- and any processing overhead must not be significant.

5.1 Information Retrieval

Information Retrieval (IR) is the art and science of searching for information in documents, searching for documents themselves, searching for metadata which describe documents, or searching within databases, whether relational stand-alone databases or hypertext networked databases such as the Internet or intranets, for text, sound, images or data.

In the context of GLScube, we use the term Information Retrieval to describe searching information, namely metadata and text content, that has previously been indexed by the IR library, and also for retrieving information from the database.

In our search for an IR library, the following were our candidates:

- Lucene, from the Apache Jakarta project,

Table 5.1: Comparison of Information Retrieval libraries

Feature	Lucene	Xapian	Zebra
Wildcards (e.g. fishe*, m?re)	Yes	No	Yes
Range operators	Yes	No	Yes
Fuzzy searching	Yes	No	Yes
Term boosting	Yes	No	No
Arbitrary fields	Yes	No	Yes
Stemming Search	Yes	Yes	No
Thesaurus expansion	No	No	No

- Xapian, based on Muscat,
- Zebra, GPL structured text/XML/MARC Boolean search IR engine.

Table 5.1 shows a comparison between the previous IR libraries. More detailed comparison is in Martin Haye's comparison [11].

Lucene was initially implemented in Java, but since then, ports for different languages have been created. CLucene is the name of the C++ port.

Performance benchmarks show that Lucene is considerably faster than Xapian and Zebra in searching. As for indexing, there is a slight difference between Lucene and Zebra.

After evaluating the performance results, the detailed documentation available for Lucene, its wide adoption, we selected CLucene as the IR library to use.

5.1.1 Lucene

Lucene is a free open source, information retrieval API originally implemented in Java by Doug Cutting. It is supported by the Apache Software Foundation and is released under the Apache Software License. Lucene has been ported to other programming languages including Perl, C#, C++, and PHP.

While suitable for any application which requires full text indexing and searching capability, Lucene has been widely recognized for its utility in the implementation of internet search engines and local, single-site searching. This has occasionally led to the misperception that Lucene is itself a search engine with built-in crawling and HTML parsing functionality. Instead, any such application utilizing Lucene would have to provide this functionality independently.

At the core of Lucene's logical architecture is a notion of a document containing fields of text. This flexibility allows Lucene's API to be agnostic of file format. Text from PDFs, HTML, Microsoft Word documents, as well as many others can all be indexed so long as their textual information can be extracted.

Lucene is a highly scalable, high-performance IR library. It has a small memory footprint, only 1MB heap size, it index size is roughly 20-30% the size of the indexed text, it ranks search results, supports several query types like wildcard queries and proximity queries, it has field searching, date-range searching, the ability the search multiple indexes and getting a merged result, and it allows for simultaneous update and searching).

5.2 Choosing an Information Retrieval Library

5.2.1 How Lucene Works

Also using Lucene is simple yet the indexing of a document undergoes some steps. The data should be text only, so we need to extract text content from files before indexing, which is responsibility of Importer module. So now we will assume that data is already in textual format.

Analysis

Lucene first analyzes the data to make it more suitable for indexing. To do so, it splits the textual data into chunks, or tokens, and performs a number of optional operations on them. For instance, the tokens could be lowercased before indexing, to make searches case-insensitive. Typically its also desirable to remove all frequent but meaningless tokens from the input, such as stop words (a, an, the, in, on, and so on) in English text. Similarly, its common to analyze input tokens and reduce them to their roots.

Indexing

After the input has been analyzed, Lucene stores it in an inverted index data structure. This data structure makes efficient use of disk space while allowing quick keyword lookups. What makes this structure inverted is that it uses tokens extracted from input documents as lookup keys instead of treating documents as the central entities. In other words, instead of trying to answer the question "what words are contained in this document?" this structure is optimized for providing quick answers to "which documents contain word X?".

5.2.2 Data organization in Lucene

Lucene deals with data in the form of documents. Each document contains fields - each has a name and one or more values. Fields could be marked for indexing, storing or both. Storing means that the original content that was indexed is Stored. Since we are only interested in searching, we do not Store the contents of the metadata or full-text. Each document has a unique internal ID, which could change due to index changes.

Queries retrieve documents form the index, were only the Stored fields are accessible through the search result. The simplest query is called Term query which is in the form "Field = Value." More complex queries could be built by combining different Term queries with Boolean operators.

Lucene deals with numbers as strings so to support range queries on numbers, numbers should have fixed number of digits zero padded from the left.

Adding a new document to an index is possible using an IndexWriter, addition has small cost as it doesnt change the index files, instead it appends to it and possibly few minor entry changes. Even more, bulk addition has an added performance benefit, for documents are indexed in memory and flushed once a preconfigured document counter is reached or the IndexWriter is closed.

Document deletion is possible using an IndexReader and the knowledge of its internal ID. Since it is dynamic, the internal ID should be retrieved by searching. The problem is that IndexReader, unlike its name, modifies the index to remove a document, and causes significant performance overhead in comparison to addition. Here as well, deleting in batches could save a lot of the cost by reducing

Input/Output.

Updating a document is not implemented as a standalone operation in Lucene, so we need to split this operation into deletion and addition. The cost of batch updates could be very high as we would not be able to benefit form batch deletes or additions in a straightforward way.

5.3 Design Constraints

Again, here are our requirements of an IR library:

- Support for full-text search,
- boolean queries,
- addition, deletion and update capabilities,
- high performance and scalability for large data,
- small storage space compared to actual, non-indexed data,
- Keyword Substitution, for example, UN is expanded to United Nations,
- and Unicode support,
- Search is consistent with real system data,
- and any processing overhead must not be significant.

The first five requirements are supported by Lucene. For Unicode support, we would have to convert data to wide characters, as used by Lucene, and not by other modules of GLScube.

The main problem is consistency and performance, with both going the opposite ways. Here we are not considering search performance, but system performance due the overhead of index manipulation.

5.4 Design

To achieve the desired requirements we see that the Indexing and Searching module needs to provide the following functionality:

- Creating a new index for new Stores,
- adding Documents,
- deleting Documents,
- updating Documents,
- and searching.

The class methods of the Indexing and Searching module are called through the Data Model, and since the Data Model could be executing several threads simultaneously, and thus, may eventually make concurrent requests to the Indexing and Searching Module. The most straightforward approach to this situation would be to carry out an operation like indexing, in the calling thread of the Data Model, however, this could cause several performance penalties as we will show next, because the pool of Data Model threads could eventually end up with each of them indexing a large piece of data from the same I/O device, this would lead to a long execution time and further, possibly simpler requests would be blocked till the active requests are indexed.

Since Indexing is expected to be a bottleneck to system response time, we isolate its execution path from Searching, which we leave to be executed in the calling thread. Indexing requests on the other hands are queued, and a single Indexer thread is responsible for performing the queued actions. This approach would also eliminate index mutual exclusion problems with Lucene, which is the restriction of what concurrent operation may be executed on one index.

Table 5.2 shows Lucenes valid simultaneous action combinations. Note that update is a combination of an addition and deletion actions.

Although searching could be executed regardless of the current index state, the active action may not affect the search result causing a minor inconsistency, which could be compromised at client side by reapplying the search.

Table 5.2: Lucenes valid simultaneous action combinations

Action	Add	Delete	Search
Add	Valid	Invalid	Valid
Delete	Invalid	Invalid	Valid
Search	Valid	Valid	Valid

Before designing the Indexing and Searching module we need to decide the mapping of Data Model Documents to Lucene documents. Each Data Model document has an ID, Type, Store ID, Metadata, Tags, possibly Text Content, and a Document Type; whether it be a Content Document, an Empty Document, a Virtual Collection or a Composite Document. Search queries could be applied on any of these fields. But the search results should only contain the Document ID, Store ID, and Document Type, which are enough to identify a document uniquely. Hence, these three fields are Stored and indexed, while other fields are indexed but not Stored.

One of the important indexing parameters that could affect both searching and indexing is the Analyzer that will be used for indexing. Lucene implements four analyzers each with different behavior. These are:

- White Space Analyzer
  - Removes white spaces and punctuation marks.
- Simple Analyzer
  - Removes white spaces and punctuation marks.
  - Converts string to lowercase.
  - Removes symbols like (@, +, -) from the string.
- Stop Analyzer
  - Removes white spaces and punctuation marks.
  - Converts string to lowercase.
  - Removes symbols like (@, +, -) from the string.

  - Removes stop words (the, that, a , of ,).

- Standard Analyzer
  - Removes white spaces and punctuation marks.
  - Converts string to lowercase.
  - Removes stop words (the, that, a , of ,).

The following example shows the difference. Consider these two sentences:
`‘‘The quick brown fox jumped over the lazy dogs’’ ‘‘XY&Z Corporation - xyz@example.com’’`
The output for the analyzer will be as follows:

- White Space Analyzer

  The [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

  XY&Z [Corporation] [-] [xyz@example.com]

- Simple Analyzer

  the [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

  xy [z] [corporation] [xyz] [example] [com]

- Stop Analyzer

  quick [brown] [fox] [jumped] [over] [lazy] [dogs]

  xy [z] [corporation] [xyz] [example] [com]

- Standard Analyzer

  quick [brown] [fox] [jumped] [over] [lazy] [dogs]

  xy&z [corporation] [xyz@example.com]

We selected the Standard Analyzer as the most suitable of the four. It will generate a case-insensitive index, remove stop words and keep symbols (searching for an e-mail wont be possible using Simple or Stop Analyzer). Using this analyzer in indexing means that the queries should also be analyzed using same Analyzer to have perfect matching of results (query for The Matrix should be Matrix. Since the was removed from index by the Analyzer, it must also be removed from the query).

5.4.1 Indexing

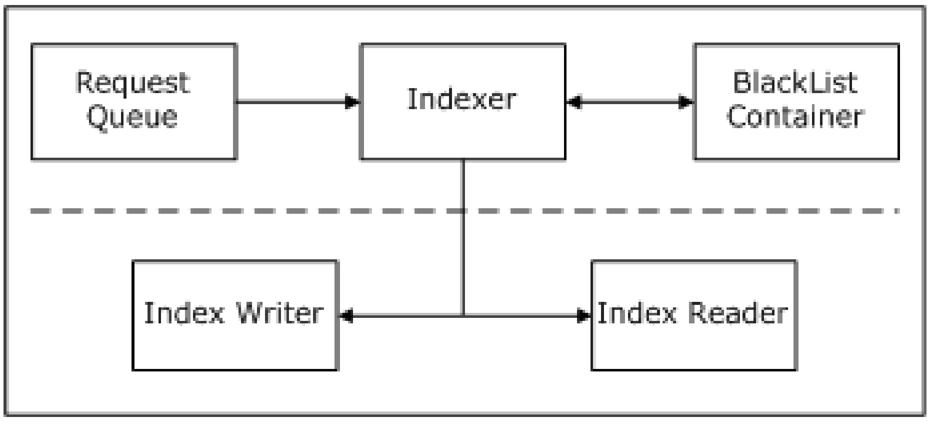

Figure 5.1: Indexing submodule

An indexing request arrives from the Data Model requesting one of three operations: Add, Update, or Delete. The first approach we evaluated for the Indexing submodule was to directly put Add and Delete requests in a request queue for execution, and replace Update requests with a Delete and an Add request. In this approach, the Indexing sub-module is always reading requests from the request queue. It opens a Lucene IndexReader or IndexWriter for the Store of the current request, registers the opened Store, and then executes the request. After executing the request it checks the next one, if it is for the same Store it reuses the same opened IndexReader or IndexWriter, otherwise it closes it and opens a new one. The Input/Output of indexing is only applied when the IndexReader or IndexWriter is closed, or after its buffer is full.

The advantage of this approach are that for a long sequence of Addition requests, or a long sequence of Deletion requests, belonging to the same Store, expensive Input/Output will be saved. However, for a long sequence of Updates, a sequence with mixed Addition and Deletion requests, whether they are for the same Store or for different Stores, the cost of Input/Output operations will be considerably higher.

The second approach, which we undertook, is to create a request queue for Addition, and a request queue for Deletion, and since the cost of Deletions in Lucene is high, as described above, performing the bulk Deletions will be Delayed till either there no no more Addition requests to process, or, the number of queued Deletion requests has met a certain threshold.

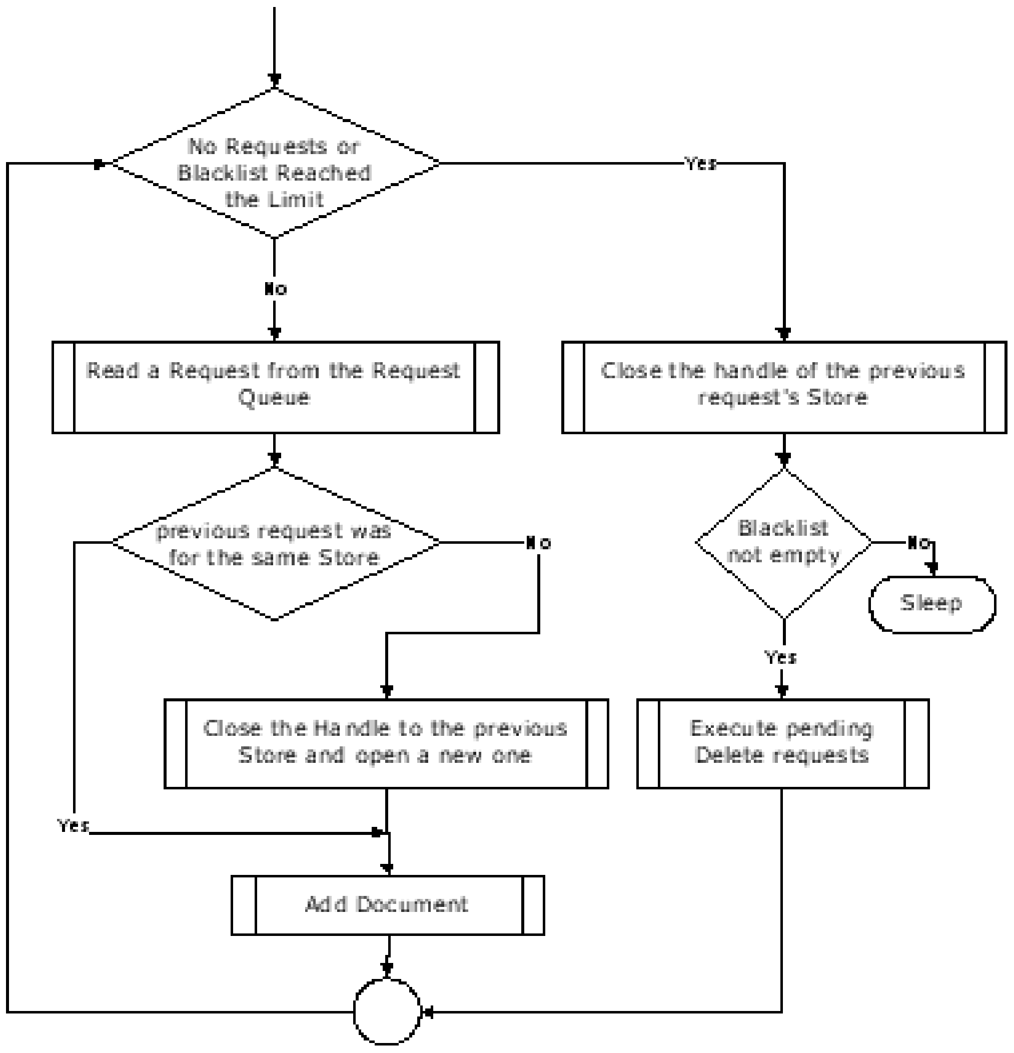
No Requests or Blacklist Reached the Limit
Yes
No
Read a Request from the Request Queue
Close the handle of the previous request's Store
previous request was for the same Store
No
Blacklist not empty
No
Sleep
Yes
Close the Handle to the previous Store and open a new one
Execute pending Delete requests
Yes
Add Document

Figure 5.2: Flowchart of Adding a Document to the index

This approach, although would yield better performance, might lead to inconsistent search results. Since a subset of the indexed information must be removed, but was not because their Deletion request has been delayed, these information may match against search queries until they are explicitly deleted. Thus, the search results are filtered from any Documents that have pending Deletion requests. The Blacklist is a container for storing information about the Documents currently pending Deletion. This way, the use of the Blacklist allows for achieving high system consistency while in the same time retaining good performance form delaying deletions.

Another important issue is that when we start executing the queued Deletion requests, we do not want to cause starvation for the arriving Addition requests, which as described above, are more critical and should have higher priority. Hence, there is a limit for the maximum number of Deletion requests performed sequentially without reverting to executing Addition requests.

Splitting the Add and Delete requests to two queues raises another issue. Since an Update request is converted to an Add and Delete request, there is no guarantee that the Delete request would execute before the Add request. To solve this issue, we added an internal version number for each Document indexed. Initially, this number is set to 0 for Addition requests. When an Update operation arrives, it is split to a request to Delete the highest version of the required Document, and an Addition request for a version 1 higher than the highest version. This means that there could possibly be more than one version of a Document in the index, till the Deletion of the older versions takes place. However, only one result is returned for any Document if it matched a search query.

Figure 5.2 shows the flowchat of Adding a Document to the index. Figure 5.3 shows the flowchat of Deleting a Document from the index.

The Searching and Indexing module has 4 types of index related requests:

- Create New Index
  - This is not queued, but executed directly.
- Add Document
  - An addition request is added to the request queue, with version of (0).

- Delete Document
  - Last documents version is retrieved.
  - The retrieved documents version is added to the blacklist.
- Update Document
  - Last document version is retrieved.
  - The retrieved documents version is added to the blacklist.
  - An addition request is added to the request queue, with a version one higher than the Last documents version.

5.4.2 Searching

The first step in performing a search is verifying the passed query is valid, according to the set search syntax. Table 5.3 describes the search syntax.

Before executing the query it must be Normalized to a uniform format. The following are the steps undertaken by the Search sub-module to Normalize the query:

- Lower case conversion,
- syntax checking and correction,
- adding Lucene related operators,
- and conversion to wide characters.

To show an example of Normalization, consider the input query “Name :: algorithm and not type : pdf not java and size :”. First, the query was converted to lower case Then, its syntax was checked and corrected, in this case the repeated colon was replaced by one, and the last field specifier was removed. Next, wild cards were added to the keywords, so that algorithm* would match both algorithm and algorithms, and similary pdf and java. Also in this step operators were added like the and in and not java.

Table 5.3: Search Syntax

Syntax	Example	Description
Keyword	UN	Search of any existence of UN in all fields.
Quoted String	“The Brown Fox”	Search for “The brown Fox” in all fields.
Keyword: Keyword	Title: UN	Searches for UN in all fields named “Title”
Keyword: Sentence	Title:“The Brown Fox”	Searches “The brown Fox” in all fields named “Title”.
(BooleanQuery)	(Brown OR Grey)	Enforces ordering.
(SimpleQuery)	(Fox)	Enforces ordering.
SimpleQuery SimpleQuery	The “Brown Fox”	Searches for the union of the result of two queries separated by a space.
SimpleQuery BooleanQuery	The Brown OR Fox	Searches for the union of the result of two queries separated by a space.
SimpleQuery OR SimpleQuery	Brown OR Fox	Searches for the union of the result of two queries.
SimpleQuery OR BooleanQuery	Brown OR Fox	Searches for the union of the result of two queries.
SimpleQuery AND SimpleQuery	Brown AND Fox	Searches for the intersection of the result of two queries.
SimpleQuery AND BooleanQuery	Brown AND Fox	Searches for the intersection of the result of two queries.
SimpleQuery AND NOT SimpleQuery	Brown AND NOT Fox	Searches for the difference of the result of two queries.
SimpleQuery AND NOT BooleanQuery	Brown AND NOT Fox	Searches for the difference of the result of two queries.

The resulting query after Normalization would be “name : algorithm* and not type : pdf* and not java*”.

After the query is executed, the search results are then categorized into different Document Types (as defined by the Data Model), and Blacklisted Documents are removed from the result.

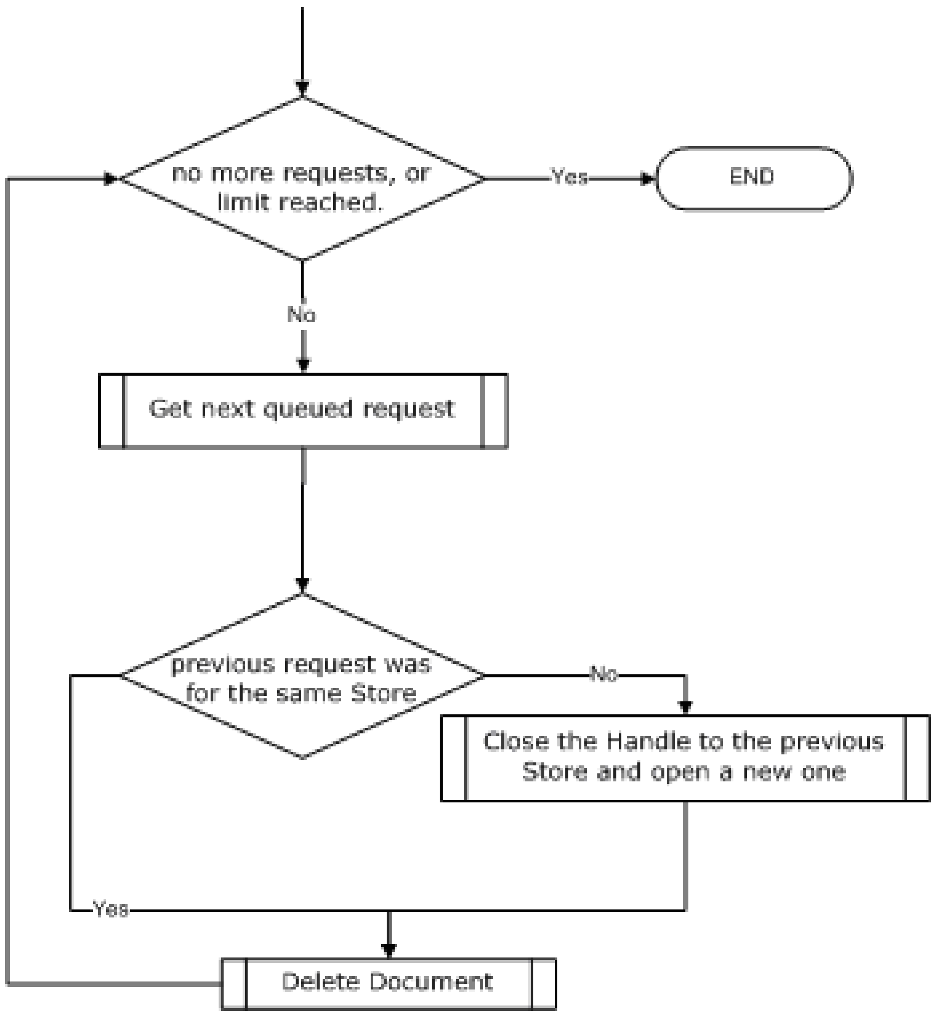
no more requests, or limit reached.
Yes
END
No
Get next queued request
previous request was for the same Store
No
Close the Handle to the previous Store and open a new one
Yes
Delete Document

Figure 5.3: Flowchart of Deleting a Document from the index

Part III

Experimental Studies

Chapter 6

Performance Analysis

We ran a set of experiments to investigate the performance of GLScube, and estimate the added overhead. The experiments was carried out on a Pentium 4 clocked at 2.4 Ghz, 384MB of RAM and a 5400 RPM 40GB disk drive with 2 MB of cache.

The experiment was divided into two parts. The first measures the cost of indexing, and the second measures the cost of searching.

In the indexing experiment, we performed full indexing on sample of 678 MB. Statistics of the sample data is shown in Table 6.1. Of all these files, only the "chm" type does not have an associated Importer, and thus, it will be imported to a Generic Type.

Table 6.1: Statistics of the sample data used to test full indexing performance

Type	Count	Size (MB)
MP3	85	374.7
AVI	13	122.6
TXT	19	8.4
CHM	6	42.1
PDF	16	50.2
JPG	771	79.9
Total	910	677.9

Table 6.2: Indexing Statistics: Storage Overhead for sample data of size 678 MB

	Size (MB)
Database Tables	5.3
Lucene Index	5.1
Total	10.4

6.1 Testing Process

To perform the Indexing and Searching experiments, custom programs where written to calculate the required time (in microseconds) of performing some actions. The programs used the GLScube API, which means that the measured cost covers the cost of not only, for example searching, but also the cost of formating the result and sending it back to the client applications.

Measurement of the disk I/O and CPU usage was done using dstat, an alternative to the iostat, vmstat and netstat suite of performance measuring applications.

6.2 Indexing Experiment

This experiment tests the system performance when a relatively large sample of data is copied to a GLScube Store, and what are the penalties associated compared to copying the same data without the existence of the GLScube services.

However, it must be noted that this experiment is not an indicator of system performance. Copying a large amount of data, although not uncommon, is not the most common operation. The most important factor would be the cost of incremental indexing, but whose cost would be much more difficult to analyze.

Table 6.3 shows the result of full-indexing the sample data shown above. In this experiment, copying the data took 183 seconds, during which the indexing operation was interleaved. After copying, indexing continued for an additional 95 seconds. The resulting size of the database and Lucene index are 10.4 MB, as shown in Table 6.2, around 1.5% the size of the original data, which we considerable an acceptable cost in storage.

To draw more clear understanding of the result, we performed the same ex-

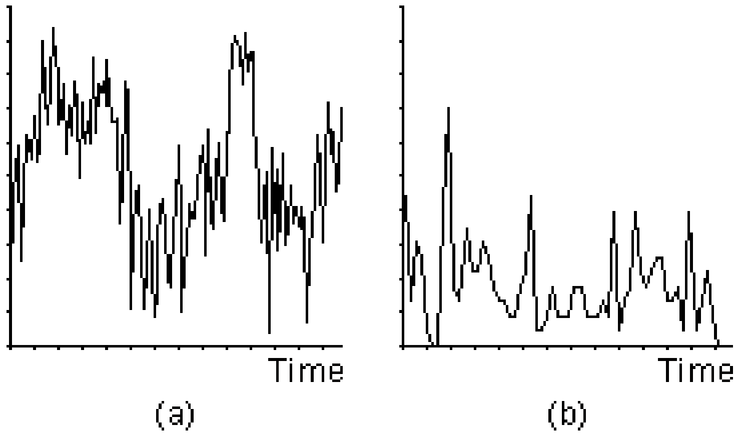

Figure 6.1: Comparison of the Average User CPU usage, both with and without Indexing

periment with the GLScube daemon closed. In this case, copying the same data took 109 seconds, around 60% of the time it took to copy the files while the daemon was running, and hence, indexing. Furthermore, the average user CPU usage dropped to 3.8%, compared to a much higher usage of 40% when the GLScube was running. I/O performance was also better in the absense of indexing, because the hard drive was all dedicated to writing the data, and unlike indexing, there was no need to read the written data again. A comparison of the Average User CPU usage is shown in Figure 6.1.

Although the results might seem inefficient, we find them excusable. First, the I/O performance can never be improved unless we overcome the necessity to read the files from the hard drive. With Direct Memory Access (DMA), this may never be possible. With DMA, the data would never pass through the kernel and thus there is no mean for intercepting it while in memory, and indexing it without having to re-read it from the disk drive.

There could be a room for improvement in the CPU performance. However, it must be noted that extracting information from some file formats is very expensive. Take for example PDF, in which the text is compressed and in order to index it, must be decompressed.

Table 6.3: Indexing Statistics: Performance Overhead

Time to Copy with Indexing	183 seconds
Time to Index (including Copying)	278 seconds
Average User CPU Usage while indexing	40%
Average System CPU Usage while indexing	20%
I/O Read Rate while indexing	26.2 Mbps
I/O Write Rate while indexing	28.48 Mbps
Time to Copy without Indexing	109 seconds
Average User CPU Usage without indexing	3.8%
Average System CPU Usage without indexing	3.2%
I/O Write Rate while indexing	48.6 Mbps

Table 6.4: Searching Statistics: Response time to executing 100 Queries

Total Time	782 milliseconds
Average Time	7.8 milliseconds
Standard Deviation	2.8 milliseconds

6.3 Searching Experiment

As shown in the Indexing experiment, there is a considerable cost associated with Indexing. In this experiment, we show that this high cost could be put into a different prespective when it is evaluated with the performance of searching.

To test the performance of searching, the same sample of data, described in Table 6.1 is used. Now that it is indexed, we test the performance of searching by executing 100 search queries sequentially, through an application that uses the GLScube API to receive results for the requests. The requests where randomly created to provide a mix of boolean operators, grouping, and even contain syntactically invalid queries to account for the cost of correcting search queries. The result to this test is shown in Table 6.4.

From this result, we can definitely conclude that the performance gained in searching is very interesting. A search query through almost 1000 files took 7.8 milliseconds on average, and the deviation from this time is only 2.8 milliseconds.

Table 6.5: Searching Statistics: Comparison of response time to executing 100 queries

Number of Files	Number of Terms	Size (MB)	Average Response
55	34,789	105.7	5.7 milliseconds
253	51,501	334.5	6.1 milliseconds
910	103,047	677.9	7.8 milliseconds
978	126,095	971.8	8.8 milliseconds

It is important however to show that the search performance does not linearly degrade with the amount of indexed information. Thus, we performed another experiment to verify that performance degrades at a rate slower than a linear function. The results to this experiment against samples of size 105MB, 334MB, 677MB and 974MB is shown in Table 6.5.

Appendix A

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with

released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ”History” in the various original documents, forming one section Entitled ”History”; likewise combine any sections Entitled ”Acknowledgements”, and any sections Entitled ”Dedications”. You must delete all sections Entitled ”Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Appendix B

Unicode

Unicode is a standard designed to allow text and symbols from all languages to be consistently represented. A crucial part of GLScube is localization, and thus, Unicode is relied upon for text representation. Next, we shall describe the history behind Unicode and its usage in GNU/Linux.

ISO 10646 and Unicode

ISO 10646 and Unicode where two independant efforts to create a single unified character set for encoding encode text of all forms and languages, so that it could be used, stored and transfered by a computer. The projects started in the late 1980s, and arround 1991, the Organization for Standarization (ISO) and the Unicode Project joined their efforts and worked together on creating a single code table, and maintaining compatibility.

ISO 10646 is only a character map that provides a unique code point - a number - for each character, and leaves the visual interpretation of the glyph to the software application. Unicode adds on ISO 10646 by supporting combined characters (represented as two code points, e and a combining acute), precomposed characters (as one code point), text normalisation, bidirectional display and collation (the last two features being cruical for scripts like Arabic).

As of now, ISO 10646 defines two forms of encoding: the UCS (Universal Character Set), namely UCS-2 and UCS-4. UCS-2 uses a single code value between

0 and 65,536 for each character, so that each character would occupy exactly two bytes. UCS-2 permits representation of every code point in the BMP (Basic Multilingual Plane, or Plane 0), and cannot represent code points outside it (like the Chinese GB18030). UCS-4 uses a single code value between 0 and hexadecimal 7FFFFFFF for each each character, so that each character would occupy exactly four bytes. USC-4 permits representation of every code point in the UCS.

An application supporting ISO 10646 does not necessarily fully support Unicode. Thus, an application could properly display ISO 10646 characters which have a one-to-one character-to-glyph mapping and single directionality, but would not correctly display Arabic characters with its bidirectional and one-character-to-many-glyphs script.

The UTF-8 encoding uses one to four bytes per code point. UTF-16 uses two bytes per code point and is nearly identical to UCS-2, with the difference that it allows encoding characters above hexadecimal FFFF as a pair of code values from the unused range D800-DFFF. UTF-32 uses four bytes per code point and is nearly identical to UCS-4.

UTF-8 is the native internal representation of text in GNU/Linux, BSD and Mac OSX, while UTF-16 is adopted in Microsoft Windows NT, and the Java and .NET bytecode environments.

Before the introduction, and widespread usage of Unicode, ISO-8859 was the most commonly used standard to provide multi-lingual applications and documents. ISO-8859 was an extension to the ASCII character set, where it made usage of the unassigned upper 128 values in a byte to store non-English characters. The standard itself was divided into 15 standards, each named in the form of ISO-8859-n. For example, ISO-8859-6 is the Arabic language standard. However, ISO-8859 had many disadvatages, some of which were its small table space - 128 bits - was not big enough to represent many languages, and there was no direct way to use two languages' character sets in the same scope simultaneously.

GNU/Linux Locale

The locale environment is the set of parameters that describe the user's language, how the time is displayed, and other language and cultural rules. A specific locale specifies preferences like how a character is converted to uppercase (using the toupper function), how mblen should count a multi-byte string, whether to use a point or a decimal point or decimal comma in numbers and various other conventions.

Each of the possible preferences is mapped to an environment variable; for example, the character encoding is defined by the LC_CTYPE environment variable and the format of time and date is defined by LC_TIME. The environment variable LC_ALL, if set, acts as a subistitution for all the other variables, and only if it is not set that the other variables are looked up.

```
char * setlocale(int category, const char *locale);
```

If the second argument to the setlocale function is empty, the locale is selected as the value of the LC_ALL environment variable. If it was empty, the environment variable with the same name as the passed category (LC_CTYPE, LC_COLLATE, ..) is used, if this was empty too, the LANG environment variable is used, otherwise, the function fails. For example, if the environment variable LC_ALL was set to en_US then a call to setlocale(LC_ALL, “”); will set the locale to en_US.

When a C/C++ program starts up, it initially uses the “C” locale by default.

As of today, many GNU/Linux distributions have switched their default locales to UTF-8, including, but not limited to, Red Hat Linux 8.0 (and higher), SUSE Linux 9.1 (and higher) and Ubuntu Linux.

To determine the currently used locale, execute the locale command. To get a list of the locales supported by your system, pass “a” as a parameter to the command:

```
locale -a
```

Wide Characters and Multi-byte Characters

Multi-byte strings are sequences of characters where each character is encoded in a varying number of bytes. The unit of storage of multi-byte string is a single byte (char datatype). In order to interpret a multi-byte string, the encoding of the string must be known ahead.

Wide strings on the other hand are sequences of characters where each character has a platform dependant fixed length that is longer than one byte. The unit of storage of a wide string is a single wide character (wchar_t datatype). In GNU/Linux, wchar_t is 32-bits long, contrary to Microsoft Windows where it is 16-bits long.

Functions that convert between multi-byte strings and wide strings are locale-sensitive. This is mandatory, because otherwise the conversion procedure would not be able to decide whether a character is one byte, two bytes or more. It is important to notice that generally, dealing with multi-byte string is dependant on the current locale, while wide character strings are independant from the current locale.

GNU/Linux, C and Unicode

In a GNU/Linux terminal with UTF-8 support, key strokes are transformed into the corresponding multi-byte UTF-8 sequence and sent to the stdin of the foreground process. Similarly, any output of a process on stdout is sent to the terminal where it is processed with a UTF-8 decoder and then displayed.

There are two approaches to add Unicode support to an application. In the first, data is left UTF-8 everywhere. The second approach is to convert the input data into wide strings, store and process in the application as such, and only convert it to UTF-8 at output time.

In the first approach, where data is left UTF-8 everywhere, the standard C char data type can be used to store the multi-byte string. All that is required is to set the application's locale as the user's at initialization. Internally, data is stored and processed as multi-byte strings, and the local-dependant functions mbstowcs and wcstombs can be used at any time to convert from multi-byte to wide strings or vice versa. Many of C's functions are locale-independant, so there usage will not be different in case of a Unicode applications, some of these are strcpy, strcat, strcmp, strstr and strtok. However, others will no longer work as anticipated, such as the strlen function, as strlen works by counting the number of bytes. To determine the length of a multibyte string, make a call to mbstowcs with NULL as the destination parameter. This would return the number of wide characters that would have been required if conversion took place.

```
size_t requiredWChars = mbstowcs((wchar_t *) 0,
   sourceMBString, strlen(sourceString));
```

The following code demonstrates some of the mentioned topics:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   setlocale(LC_ALL, ‘‘");

   // Get the current locale
   char * currentLocale = setlocale(LC_ALL, NULL);
   printf(‘‘Locale: %s\n", currentLocale);

   // convert a wide character string to a multi-byte string
   wchar_t * source = L‘‘";

   size_t destinationLength = wcstombs((wchar_t *) 0,
      source, wcslen(source));
   char * destination = new char [destinationLength];

   wcstombs(destination, source, 100);

   // output...
   printf(‘‘String Length: %d\n", strlen(destination));
   return 0;
}
```

Compile this program and execute the following commands in the shell:

```
$$ export LC_ALL=en_US.utf8
$$ ./unicode_test
String Length: 6
$$ export LC_ALL=ar_EG.utf8
$$ ./unicode_test
String Length: 6
$$ export LC_ALL=ar_EG.iso88596
$$ ./unicode_test
String Length: 3
```

Now what happened is as follows: In the first and second run, we had the current locale set to UTF-8, consequently, the wcstombs function call in the program converted the wide character string "source" to a multi-byte string encoded as UTF-8, because that was the locale the application was set to. In the third run however, the application is initialized to the ISO-8859-1 locale, which is, as described before, an extension to the ASCII standard that uses 8-bits per character. Hence, the wcstombs function call mapped each of the wide characters to their one byte representation in the ISO-8859-1 character set.

Bibliography

[1] *OdeFS: A File System Interface to an Object-Oriented Database*, volume 18, 2000.

[2] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: File organization from the desktop. *SIGCHI Bulletin*, 27(3):39–43, 1995.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2nd edition, 2002.

[4] Luke Cavanagh. Automatic categorization: Next big breakthrough in publishing? *The Seybold Report*, 2(6):3–7, 2002.

[5] Apple Computer. Mac osx: Spotlight - technology brief. 2005.

[6] Paul Dourish et al. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems (TOIS)*, 18(2):140–170, 2000.

[7] Dominic Giampaolo. *Practical File System Design with the Be File System.* Morgan Kaufmann Publishers, Inc, San Francisco, California, 1999.

[8] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Semantic file systems. *Proceedings of the Symposium on Operating Systems Principles*, pages 16–25, 1991.

[9] Otis Gospodnetic and Erik Hatcher. *Lucene in Action.* Manning Publications, Greenwich, 2004.

[10] The Shore Project Group. Shoring up persistent applications. 1997.

[11] Martin Haye. Cross-instance search system: Search engine comparison. 2004.

[12] Gary Marsden and David E. Cairns. Improving the usability of the hierarchical file system. *Proceedings of SAICSIT*, pages 122–129, 2003.

[13] Steve McCarthy, Mike Leis, and Steve Byan. Larger disk blocks or not? *Proceedings of the USENIX FAST Conference, Monteray, CA*, 2002.

[14] Microsoft Corporation. *Microsoft WinFS SDK Beta 1 Documentation*, 2005.

[15] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming.* New Riders, Indianapolis, IN, 2001.

[16] Bruce Momjian. *PostgreSQL - Introduction and Concepts.* Addison-Wesley, 2001.

[17] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. 2002.

[18] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide - C++ Programming with the Standard Template.* 2nd, 2nd edition, 2001.

[19] Michael A. Olson. Proceedings of the winter usenix conference. 1993.

[20] Dennis Quan, Karun Bakshi, David Huynh, and David R. Karger. User interfaces for supporting multiple categorization. *Proceedings of INTERACT 2003*, 2003.

[21] David A. Rusling. *The Linux Kernel.* Version 8.0-3. 1999. 26 july 2005 <http://www.tldp.org/ldp/tlk/tlk.html >edition, 2005.

[22] Doug Schaffer and Saul Greenberg. Sifting through hierarchical information. *INTERACT '93 and CHI '93 Conference Companion on Human Factors in Computing Systems*, pages 173–174, 1993.

[23] Ivan Smith. Historical notes about the cost of hard drive storage space. *3 Dec. 2005 <http://www.littletechshoppe.com/ns1625/winchest.html >*, 2004.

[24] Kent Sullivan. The windows 95 user interface: a case study in usability engineering. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Common Ground*, pages 473–480, 1996.

[25] David Thomas and Andrew Hunt. *Pragmatic Version Control Using CVS*. The Pragmatic Programmers, 2nd edition, 2003.

[26] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. Sams, 1st edition, 2000.

X

[2, 5, 3, 4, 6, 23, 1, 7, 8, 9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 10, 25, 11, 26]