MWFS

Operating System Update: Project Report

Matthew Sackman, Francis Russell, Sam Richards, and Will Osborne

January 5, 2005

Copyright © 2005 Matthew Sackman, Francis Russell, Sam Richards, Will Osborne

Abstract

When looking at the applications we use on a computer every day, it is unavoidable to see how so many applications present information about the file-data they are manipulating that is simply extracted from the file-data itself. An email application shows you who the email is from, who it was sent to, the subject and the date when it was sent; a music player will show you the name of the piece of music, the album name and the artist. These are all attributes which, if they were supported by the file system itself, would allow greater power and flexibility when dealing with files: for example, it makes complete sense to want to be able to see all the emails sent on a particular date, or from a particular person. Many email client applications support this functionality, but why not the file system? It makes perfect sense to want to group together all the pieces of music by a particular artist, but the file system doesn't cater for this, instead we are forced to create a *directory* and use that instead which has no relation whatsoever to the actual value of the artist attribute of the files inside the directory.

In this project we attempt to solve these problems, creating a filesystem where you can define attribute-types upon file-types. By using a database we can then filter the *view* of the filesystem by specifying criteria to match against attribute values. We then extend the filesystem by adding support for well known file-types, including JPEG, MP3, Ogg Vorbis and email so that new files will helpfully have attributes created for them. Our solution is multi-user and network transparent and features a full notification system based around research on multi-user database systems so that changes to the filesystem that affect users' views are immediately sent to the relevant users' clients, informing them of the changes. Finally we present several client applications, including file browsers, an audio player and a text editor, that allow us to make full and effective use of the filesystem's increased flexibility and power.

Acknowledgements. We would like to acknowledge the following people for helping us with this project: Susan Eisenbach for supervising us, the entire Eclipse[1] team for creating such a superb Java IDE, Tristan Allwood, Ivan Ianakiev and Matthew's long suffering girl friend Amy for tollerating Matthew's ranting, Matthew's parents for tollerating the endless late nights during christmas, Sam's flatmates for cooking for him so many times, Sam's family for not missing him too much over the holidays and Sophie Ellis-Bextor for the music that kept Sam going.

[1] <http://www.eclipse.org/>

Contents

1 Introduction **7**

2 Specification **9**
2.1 Minimum Requirements . . . 9
2.1.1 Server . . . 9
2.1.2 Clients . . . 10
2.2 A-grade Requirements . . . 10
2.2.1 Server . . . 10
2.2.2 Client . . . 11
2.3 Nobel Prize Requirements . . . 11

3 Background **13**

4 Implementation Structure and Design **15**
4.1 File Types . . . 15
4.1.1 The File Type Hierarchy . . . 15
4.2 Future Developments . . . 15

5 Metadata **19**
5.1 Definition . . . 19
5.2 Technologies . . . 19
5.2.1 Author Supplied Metadata . . . 19
5.2.1.1 Formats . . . 20
5.2.2 Third-Party Supplied Metadata . . . 20
5.2.3 Automated Derived . . . 20
5.3 Design . . . 21
5.4 Implementation . . . 21
5.5 Integration . . . 21
5.6 Testing . . . 21
5.7 Evaluation . . . 22

6 Server **23**
6.1 Purpose . . . 23
6.2 Protocol . . . 23
6.3 Architectural Design . . . 23
6.4 Views . . . 24

7 Database **27**
7.1 Purpose . . . 27
7.2 Design . . . 27
7.3 Implementation . . . 29

8 Cache **31**
8.1 Purpose . . . 31
8.2 Existing solutions . . . 31
8.2.1 File Systems . . . 31
8.2.1.1 Network File System . . . 31
8.2.1.2 Andrew File System . . . 31
8.2.2 Database Solutions . . . 32
8.3 Design . . . 32
8.3.1 Cache replacement policy . . . 33

9 Client 35
9.1 Connection Dialog 35
9.2 Launch Preference System 35
9.3 File URLs 35
9.4 MWFS Client Applications 36
9.5 Attribute Value Editing 37
9.6 Loading Utilities 37
9.6.1 File Loader 37
9.7 Email Handler 38

10 Drag and Drop 39
10.1 Purpose 39
10.2 Design 39
10.3 Implementation 39

11 File Browser with Query Builder 41
11.1 Motivation 41
11.2 Design 41
11.2.1 Overview 42
11.2.2 Query Builder 42
11.2.3 File Lister 43
11.3 Implementation 43
11.3.1 Query Builder 43
11.3.2 File Lister 44

12 File Browser without Query Builder 47
12.1 Purpose 47
12.2 Design 47
12.3 Implementation 48

13 Audio Player 51

14 Editors and Viewers 53
14.1 Introduction 53
14.2 Editors and Viewers 53
14.3 Utilities 54
14.4 Save and Attribute Dialogs 55
14.5 A Simple Browser 55
14.5.1 The Browser 55
14.5.2 The Query Builder in the Browser 56
14.5.3 The Command Line 56
14.6 Windows XP style buttons 57

15 Evaluation 59

16 Conclusions 61
16.1 Is it possible to implement? 61
16.2 Is performance acceptable? 61
16.3 Can it be made independent of the existing filesystem? 61
16.4 Is it a viable replacement for existing filesystems? 61
16.5 Was the project a success? 62

Bibliography 63

A Userguide 65
A.1 File Browser with Query Builder User Manual 65
A.1.1 Building a Query 65
A.1.2 Saving, Loading and Running Queries 66
A.1.3 Using the File Lister 67
A.2 File Browser without Query Builder 67

A.2.1 Displaying attribute types 67
A.2.2 Filtering the attribute values 67
A.2.3 Selecting attribute values 68
A.2.4 Restraining and releasing the view 68
A.2.5 Saving a view 68
A.2.6 Loading a view 68
A.2.7 Launching files 68
A.2.8 Deleting files 68
A.2.9 Editing file attributes 68
A.3 Audio Player 69
A.3.1 Controlling playback 69
A.3.2 Loading files 70
A.3.3 Adding and removing files from the playlist 70
A.4 Editors and Viewers User Guide 70
A.4.1 The Text Editor 70
A.4.2 Creating a Text File 70
A.4.3 Deleting a text File 70
A.4.4 Closing a Text File 71
A.4.5 Saving a Text File 71
A.4.6 Exiting the Text Editor 71
A.4.7 Editing Text 72
A.4.8 Toggling Syntax Highlighting 72
A.4.9 Formatting the Font 72
A.4.10 Shortcut keys 72
A.5 The Image Viewer 72
A.5.1 Deleting an Image File 72
A.5.2 Closing an Image File 73
A.5.3 Exiting the Image Viewer 73
A.6 The E-mail Viewer 73
A.6.1 Deleting an E-mail File 73
A.6.2 Closing an E-mail File 74
A.6.3 Exiting the E-mail Viewer 74
A.7 The Basic Sound Player 74
A.8 The Basic File Browser 74
A.8.1 The Browser 74
A.8.2 The Query Builder in the Browser 75
A.8.3 The Command Line 75
A.9 Utilities 76
A.9.1 File Loader 76
A.9.2 Email Handler 76

B Development logs **77**
B.1 Minutes of Meetings 77
B.2 Development Log for Matthew 78
B.3 Development Log for Francis 81
B.4 Development Log for Sam 83
B.5 Development Log for Will 85

Chapter 1

Introduction

> The most overlooked advantage to owning a computer is that if they foul up, there's no law against whacking them around a little.
>
> —Eric Porterfield

This is the final report produced in relation to our 3rd-year group project "Operating System Update". This document covers the details of our findings and implementation of our work.

This has been a difficult project which has touched on many different areas of computing, from cache invalidation techniques to human computer interaction psychology. As a result it has been challenging but also very rewarding: the very fact that I am using our created software in everyday use is immensely rewarding. That's not in anyway to suggest that the software is a finished product: there are many areas in which additional features would be more than welcome and other areas where further optimisation and reworking of the code would produce worthwhile and noticeable performance improvements. As always, there is never enough time to do everything one would want.

The filesystem of a computer system is one of the most crucial parts of the operating system. It must be reliable, it must perform acceptably and it must be easily understood and used by users. It is perhaps the one part of the operating system with which users directly interface: when using a computer, you're not aware of the scheduler or the memory manager, but you certainly are aware of the file system.

A relational database system offers huge power and flexibility: it is no mistake that they are behind almost every dynamic website, every bank and every business that uses a computer system to record actions. However, perhaps due to the perceived complexity of a database and perhaps due to inertia the typical end user does not attempt to make use of databases for their day-to-day work. However, it appears to us that the additional functionality provided by a database system would make a very strong argument for the use of a database as a filesystem. In many ways, a filesystem is already a database albeit somewhat limited when compared to the successful relational database products available today. This project is the realisation of this idea.

As file-data formats become more open and better structured, it becomes easier to extract properties of files. For example, a DocBook file may well have an author tag within the file. This is easy to extract from the file and at that point we would like to be able to search for all DocBook files which have the same author. For this to happen with a traditional filesystem, the filesystem would have to be searched for files and then every file would have to be opened and read. This is a phenomenally expensive operation. However, if we could specify that the author is extracted from the file whenever the file is written and kept as an attribute of the file, then there is then no need to open each file and the expensive operation becomes a quick search of a column of author-values.

The meta-data that can be associated with an arbitrary file is therefore fundamental to the power of this system. The tools that we have written to effectively search and browse the files in the system are more effective than traversing a folder structure in the general case. The file system is a client-server based network file system enabling distribution of files and applications over any TCP/IP network, including the internet.

It is clear that the traditional filesystem is ineffective in associating semantics with files. The traditional filesystem forces the association of a *location* in the directory structure with some meaning upon the user: the filesystem is very dumb in this respect. In our system the filesystem helps much more with the semantics as the additional attributes create implicit groupings and associations between files in a way which is simply not possible with the traditional filesystem. As a result, there is no longer any need for the directory structure, indeed our filesystem has no concept of directories. In addition, the system

has a much greater understanding of the concept of the *file type* of a file: gone is the three letter file-name extension.

Whilst this may seem a utopian notion, a side effect of this additional functionality is that it becomes harder to immediately grasp the state of the filesystem. With a traditional file-and-directory system, you always have a concept of *where* files are. Without directories you loose that knowledge and initially this makes it harder to use. However, with some thought and practise you realise that the purpose of the directory was simply to allow you to locate the file. Without directories you no longer care *where* the file is located because where it was located was never an important piece of information in the first place. The important information is simply how to locate and retrieve the file and in our system we present many more elegant and more efficient ways to achieve this than simply by directories. Therefore, our system dramatically alters the way in which we interact with the filesystem and with the computer in general, leading to a more productive and rewarding system.

Chapter 2

Specification

> Computer (noun): a device designed to speed and automate errors.
>
> —The Jargon File

The simplest aim of the project is to move the filing system into a database. This suggests several attractive possibilities but also produces some quite difficult problems, which we discuss. The given specification is as follows:

> Every component of a file other than the file data itself can be considered as meta-data. This includes the name, path, security attributes etc. If one is able to associate arbitrary meta-data with a file then the need for directories or even file names is vastly reduced and the way in which one uses a computer changes significantly as the filing system becomes closer to a database. For example, there is no need for a music file to have a file name or a path: the most useful data would be song title and artist(s) of the track.

The file system will be designed to make not only searching, but also manipulating this meta-data faster and easier than you would otherwise be able to with current file systems. To do this we will store the meta-data and the files in a relational database. This makes the file system far more versatile than a directory based structure as you can associate any piece of data with a file as an *file attribute* and then later recall that file by specifying criteria against available attributes.

The original proposal for this project can be found at `<http://www.doc.ic.ac.uk/~ih/teaching/group-projects/proposals/sue1.html>`

2.1 Minimum Requirements

These are the specifications which, if implemented, should result in a B grade for the project.

2.1.1 Server

The Server is a daemon to which multiple clients connect and communicate with, providing an interface to query, extract and modify data in the database.

- The Server must be able to be connected to by a reasonable number of Clients without any severe scaling issues. A *reasonable* number is clearly open to debate and will be a function of the power of the hardware on which the server is running. Let us simply say that it must support six or more users each using several client applications. It must be able to provide reliable operation to those Clients at all times including when other Clients join or leave the Server.

- The Server must be able to respond to arbitrarily complex queries sent from any Client, querying the filesystem, without any substantial delay. That is to say that the database itself must be designed appropriately to allow fast querying of the data in the database.

- The Server must be able to reliably transport the contents of files requested by the Client to the Client in a streaming manner which allows the Server and the Client requesting the file to remain responsive and without saturating the network. All other Clients connected to the Server must not become unresponsive.

- The Server must be able to guarantee the consistency, accuracy and integrity of the data in the database and of the data that it sends to the Client. The Server must be able to respond to and deal with errors occurring from the database and handle those errors in an appropriate manner, informing the Clients promptly at all times of any problems the Server has encountered.

2.1.2 Clients

The Clients are typically desktop applications that provide the expected functionality of a file browser, text editor, audio player etc. They connect to the Server and use the interface defined by the Server to manipulate files and their attributes.

- There must be a File Browser that allows the user to connect to a specified Server and perform user defined queries, displaying the result in a graphical manner. All saving and loading of files is performed via this File Browser.
- The File Browser must allow the User to define an arbitrary query and to save that query for later use.
- The File Browser must be able to present the details of a query in a textual and graphical form.
- The File Browser must be able to allow the user to define queries that reference other queries, ie a query that filters the results of another query.
- There must be a simple Text Editor which performs the expected functionality of a Text Editor. It will be invoked by the File Browser when a User opens a Text File. Saving will be performed by *Drag and Drop* actions between the Text Editor and the File Browser.

2.2 A-grade Requirements

These are the requirements that if met should result in an A grade (or higher) for the project.

2.2.1 Server

- All files will carry with them a File Type. The File Types will form a hierarchical tree structure.
- Attribute Types can be defined against File Types which will then force any file which is of that File Type or any sub-File Type to carry a value for that Attribute Type. For example, all Files of the most basic type will have a "Creation" attribute. All Files of type "Music" will have an "Artist" attribute. "Music" is a sub-File Type of the basic type so all "Music" files will also carry the "Creation" attribute.
- The Attribute Types will specify a Data Type which will indicate the type of the data the value of the Attribute will contain. For example, the Attribute Type "Creation" will have a Data Type of "Timestamp". Thus the value of all "Creation" attributes will be interpreted to be a timestamp. This will reflect the way the values are stored in the Database itself.
- The Server must be able to automatically extract attributes from suitable files. For example, "MP3" files contain within them information regarding the artist and title of the song. The server should be able to automatically extract this data from the files and present this data as attributes which can then form requirements in a query.
- There must be the ability to define a persistent query, the results of which the File Browser would display as normal. However, as soon as any File in the results of the query is updated, the Server will inform the Client of the update which will allow the Client to update its display immediately. This removes the need for the Client to either poll the Server or display out of date data.

2.2.2 Client

- There should be an Audio Player which is capable of playing audio files stored in the database.
- The File Browser should allow the definition of queries via either a textual or graphical manner.
- The File Browser should provide context sensitive options when building a query.
- When saving a file via *Drag and Drop* operation, the File Browser should apply to a new file all fixed attributes that the query into which the file was *dropped* specifies. This will make it much easier to quickly specify attributes on new files.
- The Text Editor and Audio Player should support the loading of files by a *Drag and Drop* action from the File Browser.
- There should be an Authentication System to allow support of multiple users.

2.3 Nobel Prize Requirements

These are the requirements that if met should result in our group being awarded a Nobel Prize or Oscar.

- File versioning: It should be possible to roll a file back to a previous known version in the database.
- Disconnected Operation: The Client should maintain an extensive cache of files and file actions to allow full operation without connection to the Server. Upon reconnection, the Client will resynchronised with the Server.
- To aid performance, the Database should be integrated in the System Kernel. At this point we should be able to reach speeds matching the *ext2* filing system.
- The entire Operating System should be rewritten to take advantage of this Filing System. Thus there should be no requirements whatsoever for a traditional Filing System. This would require the rewriting not only of the Kernel but of every application, program, library and command that makes any reference at all to the Filing System as it stands.

Chapter 3

Background

> Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.
>
> —Rich Cook

Filesystems have developed markedly throughout the history of computing. Ever since the invention of non-volatile and mutable storage, there has been the requirement for the ability to identify individual arrays of data, which we know as files. Early systems only supported file names, later, directories were created and later still file-name length limitations were eliminated. Finally, attributes such as created-date, modified-date and security and access attributes were added. Nevertheless, the file-and-directory structure remained.

Some systems did go further, most notably BeOS had the ability for files to have arbitrary attributes defined against them. There was then a filtering and notification system so that the results of a set of filters against the filesystem would be updated as soon as a relevant modification to files' attributes were made. However, firstly this was performed on top of a traditional file-and-directory filesystem and secondly, whilst there was the capability of almost arbitrary attributes, the attribute values themselves were dictated by the application. For example, the email application would dictate the values of the *from*, *to* and *subject* attributes. Further more, there was no provision for the user to define their own attribute-types, forcing users to revert to the traditional file-and-directory means of grouping files, for example as part of a project.

Perhaps as a result of the frailty of operating systems and computers in general, much recent filesystem development has been focused on filesystem resilience. This includes *journaling* of data so that the consistency of the filesystem can be guaranteed even after a power loss during a write to the filesystem. This is obviously useful and important work and is well received by the data warehouses and enterprise level systems but does little to develop the user's interaction with the filesystem.

Most recently, Microsoft have removed their much vaunted *WinFS* from their next re-invention of *Windows*, code-named *Longhorn*. WinFS was meant to have support for database-like operations, some quite similar to the work we have done in this project. The reasons why it has been dropped remain unclear, it may be available as an additional optional feature some time after the launch.

On the whole however, there has been little work on adding the sorts of functionality to filesystems that we are interested in. We can not conclude from this that users are satisfied with the traditional file-and-directory filesystem, but instead that it has attraction of being instantly understandable by analogy with a filing-cabinet. Not only is the traditional filesystem easy to understand, but it is also easy to represent on a computer screen. In contrast, the system we present has some very interesting challenges when it comes to displaying a realisation of the filesystem, indeed, this is where a great deal of effort was spent: how do you prevent the user from becoming confused when the file browser tells them that there are 100 emails in the filesystem, we are currently displaying the values of the *from* attribute and there are only 52 unique values? How can this be when there are apparently 100 emails? With a little practise and thought, the system does become intuitive to use, but there is nevertheless a learning curve that is steeper than that of the traditional filesystem.

Chapter 4

Implementation Structure and Design

> I have made this letter longer than usual because I lack the time to make it shorter.
>
> —Blaise Pascal

The general structure of the server and client can be seen in Figure 4.1 and Figure 4.2 respectively. Both are implemented entirely in Java apart from where the server interfaces with the database where SQL must be used. Both the client and the server are multi-threaded, using thread pools to maintain a group of worker threads that are assigned jobs as the jobs arrive. This allows for parallelism but also provides a useful cap on the number of threads that can be spawned to prevent exhausting the Java runtime system.

The Client-Application interface is feature complete in that there is never any reason for a client application to communicate with the server directly: all is encapsulated by the client system, massively making the situation easier for the client application author. In fact, the interface presents methods which may not ever even query the server if they can be satisfied from the client's cache. The client will never replace an item in its cache with a new item, instead it will modify the state of the cached object to match the state in the server. Therefore, == is always the correct way to evaluate whether the results of two method calls are in fact equal. It is never necessary to invoke the `equals()` method. This approach aids performance and memory usage as objects tend to become longer lived, aiding the garbage collector's work.

4.1 File Types

In our system we have decided to abstract away the fact that different file formats may be used to store one file type. For example there will be an audio file type that will encompass MP3, Ogg, FLAC and WAV files. This will simplify the experience for the user, however if the user needs to select files based on the format that they are stored in, this will be stored as an attribute.

From the user's perspective there will then simply be a audio player, a text editor, an image viewer and any other applications necessary to support the file types used.

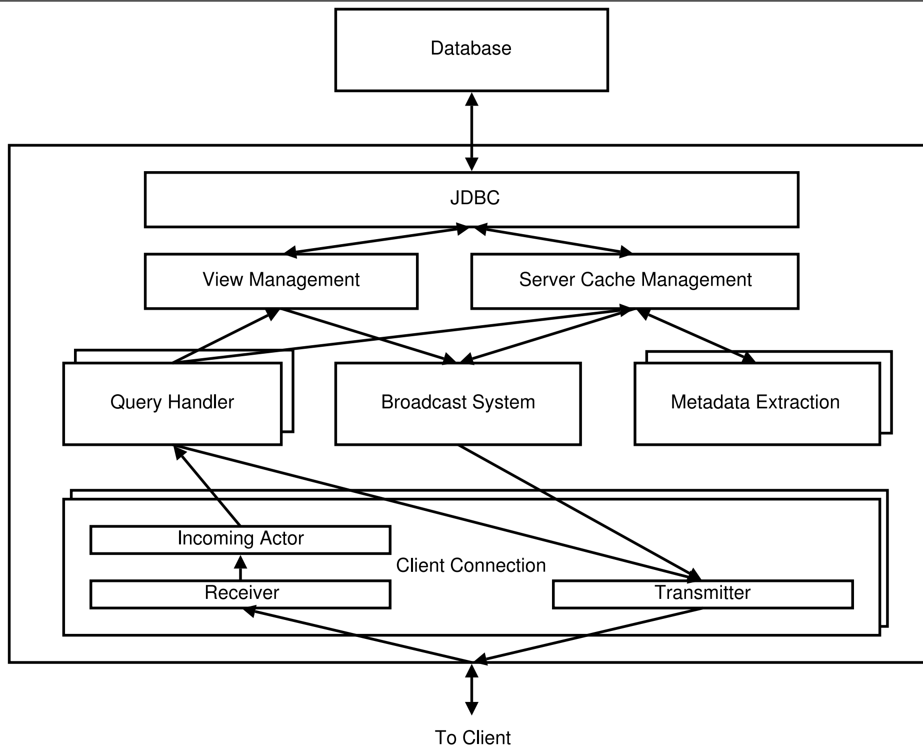
4.1.1 The File Type Hierarchy

There are many types of file that share common attributes. Multimedia files, for example share common information, such as the duration and copyright holder. We then have file types that belong to this base file type and inherit its attributes. Figure 4.3 shows the hierarchy for the base file types. This hierarchy is stored in a table in the database, see Figure 7.1. The database is never accessed by any client application and the MWFS server provides an object oriented interface to this hierarchy. The MWFS server provides an object oriented system of files that uses the SQL database as its data store.

4.2 Future Developments

In many situations it would be useful to have the ability to define multiple attributes of the same attribute type on a single file. For example, if a file was coauthored then it would be useful to assign two

Figure 4.1 The general structure of the Server



author attributes. This is not currently catered for as it would require the relaxation of database constraints, it would present some further client caching complications and it would cause client display complications. For example, in such a system, consider a file browser where you are currently viewing the author attribute values. Firstly, if you only have a single file, it would now be possible for the file browser to display each attribute value leading to the suggestion that there are multiple files. With such a system you would really have to get away from the idea that an attribute value corresponds to a file which is a difficult concept to get used to. Secondly, if you then filtered such a view of the filesystem so that the only value of the author attribute was “freddy” then do you exclude files which have “freddy” and someone else as the author? If not then upon inspecting the results of such a filter you would possibly conclude that the system was broken as you would still have values of the author attribute that are not “freddy”. For these reasons we have not implemented this functionality. It’s debatable whether the advantages of such functionality outweigh the disadvantages in terms of ease of use of the filesystem as a whole.

Currently it is not possible to have a file itself as an attribute value. One of the major problems with such an idea is it would massively complicate the client applications which would face some interesting challenges in displaying the *value* of such an attribute. This is nevertheless a nice idea, for example, consider a family-tree application which simply uses files to represent people. Each person would have a spouse attribute, mother, father and children attributes. It would be incredibly convenient to be able to have as the values of these attributes the actual file for the corresponding person. However, as soon as you provide for this, you have to contend with cyclical data structures in your file system and the rendering problems already discussed. Currently therefore, this functionality is not catered for but it is something that would be useful in a future version.

Figure 4.2 The general structure of the Client

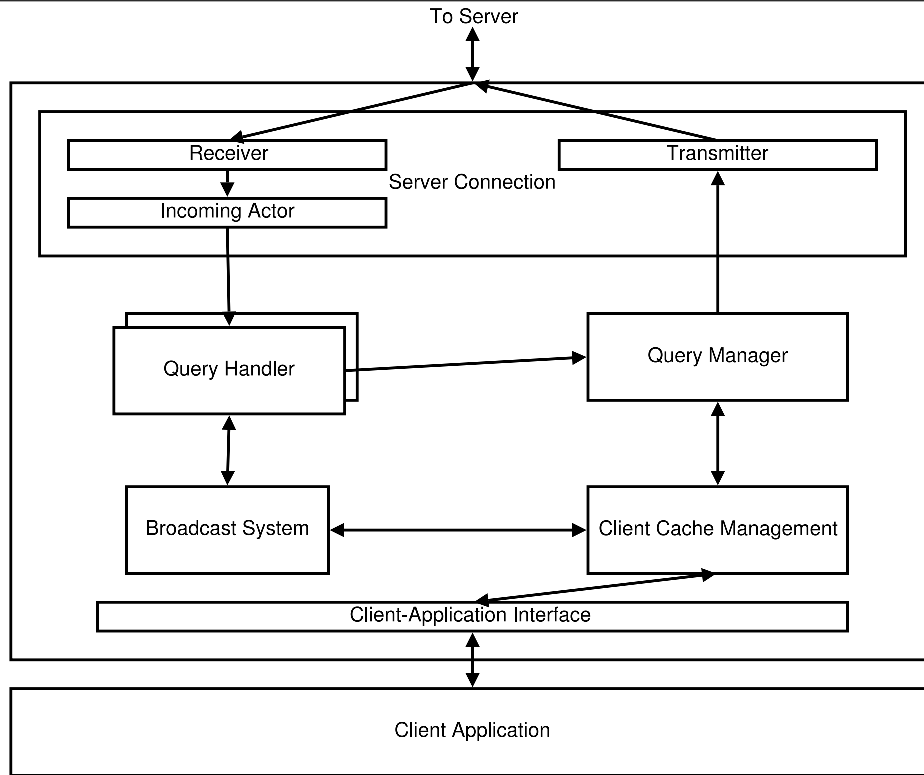
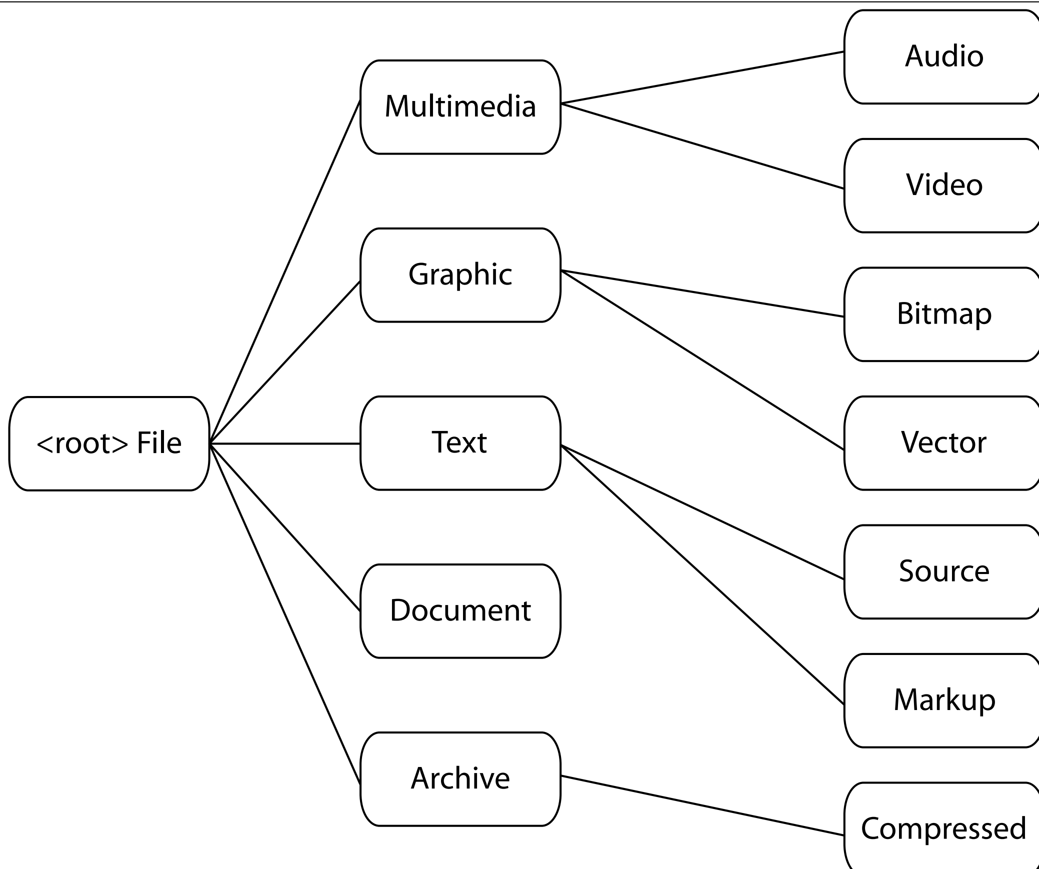


Figure 4.3 The *base file type* hierarchy.



Chapter 5

Metadata

5.1 Definition

Metadata includes a description and control information about data, or simply data about data. The theory of this system is for each file to be uniquely identifiable by its metadata, in a more common hierarchical tree based filesystem, this would be achieved by a file's path and name. By storing each file with its own metadata stored in an index, any group of files on the system can be defined by a *view query*. In a tree based file system, files may be sorted by the user and organised into a folder structure. This is an effective method of organising files if the user is careful in his saving of files. The user then should be capable of retrieving any file from his logical file structure. However if the user wanted to retrieve a group of documents defined by some parameter that isn't part of the definition of his folder structure this would be very difficult to do.

Having all the metadata for each file indexed means that arbitrary views on the files in the system can be created enabling browsing of the available files by any attribute.

5.2 Technologies

There are three classes that metadata falls in to:

- **Author Supplied.** When the author publishes a document, he may add metadata to it. Common attributes might be the author's name, the title, some keywords, the validity period and date published.
- **Third-Party Supplied.** If documents are being added to an information retrieval system or digital library, there may be a group of *metadata authors* that read the documents and write metadata in order to categorise them.
- **Automated Derived.** Much of this metadata can be extracted from a well structured data type. SGML documents being one example, other media such as images and audio are difficult to derive metadata from their binary content. It is currently unfeasible for a piece of software to derive a description of the contents of an image or audio file. Much research is being done in this area including work here at Imperial in The Multimedia Information Retrieval team. However this is beyond the current scope of the project.

We will handle the first two types of metadata in the system but also keep the framework extensible so that future work on automated metadata extraction can be added in a modular fashion to extend the system.

5.2.1 Author Supplied Metadata

This data is attached to the file in a file type dependent manner. In order to extract this data into common set of attributes, various filters will be written to handle the extraction of the data from the different formats.

In a public or widely accessible system author metadata unfortunately cannot always be relied on for an accurate description of the contents of the document. It is common for authors to add very common

keywords to their documents so they appear more frequently to a user of the system during browsing or searching. In a limited access system, as this will be, this phenomenon shouldn't be an issue.

5.2.1.1 Formats

For many file types there are built in fields that can be used as metadata. These will be dealt with using specific filters.

- **ID3.** This is the metadata format that is present in the common audio format MP3. Version 1 of ID3 provided fixed fields for specifying a small number of attributes of the audio file. These included `artist`, `title`, `album`, `year`, `genre`, `comment` and `track`. The total information that could be stored per track was a mere 128 bytes. The second version of the tagging format enabled arbitrary length fields and custom fields to be added based on an extensible frame format, similar to the more elaborate metadata format found in JPEG files.

  As both formats are very common, it will be necessary to read both in the system.

- **EXIF.** This is the format that scanners and cameras store various details about the image when it was captured. These include the conditions under which the image was digitised, information about the capture device and a timestamp. This format is present in JPEG files in the `APP1` segment.

- **IPTC.** This is the International Press and Telecommunications Council, and this format is a standardised method of storing the author, caption, copyright and other information for an image or other media file. This standard can be stored in the Adobe's XMP standard which is found in the APPD segment of JPEGs.

- **Ogg Metadata.** This is the format included in the open source Ogg transport stream that is most commonly used to encapsulate the open source Vorbis audio compression format. It defines[1] many tags for the detailed description of audio files.

- **Email Headers.** These contain metadata with which you will be familiar, such as the `subject`, `date` and the recipients. It also contains other information that is usually hidden by email readers including the route that the message has taken on the network. The most important information to uniquely identify a message in the system is the date, subject, the recipients and who sent the message.

- **Structured File formats.** These contain data that can be used in the place of metadata because strictly metadata is in addition to the file contents. It is possible to use parts of certain file types in order for them to be uniquely identified.

5.2.2 Third-Party Supplied Metadata

This data can be added by the user to an existing file in the system. The attribute management system will provide an interface for the manipulation of the file attributes by the client. An editor can be made available to edit the attributes directly of particular files.

Whilst this facility is useful, for large collections it is necessary for the users that will be manipulating metadata to try and do it in a way that is consistent among them. This can be aided through training of the users, and use of controlled vocabularies.

5.2.3 Automated Derived

This is the process of attempting to guess or calculate metadata from the raw contents of the file. In some types of file, such as textual files this is quite feasible, but associating attributes with binary content is the subject of much research and is beyond this project's scope.

[1] The comment specification is at <http://www.xiph.org/ogg/vorbis/doc/v-comment.html>. As the format allows you to specify your own fields there are many community recommended lists of fields when more detail about a track is required.

5.3 Design

The file-typing system described in Section 4.1 denotes the metadata fields that will be required for each base file type. These file types are independent from the actual format of the file. Each format of file will need a filter to read out the metadata that has been defined. Each of these is a `FileModificationHandler`. The handler is responsible for keeping the attributes up to date with the file contents. Each file may have multiple handlers associated with it, each responsible for different attributes. The handlers will be assigned to each file-type recursively so that when an audio file is modified, the server will trigger the `FileModificationHandler` for each of its file-type and of each of its parent file-types: that of the `multimedia` type and the `file` type. The `file` type handler will have support for `modified`, `created`, `size`, and `owner` attributes. As these attributes are native to the `file` type and as everything is descended from `file`, these will be the only attributes that are guaranteed be common to every file on the system.

As the system is being implemented in Java, there are many libraries available that can form the basis of the filters. Although from initial research, the majority of them are not capable of reading a file from anywhere other than the local filesystem. They can however be modified so that they will be able to read the file from the database.

5.4 Implementation

A number of Sun and third party libraries were used to build the handlers for each file format, I will cover a few here:

- **JavaSound.** was used in conjunction with third party SPIs[2] providing support for MP3 and Ogg Vorbis audio formats. The JavaSound MP3 SPI only provided support for the newer version 2 ID3 tags so to support the still common ID3 version 1 tags we wrote an extractor to read them from the files.

- **JPEG: EXIF and IPTC.** The Java Image functions were unsuitable for use to read the image files here, as in order to extract the metadata from the file they loaded the entire file. This is unnecessary to only extract the metadata from the frames at the start of the file, so a third party library was modified that only needed to scan through the start of the file. This prevented unnecessary database traffic as the Sun libraries would have loaded the whole of a potentially sizeable image from the back-end database.

- **Archives.** Java's built-in Zip, GZip and a third party TAR parser were implemented to provide `file count` and `actual size` attributes for the uncompressed and compressed archives respectively. As tar archives are commonly GZip'ed, this was handled as a special case so the attributes reflected unGZipping and untarring the archive.

File type auto-detection mappings.. Many different file types might be categorised transparently as one base file type within this filesystem. For example both an MP3 file and Ogg file will be classified as an audio file. However the two files will be distinguished by having different `audio codec` attributes but not separate types as there is no difference in the type of data that is being stored in the file.

5.5 Integration

Each `FileModificationHandler` is pluggable so that in order for them to work they simply need to be registered with the server for them to operate on the specific file type.

Example 5.5.1 shows one example of adding a file modification handler to the server.

5.6 Testing

We made a large number of test images, archives and audio files which were loaded into the filesystem using the LoadInFolder utility described in Section A.9.1. The files were then examined in the file

[2]Service Provider Interfaces, transparently provide extra functions to the JVM at runtime, without the need to recompile the application.

Example 5.5.1 Adding the audio `FileModificationHandler` to the audio file type

```
ISFileType audioFt = FileTypeFactory.getFileType("audio");
            if (audioFt != null)
               audioFt.addFileModificationHandler(new AudioFileModificationHandler());
```

browsers and checked whether they reported the expected attributes. For the initial development of the handlers they were tested with a JUnit test so that a one-click test could be performed to check that the functionality was present.

5.7 Evaluation

The framework worked very well and left much room for expansion. To add new file formats and new file types to the system would not require rewriting any of the system, solely adding the required code necessary for the format's support.

There weren't many major problems implementing this section, however a large amount of research was required to write the part. We needed to research in detail the structure of each file format, each file format that we support in the finished version is open source or the specification was freely available. We did not have the time, nor resources to attempt to handle any proprietary file formats.

Future Plans. These would include the addition of support for a wider variety of file formats and the addition of more intelligent metadata modules. These may include:

- Thumbnail module, for images and video
- Album covers for music tracks
- Fetching of artist information, such as biographies

Chapter 6

Server

> Basically, I no longer work for anything but the sensation I have while working.
>
> —Albert Giacometti

6.1 Purpose

The purpose of the Server is to define a useful API for Clients to connect to, to query and manipulate the filesystem. The Server must contend with multiple Clients.

Some attributes are known as *derived* attributes. These are attributes whose value can be suggested by analysis of the file's content. For example, an MP3 file may have within it information as to the *song name*, *artist* etc. These are derived attributes. The details of the mechanisms used to extract these attribute values are considered in Chapter 5, "Metadata" and are not further considered here.

In order to improve performance, both the Server and the Client will cache files and their relations with file types, attributes, attribute types and data types. Efficiently maintaining the Client's cache is a difficult problem and is discussed in Chapter 8, "Cache".

6.2 Protocol

Communication between the Clients and the Server is done using a TCP/IP connection. Objects are serialised and passed from the Client to the Server and responses are passed back, again through serialisation. The type of the object being serialised defines the protocol, that is, upon receiving an Object at the Server, the Server analyses the interfaces the Object implements and hands the Object to a handler class which has been registered as being able to deal with that particular type of Object. This way structured information can be easily passed from the Client to Server and vice versa and the protocol can be very easily extended as necessary. The action of sending an Object to the Server and receiving a reply is referred to as a *query*. Do not confuse this with a database query.

All communication is based on the send-and-receive primitive. That way, the client can guarantee that operations are performed in order despite a multi-threaded Server design which would allow queries to be evaluated out of order. Due to the multi-threaded design in both the Client and Server, both synchronous and asynchronous queries can be catered for and an asynchronous query can become a synchronous query by blocking waiting to receive the results of the query. In addition, any client can ascertain whether the results of a query have arrived yet without blocking. There is also the functionality to discard any response to a particular query if necessary.

6.3 Architectural Design

The general design is symmetric so the Server can query the Client in exactly the same manner as the Client can query the Server. This is primarily used for cache invalidation notification purposes and obviously the query handlers installed in the Client are very different to the query handlers installed in the Server. Nevertheless, careful design has led to a significant amount of reused code between the Client and the Server. Outgoing queries and the response to the query are matched using a simple query id. Additionally there is a broadcast mechanism in which the server can broadcast messages to all clients

without expecting any reply back from the client. This is useful for informing the clients of changes to file types and attribute types as it is almost certain that every client will need to be informed of such changes, thus it is more efficient to use a broadcast rather than a subscription based notification system in these cases.

To improve performance, Thread Pools are used both on the Client and the Server. Whenever a query is received and a thread in the pool is available, the query is passed to the thread and the thread deals with it. At that point, the *incoming actor* thread can go back to waiting for the next query to arrive. This allows for the possibility of massive parallelism and generally results in very good performance. For pure data transfers, for example transferring a block of a file, it has been found by trial and error that a large block size of 64k is preferable. This is because many smaller blocks would result in many threads being invoked to deal with the incoming blocks and the resulting rendezvous and thread-context switching overhead becomes noticeable. A larger block size gives each thread in the pool something rather more meaty to get its teeth into and therefore performs better. To avoid resource exhaustion, the thread pool is capped. There is also a limit on the number of idle threads so that a significant burst of queries would not result in vast numbers threads being spawned and then remaining on the system doing no work. It therefore becomes important to prevent all the threads in the thread pool from being allocated to long-running *daemon* jobs otherwise starvation can occur.

6.4 Views

Views of the file system are defined by *view queries*. A view query has a tree structure within, which specifies the conditions of the view: for example specifying the required file type and values of attributes. At the server, a visitor pattern is used to walk this tree and construct SQL statements that can then be used to query the database.

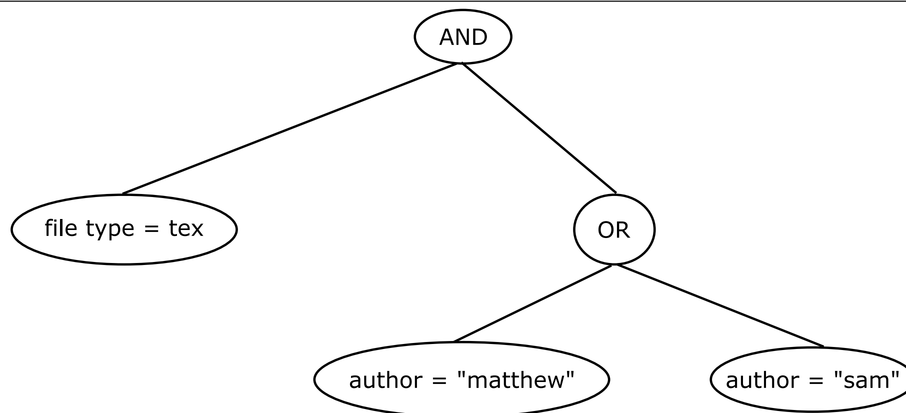
The conversion to SQL is complex but highly efficient. The process walks the tree depth first, left to right. For every leaf in the tree, a simple SQL statement is produced, for example, if a leaf specified that the file type should be `tex` then the resulting SQL is shown in Example 6.4.1. The *x* is the ID of the `tex` file type and can be obtained simply from the corresponding object. The leafs then get combined in the *where* clause of the overall SQL query. Using *exists* is extremely efficient because all we care about is whether the result is empty or not. This means that the query is halted as soon as a single result is found. Further more, we do not care about the value of the result, hence the *select 1 where...* which further improves performance as selecting a constant value is very fast.

Taking the query tree in Figure 6.1 results in the SQL statement in Example 6.4.2, where `x` is the ID of the `tex` file type and `y` is the ID of the `author` attribute type. Much analysis of the various SQL statements that can express such a query was performed: there are many ways to express these queries. However, this method, using the *exists* function is by far the fastest, in particular because it only results in one sub-select statement per leaf, which keeps the complexity of the statement to a minimum.

Example 6.4.1

```
exists (select 1 where files.file_types_id = x)
```

Figure 6.1 . An example view query tree.



Example 6.4.2

```
select files.object_id from files, attributes
  where files.object_id = attributes.files_object_id
    and
      (
        (exists (select 1 where files.file_types_id = x))
      and
        (
          (exists (select 1 where attribute_types_id = y
            and value_string = 'matthew'))
        or
          (exists (select 1 where attribute_types_id = y
            and value_string = 'sam'))
        )
      )
```

Chapter 7

Database

In theory, there is no difference between theory and practice. But, in practice, there is.

—Jan L.A. van de Snepscheut

7.1 Purpose

The database is responsible for storing the files themselves and all of their attributes.

7.2 Design

To improve usability, a formal structure will be imposed. A hierarchy of file types will be created and attribute types will be defined for each file type. Every file is of a single file type and can only have defined upon it attribute values for attribute types defined on the file's file type or any parent file type.

The type of the value of an attribute is defined by the data type of the attribute type. Thus the attribute type `creation` would have a data type of `time stamp`. Data types can be given meaningful names. This would allow a client-side application to make some reasonable guesses at how to display the attribute value. For example, all files will have an `owner` attribute and some files may have an `author` attribute. The data type of values of these attributes would ultimately be a string type but you could declare a data type called `username`. This would, for example, allow the client to look up some information about that user, when they last logged in, their email address, present a link to the user's homepage etc. This would not be possible if the data type of the attribute was simply defined as `string`. Similarly, a data type of `image` means far more than a byte array to a client application. Note that the abstraction is correct in that this information is inferred from the data type and not from the attribute type. Many attribute types of different names may use a data type of `image`. It is the data type that informs the client to render the data as an image, not the name of the attribute type.

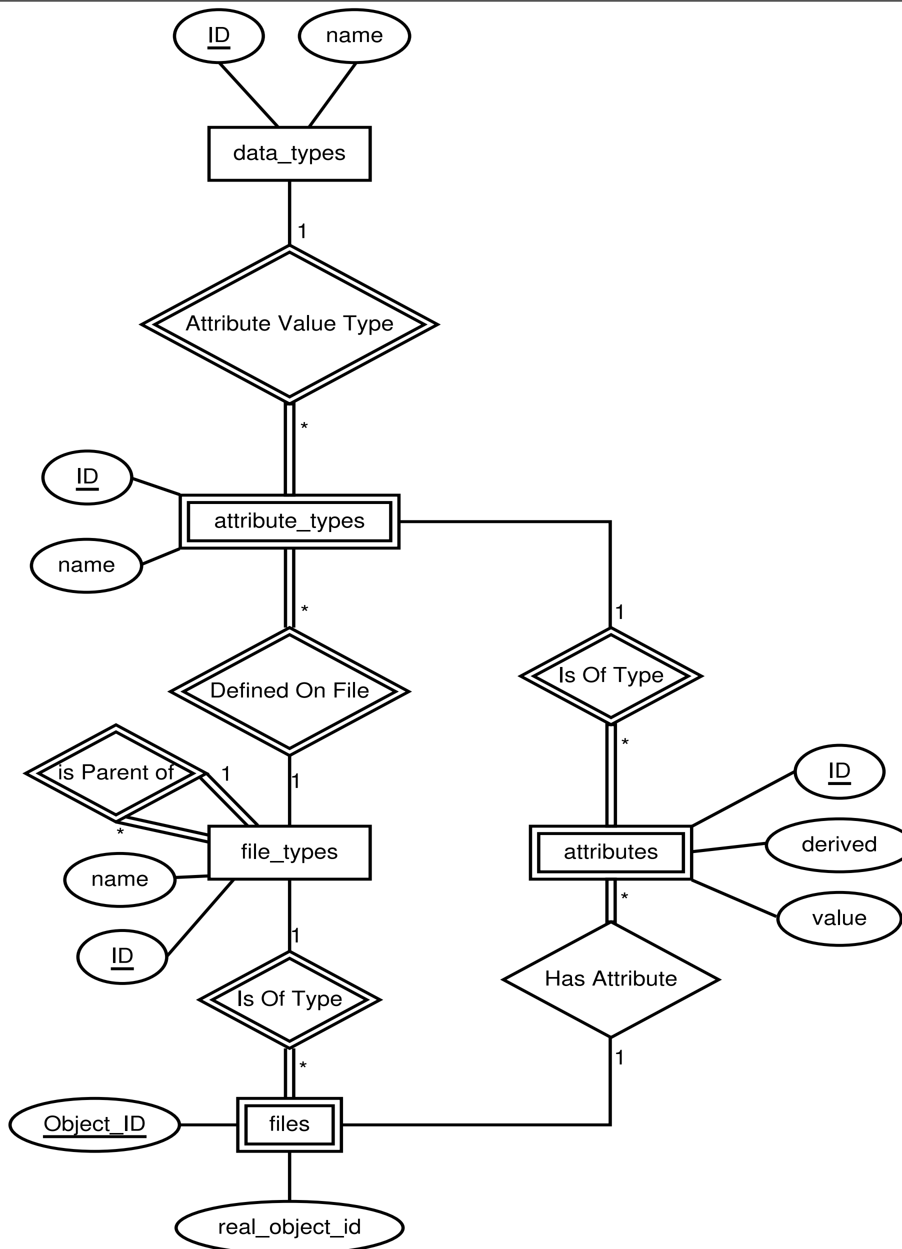
Figure 7.1 shows an ER diagram of the database design. This is in fact a simplification for several reasons. Firstly, you will notice that the file data itself is not present in the diagram. This is because we have chosen to use Postgresql's Large Object[1] support. This presents an API entirely external to the usual SQL interface. The advantage is that it becomes possible to do raw seeks, reads and writes on the Large Object which would not be possible with SQL. Every Large Object is given a unique Object ID. It is this Object ID that you can see as the key attribute in the *files* relation. The alternatives to the Large Object API is to either use byte arrays in the database which is impractical as the entire byte array has to be read into memory during an SQL operation which is hardly practical for massive files, or to simply use numbered files on the server. However, as far as possible we wish to avoid the use of the traditional file system and if we did use numbered files then we would loose the ACID[2] properties that Postgresql ensures via the Large Object API.

Unfortunately we discovered quite recently that Postgresql's Large Object API does not support the truncation of files. Thus if you wrote a file, then wrote over it, you would have to guarantee that the new data was at least as long as the old data otherwise you would have the end of the old data appended to

[1] `<http://www.postgresql.org/docs/7.4/static/largeobjects.html>`

[2] A mnemonic for the properties a transaction should have to satisfy the Object Management Group Transaction Service specifications. A transaction should be Atomic, its result should be Consistent, Isolated (independent of other transactions) and Durable (its effect should be permanent).

Figure 7.1 . ER diagram of the database design.



the new data. The only solution to this problem is to delete and create a new file whenever you write to a file. This is the reason for the `real_object_id` attribute of the *files* relation. Had we known this earlier, the `object_id` attribute would have become a simple increasing numerical `id` and then the `real_object_id` attribute could be renamed to `object_id`. This limitation is not anywhere near as limiting as might first seem. Analysis of file access patterns shows that random access to files is rarely used and sequential access is far more common. This is even more pronounced for writing files than for reading files and it is also extremely common when writing files to rewrite the entire file rather than a small portion of it.

Secondly, Postgresql supports seven base data types and whilst a type hierarchy does exist, it doesn't go as far as presenting an overall data type which can be used to store any data.[3] This is a great shame as it results in the necessity to define seven `value` attributes on the *attributes*[4] relation in order to be able to store all possible attribute values correctly. For example, a `size` attribute would obviously have numeric data, a `creation` attribute would have a time stamp and a `thumbnail` attribute would have

[3]As opposed, say, to Java, where every object can be cast up to the single root Object class.
[4]It is confusing that relations have attributes and that we also have an attribute relation. To aid understanding, all relation names are *emphasised*.

a byte array. There is no single data type in Postgresql which is capable of storing all these different values, so the inevitable solution is to have `value_numeric`, `value_time_stamp`, `value_bytea` etc attributes on the *attributes* relation.

Whilst SQL is a standard, there are no two database products which implement SQL in the same way. To reduce dependencies on any one database product much of the heavy SQL is performed by *Stored Procedures*. These are functions which are loaded into the database. They can then be called as a normal function call in any SQL statement. Various languages are supported by Postgresql for writing *Stored Procedures* including C, Perl and Tcl but for our purposes the PL/pgSQL[5] language is perfectly suitable. It simply moves the control logic for setting, removing and modifying attributes out of the Java-SQL side and into the database. Thus any database product that supports *Stored Procedures* can now be used, all that is required is the re-implementation of the defined procedures in which ever language is supported by the chosen database product. It also results in greater speed, efficiency, and as PL/pgSQL supports both recursion and function overloading, highly complex or impossible queries can be greatly simplified. For example, every file type has a parent file type. It is impossible to write a single SQL statement that can find all the parents of a given file type.[6] However, with a stored procedure, this functionality can be implemented and then used via a call to the *Stored Procedure* in an SQL statement.

There are three other relations that are required which are not included in the ER diagram. These relations are required by the cache management system which is discussed in Chapter 8, "Cache"

7.3 Implementation

My own opinion is that the hardest possible challenge in programming today is writing well-behaved multi-threaded programs. This is even more pronounced when dealing with concurrent access to a database.

In Java, the mechanism with which you connect to the database is specified by the JDBC[7]. This specifies amongst other things that each connection can only be used by one thread at a time. In addition, in practice there is a not insignificant overhead in creating a connection to the database. Therefore, a connection pool is recommended. When obtaining a connection from the connection pool, we must first look at the current thread. In the interests of efficiency, it would be sensible that if a thread executed different methods and each method requested a connection to the database that the same connection be assigned to the thread. However, database connections can be changed between *auto-commit* where each statement is immediately committed to the database and *manual-commit* where each statement builds up a transaction which is committed atomically. If the first method a thread visits obtains a connection, sets it into *manual-commit* mode, then calls a second method, the second method may well rely on the connection being as by default in *auto-commit* mode. Thus each thread must in fact have a set of connections so as to be able to ensure that when each thread asks for a connection it can be assigned a connection which is in the default state of *auto-commit*. Thus what seemed at first like a simple connection pool turns out to be very complicated in detail.

When we update a tuple in the database, the database system will obtain various locks. When the update or delete affects several relations, a particular type of lock must be obtained on each relation referenced. When the relation has foreign-key constraints[8] there are further requirements on the locks obtained. Yet more constraints are required when the relations have triggers defined on them. As a result of all this, it is entirely possible to deadlock the database due to locks being acquired in different orders by different transactions. To make matters worse, the query optimiser can decide to optimise a particular query under particular circumstances, potentially altering the order in which locks are obtained. Postgresql detects deadlocks and aborts one of the transactions causing the deadlock. This causes the JDBC to throw an exception which we can then catch. The Postgresql documentation simply suggests to wait a little while and then to try the transaction again. This is the approach we take. Currently we simply sleep for 250 milliseconds, but with further work this should be replaces by an exponential binary back-off mechanism.

[5] `<http://www.postgresql.org/docs/7.4/static/plpgsql.html>`

[6] Note that SQL is not Turing Complete, whereas PL/pgSQL and all languages that can be used for *Stored Procedures* are.

[7] Java Database Connection.

[8] This is where you specify, for example, that the only valid values of an attribute in relation x must be present in a particular attribute of relation y.

Chapter 8

Cache

> Problems worthy of attack prove their worth by fighting back.
>
> —Paul Erdos

8.1 Purpose

To avoid the interrogation of every attribute of every file not only traversing the network to reach the server but also potentially traversing the network again to reach the database, server-side and client-side caching is employed. However, all caches suffer from becoming dirty: that is the data they contain becomes out of date and should be invalidated. How best to do this quickly becomes a complicated affair hence the separate discussion of these problems outside of the server section.

8.2 Existing solutions

Existing solutions can be seen from two view points: firstly, solutions that deal with file systems and secondly from solutions that deal with databases. The reason that database solutions are appropriate is because we are effectively implementing the file system within a database, so solutions that solve similar problems for databases in general may be appropriate to this situation.

The basic problem is, if two clients are looking at the same file and one client modifies an attribute of that file then how is the other client informed. Various solutions have been employed in existing systems.

8.2.1 File Systems

8.2.1.1 Network File System

The Network File System (NFS) does client-side caching. However, there is no concept of cache-invalidation, so if two clients are viewing the same file and one of those clients updates the file, the server makes no effort whatsoever to inform the other client that the file has been updated. It is up to the client to poll the server to make sure that its cache is not out of date. Typically, NFS clients poll the server between every 3 and 30 seconds depending on how frequently the file is updated.

Whilst this approach is perhaps acceptable for files that are only ever limited to a very small set of users, it quickly becomes ridiculous when the file's contents or attributes are changing rapidly. Consider a single log file that is being written to by many clients. The resulting performance degradation would be unacceptable. This solution is nevertheless simple to implement and generally performs well in real life because few files which are shared between users are updated frequently by multiple users.

8.2.1.2 Andrew File System

The Andrew File System (or AFS) was designed from the ground up to deal much more successfully with high workloads. As such the caching design is somewhat different.

When each client wants to access a file, the entire file is transferred to the client from the server. That way, access to the file becomes local[1] and only writes to the file need to be committed up to the server. In addition, the server guarantees to inform the client if any other client modifies the file, thus all clients are informed if their cache of the file becomes dirty. This can roughly be viewed as an implementation of the observer pattern: each client subscribes to events relating to each file they cache. To avoid excessive delay in opening large files due to the need to copy the entire file to the local machine, for files bigger than 64KB, only 64KB of the file is ever cached at one time.

Whilst this works much more successfully than NFS, the overhead of subscribing and unsubscribing from each file's events induces a significant load on the server.

8.2.2 Database Solutions

In databases, in order to reduce loading on the server, clients maintain their own cache of the database. However, the same problem arises: if two clients have a cache of the same tuple and one client updates the tuple, how is the other client informed?

There is the simple observer-pattern subscription based solution available where the server keeps track of which clients have cached which tuples and then distributes notifications to clients informing them when their cache becomes dirty but, as with AFS, the overhead of maintaining such a map in the server quickly becomes prohibitive. Consider maintaining such a mapping for several million tuples. The memory requirements of such a mapping can quite easily exceed the memory requirements of the tuples themselves! Such approaches have been investigated in [WN90], [WR91] and [CFZ94].

Instead, if we look at how client caches are populated, then we quickly see that they are populated by the results of a query that the user performs. Thus all the server needs to remember are the details of the query. Upon each modification to the database, the server can calculate whether each query result has changed and inform each client as necessary. The overhead of maintaining a detailed map of tuples in client caches is removed.

The manner in which the possible modification of the query results are calculated has been the subject of much research. Some algorithms that have been proposed are efficient in that they can benefit from large amounts of pre-computation but are otherwise inflexible and suitable only for predetermined queries. Once such algorithm, the *counting* algorithm is discussed in [GMS93]. This works off a differentiation of the query which is complex to calculate and at each stage in the evaluation is required to be able to access the set of changes to the table, the table prior to the changes and the table after the changes. Whilst this may be possible within a database product implementation, it is not possible for a user of a database product.

Other algorithms, for example suggested in [KB96], use an amalgamation of query predicates to maintain the cache. However, as with many of these papers, very little consideration is made to the actual implementation.

8.3 Design

By analysing the Client-Server API, it can be seen that the only possible modifications are:

- A modification to an attribute could remove the corresponding file from the query's results.
- A modification to an attribute could add the corresponding file to the query's results.
- A new file could be created which is a valid result for the query.
- A file could be deleted that was previously in the query's results.

If we maintain a table which has view query IDs and view query results then it would be possible to calculate the changes. As the file that is affected is known in advance (ie the attribute value update is linked to the file upon which the attribute is defined), this can be used as an effective filter to the queries. Thus all that is required is to perform the view query SQL query with the added condition restricting the file's *object ID* and compare the results with the stored (cached) results. If the file is in the results of both the query's new results and the query's old results then the client who registered the view query must be informed that the modified attribute has been updated. If the file is in the query's new results

[1]That is, the file is physically accessed from a local disk, hence the performance is as good as you would expect with a file on a local disk.

but not in the query's old results then the file has become a member of the results of the view query where it was not before; and if the file is not in the query's new results but is in the query's old results then the file has ceased being a member of the results of the view query where is was before. If the file is neither a member of the old or new query then no action should be taken.

Note that it is not necessary to store the previous values of attributes in the query results table, only the file identifier. The draw back to this is that, upon notification that the file's attributes have changed, the client has no choice but to re-fetch all the attributes of that file. This inefficiency is however considered acceptable as it massively simplifies the server's workload. A possible solution to this problem would be to have the query results table contain the `id` of the last attribute belonging to the file to be updated. However, if several updates were performed in quick succession it would be very easy for updated attribute values to be ignored leading to invalid client caches. In practice this limitation of having to fetch all attribute values is small. This is due to the design of the client-server protocol which transfers many attribute values in a single object and the extensive attribute caching on the server eliminating the need to query the database for such queries.

To avoid the overhead of making Java do this calculation via the SQL interface, database *triggers* are used. A trigger is applied to a table and fires a function.[2] The conditions for the trigger can be set to any combination of `INSERT`, `UPDATE` and `DELETE`, the trigger can be fired either once per SQL statement or once per row modified. Also, the trigger can be fired either after the statement has been executed or before it has been executed. For our purposes, we define the trigger on the *attributes* and *files* relations, to fire once per row after the statement has been executed on all `INSERT`, `UPDATE` and `DELETE` statements.

The trigger is supplied with the new row for `INSERT` and `UPDATE` statements and with the old row for `DELETE` statements. From this, the trigger function can extract the file's object ID and use that to complete pre-defined SQL statements in the function body which then calculate the changes. It then puts the changes into a table which a Java thread examines when prompted by attribute value changes and file creations and deletions. This Java thread then commits the changes to the view results table and informs the relevant clients. There is one trigger for each *view query*. This is an unfortunate limitation. It would be nice for the trigger to be able to inform the server directly of any necessary query results changes. Because you can write the functions in Perl and Perl supports network connections, it would be possible for the trigger function to open a connection to the server to inform the server of the view query ID and attribute ID. However, this would require the definition of another protocol (albeit a very simple one) and further work to deal with the socket connections. Given more time, this would be implemented as would allow quickly propagation of changes to view query results to the clients.

Because the SQL view query statements are limited by the file *object ID*, the performance of these queries is greatly improved. This allows many dozen view queries to be defined resulting in an equal number of triggers being defined on the *files* and *attributes* relations without causing noticeable performance degradation.

8.3.1 Cache replacement policy

In order to cache file data itself, it is necessary to have somewhere on the client to store the file data. Obviously, RAM is not suitable as it would quickly become exhausted. The traditional solution, as taken by AFS, is to use the hard disc and store files there. However, we want to avoid any dependency on the traditional filesystem so therefore we would have to install a database product on every client and use that. This is currently considered excessive and would detract from the project. In particular, if you were going to simply use the system on a single user workstation with no multi-computer installation, you would have to have the database on the computer for the Server and again for the Clients. This would be extremely inconvenient.

However, one possibility would be that the Client could take advantage of a local database if it was configured that way. This would help performance over a slow network or with a heavily loaded server but would not be a requirement. Such a solution may be an effective compromise, is subject to more research and would require much more time for implementation.

As a result of the current non-existence of file data caching, there is no need for any cache replacement policy. If file data caching on the client ever becomes a possibility then this will become an issue and a solution will have to be found.

File attributes and the general hierarchy of files and their attributes, file types, attribute types and data types are cached in memory and are not considered large enough to exhaust memory, so again, no cache replacement policy is considered. This may change with testing if it becomes clear Clients are

[2] This is just a normal stored procedure function as discussed in Chapter 6, "Server".

running out of memory, however even if this is the case, it is difficult to find an acceptable solution because it is difficult to find out when Java is about to run out of memory. The only thing possible is to catch `OutOfMemoryError` exceptions and remove items from the cache when caught. The problem is that you would have to take this approach with every object creation which would be difficult and it would then be very tricky to get back to the part of code that was being executed prior to the exception. More research is necessary.

Chapter 9

Client

> We don't see Windows as a long-term graphical interface for the masses.
>
> —Lotus Development official

There are a number of features and utilities that all client applications will typically make use of. Among them is drag and drop, documented in Chapter 10, "Drag and Drop". The others are documented here.

9.1 Connection Dialog

The connection dialog is used by the first application started by the user. The user can select the host to connect to, and specify a username and password.

Figure 9.1 Connection Dialog

9.2 Launch Preference System

The launch preference system is the method MWFS uses to associate file types with the user's preferred application for viewing them. The first time a user launches a file, or files, of a particular type, the system will bring up a dialog from which the user chooses the application they wish to view the file(s) with. The association is saved to the database in a file of type *launch*, which contains lines associating the name of the file type with the *File ID* of the application to be launched.

9.3 File URLs

URLs are used to identify individual files within the filesystem. They are used by the Drag and Drop mechanism when saving files and the application loading system when loading Java classes directly out of the filesystem. They are specified in the form of `mwfs://server/file-object-id`. As such, they are very simple to parse. Upon receiving a URL, the client application simply needs to obtain a

Figure 9.2 Launch Dialog

connection to the indicated server and can then create a *view* which simply specifies the file object ID. The *view* is used for cache management purposes: without it, there would be no indication from the server when the file's attributes are updated or when the file itself is deleted.

Java's URL subsystem can be very easily extended. One must simply create a `Handler` class in the `url.<protocol>` package where `<protocol>` is the name of the protocol to be handled, in our case `mwfs`. Once created it is a simple case of appending to the Java runtime system's search path for URL handlers.

9.4 MWFS Client Applications

Java features a *URL Class Loader*. This allows us to load Java class files from a URL. Thus with our URL Handler created, we can now load classes directly out of the filesystem. This is an awesome feature as it means that once we have a single file browser application up and running, all further client applications can now be loaded out of the filesystem. Therefore, other than for *bootstrapping*[1] purposes, there is no dependency on the traditional filesystem for client applications. The mechanism that is used to actually launch the applications is slightly complex and worth further discussion.

All MWFS Client Applications are required to implement a particular interface in one class. This interface, `IMWFSClientApplication`, has just one method: `public void start(Client c, IViewQuery vq)`. It is also required that the class that implements this interface has the class name recorded as the attribute value for the `launch class` attribute type defined on the `java` file type. In addition, this class must have a constructor that does not require any parameters.

When you launch an application in our filesystem, a new process is created. This first receives the connection details of the Server to connect to. To avoid the need to prompt the user for a username and password for authentication purposes, an already authenticated Client can ask the Server for a new random number to be associated with the authenticated user. This random number is passed back to the authenticated Client which then sends it to the new process's *STDIN* data stream. Thus the new process reads from its *STDIN* data stream the server host and this number. It connects to the server and presents this number which the server then looks up and finds to be associated with the authenticated user. The new process is now correctly authenticated with the server.

The new process then receives a *File object ID* on its *STDIN* data stream corresponding to the actual file in the database we wish to be executed. From this ID, a URL to the file is created which is then passed into a *URL Class Loader*. The name of the class to be loaded is, as required, the value of the `launch class` attribute. This allows us to cater for jar files as well as single class files. The *URL Class Loader* is then used to load and create an instance of the class (using the null constructor).

At this point the new process checks to see if there is any more data available on its *STDIN* data stream. If it is then it is assumed to be a serialised view query tree. This is de-serialised and a new view query is created using the provided view query tree. This is then passed along with the authenticated Client to the `start` method which is invoked via Java's reflection system. At this point, the launched application is now up and running.

The use of the required interface means that if the application is launched from the command line then all the body of the `main` method needs to do is create a new instance of the class and invoke the `start` method upon it. The expected approach is that in this case, the Client passed in is null which then causes the application to present the standard Connection Dialog to obtain and authenticate a connection to a Server. Furthermore, the `start` method should examine the View Query it is passed.

[1]This can be thought of as roughly similar to the base case in a proof by induction: consider writing a compiler for the language *X* by using the language *X*. This is completely acceptable: the *GCC* compiler is written in *C* and compiles *C*. The only problem, and this is the *bootstrapping* problem, is if you don't have a *C* compiler to start with, how do you compile your compiler?

If it is null then it need take no further action. If however it is not null then it should be examined and appropriate action taken. For example, the Audio Player (Chapter 13, "Audio Player") will add all the results of the view query to the play list, whereas the text editor will open a new window for every file found in the results of the query.

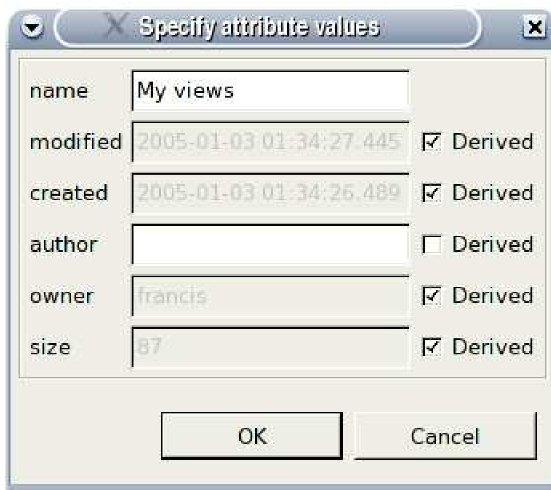
9.5 Attribute Value Editing

Once a file has been created it is obviously required functionality to be able to edit the attributes defined on a file, and to define new attributes for *attribute types* which are defined on the *file type* of the file but have yet to be used for *attributes* of the file. However, it is also extremely common to wish to edit common attributes of a set of files. This functionality is catered for in our system.

The *Attribute Editor* is constructed with a View Query. It interrogates the results of the view query, extracting the common *file type* for all the files in the results of the View Query.[2] For this *file type* the set of available *attribute types* is obtained. Then, for each available *attribute type* the set of files is examined to obtain each file's value for each *attribute type*. If every file has the same value for a particular *attribute type* then that *attribute* can be edited by the attribute editor. Also, if every file does not have a particular *attribute type* defined then that *attribute type* can also be edited by the attribute type.

Because the *Attribute Editor* was created with a View Query, it will immediately be informed of any changes made either to attribute values or changes to the results of the view query. At such points, the *Attribute Editor* reanalyses the results of the View Query and updates the display as necessary. Therefore, the user can see the results of the editing of the attributes in real-time.

Figure 9.3 Attribute Editor



9.6 Loading Utilities

We wrote some utilties to import large numbers of files into the filesystem and automatically generate appropriate attributes for them. We also wrote an email handler that processed emails as they are received so that they can be automatically added to the filesystem.

9.6.1 File Loader

The file loader must be capable of detecting the file type from the file data. This is necessary as some platforms do not support file extensions[3] and it enables file to be loaded from any source, such as an `InputStream`. This can be achieved with reasonable accuracy by reading the first bytes of a file and along with a database of known magic numbers the file type can be derived. The `file` command

[2] The algorithm to do this is too simple to be worth including here.

[3] These ought not to be trusted as the filesystem could attempt to process it incorrectly and frequently files have incorrect file extensions

on linux provides this functionality to files on a standard filesystem and to maintain an entirely Java solution some of this functionality had to be reimplemented.

With the XML formatted magic numbers from the JMimeMagic[4] project we built a detection class that returned one of our *base file types*. This was the bulk of the work required to implement the file loader as we could then simply write small applications to go through every file in a directory: detecting its filetype, creating a new file in the filesystem with this type and then copying the file data across. Once the file is in the filesystem the *File Modification Handlers* take care of deriving the attributes.

9.7 Email Handler

This utility can take emails that are piped into it and add them to the filesystem. This is very useful to put in your `.forward` file so any emails that you are sent can be automatically added to the filesystem. This is a standalone client application that connects to the server and creates a new file of type `email` and copies what it receives on its `STDIN` into the file on the server. The email's attributes are then added by the server as with the File Loader.

[4] Unfortunately this project was not complete enough for us to be able to use it without modification.

Chapter 10

Drag and Drop

10.1 Purpose

Drag and drop is used for loading and saving all files within the system. Every client application must support the Drag and Drop primitives.

10.2 Design

Almost all work we do on a computer system involves manipulating files in some way. Therefore, the actions of saving and loading files becomes incredibly important in terms of ease of use and efficiency. I have always been a fan of Drag and Drop since my days working with Acorn computers running RISC OS which used Drag and Drop extensively. I particularly object to the *Windows* saving mechanism where you are constantly forced to navigate a view of the file system always starting from a default state. This discourages people to organise and categorise their files. In contrast, saving implemented by Drag and Drop requires that you have one file browser type application open with the relevant view of the filesystem and then you simply *drop* every file you wish to save into this single view: after initially creating the view, no further navigation action is necessary.

Given that *views* in our system are active components which are always updated by the server as and when necessary, it seemed that being able to load a *view* was crucial and in particular, crucial to load the *view* itself rather than the contents of the *view* at any one point. This has profound implications. Consider your favourite music player. This will typically support a *play list* which will contain several songs. Each song will be played in turn by the application. However, whenever you download or otherwise obtain a new song, you must manually add it to the play list. If, when loading a *view* in our system we simply loaded the contents of the *view* at the time of loading, we would have the same behaviour. Instead, because we load the *view* itself, as further files are created, the client application into which the *view* has been loaded it informed and will update itself. Therefore, if we are ripping an audio CD, we initially set our audio player to play all files of the CD's author and album. It doesn't matter if initially there are no files in the results of the *view*: as the rip and encoding of the files progresses, the audio player will be informed of the changes and will automatically add the new files to the play list. This improved functionality is achieved because we are now passing *views* of the filesystem around as opposed to single files or lists of files.

10.3 Implementation

The Drag and Drop subsystem is implemented as four separate components allowing client applications to implement loading and saving actions as appropriate to the nature of the application:

- **Drag Load From.** When a drag action is detected, the Drag Load From mechanism extracts the current View Query from the client application in which the drag action was started and serialises the View Query Tree and the server connection details. These are then provided to the Drag Load To component.
- **Drag Load To.** When a drag-finished ("drop") action is detected and the corresponding drag start action was a Drag Load From, the Drag Load To component takes the passed data and de-serialises

to obtain the server connection details and the View Query Tree. It then checks that a connection to the specified server is available in the current client application and if so, builds a full View Query and passes it to the client application. The client application is then free to act on the provided View Query as it wants: typically this would involve either loading the files contained in the View Query or updating the current display in some way to reflect the contents of the View Query.

- **Drag Save From.** When an application wants to save a new file, the first thing it must to is create the file and write the contents of the file to the filesystem. As soon as this is complete, the file will be analysed and attributes may be automatically created as discussed in Chapter 5, "Metadata". Once this is done, the application pops up a simple dialog box with a single icon. This icon is then dragged and dropped into a client which implements the Drag Save To component. The data transferred in this Drag and Drop action is the URL of the new file. If the save is aborted, the application must delete the file. This may seem a strange order to do things in: why not delay the potentially expensive write to the filesystem until after the drop has completed? The answer is that the meta-data auto extraction can not function on an empty file: it must have the complete file available for analysis. Thus the writing is done first so that the Drag Save To component can present to the user the values of the extracted meta-data attributes and allow the user to override these if the user so wishes. We don't consider this to be a large problem as from analysis of file usage patterns, the sizes of the files that users create themselves tend to be small - emails, text files etc are not large and can be saved very quickly. It would only be the music editing and video editing type of application that would possibly suffer from this approach.

- **Drag Save To.** This component takes the URL of the newly created file, extracts the details of the server connection and obtains a suitable connection to the server. It then analysis the *file type* of the file and obtains a list of *attribute types* which are defined on the *file type* and displays these in a dialog box. For each *attribute type*, the current view query is analysed. If the current view query has an unambiguous restraint on the value of the attribute type[1] then the value of the restraint is suggested. If not, then if the file has an *attribute* value defined for the *attribute type* then that value is displayed. Otherwise a blank value is displayed. This means that if, for example, you have a file browser in which your view has a restraint on the value of a `project` attribute type then when you drag and drop into the file browser, this restraint will be reflected in the suggested attributes for the file. This reduces the number of attribute values the user is likely to have to specify manually. *Attribute types* can be *derived* in which case their value is specified by analysis of the file: for these *attribute types* the user can not alter the value of the *attribute* until they mark that *attribute* as non-derived. This can be done on a per-attribute basis to avoid the need to redefine the whole *attribute type* as non-derived. Thus if in general the meta-data extraction is correct for a particular *attribute type* but is only occasionally wrong, necessitating the *attribute* value to be set manually, this can be achieved. When the user confirms the attribute values the save operation is complete and the attribute values are created or modified as necessary.

[1] Unambiguous in the sense that it must be specified by traversing the view query tree without passing through an *OR* or *NOT* node.

Chapter 11

File Browser with Query Builder

11.1 Motivation

Almost every operating system that has had a GUI has considered it essential to include programs that use that GUI for the purposes of manipulating the filesystem. These programs typically allow the user to browse the structure of the filesystem, as well as searching for particular files. In MWFS, there is no distinction between these two activities. Browsing the filesystem becomes synonymous with searching, as the only way of locating groups of files is though the usage of searchable attributes.

Hence, it becomes obvious that in order to experience the benefits of such a filesystem, there must be a method for the user to interact with it. It must be fast, intuitive and be able to cope with the requirements of locating a file in a filesystem structured in this manner.

11.2 Design

The file browser will contain two important elements. These are:

- A query builder, the purpose of which is to provide an intuitive way to design queries to locate files within the filesystem
- A file lister, used to display the results of the queries, and capable of handling the intricacies of displaying files in a filesystem where paths do not exist, filenames and other attributes are optional, and file-types have a hierarchy

As the client will be written in Java, there is a choice as to which Java GUI toolkit to use. The choices are:

- **Abstract Window Toolkit.** AWT was designed for simple applets. It wraps the native GUI widgets in a platform independent API. For this reason, only the basic building boxes of GUIs, such as text boxes and buttons, are supported. This makes developing complex GUIs in AWT very difficult.
- **Swing.** Swing is one of the most complex GUI frameworks ever created. It supports a wide range of GUI components. Instead of using native GUI components like AWT, Swing paints its own components using primitive graphics operations. Furthermore, Swing's architecture uses the Model-View-Controller pattern of GUI component interaction, making it extremely flexible, although Swing can have poor performance at times.
- **Standard Widget Toolkit.** SWT is the toolkit developed by IBM for its Eclipse IDE. SWT is implemented using native widgets, giving it better performance than Swing, and emulates widgets if they are unavailable, avoiding the restrictions imposed by AWT. As SWT is platform dependent, each OS running an application using SWT will require the appropriate libraries.

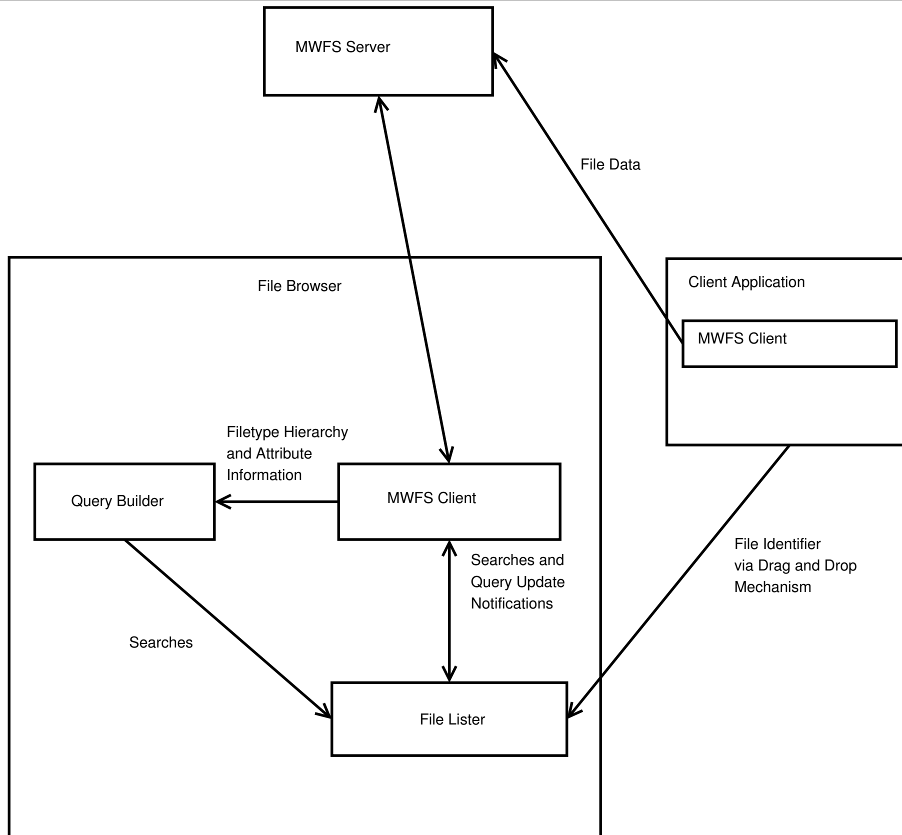
As the file browser needs to be fast, and most likely will have an interface requiring complex widgets, it was decided to use SWT. Although SWT libraries are platform dependent, they are maintained for a number of OSs, allowing the file browser to run on a number of platforms.

Furthermore, JFace, a platform independent API which inter-operates with SWT, can be used. JFace provides components and utilities designed to simplify common problems encountered while programming user interfaces in SWT.

11.2.1 Overview

Figure 11.1 details the interaction between the client, the server, the file browser and the client applications. The file browser possesses an instance of the client, through which all communications with the server are performed. The query builder is used to design searches for files, and the file lister to used display the results. The results of the queries are relayed to client applications for the purposes of saving and loading via the drag and drop mechanism.

Figure 11.1 Overview of file browser design and interaction



11.2.2 Query Builder

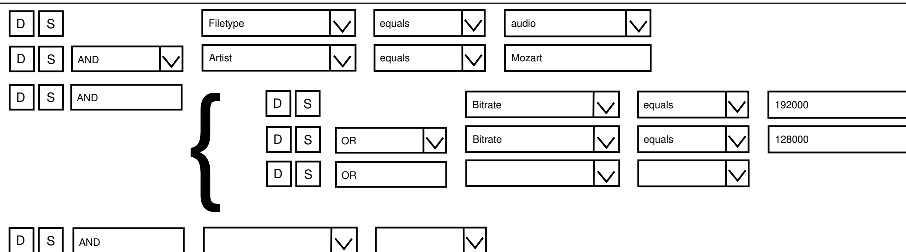
Queries are submitted to the server in the form of a *view query* which although capable of handling queries flexibly, use binary operators, which result in even simple queries translating into a complex graphical representation. A user unfamiliar with expression trees would be unable to comprehend the meaning of a query with a direct correspondence to the *view query* tree. Instead, the query builder will treat ANDs and ORs as n-arity operators, instead of binary, which allows for a more compact and understandable display representation, without loosing any of the meaning of the query.

Figure 11.2 shows the planned design for the query builder. Each clause contains expressions which define constraints on the files found by the query. Each clause can contain further sub-clauses. The ”D” button will delete lines, and the ”S” button moves lines into a new sub-clause. For each clause, the user specifies if all constraints in the clause should hold, or alternately, one or more, using the combo box to select the ”AND” and ”OR” operators.

As all lines in a clause are joined by the same operator, there is no need for controls to reorder the lines. New lines are automatically created in each clause when a valid line is completed, negating the need for a line insertion control. Note, although it is technically unnecessary, all lines in a clause are prefixed by the operator joining them, improving query readability.

Loading and saving of queries will probably be done by converting the query into an XML representation, as parsers already exist for Java to handle this.

Figure 11.2 Planned query builder design



11.2.3 File Lister

This is responsible for displaying the results of the queries built. The lister will display files in a table with each row in the table representing a group of files. Files are grouped by two things, their file-type, and the attributes the user has decided to display. Hence, each row in the table represents a group of files that all have the same file-type and have the same value for all the attributes the user has decided to display. Hence, as the user chooses to display more attributes, the number of groups become larger, as the information available to distinguish files from each other increases.

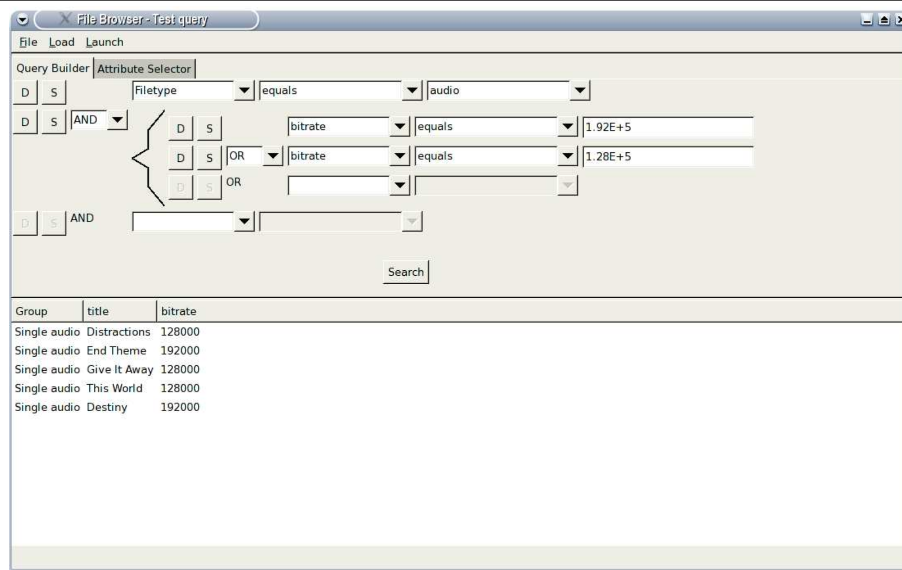
The user will also have the option to click on particular groups of files, in order to refine their search, delete files, and open them is a selected application.

The file lister will listen to changes in the query allowing it to be instantly informed about files being added, removed and modified and update the listing accordingly.

11.3 Implementation

11.3.1 Query Builder

Figure 11.3 Query Builder and File Lister



The placement of controls onto the form is done using *FormLayout*s which allow the positioning of controls on a form to specified in terms of the positions of other controls. This extremely powerful way of laying controls out avoids the complexities of placing controls using absolute positioning

The user can select which query to display using the *Load* menu, however, saving and loading queries can also be done using drag and drop, covered in Chapter 10, “Drag and Drop” and Section A.1. The file browser also has the ability to start any applications loaded into the system though the *Launch* menu.

File-type and attribute data is downloaded from the server and used to provide the combo box controls with the appropriate choices for attribute types and data types. Also, information about attribute

data types is downloaded from the server in order to validate and format search parameters for attributes.

In order to separate the classes modelling the query, from those handling the display of it, a notify-listener model is used, in which the renderers displaying the clauses and lines register as listeners to changes in the model. Changes to the line renderer notify the line model of the change, and them the model notifies a wrapper class around the line renderer of the change, and changes the line renderer if the condition type has changed. The clauses also register as listeners to changes in the lines they contain so that they know when to create new blank lines.

Saving of the queries is done using *Serialisation*, a Java feature in which Java objects can be saved and retrieved, to and from disk, respectively. This allows for extremely simple code for saving and retrieving of queries. Instead of saving the model used by the query builder to disk, the *view query tree* is saved to disk, allowed for an application independent way of expressing queries. Conversion to and from this format is discussed in the following chapters.

Conversion to the *view query tree* format is relatively simple. Each line in the query can be directly converted to a *view query* node, or one prefixed by a *NOT* node. Clauses are converted by taking the corresponding view queries for each line, joining pairs with the appropriate *AND* or *OR* nodes, and repeating the process until all the lines are exhausted.

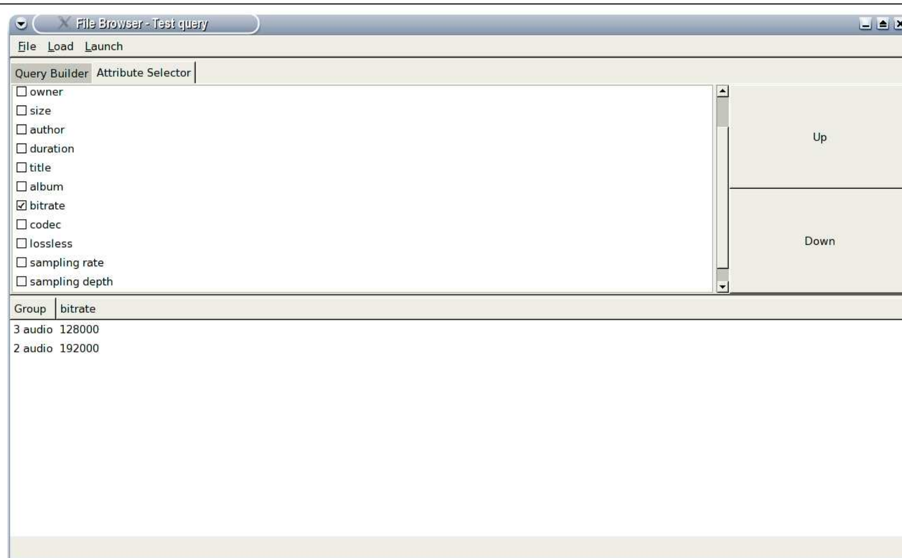
Conversion from the *view query tree* to the representation used by the query builder is slightly more complex. A visitor class, is used to visit the *view query tree*, which supports the *visitor design pattern*. This coalesces binary operators back into n-arity operators, and uses DeMorgan's laws to handle *NOT* nodes at any location in the tree.

Handling references to other queries required an extension to the *view query tree*. A new node was created which contains the name of the query being referenced, and the interface for the *view query tree* visitor extended to handle the new node. Query trees saved to disk may contain this node, but any query trees sent to the server are rebuilt so that the nodes are replaced which the appropriate query's *view query tree*. Errors due to missing queries or cyclic dependencies are detected and an appropriate error message given.

Additional: The extensions made to the view query tree have since been incorporated into the client, allowing all client applications to handle queries which refer to other queries.

11.3.2 File Lister

Figure 11.4 Attribute Selector and File Lister



The file lister is shown in Figure 11.4. The user can select which attributes to display though the usage of the attribute selector, and the files are grouped appropriately.

The file lister implements the drag and drop mechanism, making it possible for file groups to be dragged from the lister onto other applications and for applications saving files to drag onto the query and have attribute values for the save to be filled in automatically. Consult Chapter 10, "Drag and Drop"

for further information. The lister also supports opening files, deleting files, and opening file groups. Consult Section A.1 for further information.

When the user opens a file group, they should be able to keep track of the what files they are now viewing. As each file group knows what file-type its files are, and the attribute values a file must have to be in the group, the file group can be used to create a new query showing only the files in that group. A local *view query tree* is created by the group, and joined with an *AND* node to the existing query. This new query tree is displayed in the query builder, and the results displayed in the lister. Hence, as the user expands file groups while refining their search, the query displayed by the query builder is updated to match.

For all files being displayed by the lister, the lister determines the file-type common to all files and only allows the attributes on these files to be selected for display in the lister. For each file group in the lister, the most specific common file-type is calculated and shown for each group.

The file lister implementation was made significantly simpler though the functionality available in JFace. The *TableViewer* class allows *FileSet*s (the class responsible for placing the files into *FileGroup*s) to be set as an input to the *TableViewer*. An instance of the *IStructuredContentProvider* interface then supplies the *TableViewer* with the *FileGroup*s and registers as a listener with the *FileSet* so that it can be informed about additions, changes and removals of groups and update the *TableViewer* appropriately. The *FileSet* in turn has registered as a listener on the query for addition and removal of files or changes in their attributes. An instance of the *ITableLabelProvider* interface is used to supply column values for the table, given *FileGroup*s. The abstractions provided by JFace also lend themselves to changing the viewer in future, easing the task of adding a different viewer, such as a more graphical one.

Chapter 12

File Browser without Query Builder

There is no way to satirize a map. It keeps telling you where you are. And if you're not there, you're lost. Everything is reduced to meaning. A map may lie, but it never jokes.

—Howard McCord

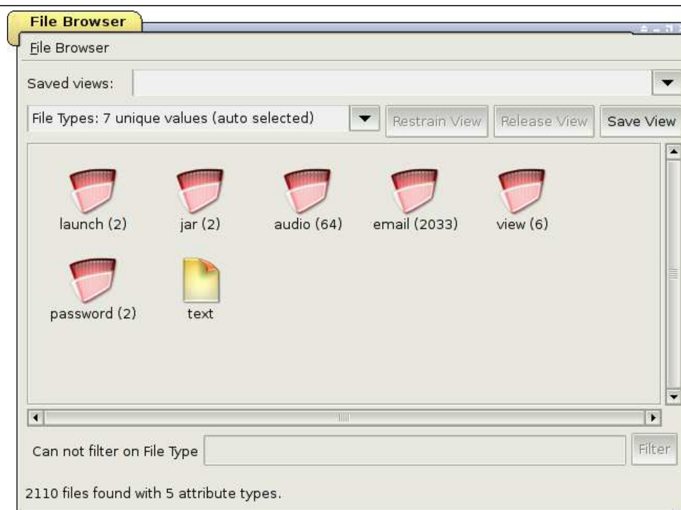
12.1 Purpose

Whilst the file browser with query builder as discussed in Chapter 11, "File Browser with Query Builder" is very capable and, with a little practise easy to use, it nevertheless is a two-stage application in which you must first define the query and then define the attributes which you want to view. Users in general are not used to this design of a file browser and because there are no file browsers available for this type of a filesystem which we could analyse or model on, we decided to create a second file browser based around the kind of navigational mechanisms users are already familiar with.

12.2 Design

This file browser displays a single attribute type at a time. This can be chosen from a drop down menu and a particular attribute type will be automatically selected by the browser as the most suitable attribute type to display by default based around the number of unique values found for that attribute type. In addition it is possible to choose to view the file types of the files in the current view. Initially, the current view displays all the files in the system as can be seen in Figure 12.1

Figure 12.1 Default view of the file browser



The choice of the automatically selected attribute type is based around decision trees in AI. Whilst a large number of values would result in a very shallow decision tree, humans are rubbish at dealing with

lots of pieces of information at any one time. A small number would present the user with less choice at each stage and would therefore make life easier for the user but it may result in an inefficient system as the user is then required to make too many choices. As a result, the automatically selected attribute type is the attribute type which has, in the current view, closest to 25 unique values. Furthermore, a non-derived attribute will always be chosen over a derived attribute where a draw is detected. This is based on the grounds that for a non-derived attribute, the user will have had to have entered the value of the attribute manually and will therefore be more likely to be able to identify with that value.

Where a file has a unique attribute value for the current attribute type, that file is shown with a file icon. Where more than one file have the same attribute value, the attribute value is shown with a traditional "directory" icon and the number of files indicated in brackets next to the attribute value.

Arbitrary selections are possible by two methods: firstly by single left-clicking with the mouse, holding down shift will add the current attribute value to the selection; secondly by using the middle mouse button, you can drag a rubber-band around several attribute values, and again, holding down shift will add to the current selection. Once you have a selection, you can restrain the current view by clicking on the *restrain* button. This takes all the values of the currently displayed attribute type that you have selected and ORs them together and then ANDs that with the existing view. The display is then updated with the new results. Alternatively, you can double click with the left mouse button on any attribute value which is not unique to a single file and you will cause the view to be restrained with the value of the attribute type that you selected.

Furthermore, if the attribute type you have chosen results in a massive number of attribute values being displayed then there should be some way of filtering or searching through these attribute values. This catered for with the *Filter attribute values* component which takes the text you enter and uses it in an *attribute value contains (case insensitive)* restraint which it ANDs with the current view. Thus you can very quickly cut down a massive number of attribute values to only a few by use of this feature. This is very useful if you need to locate a file but can remember very little about it, for example you know who sent the email but nothing about when or what the subject was: you would select the *from* attribute type and find you are presented with possibly thousands of values. This you can very quickly limit by filtering on the information you can recall.

At any point you should be able to return to the previous view. This is achieved through the *release* button. Whenever you have any selection of attributes, you can click the right mouse button and you will see a context-sensitive menu. This will always contain an *Edit attributes* entry which is discussed in Chapter 9, "Client". It will also contain a *Launch files* entry which allows you to load the files which posses the attribute value(s) selected. This is discussed in Chapter 9, "Client". Finally, for selections which consist of attribute values which are unique to single files, the menu will contain a *Delete File* entry which allows you to delete the files (surprisingly). The only reason this is constrained to selections consisting of attribute values which are unique to single files is one of safety. We have not yet decided whether we should allow deletions based on attribute values which are common to many files. Time will tell whether this is too dangerous or not.[1]

Views can be saved through the *Save View* button. This pops up the standard Drag and Drop save dialog. Once saved, views can be loaded through the drop-down combo-box at the top of the file browser or by double-clicking on the individual view file in the file browser. This can be seen in Figure 12.2 which is currently showing the saved view "audio" and the attribute type `title`.

The file browser supports the *Drag Load From* component as discussed in Chapter 10, "Drag and Drop" so you can load the contents of a view or arbitrary selection by dragging from the file browser into an application which supports the *Drag Load To* component. This includes the Query Builder (Chapter 11, "File Browser with Query Builder"), which is extremely useful as it allows you to examine the exact conditions of the current view.

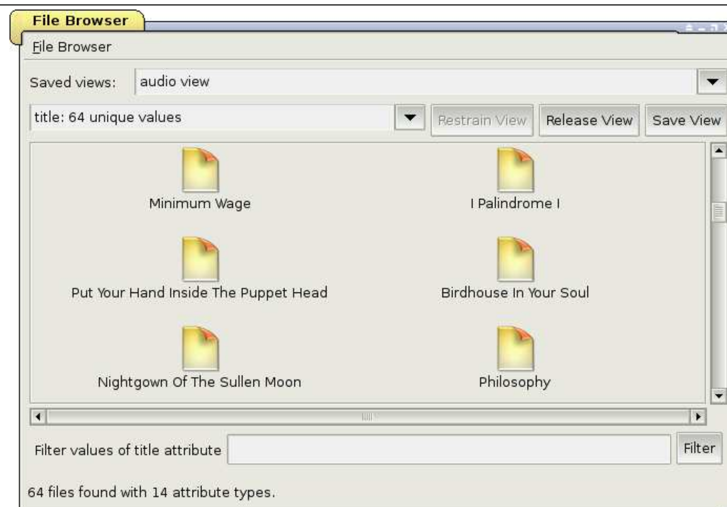
12.3 Implementation

As with the Query Builder, we decided to use SWT as the graphical toolkit. In fact, this has been used for all the graphical client applications giving a good consistent feel to all the applications.

Unfortunately there is no existing SWT component that allows for the icon and text string combination in the traditional graphical file browser manner. However, SWT does provide a general canvas onto which you can draw. This required us to perform analysis of the chosen attribute type to discover the maximum width of the text string representation of the attribute values so as to be able to layout the

[1] An amusing cartoon in a recent *Linux Format Magazine* issue ended with "...so your impromptu test of *rm* worked out well then!"

Figure 12.2 File browser showing the "audio" view and the `title` attribute type.



icons and text. We also had to deal with the scroll bars and resizing of the display. Additionally, SWT doesn't provide the ability to invert an image so we added that as well. Currently this only works with 24-bit images but this can be extended as necessary.

Given that we are drawing the entire canvas ourselves, there was work needed on the optimisation of the drawing routines in order to keep the application responsive even with very large numbers of attribute values to display. In this we are largely successful although further work can be done here. We also have to detect mouse clicks and match them to the attribute of the icon clicked upon as well as dealing with dragging rubber bands and selection logic. In all, it turned into quite a maths exercise and reminded us why we like nice high-level graphical libraries and layout managers which don't require any such work!

Finally, it is necessary to provide a context sensitive dragging mechanism and menu. Therefore, whenever you make a selection you are infact altering an internal view query so that when you request the menu, it is requested upon the selection you have made as opposed to the current view of the display. Similarly, when you drag a selection into another application, the view that you are dragging is the combination of the current view of the display and the attribute value(s) you have selected.

Chapter 13

Audio Player

Without music, life is a journey through a desert.

—Pat Conroy

The Audio Player was originally intended to be a full media player capable of playing video. To do this we were going to use the Java Media Framework[1] (JMF). However, some time ago, Sun removed their codecs from the JMF which catered for popular video formats and MP3 audio format. We have not been able to find free replacement codecs that are designed with the same mentality as the rest of the JMF has been designed: notably, the codecs that we have found have been restricted to playing files stored within a traditional filesystem only because they are simply wrappers around established codec libraries written in C and C++ and therefore use the C and C++ IO systems for reading the files. This is clearly not appropriate for our needs unless we were going to provide new IO libraries for C and C++ which would use our system. We came to the conclusion that this may have taken a little too long to implement. Therefore, the media player was abandoned and in its place an audio player was created.

Confusingly, Sun's Java libraries cater for sound through at least two different systems. Firstly, as already discussed, the Java Media Framework can deal with audio streams. However, again, Sun removed their MP3 codec from the JMF for licensing reasons. They have recently released a new MP3 codec after much discussion with the patent holder for the MP3 codec. However, this is limited in functionality and license. Secondly, there is the `javax.sound` framework. This is somewhat more successful because it only deals with sound and as a result is much simpler to develop with. As a result there are codecs for MP3 and Ogg Vorbis available for the `javax.sound` framework freely available from Java Zoom[2]. These we have used in the audio player.

The audio player is designed to demonstrate the full functionality of Drag and Drop and active view queries as discussed in Chapter 10, "Drag and Drop". The audio player has been a good way in which to test the reliability of the file access system: the `javax.sound` system simply takes an `InputStream` which it reads from and plays. This `InputStream` is in fact a `BufferedInputStream` which wraps a custom `InputStream` class written specifically for our filesystem. The buffering is highly effective in reducing the number of round trips to the server - each trip to the server fetches the maximum amount of file data permitted in a single query which is currently 64KB. This is then buffered and fed to the `javax.sound` system which has had no problems whatsoever in dealing with our custom `InputStream`. When the server is placed under high loads by several clients performing heavy rapid reading and writing of the database, it is sometimes noticed that the audio playback stutters as the buffers run dry, but this seems to be fairly rare. Increasing the buffer size would attenuate this problem at the expense of increased memory usage. On the whole however, this application works very well.

The audio player has the standard play list functionality. You can also use Drag and Drop in the *Drag Load From* mechanism as discussed in Chapter 10, "Drag and Drop" which allows you to get hold of the View Query that the audio player is currently using. Thus you can take the view query and drop it into a query editor, refine it there and then drag and drop it back into the audio player. This is a very nice way of performing alterations to the contents of the view query tree: certainly nothing as crass as selecting individual files to be added to or removed from the play list. It would probably be beneficial to add the functionality to save the current view query to a file from within the audio player: currently we have to drag from the audio player to a view query and from there save the view. Whilst inconvenient,

[1] <http://java.sun.com/products/java-media/jmf/index.jsp>

[2] <http://www.javazoom.net/>

it certainly reinforces the idea that the audio player is simply playing the results of a view query, with the added restraint that the files must also have the `audio` *file type*.

Chapter 14

Editors and Viewers

14.1 Introduction

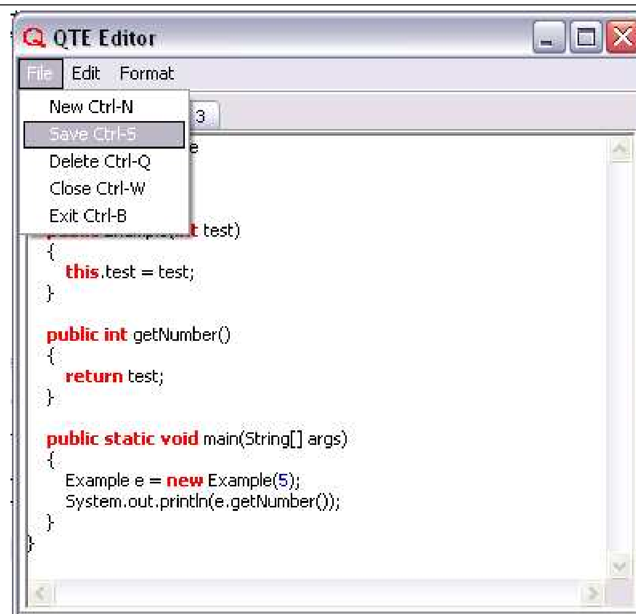
These tools and utilities are designed to make the users life easier by performing useful tasks such as populating the database with files and viewing files in the database.

14.2 Editors and Viewers

There are a number of applications designed to view images, text files, e-mails and play sound files. It is important to be able to edit text files since this is likely to be a very common task for users to perform. The viewers are by no means complete. The image viewer will only load a subset of the different image types, however, these viewers and editors will provide classes that could be expanded upon to produce new applications. All of the viewers have basic functionality, for example the text editor has the ability to create multiple text files, edit them, save them and then close them. Any files used by the editors/viewers are loaded from the database. For example, it can be seen that most of the editors/viewers have icons in the top left corner (similar to many applications in Windows) as a simple means of identifying them. These icons are image files loaded from the database.

An example of the text editor can be seen below. Figure 14.1

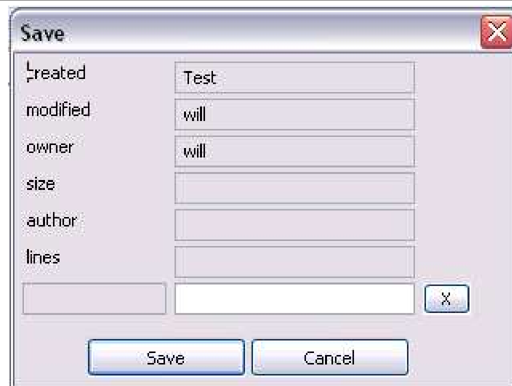
Figure 14.1 . An example of a text editor



Notice the icon in the top left. This was loaded from the database. The editor contains basic functionality including: save files, close files, create new files, delete files, exit application, format font, cut text,

copy text and paste text. The save mechanism used is the same as the method used for saving queries. A dialog box will be displayed as follows: Figure 14.2

Figure 14.2 . Saving a text document



An example of a simple image viewer can be seen. This purely has the ability to load, close and delete images. Figure 14.3

Figure 14.3 . An image viewer (image courtesy of digitalblasphemy.com)

14.3 Utilities

It is important for the user to move files they have created from the local file system (for example NTFS, the method by which the majority of files are stored in Windows XP) to the database file system. The transfer utility displays a simple save dialog so that the user can specify which attributes to give the file in the database, remember files aren't distinguished by there name, but by a collection of attributes.

To run this program use the command:

```
java -Djava.library.path=C:\eclipse\plugins\org.eclipse.swt.win32_3.0.1\os\win32\x86 Trans
```

Then follow the on-screen instructions. The first parameter is the SWT library (alter to point to the correct place).

14.4 Save and Attribute Dialogs

As already mentioned, with such important tools it is important to produce more than one to ensure the users will be satisfied. Two methods have been implemented for a save dialog, drag and drop is discussed in another section. The other method purely allows the user to enter the attributes in manually. It alerts the user if a value is entered over a derived field (the attribute may automatically be filled out by the server if not filled in). This method was implemented for two reasons. The first reason is that it is simpler to implement and was therefore implemented first just in case the drag and drop method could not be completed for one reason or another. The second reason is that it does not require a mouse, many users may prefer this. A second simple attribute selector was also created to test the file browser.

The save dialog allows users to fill in as many of the attribute values as necessary. If the attribute value is shaded grey, if it is empty it may automatically be filled in by the database. The last line of the dialog allows the user to specify additional attributes to add to the file type. For example lets suppose all text files now need to contain an additional attribute ”name”. The user would type name in the left box and the value in the right box. If the user makes a mistake he can simply delete the line. Clicking ’Save’ will add all of the necessary attributes and save all of the values. Clicking cancel will have no effect on the file (if the files does not exist it will not be created).

An example of a save dialog is shown below. Figure 14.4

Figure 14.4 . An example of a save dialog

The attribute selector allows the user to select which attributes to view on a group of files. As an example let’s assume that the user is viewing 5 audio files. It is unlikely the user will want to view every single attribute (which may not be filled in, the attribute may be undefined). It is more likely that the user will only want to view the title and artist for example.

14.5 A Simple Browser

14.5.1 The Browser

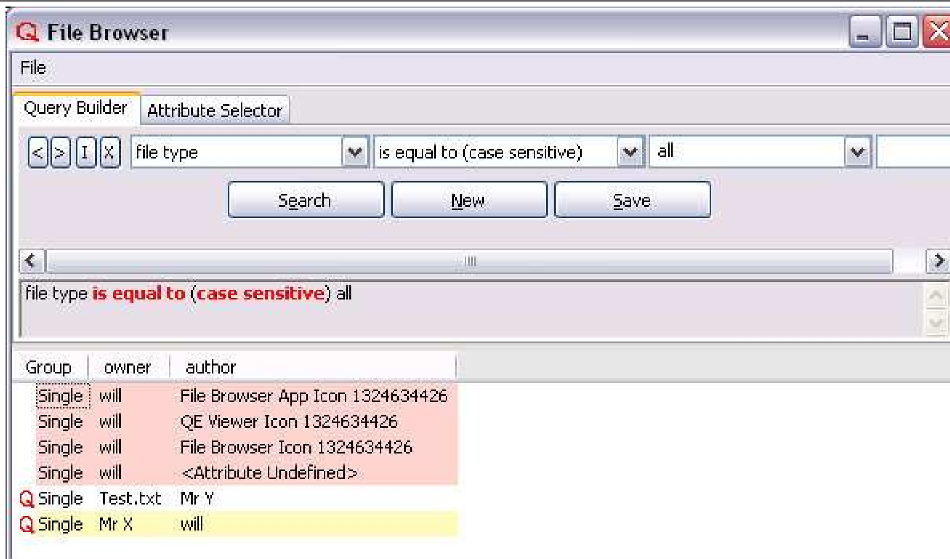
A simple browser is available to test these applications. Since loading applications from the database can be slow under very heavy workload, this browser loads the applications from disk directly. Everything that the editor uses whilst active (for example icons) is loaded from the database. This means that the files are still stored in the database, it is just the application that is not.

An example of the basic file browser. This is used to load a set of applications such as a text editor, an image editor, an e-mail viewer and a sound player. Figure 14.5

Some of the rows are highlighted red and some yellow. The yellow rows represent files that have been authored by the current user. The red rows indicate files that are owned by the current user. It can be seen that the text files have a ’Q’ next to them. This is just to distinguish them from other files.

When the search button is clicked the attribute selector is automatically selected (shown below). This enables the user to select which attributes to view on the files. Files with the same attributes are grouped together. For example if the user is only viewing the owner attribute, all of the files with the same owner will be grouped together. When the user clicks search for the first time, only 2 attributes are (by default)

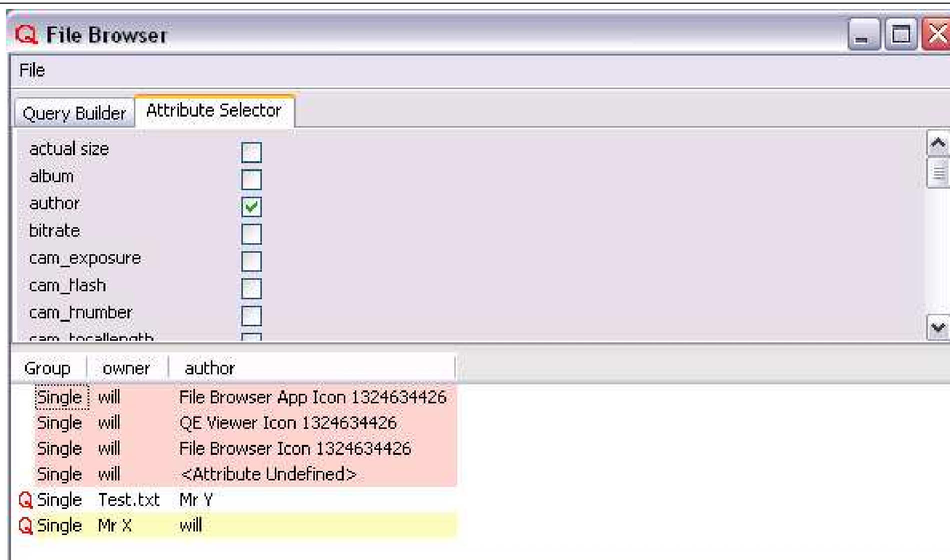
Figure 14.5 . An example of the basic file browser



selected, owner and author. The user can then select other attributes. If the user creates a new query, the set of attributes the user used for the previous query will remain selected, this is to try and aid usability.

An example of the basic file browser's attribute selector. Figure 14.6

Figure 14.6 . An example of the basic file browser's attribute selector



14.5.2 The Query Builder in the Browser

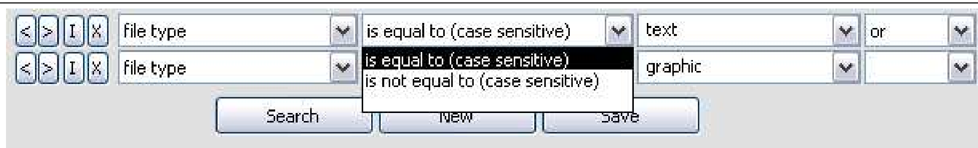
It was decided not to enable the direct typing in of any text string by the user to build the query. Instead, it was decided to allow the user to only type in correct syntax. This was achieved by using combo-boxes to restrict input. Thus when the user types into a combo-box the focus may move to a different combo-box automatically. This will enable the user to type phrases in without the need for a mouse, therefore speeding up input.

An example of how a query is entered is demonstrated below. Figure 14.7.

14.5.3 The Command Line

A textual representation of the query will appear on a command line. Using the example above, the resulting output on the command line will be: "file type is equal to (case sensitive) text and file type is

Figure 14.7 . An image of the Query Builder's combo boxes used for data input



not equal to (case sensitive) graphic". This command line output is largely to assist the user by providing an English description of the query. When the query builder is created, a reference to the command line object is passed to it. This means that when the query is asked for or the user edits the query the query builder informs the command line object and it is updated. In the following figure, the query can be created using one key press ('f'). This is because the query builder will automatically update as much as it can before informing the command line. The command line will update after every change to the query. This means that when typing a value in for an attribute, the command line will update letter by letter. Figure 14.8

Figure 14.8 . The query updating the command line

14.6 Windows XP style buttons

To enable modern Windows XP buttons as seen in some of the screenshots, simply create a new file and paste the following text into it. Rename the file 'javaw.exe.manifest' and place it in the jre bin folder (Java Runtime Environment). This is is ussually: 'C:\Program Files\Java\jre1.5.0\bin'.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" processorArchitecture="X86" name="XP.javaw" type
  <description>XP Buttons</description>

  <dependency>
     <dependentAssembly>
        <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" versio
     </dependentAssembly>
  </dependency>
</assembly>
```

Chapter 15

Evaluation

Funny sounds are not funny.

—Bart Simpson

Evaluating this project is a difficult task because there are no existing products available to which this project can in any way be compared.

- The created software on the whole works well. Where there are known deficiencies, these have been documented in the relevant section. In many cases, such deficiencies are a result of both lack of time and of not quite knowing if such a feature is desirable or necessary, so it may take us a little while for us to decide for certain whether or not a particular feature is a good idea or not.

- The server functions correctly. It has been very difficult to iron out some bugs. In particular the PostgreSQL deadlock detected bugs were very frustrating as were many race-condition bugs due to the multi-threaded design of the server. Deadlocks may still be possible in the server, but we have not suffered any for some while now. The server supports all the required functionality as per the specification. The performance is adequate but could be beneficially improved in some areas. However, in many cases the bottleneck appears to be PostgreSQL but proper analysis and profiling of the server will be required before any firm conclusions can be made as to the locations of performance problems.

  The design of the server is good and lends itself well to extensions. In practice the server behaves well and survives heavy usage without dropping connections or failing in anyway.

  One very interesting side effect of the design is that it's now quite easy to extend the system so that it will automatically version files. This is as a result of the fact that whenever you open a file for writing, the old version of that file is deleted. Instead, we could simply have attributes which record the *large object ID* of the previous version of the file and not delete the *large object* at all. A queueing type structure could, for example keep the last 10 versions of each file. There would then be some client API to allow the user to roll back to a previous version of the file. The only drawback to this is that it is the complete file that is kept as the previous version rather than the set of differences between the versions. This can quickly become very expensive in terms of disk space so an obvious requirement would be to be able to turn off this versioning functionality.

- Our decision to store all the attributes and files in the database was a good one as it made managing and querying the attributes easy using SQL queries. As the database server handled the indexing and complex queries on the attributes this meant that we were able to concentrate on other areas. However our decision to store the file data in the database as well did cause some problems: problems with PostgreSQL not truncating files, only having an *InputStream* available to access the data. An alternative method would have been to store all the file data as normal files on the file system and managing all the attributes and file typing using the SQL database; this would have provided us with an easier API to deal with when accessing files, and one which would have been potentially faster. The reason that this wasn't chosen to start with was the tidyness that we have gained by keeping all the data in the system in one place.

- The Meta-data automatic extraction is probably the major reason why this project could result in a major change in the way in which people interact with their computer filesystems. It works well and is extremely effective. It completely removes, in many cases, from the user, any necessity for

the user to enter any attribute values, or at least reduces it to no more than one or two attributes. Thus the user is not expected to type any more than they would be under a traditional file-and-directory filesystem. This is crucial. When you combine this with the fact that because the user would tend to keep a file browser open which shows the files they are working on and that saving is done with drag and drop into this file browser, the entire saving mechanism actually turns out to be more efficient and faster than with the traditional file-and-directory filesystem, despite all of the additional attribute values that are being kept. This is a huge achievement as it eliminates any burden of maintaining these attributes.

The design again has proven to be good, and lends itself well to extension. It would be useful to slightly alter the design so that the server, on start up, reads a configuration file in the database which then instructs it on which `File Modification Handlers` it should load and which `File Types` they should be associated with. This would effectively form a plug-in system with the ability for users to write, buy or download additional `File Modification Handlers` for particular `File Types`.

- The filesystem itself is extremely successful. The sheer number of features we have managed to implement is very satisfying. In particular, the ability to launch client applications from the database is something I never expected we would achieve, but it works well, and the authentication system is also very successful. The elegance of simply having a single file of the `file type` *password* which contains the hash of the user's password is extremely satisfying. More work is needed, particularly in terms of security: currently there is no means of restricting access to file-data or attributes. It is entirely possible that you would want to prevent other users from even viewing the attribute values of files in addition to preventing other users from having access to the contents of files. Currently all files have an *owner* attribute value created automatically, but to extend this into a general security system within the server would require more development.

- The file browsers are effective. These were perhaps the hardest to design as we really were working on a blank canvas with very little to guide us in terms of existing solutions. They have both turned out well and are suited for slightly different tasks.

  It would probably be beneficial to combine both file browsers into a single file browser which offers a choice of display types (list or icons) and both the attribute-value-selection method and the query-builder method of manipulating views of the filesystem. With the file browser in particular, they have only become fully functional relatively recently so there is some way to go in terms of refining and improving their behaviour and functionality. They are certainly highly effective as a first attempt at what is a very difficult visualisation problem.

- The audio player functions well though additional features would be welcome. In particular a volume control and the ability to pause the playback rather than stopping it would be an idea. There are also some known logic problems associated with the play-list under some circumstances. However, these problems or missing features were not considered important enough to warrant the necessary time as they didn't relate to the audio player in terms of its interactions with the filesystem and therefore were considered secondary to the purpose of the audio player which is to demonstrate interactions between clients and with the server.

  The design is very simple and should withstand fixing the existing bugs and implementing additional features with ease. It would be useful to obtain further plug-ins for the `javax.sound` system so that other audio file formats can be supported.

- The text editor again, for the same reason as the audio player, is rather lacking in features. However, again, as with the audio player, it is highly effective for what it was designed for which is simply a demonstration of the interactions between clients and the server.

  Depending on what we want to use it for, we would need to add lots of different features. It could be developed into a full email client. It could be developed into a general purpose text editor in the style of Vi or its numerous improved versions. It could be made into a complete kitchen sink like Emacs. Currently it simply doesn't have any real functionality other than to load and save text files.

Chapter 16

Conclusions

Whilst this project had a detailed specification, and has been evaluated in regard to this, this project was undertaken with the purpose of answering questions about the feasibility of such an unconventional filesystem. This chapter states some of these questions, and attempts to answer them as well.

16.1 Is it possible to implement?

A Postgres backed Java server was written capable of handling views on the filesystem defined by queries. The filesystem is capable of handing the storage of files, and their attributes. Client applications have been written that can use the the filesystem, and take advantage of its benefits. The implementation has been completed and functions correctly.

16.2 Is performance acceptable?

In tests, the system has certainly performed fast enough to be useable. However, the system has only been tested with a maximum of a couple of thousand files, whereas it is not uncommon for a conventional filesystem to hold hundreds of thousands of files.

With a filesystem no longer limited by a directory based hierarchy, it is very easy for users to create views which may contains hundreds of thousands of files, or even every file in the system. Users of a conventional filesystem may accept sorting through every file in the filesystem taking a number of minutes, however, tasks such as these in MWFS must nearly as fast as much simpler operations.

Testing has shown that initial retrieval of large number of attributes to be slow, but significantly faster after this data has been cached in the MWFS server. There are significant optimisations that can be performed at the server side in relation to the SQL statements used to retrieve data from the database. With these optimisations, we believe the system will scale effectively to handle a significantly larger number of files, and perform transfers of large numbers of attributes quickly.

16.3 Can it be made independent of the existing filesystem?

Although the Java virtual machine, and initial client application must be launched from the system's native filesystem, the ability to launch application held in jars or class files in a new virtual machine without referring to the native filesystem has been developed. With the correct bootstrap code, there needn't be any need for the system to use the native filesystem at all.

16.4 Is it a viable replacement for existing filesystems?

We have shown that the filesystem is capable of holding both files, and realistic user applications. The degree of dependence on a conventional filesystem that other client applications may have will vary, but we believe any problems regarding file system dependence may be solved. Programs with only minimal filesystem dependence should be easy to port. Programs with heavier filesystem dependence are probably inappropriate for MWFS, and a new restructured program should be designed if appropriate.

16.5 Was the project a success?

In terms of determining the answers to the above questions, the project was definitely a success, and even better, in terms of meeting the specifications, the project was also successful. We have proved that a metadata based filesystem is feasible, and in doing this wrote applications, utilities and the appropriate software to show how an implementation of this kind of filesystem could work.

Bibliography

[BPK97] *Performance Analysis of an Associative Caching Scheme for Client-Server Databases*, Julie Basu, Meikel Pöss, and Arthur M. Keller, 1997.

[CFZ94] *Fine-Grained Sharing in a Page Server OODBMS*, M. Cary, M.J. Franklin, and M. Zaharioudakis, 1994.

[GM95] *Maintenance of Materialized Views: Problems, Techniques, and Applications*, Ashish Gupta and Inderpal Singh Mumick, 1995.

[GMS93] *Maintaining Views Incrementally*, Ashish Gupta, Inderpal Singh Mumick, and V.S. Subrahmanian, 1993.

[KB96] *A Predicate-based Caching Scheme for Client-Server Database Architectures*, Arthur M. Keller and Julie Basu, 1996.

[WN90] *Maintaining Consistency of Client-Cached Data*, K. Wilkinson and M.-A. Neimat, 1990.

[WR91] *Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture*, Y. Wang and L.A. Rowq, 1991.

Appendix A

Userguide

In this appendix we have basic user guides for the client applications we have written. Some of these applications, as discussed in Chapter 15, “Evaluation” do not have the level of functionality you might expect. This is largely deliberate: the purpose of many of these applications is to demonstrate how various operations are achieved within our system rather than to produce yet another audio player or yet another text editor.

A.1 File Browser with Query Builder User Manual

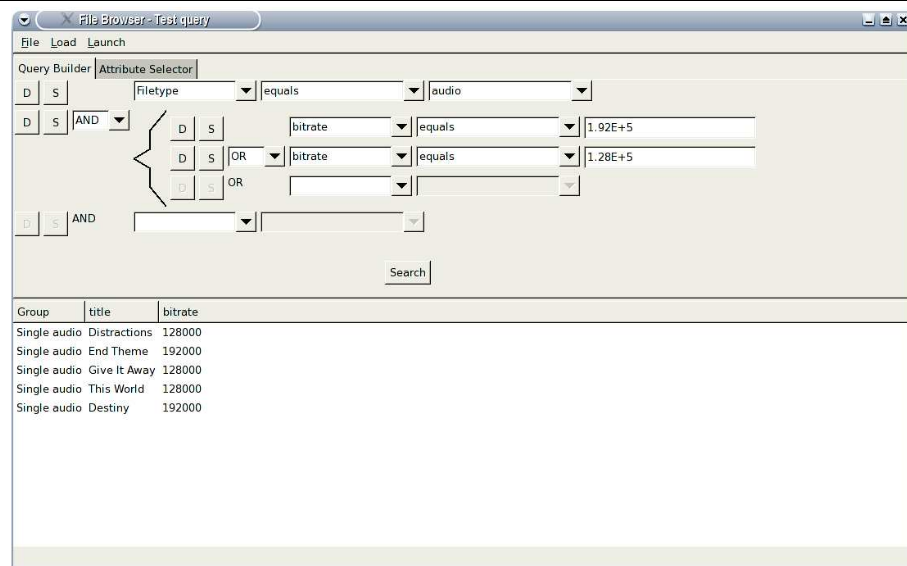
This guide is split into three sections that will tell you

- How to build queries
- How to load, save and run queries, and
- How to use the file lister

However, the file browser also possesses the ability to start other applications without having to give them a file, or files, to load. Simply choose the application you want to start from the *Launch* menu.

A.1.1 Building a Query

Figure A.1 Query Builder and File Lister



The query builder is shown in Figure A.1 The *D* button is used to delete lines or sub-clauses. The *S* button places a line into a new sub-clause, to which further lines can be added. When a line is correctly filled out, a new line will be automatically inserted.

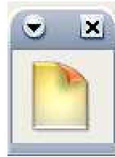
For each line, there are three boxes that must be filled out. The first is a combo box, from which must be selected what condition the line will be defined on. You can choose:

- **File-type.** You can then select whether the files found should be this file type or a child of it, exactly this file type, or the exact opposite of these conditions. The final combo box allows you select the file type.
- **File.** This option allows you to refer to other queries. The next combo allows you to specify whether the file should or should not be in the specified query. The last combo allows you to choose the name of the query.
- **An Attribute Type.** All attribute types are listed, and after specifying one, the next combo box will allow you to choose conditions appropriate to that attribute's type. The last box will let you enter values for the attribute and format them appropriately.

A.1.2 Saving, Loading and Running Queries

The *File* menu allows the creation of blank queries using the *New* command, and the option to save queries. To save a query, select the *Save As* option from the *File* menu. A new window will appear as in Figure A.2.

Figure A.2 Save As Window



This icon can be dragged onto any file browser displaying the results of a query. This will bring up a window to specify attribute values as in Figure A.3. Any future usage of the *Save* command will save any changes made to the query to this file.

Figure A.3 Attribute Editor

Specify attribute values
name My views
modified 2005-01-03 01:34:27.445 Derived
created 2005-01-03 01:34:26.489 Derived
author Derived
owner francis Derived
size 87 Derived
OK Cancel

Specify the name and any other attributes you wish, then click *OK* to save the file, or *Cancel* to abort.

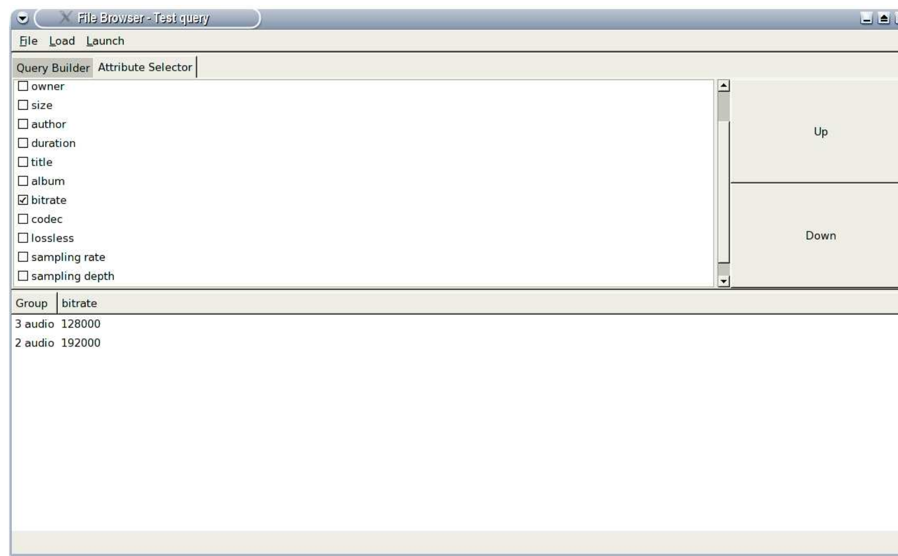
Loading queries is easy. Simply select the name of the query from the *Load* menu and the query will be loaded in the query builder, and the search run. Alternatively, it is possible to use drag and drop to load a query. Dragging a file group from the lister, or a search from another file browser, will cause

that search to be displayed in the query builder, and that search run. If the search contains only a single query, that will be loaded and run instead.

To run a query, click the search button underneath it, and the file lister will update itself with the results.

A.1.3 Using the File Lister

Figure A.4 File Lister and Attribute Selector



The file lister displays files grouped together by common file-types and attribute values. The attribute selector allows you to choose which attributes to show and to change the order that they appear in the lister.

If a file group contains a single file, or it is the only group being displayed, then double clicking it will cause the file(s) to be opened using the file launch preferences mechanism. If there is more than one group listed, then double clicking on a file group with multiple files, will cause the query builder to update itself with a new query to show only those files, and the file lister to display the files in that group.

Right clicking on any file group will bring up a menu that will allow, the files in the group to be opened, deleted, expanded, or have their attributes edited. Figure A.3 shows the attribute editor window, which is identical to the one used when saving files. Users can select whether attributes should be derived from the file or not, and if not, what their value should be.

A.2 File Browser without Query Builder

This file browser has been designed to be very intuitive to use and behaves similarly to a traditional file browser.

A.2.1 Displaying attribute types

With the file browser loaded, the combo box on the left hand side of the window can be used to choose the attribute type being displayed. Note that the only attribute types available are the attributes types available to the common file type of all of the files contained in the results of the current view. File Type can always be chosen to be displayed even though it strictly isn't an attribute type.

A.2.2 Filtering the attribute values

When the file browser is displaying a real attribute type (ie not the File Type), you can use the text entry box at the bottom of the window to enter any text string that you wish to constrain the attribute values

by. This is a case-insensitive arbitrary match - it will match against any part of the attribute values, in no way is it restricted to matching against the start or end of the attribute value.

A.2.3 Selecting attribute values

Use the left mouse button to click on an attribute value or its icon to exclusively select that icon. Click in a blank area of the display to clear the selection. Hold down shift whilst clicking with the left mouse button to add attribute values to the current selection. Use the middle mouse button to drag a rubber band around attribute values to select a group of attribute values. Again, using shift whilst dragging will add the group of attribute values to the current selection.

A.2.4 Restraining and releasing the view

With a selection made, click on the *Restrain View* button to add a conjunction of the disjunction of the selected attribute values with the current view query. Alternatively, double click on a non-unique attribute value (ie, not an attribute value which corresponds to a single file). Clicking on the *Release View* button will always take you back to the previous view.

A.2.5 Saving a view

To save the current view, click on the *Save View* button. The standard save file dialog box will appear. Drag this file onto any file browser and the standard edit attributes dialog will appear. Modify these as necessary and click on *OK* to complete the save action. The *view* combo box at the top of the file browser window will update to show the `name` of the view you have just saved.

A.2.6 Loading a view

Use the *Save View* combo box at the top of the file browser window to select the view you wish to see. Alternatively, double click on a unique attribute value (ie, an attribute value corresponding to a single file) of a file with the *view* file type.

A.2.7 Launching files

To launch a single file, simply double click on the unique attribute value. If you haven't launched a file of the file type before then you will be prompted to choose which application to launch the file with. Otherwise, the file will be launched with the application you chose (unless that application is no longer available). To launch a selection of files, with the selection made, right click. The menu that will appear will have a *Launch Files* entry. If you choose that entry, the files will be launched with the application associated with the common file type of the files selected. You will be prompted to choose an application if no such association currently exists.

A.2.8 Deleting files

You can not delete a selection containing non-unique attribute values for safety reasons. Every attribute value in the selection you wish to delete must be unique (ie corresponding to a single file). With such a selection made, right click. The menu that appears will have a *Delete files* entry. Upon choosing this, the files and all associated attributes will be removed.

A.2.9 Editing file attributes

With any selection made, click the right mouse button. The menu that appears will contain an *Edit file attributes* entry. Upon choosing this, the attribute editor will appear. This will allow you edit attribute types for the common file type of the selection where the attribute value for each attribute type in turn has the same value for every file in the selection. To remove an attribute, simply make the value of the attribute empty. You can not edit the value of a derived attribute. If you want to change the value of a derived attribute you must first make the attribute non-derived. The attribute editor can be seen in Figure A.5. Note that every attribute is marked as being derived - when importing this file there was no need to manually specify any attribute value as they were all extracted automatically from the file data itself.

Figure A.5 The attribute editor showing the attributes of an audio file.

A.3 Audio Player

The audio player can be launched from any file browser as a normal application or can be associated with the *audio* file type in which case it will be used to launch any audio file.

A.3.1 Controlling playback

The audio player features the usual array of buttons allowing you to stop and start playback and cycle forwards or backwards. When the end of the playlist is reached, the first item in the playlist will then automatically be selected. The audio player can be seen in Figure A.6. Currently it is not possible to choose which columns are shown in the playlist. However, this functionality can be added in future versions.

Figure A.6 The audio player

A.3.2 Loading files

Simply make any view in a file browser and drag from the file browser into the audio player. The audio player will analyse the results of the view and load any *audio* files found. Note that the audio player will automatically adjust the playlist as the results of the view change.

A.3.3 Adding and removing files from the playlist

This is not catered for directly in the audio player. However, you can drag the view the player is playing from the playlist to the query builder and then use the query builder to include or exclude additional files. You can then drag the view from the query builder back to the audio player and the audio player will update the playlist.

A.4 Editors and Viewers User Guide

Each of these programs can be run from the command line. If you type the wrong number of arguments, a usage message will appear.

A.4.1 The Text Editor

The text editor is designed to edit text files. To start the application, either use the file browser to load text files, or from the command line type:

```
java -Djava.library.path="C:\Program Files\Java\eclipse\plugins\org.eclipse.swt.win32_3.0.
```

Then follow the instructions. Note: the filename of the library may need to be altered.

A.4.2 Creating a Text File

To create a text file click 'File' on the menu, followed by 'New'. The new file's tab will then automatically be selected.

A.4.3 Deleting a text File

To delete a text file click 'File' on the menu, followed by 'Delete'. You will then be faced with a message box asking whether you want to delete the file. If you click 'Yes' the file will be deleted and the document closed. If you click 'No' the document will remain open and the file will not be deleted.

A demonstration of a file being deleted. Figure A.7

Figure A.7 . A file being deleted

A.4.4 Closing a Text File

To close a text file click 'File' on the menu, followed by 'Close'. You will then be faced with a message box asking whether you want to save the file. If you click 'Yes' the file will be saved (refer to 'Saving a text file') and the document closed. If you click 'No' the document will be closed and the file will not be saved. If you click 'Cancel' the file will not be closed or saved.

A demonstration of a file being closed. Clicking 'No' will not save the file, clicking 'Yes' will save the file, clicking 'Cancel' will not close the file. Figure A.8

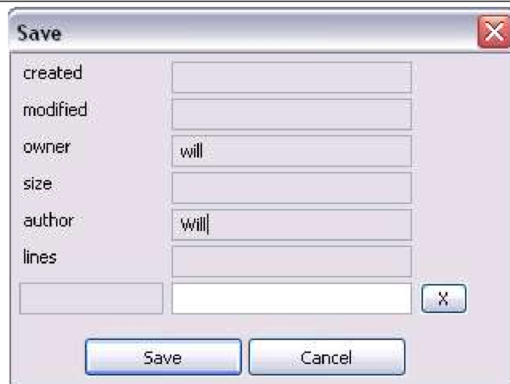
Figure A.8 . A file being closed

A.4.5 Saving a Text File

To save a text file click 'File' on the menu, followed by 'Save'. A dialog will be displayed as shown below. Simply fill in as many of the attribute values as you want and click save (or cancel to close the dialog). Keep in mind that if the attribute value is shaded grey, if it is empty it may automatically be filled in by the Server. The last line of the dialog allows you to specify additional attributes to add to the file type. For example lets suppose all text files now need to contain an additional attribute "name". Simply type name in the box on the left and the value in the box on the right. If a mistake is made simply delete the line or correct the error. Clicking 'Save' will add all of the necessary attributes (creating them as necessary) and save all of the values. Clicking cancel will have no effect on the file.

A demonstration of a file being saved. Notice the grey attribute values are derived attributes and may be filled in by the Server if left empty. Figure A.9

Figure A.9 . The save dialog used to save the file



A.4.6 Exiting the Text Editor

To exit the text editor click 'File' on the menu, followed by 'Exit'.

A.4.7 Editing Text

Standard facilities such as 'Cut', 'Copy' and 'Paste' are included to edit text. Clicking 'Cut' on the 'Edit' menu will remove any text that is highlighted whilst at the same time copying it. Clicking 'Copy' will copy the text into memory. Text that has been cut or copied can be placed in a different place in the document by clicking 'Paste' on the 'Edit' menu.

A.4.8 Toggling Syntax Highlighting

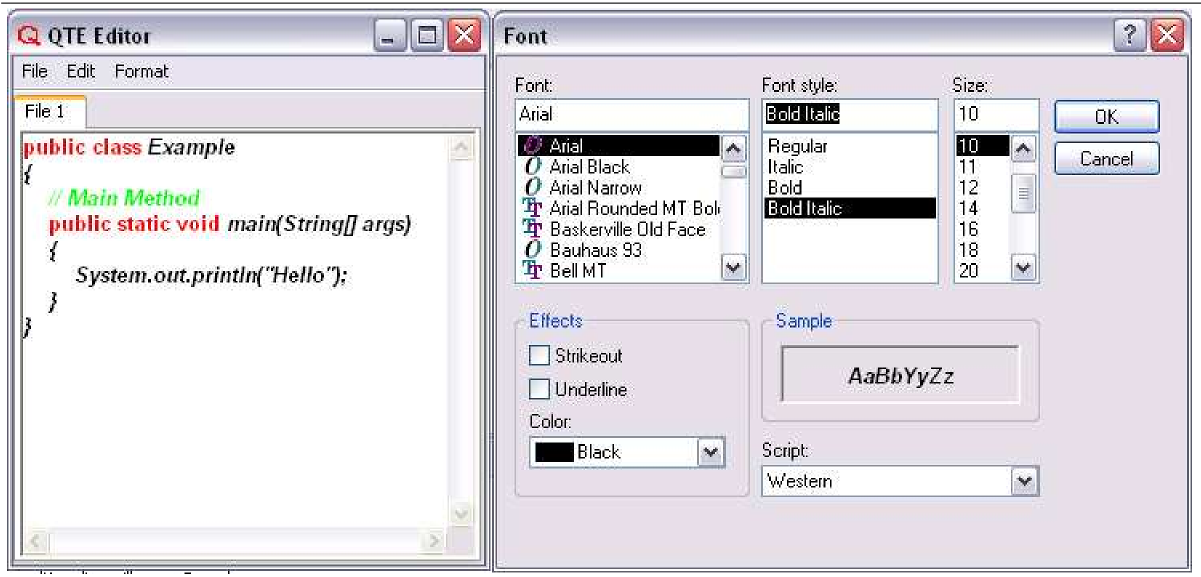
Clicking 'Edit' followed by 'Toggle syntax highlighting'. When syntax highlighting is enabled keywords in Java are coloured and made bold. Numbers and comments are also highlighted. When syntax highlighting is disabled all of the text is coloured in the standard colour.

A.4.9 Formatting the Font

The font can be altered by clicking 'Format' followed by 'Font' on the menu.

A demonstration of a file's font being changed is shown below. Figure A.10

Figure A.10 . An example of the font being formatted



A.4.10 Shortcut keys

All of the operations in the text editor can be accessed via a shortcut key. As you can see from the images above the menu items have text besides them for example (Save Ctrl-S). Ctrl-S is the shortcut key (hold down the 'Ctrl' key and press the 'S' key).

A.5 The Image Viewer

A demonstration of an image file being displayed. Figure A.11

The image viewer is as the name suggests is to view image files.

A.5.1 Deleting an Image File

To delete an image file click 'File' on the menu, followed by 'Delete'. You will then be faced with a message box asking whether you want to delete the file. If you click 'Yes' the file will be deleted and the image closed. If you click 'No' the image will remain open and the file will not be deleted.

Figure A.11 . An image being displayed (image courtesy of digitalblasphemy.com)

A.5.2 Closing an Image File

To close an image file click 'File' on the menu, followed by 'Close'.

A.5.3 Exiting the Image Viewer

To exit the image viewer click 'File' on the menu, followed by 'Exit'.

A.6 The E-mail Viewer

The e-mail viewer is as the name suggests is to view e-mails. If the e-mail contains no content, a message is displayed.

A demonstration of an e-mail being displayed. Figure A.12

Figure A.12 . An e-mail being displayed

A.6.1 Deleting an E-mail File

To delete an e-mail file click 'File' on the menu, followed by 'Delete'. You will then be faced with a message box asking whether you want to delete the file. If you click 'Yes' the file will be deleted and the e-mail closed. If you click 'No' the e-mail will remain open and the file will not be deleted.

A.6.2 Closing an E-mail File

To close an e-mail file click 'File' on the menu, followed by 'Close'.

A.6.3 Exiting the E-mail Viewer

To exit the e-mail viewer click 'File' on the menu, followed by 'Exit'.

A.7 The Basic Sound Player

The basic sound player will play sound files of a basic format such as wav files (simple uncompressed sound files typically used when speed is important).

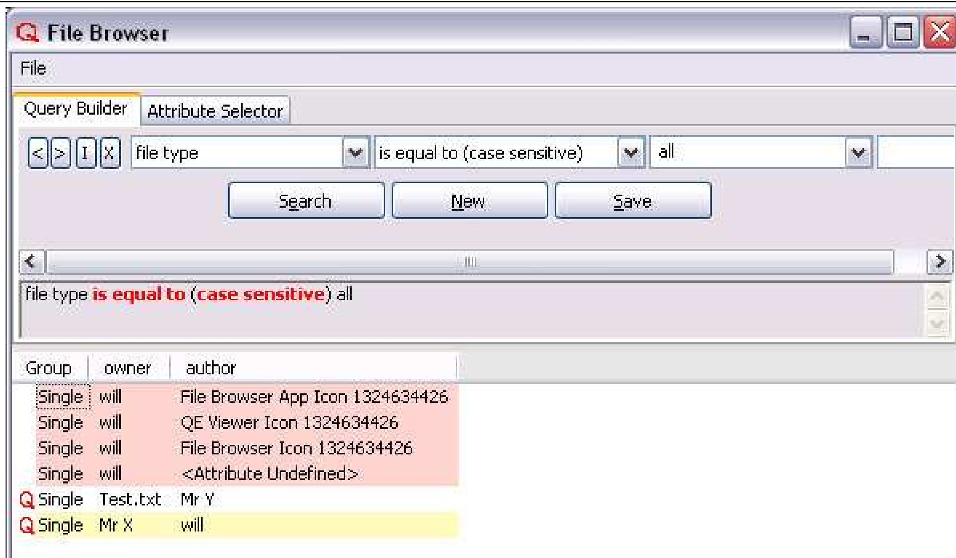
A.8 The Basic File Browser

A.8.1 The Browser

A simple browser is available to test these applications and is explained below. To start the application, from the command line type: 'java -Djava.library.path="C:\Program Files\Java\eclipse\plugins\org.eclipse.swt.win32_3.0.1
FileBrowser' from the command line and follow the instructions. Note: the filename of the library may need to be altered.

An example of the basic file browser. This is used to load a set of applications such as a text editor, an image editor, an e-mail viewer and a sound player. Figure A.13

Figure A.13 . An example of the basic file browser

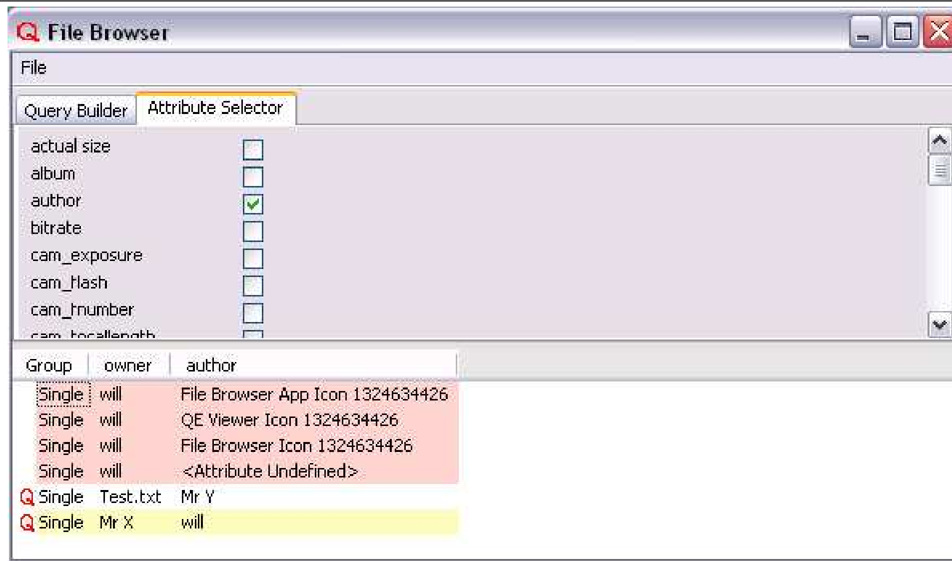


Some of the rows are highlighted red and some yellow. The yellow rows represent files that have been authored by the current user. The red rows indicate files that are owned by the current user. It can be seen that the text files have a 'Q' next to them. This is just to distinguish them from other files.

When the 'Search' button is clicked the attribute selector is automatically selected (shown below). This enables you to select which attributes to view on the files. Files with the same attributes are grouped together. For example if you are only viewing the owner attribute, all of the files with the same owner will be grouped together. When you click the 'Search' button for the first time, only 2 attributes are (by default) selected, owner and author. The user can then select other attributes. If you create a new query, the set of attributes the user used for the previous query will remain selected.

An example of the basic file browsers attribute selector. Figure A.14

Figure A.14 . An example of the basic file browsers attribute selector

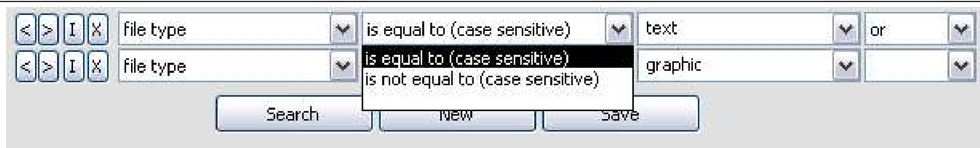


A.8.2 The Query Builder in the Browser

To create a query simply enter values in the boxes, the values in other combo-boxes may change as a result. This is because it may not be sensible to enter particular sets of values. The focus may move to a different combo-box automatically. This will enable the you to type phrases in without the need for a mouse, therefore speeding up input.

An example of how a query is entered can be demonstrated below. Figure A.15.

Figure A.15 . An image of the Query Builder's combo boxes used for data input



A.8.3 The Command Line

A textual representation of the query will appear on a command line. Using the example above, the resulting output on the command line will be: "file type is equal to (case sensitive) text or file type is not equal to (case sensitive) graphic". This command line output is largely to assist by providing an English description of the query. In the following figure, the query can be created using one key press ('f'). This is because the query builder will automatically update as much as it can. The command line will update after every change to the query. This means that when typing a value in for an attribute, the command line will update letter by letter. Figure A.16

Figure A.16 . The query updating the command line

A.9 Utilities

We wrote a few utilities which were designed to import many different files into the filesystem in bulk from any other filesystem.

A.9.1 File Loader

This client application can recursively load folders of files into the database and by detecting their file-type automatically, the server can generate their attributes automatically. It takes four commandline arguments:

```
java foobar.misc.LoadInFolder <server> <username> <password> <folder>
```

Files can added to the database while you work, and for any queries that you have open that the files match, the view will be updated on the fly.

A.9.2 Email Handler

This applications takes individual emails that are piped into it and adds them to the filesystem. This can be used to great effect having it in your `.forward` file because if you have a view open of all your emails any emails that arrive will automatically be added to your view. This removes the need for "checking your email" or even polling the email server as you are notified as they arrive.

For a common *nix `.forward` file you can have your emails copied to the filesystem as shown:

Example A.9.1 .forward file

```
sjr02, "|/usr/java/jre1.5.0/bin/java foobar.misc.EmailHandler \
        <server> <username> <password>"
```

Appendix B

Development logs

These are the minutes of the meetings we held and the individual development logs we kept. It should also be noted that we made extensive use of a mailing list to communicate and we all developed using CVS which both went some way to reducing the number required and length of meetings.

B.1 Minutes of Meetings

- **Monday 11th October.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Early planning stage. The following issues were decided upon:
  - the database to be used will be PostgreSQL due to previous experience
  - files are to be stored in the database
  - different views of file system
  - metadata and derived attributes e.g. thumbnails
  - buckets - ”pieces of a file”
- **Tuesday 12th October.** Meeting with supervisor: Susan Eisenbach. Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Explained the basic modular design of the system:
  - Client
    - User Interface (UI)
    - Background loading module and application programs
  - Server
    - Implementation of interfaces that the client uses
    - Libraries to be used by the server
  - Discussed benefits:
    - Can still have directory structure
    - Can create multiple views of file system
    - Very fast searching
    - Background loading
- **Wednesday 13th October.** Present: Matthew Sackman, Will Osborne, Francis Russell. Discussed how the UI might be implemented. Decided against a command line. Difficulties were realized due to there being no requirements for files to have names. Query builder discussed i.e. how to implement ”search” in the UI. A simple tree structure was decided to build searches.
- **Monday 25th October.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. It was decided that the Client should be split up. The query builder and file browser (including the drag ’n drop save dialog) will now be handled separately by Will and Francis respectively.
- **Friday 29th October.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. New allocation of Client elements proceeding as planned.

- **Monday 1st November.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Core elements of the server largely complete. Explanation of these features given to enable the Client to use them.
- **Tuesday 2nd November.** Meeting with supervisor: Susan Eisenbach. Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Explained changes since the project specification (the main change being the client split-up).
- **Friday 5th November.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Checking up on what everyone was doing. Reminder to document work.
- **Friday 12th November.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. File browser undertaken by Francis and Matthew to test which idea would be better. The 2 ideas are as follows:
  - Specify a query and the results are displayed in the file browser
  - Gradually build up results from a sequence of queries (refining the result each time).
- **Monday 15th November.** Present: Matthew Sackman, Will Osborne, Francis Russell. Discussed drag and drop and decided to get a basic Save API implemented first.
- **Friday 19th November.** Present: Matthew Sackman, Will Osborne, Francis Russell. Talked about hooking up the query builder to a file browser.
- **Tuesday 23rd November.** Meeting with supervisor: Susan Eisenbach. Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Talked about what had been done and what would be done over Christmas.
- **Monday 13th December.** Present: Matthew Sackman, Will Osborne, Francis Russell, Sam Richards. Talked about what to do over Christmas in detail.

B.2 Development Log for Matthew

- **9 October 2004 - 4 hours.**
  - Initial research and creation of object serialisation and deserialisation of objects.
- **10 October 2004 - 4 hours.**
  - Creation and testing of protocol elements, serialisation mechanism now working over the network.
- **12 October 2004 - 3 hours.**
  - Added code to make file transfer possible across the network. Full testing of a variety of file transfer mechanisms.
- **13 October 2004 - 2 hours.**
  - Some code fixes to solve some bugs discovered yesterday in the file transfer. Massive increase in efficiency.
- **17 October 2004 - 4 hours.**
  - Initial work on the Database design. Research and testing of stored procedures.
- **20 October 2004 - 3 hours.**
  - Finalisation of the Database design. Import of PostgreSQL JDBC drivers and initial work making the server talk to the database.

- **25 October 2004 - 3 hours.**
  - Lots of work on the client-server query mechanism. You can now query the server from the client and it will hit the database if necessary. You can now read files from the client.
- **26 October 2004 - 5 hours.**
  - More work on client-server communication. You can now read and write files from the client, change attribute values, create and remove attribute-types on a file-type, create and remove attributes on a file.
- **27 October 2004 - 3 hours.**
  - Yet more work on client-server communication. You can now create and delete files. Plus improvements in Database connection handling code.
- **30 October 2004 - 6 hours.**
  - Research into caching issues, cache invalidation techniques and associative database client caches based on predicates, specifically, the maintainance of materialised views in client caches.
- **1 November 2004 - 4 hours.**
  - Work on caching mechanism.
- **2 November 2004 - 10 hours.**
  - Massive implementation of caching system, work on update notification. Trigger function auto creation, addition and removal. View Query manager work. Modification of Data Types to support more flexible typing system.
- **3 November 2004 - 5 hours.**
  - Research and work on JUnit unit testing. Creation of testing suite. Further bug fixes in Client and Server resulting!
- **4 November 2004 - 5 hours.**
  - JUnit testing and resulting fixes to code. Mental note: Unit testing is actually a good idea.
- **5 November 2004 - 3 hours.**
  - More testing and more fixes. Code is almost reliable now!
- **7 November 2004 - 4 hours.**
  - Work on the next report.
- **8 November 2004 - 4 hours.**
  - Work on the next report and finally getting derived support for attributes.
- **9 November 2004 - 6 hours.**
  - Work on the next report and massive efficiency improvements.
- **10 November 2004 - 2 hours.**
  - Work on the next report.

- **11 November 2004 - 2 hours.**
  - Work on the next report.
- **12 November 2004 - 4 hours.**
  - Work on and submission of the next report.
- **14 November 2004 - 6 hours.**
  - Start work on a file browser.
- **15 November 2004 - 5 hours.**
  - Lots of work with concurrency fixes and caching issues.
- **16 November 2004 - 6 hours.**
  - More work on caching issues, broadcasting file type changes and concurrency issues.
- **17 November 2004 - 2 hours.**
  - Consideration of attribute value issues, in particular, how to cast down the value to a usable state.
- **18 November 2004 - 6 hours.**
  - Creation of attribute value visitor using reflection and visitor pattern to invoke the correct method using runtime type information. AttributeCanvas can now do rubber band selection work.
- **21 November 2004 - 8 hours.**
  - Massive refactoring and work on File Browser and fixes elsewhere throughout the project. File Browser nearly finished.
- **22 November 2004 - 2 hours.**
  - Work on file browser.
- **23 November 2004 - 2 hours.**
  - Work on file browser.
- **25 November 2004 - 8 hours.**
  - Work on optimisations and parallelisations.
- **28 November 2004 - 8 hours.**
  - Attempts to use GZIP compression routines to compress protocol to improve performance. Abandoned due to insurmountable problems.
- **29 November 2004 - 4 hours.**
  - Work on optimisations and parallisations: modifications to protocol to improve performance and decrease round trips.
- **30 November 2004 - 4 hours.**

- Massive reworking of Thread Pool functionality to solve problems and improve performance. Generally successful.

- **2 December 2004 - 4 hours.**
  - Bug fixing of Client to solve problems raised by Francis relating to cache consistency. Solution found.

- **3 - 17 December 2004 - 0 hours.**
  - Development halted whilst we do exams. They were fun.

- **18 December 2004 - 2 January 2005 - 6 hours per day on average. I think..**
  - I don't know what I did as I didn't keep a log as things were developing too fast.
  - There were many many bug fixes.
  - Finished my file browser.
  - Wrote an audio player.
  - Wrote a text editor.
  - Added authentication functionality.
  - Added the ability to load an application out of the database.
  - Added the ability to create a new process and have it run an application out of the database.
  - Added Drag and Drop functionality.
  - Wrote an attribute editor.
  - Wrote parts of the final report.

B.3 Development Log for Francis

- **21 October 2004 - 1 hour.**
  - Created file browser form
- **22 October 2004 - 2 hours.**
  - Continued file browser form development
- **26 October 2004 - 1 hour.**
  - Created draft file type hierarchy list
- **31 October 2004 - 1 hour.**
  - Started work on file lister
- **1 November 2004 - 4 hours.**
  - Continued work on file lister
- **2 November 2004 - 4 hours.**
  - Continued work on file lister.
- **4 November 2004 - 3 hours.**
  - Continued work on file lister
- **8 November 2004 - 1 hour.**

  - Started work on alternative query builder

- **9 November 2004 - 2 hours.**
  - Continued work on query builder
- **10 November 2004 - 1 hour.**
  - Started documentation of file browser in report
- **11 November 2004 - 3 hours.**
  - Continued documentation of file browser
- **12 November 2004 - 1 hour.**
  - Added file browser overview diagram and documented drag and drop in report
- **16 November 2004 - 3 hours.**
  - Continued work on query builder
- **20 November 2004 - 3 hours.**
  - Continued work on query builder
- **22 November 2004 - 4 hours.**
  - Continued work on query builder
- **25 November 2004 - 10 hours.**
  - Integration of Will's query builder with file lister
  - Lister optimisations
  - Bug fixes
- **27 November 2004 - 7 hours.**
  - Work on query builder
- **29 November 2004 - 4 hours.**
  - Modified file lister to use the client's new more efficient attribute value retrieval call
  - Helped Matthew with server debugging
- **2 December 2004 - 1/2 hour.**
  - Wrote code to prove server had a caching bug
- **3 - 17 December 2004 - 1/2 hour.**
  - Development suspended due to examinations
- **18 December 2004.**
  - Refactoring query builder code
- **20 December 2004 - 13 hours.**
  - Adding query builder functionality
- **21 December 2004 - 6 hours.**
  - Development and debugging of query builder GUI

- **22 December 2004 - 11 hours.**
  - Development and debugging of query builder GUI
- **23 December 2004 - 11 hours.**
  - Implementing query saving and loading
- **24 December 2004 - 11 hours.**
  - Finished saving and loading
- **25 December 2004 - 9 hours.**
  - Wrote class to handle time-spans to replace java.sql.time
  - Rewriting file lister
- **26 December 2004 - 10 hours.**
  - Rewriting file lister
  - Rewriting attribute selector
- **27 December 2004 - 8 hours.**
  - Added file lister group expansion
  - Added some drag and drop support
- **28 December 2004 - 8 hours.**
  - Corrected and improved file grouping behaviour
  - Added File Object ID support to query builder
  - Started rewriting save and load menus
- **29 December 2004 - 9 hours.**
  - Completed new load menu
  - Modified saving to use drag and drop
  - Fixed GUI under windows
  - Added file launching support
  - Added attribute editing support
  - Modified file browser to use server connection dialog
  - General bug fixes

B.4 Development Log for Sam

- **Friday 29th Oct - 5 hours.**
  - Researched magic file-type libraries in Java.
  - Rewrote Metadata parser structure to abstract class.
- **Sunday 31st Oct - 1 hour.**
  - Researched image resizing in Java.
  - Reviewed database
- **Monday 1st November - 4 hours.**

  - File types and attributes tables.
  - Tested jmimemagic library.

- **Thursday 4th November - 4 hours.**
  - XML DTD for file types hierarchy.
  - Translation of file type to our file-type tree.

- **Friday 5th November - 1 hour.**
  - File-type tree and translation.

- **Tuesday 9th November - 2 hours.**
  - Documentation for middle report.

- **Thursday 11th November - 6 hours.**
  - Finalised report.

- **Sunday 21st November - 6 hours.**
  - Completed XML for file type tree.
  - Work on Mime type lookup in tree.

- **Monday 22nd November - 4 hours.**
  - Started sub-matching.

- **Tuesday 23rd November - 3 hours.**
  - More work on sub-matching.

- **Saturday 27th November - 5 hours.**
  - Hacked up matching to finish it, dropped certain sub-matching features.
  - File type detection finished.

- **Tuesday 30th November - 2 hours.**
  - Work on ID3 parser.

- **Wednesday 1st December - 9 hours.**
  - Work on EmailHandler.
  - Fixed attribute updater.
  - Correct typing of attributes in exif_parser.

- **Friday 3rd December - 3 hours.**
  - Multiple addresses and typing of fields, finished EmailImporter.

- **Sunday 5th December - 3 hours.**
  - Converting ID3 V1 tag reader.

- **Monday 20th December - 3 hours.**
  - Setup home db
  - Finished ID3.
  - Researched Ogg tags.

- **Wednesday 22nd December - 5 hours.**
  - Ogg extractor, and MP3 extractor work using JavaSound SPIs.
  - Tested playing of audio using JavaSound from DB.
  - Started Audioplayer.
- **Thursday 23rd December to Sunday 2nd January - Roughly 50 hours.**
  - Mass file loader.
  - Testing of all extractors.
  - Archive metadata extractors.
  - Documentation

B.5 Development Log for Will

- **Monday 11th Oct - 2 hours.**
  - Designed and built project website
- **Monday 18th Oct - 2 hours.**
  - Started work on project report
- **Wednesday 20th Oct - 3 hours.**
  - Researched and experimented with the IBM Standard Widget Toolkit (this will continue throughout the project)
- **Thursday 21st Oct - 3 hours.**
  - Started work on QTE (text editor) as a demonstration of database features
- **Sunday 24th Oct - 3 hours.**
  - Started work on Query Builder to build searches and views on the client
- **Wednesday 27th Oct - 3 hours.**
  - Work on Query Builder
- **Thursday 28th Oct - 5 hours.**
  - Finished work on graphical part of Query Builder
- **Sunday 31st Oct - 2 hours.**
  - Documentation
- **Wednesday 3rd Nov - 2 hours.**
  - Added functionality to query builder
- **Wednesday 10th Nov - 3 hours.**
  - Added basic filtering to the query builder
- **Thursday 18th Nov - 1 hour.**
  - Got XP-style buttons working in SWT

- **Sunday 21st Nov - 3 hours.**
  - Added a save API to allow files to be saved (and attributes filled in)
- **Monday 22nd Nov - 3 hours.**
  - Added new/save/load functionality to the text editor
- **Thursday 25th Nov - 5 hours.**
  - Integrated Query Builder into File Browser
  - Added tabs to Text Editor
- **Saturday 27th Nov - 3 hours.**
  - Added the ability to load images into Text Editor
  - Save dialog now saves file attributes
- **Friday 3rd Dec - 5 hours.**
  - Explored the possibility of using JMF to load videos and sound
  - Simple to use if you have a URL, the difficulty arises when you try to get an Input Stream from it
- **Sunday 5th Dec - 3 hours.**
  - Text editor now loads e-mails
- **Tuesday 7th Dec - 2 hours.**
  - Added the ability to load basic sounds to test streaming. Icon now loads from the database
- **Thursday 9th Dec - 3 hours.**
  - Query Builder altered to accept references
- **Monday 13th Dec - 3 hours.**
  - Query Builder load and save capability added (i.e. save the query to a file on the database and load it from a file)
- **Wednesday 22nd Dec - 5 hours.**
  - Work on File Browser and split up image viewer, text editor etc.
- **Thursday 23rd Dec - 4 hours.**
  - Work on File Browser - added colouring to rows and icons to the columns