5-18-2007

Indexing and Searching Document Collections using Lucene

Sridevi Addagada
University of New Orleans

Indexing and Searching Document Collections using Lucene

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Sridevi Addagada

B.Tech. Jawaharlal Nehru Technology University, 2002

May 2007

ACKNOWLEDGEMENTS

I take this opportunity to thank my thesis advisor Dr. Shengru Tu. I thank him for giving me the freedom to explore the various possibilities in this fast growing field. I would also like to take this opportunity to thank my thesis committee members, Dr. Jing Deng and Dr. Adlai DePano. I once again thank Dr. Shengru Tu, graduate coordinator, Department of Computer Science, UNO for guiding me all through my Master's program. I would also like to thank Dr. Mahdi Abdelguerfi, Chairman, Department of Computer Science, University of New Orleans, for his support. I would also like to express my gratitude to all the other professors in the department.

I finally thank my parents and friend for their overwhelming support all the way.

Table of Contents

List of Figures

ABSTRACT

The amount of information available to a person is growing day by day; hence retrieving the correct information in a timely manner plays a very important role. This thesis talks about indexing document collections and fetching the right information with the help of a database. The primary role of a database is to store the additional information which may be or may not be available in the document collection by itself.

The indexing of document collection is performed by Lucene, while the search application is strongly integrated with a database. In this thesis a highly efficient, scalable, customized search tool is built using Lucene. The search tool is capable of indexing and searching databases, PDF documents, word documents and text files.

1. INTRODUCTION

The amount of information available to a human being is growing exponentially every day due to the advancement in technologies and the internet. The increased availability of documents in digital form has contributed significantly to the immense volume of knowledge and information available to people. Initially, the raw documents were given in various formats such as HTML, PDF, and WORD without following any schemas. They are called unstructured documents. Due to their dynamic nature, fast growth rate, and unstructured manner, it is increasingly difficult to identify and retrieve valuable information from these documents. This problem has attracted a number of research efforts in the Information Retrieval (IR) society [Crane]. Significant research has been placed on the efforts to combine the IR techniques with the database technologies that are for structured information searching [Moens]. Leading software firms such as IBM has developed a powerful software development kit called Unstructured Information Management Architecture (UIMA) [IBM]. UIMA supports ontology-based information retrieval for texts, audio and video media. As powerful as UIMA is, the deployment of UIMA carries a reasonable technical burden including annotation based on the common analysis structure. Unfortunately, such complexity could prohibit a quick deployment in many simpler applications.

In this thesis, I report a simpler, more practical approach that combines one of the IR techniques – keyword searching – with the database techniques through a filtering process. The utilization environment of my development is to support management of training courses from tens of thousands of entries owned by a military branch. Users need search for courses against specific training needs. Each search typically concerns two aspects of information. On one hand, the user has a set of requirements regarding the properties of the courses such as the delivery form, the course length, and standards compliance. These properties are stored in the course database. One the other hand, keyword search is a practice of the military training professionals. The keyword search needs to be supported in two aspects. First, searching keyword is facilitated in four limited-length fields, namely title, abstract, description, and audience. A more desirable feature is to support keyword search from the entire contexts of the courses. Typically, each course has one or more large documents in PDF, Word or plain text formats. In my implementation, the constraints regarding the courses' properties and the keywords occurrence in the four limited-length fields are carried out by the typical database queries. Searching keywords

from the entire documents is supported by full-scale indexing. Considering the size, the variations and the number of the documents, this indexing is a non-trivial task. I have adopted an Apache open-source indexing toolkit, Lucene.

Using off-the-shelf commercial or non-commercial software products as building blocks has been a strong trend in businesses and organizations, as highlighted by the COTS-based systems initiative at the Carnegie Mellon Software Engineering Institute [Carnegie]. The advantages of using off-the-shelf software products are numerous including market-tested reliability, market-approved features, and an opportunity for expanding software capabilities and improving system performance by the marketplace. Using off-the-shelf software products often promises a rapid system deployment. Yet, the promise of off-the-shelf products is too often not realized in practice. There have been more failures than successes in using COTS software products [Brownsword]. To a large extent, my project was an implementation based on utilizing off-the-shelf software products. In my experiments, even though adopting Lucene was straightforward, integrating the Lucene indexing and the database querying were challenging.

My implementation has illustrated a framework of integrating the full-scale text indexing engine Lucene with the database query techniques for a general approach of keyword search followed by properties filtering. This can be a very useful framework for many applications for small and middle businesses or organizations.

The remaining parts of this thesis are organized as the following. Chapter 2 provides the background information about Lucene. Chapter 3 is Application design where I explain more about my user interfaces and Chapter 4 provides insight about my system implementation.

2. BACKGROUND

Lucene is a Java library that adds text indexing and searching capabilities to an application. It is not a complete application that one can just download, install, and run. It offers a simple, yet powerful core API. To start using it, one needs to know only a few Lucene classes and methods. Lucene offers two main services: text indexing and text searching. These two operations are relatively independent of each other.

2.1 History of Lucene

Doug Cutting is the primary developer of the Lucene and Nutch open source search projects. He has worked in the search technology field for nearly two decades, including five years at Xerox PARC, three years at Apple, and four years at Excite. Lucene was initially available for download from its home at the Source Forge web site. Lucene joined the Apache Software Foundation's Jakarta family of high-quality open source Java products in September 2001. With each release since then, the project has enjoyed increased visibility, attracting more users and developers. Lucene derives its name from Doug's wife's middle name; Lucene is also her maternal grandmother's first name.

2.2 Installing Lucene

Lucene is distributed as pre-compiled binaries or in source form. One can download the latest release from Lucene's release page. After downloading the Lucene jar file, the jar file is added to the CLASSPATH environment variable.

2.3 Lucene ports

Lucene can be ported to other programming languages. Lucene was originally written in Java, Lucene implementations in other languages are given in the following table. Lucene porting helps developers to access Lucene indices from applications written in different languages.

Lucene Port	Description
CLucene	Lucene implementation in C++
Lucene4c	Lucene implementation in C
Lupy	Lucene implementation in Python
Zend Search	Lucene implementation in the Zend Framework for PHP 5
KinoSearch	New version of Lucene implementation in PERL
MUTIS	Lucene implementation in Delphi
dotLucene	Lucene implementation in .NET
LuceneKit	Lucene implementation in Objective-C (Cocoa/GNUstep support)
NLucene	Another Lucene implementation in .NET (out of date)
Plucene	Lucene implementation in PERL
PyLucene	GCJ-compiled version of Java Lucene integrated with Python via SWIG
Ferret	Lucene implementation in Ruby

2.4 Lucene Classes

Lucene consists of a set of key classes that are used to build a search application. The various key classes are given below

IndexWriter – The IndexWriter class is used to create and maintain indices. This class also determines whether a new index is created or whether an existing index is opened for the addition of new documents.

Directory – The Directory class represents the location of a Lucene index. A Directory is a flat list of files. Files may be written once, when they are created. Once a file is created it may only be opened for read, or deleted. Random access is permitted both when reading and writing.

Document - The Document class represents a document in Lucene. Documents are the unit of indexing and search. A Document is a set of fields. Each field has a name and a textual value. A field may be stored with the document, in which case it is returned with search hits on the document.

Field - The Field class represents a section of a Document. The Field object will contain a name for the section and the actual data. Values may be free text, provided as a String or as a Reader, or they may be atomic keywords, which are not further processed. Fields are optionally stored in the index, so that they may be returned with hits on the document.

Analyzer - The Analyzer class is an abstract class that is used to provide an interface that will take a Document and turn it into tokens that can be indexed. There are several useful implementations of this class but the most commonly used is the Standard Analyzer class. A typical implementation involves building a tokenizer first, which breaks the stream of characters from the reader into raw tokens.

IndexSearcher - The IndexSearcher class is used to search through an index. This class implements a search over index reader.

Term - The Term is a basic unit for searching, consist of a pair of string elements, name of the field and he value of that field. A Term represents a word from text. Terms may also represent more than words from text fields, they can also represent things like dates, email addresses, URL's, etc.

QueryParser - The QueryParser class used to build a parser that can search through an index. A Query is a series of clauses. A clause may be either a term, indicating all the documents that contain a particular term or a nested query enclosed in parentheses. A nested query may be used with a (+) or (-) prefix to require any of a set of terms. The BNF Query grammar can be written as

```
Query  ::= ( Clause )*
Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")" )
```

Query - The Query class is an abstract class that contains the search criteria created by the QueryParser.

Term Query – The term query class is used for matching documents that contain fields with specific values. This may be combined with other terms with a Boolean Query.

Hits - The Hits class contains the Document objects that are returned by running the Query object against the index. It is a ranked list of documents that is used to hold search results.

2.5 Indexing

The heart of all search engines is the concept of Indexing; Indexing in short can be defined as the processing of original data into a highly efficient cross reference lookup in order to facilitate rapid searching.

Indexing can be defined in a much better manner by taking an example. Let us assume that a person wants to search for a word or a phrase among a large number of files. The simplest method would be to sequentially scan each file for the given word or phrase. The main disadvantage with this approach is that it does not scale to larger file sets or cases where files are large; hence to search large amounts of text quickly, one must first index that text and convert it into a format that will let the person search it rapidly, eliminating the slow sequential scanning process. This conversion process is called indexing, and its output is called an index.

2.5.1 Structure of a Lucene index

A Lucene index is stored in a single directory in the file system on a hard disk. The core elements of a Lucene index are segments, documents, fields, and terms. In a Lucene index every index consists of one or more segments. Each segment contains one or more documents. Each document has one or more fields, and each field contains one or more terms. Each term is a pair of Strings representing a field name and a value. A segment consists of a series of files. Pictorially the entire structure of a Lucene index can be represented as shown in figure.2.1.

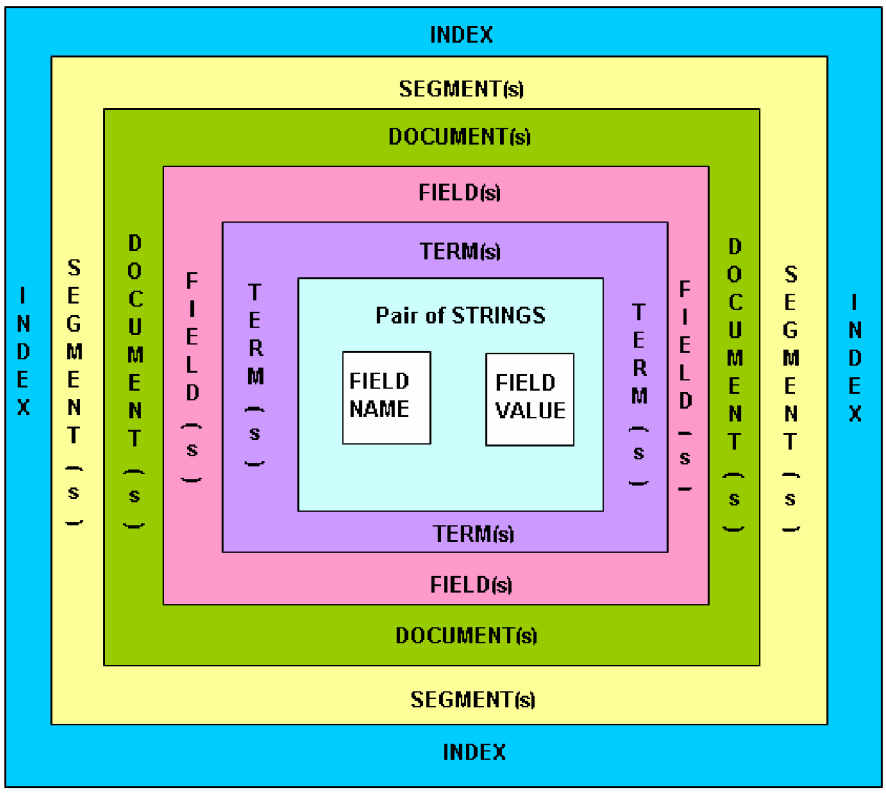

Figure 2.1: Structure of a Lucene index

The exact number of files that constitute each segment varies from index to index, and depends on the number of fields that the index contains. All files belonging to the same segment share a common prefix and differ in the suffix. One can think of a segment as a sub-index, although each segment is not a fully-independent index.

2.5.2 Factors affecting Indexing Speed

The bottleneck of a typical text-indexing application is the process of writing index files onto a disk. When new documents are added to a Lucene index, they are initially stored in memory instead of writing immediately to the disk. The simplest way to improve Lucene's indexing performance is to adjust the value of Index Writer’s merge Factor instance variable. This value tells Lucene how many documents to store in memory before writing them to the disk, as well as how often to merge multiple segments together. The default value is 10; Lucene will store 10 documents in memory before writing them to a single segment on the disk. The merge Factor

value of 10 also means that once the number of segments on the disk has reached the power of 10, Lucene will merge these segments into a single segment. When the merge Factor value is set to 10, a new segment will be created on the disk for every 10 documents added to the index. When the 10th segment of size 10 is added, all 10 will be merged into a single segment of size 100. When 10 such segments of size 100 have been added, they will be merged into a single segment containing 1000 documents, and so on. Therefore, at any time, there will be no more than 9 segments in each power of 10 index size.

2.5.3 In-Memory Indexing

Lucene distribution contains the RAMDirectory class, which gives even more control over this process. This class implements the Directory interface, just like FSDirectory does, but stores indexed documents in memory, while FSDirectory stores them on disk. Because RAMDirectory does not write anything to the disk, it is faster than FSDirectory. However, since computers usually come with less RAM than hard disk space, RAMDirectory is not suitable for very large indices.

2.5.4 Multi-Threaded Indexing

While multiple threads or processes are used to search a single Lucene index simultaneously, only a single thread or process is allowed to modify an index at a time. If the indexing application uses multiple indexing threads that are adding documents to the same index, one must serialize their calls to the IndexWriter.addDocument (Document) method. If the calls are not serialized then it may cause threads to get in each other's way and modify the index in an undesired manner causing Lucene to throw exceptions. To prevent misuse, Lucene uses file-based locks in order to stop multiple threads or processes from creating Index Writers with the same index directory at the same time.

2.5.5 Optimizing a Lucene Index

To optimize an index, one has to call optimize () on an IndexWriter instance. When optimize() is called, all in-memory documents are flushed to the disk and all index segments are merged into a single segment, reducing the number of files that make up the index. However, optimizing an index does not help improve indexing performance. As a matter of fact, optimizing an index

during the indexing process will only slow things down. Despite this, optimizing may sometimes be necessary in order to keep the number of open files under control. For instance, optimizing an index during the indexing process may be needed in situations where searching and indexing happen concurrently, since both processes keep their own set of open files. In fact, if more documents are added to the index, one should avoid calling optimize (). If, on the other hand, one should know that the index will not be modified for a while, and the index will only be searched, and it should be optimized. That will reduce the number of segments, and consequently improve search performance, the fewer files Lucene has to open while searching, the faster is the search.

2.6 Searching

In Computer Science, searching can be defined as an algorithm that takes a problem as an input and returns a solution to the problem, usually after evaluating a number of possible solutions. The set of all possible solutions to a problem is called the search space. Searching can be broadly classified as follows

2.6.1 Uninformed search

An uninformed search algorithm does not take into account the specific nature of the problem. Uninformed search algorithms are implemented in general, and then the same implementation is used for a wide range of problems with the help of abstraction. The disadvantage in uninformed search is that the search spaces are extremely large, and an uninformed search will take a reasonable amount of time for small examples. Hence to speed up the process, one has to use an informed search. Examples of uninformed search includes list search, tree search and graph search.

2.6.2 Informed search

An informed search algorithm uses a heuristic that is specific to the problem. A good heuristic will make an informed search really faster than a uniformed search. Examples of informed search include Best-first search, and A*

2.6.3 Adversarial search

Adversarial search is basically used for games; Adversarial search takes a unique characteristic that will account for any possible move that the opponent may take. Adversarial search is often used in games and artificial intelligence.

2.6.4 Interpolation search

An interpolating search attempts to find the item by approximating how far the item is likely to be from the current position. Interpolation search is analogous to searching a dictionary.

2.7 Analyzers

Analyzers can be defined as components that pre-process input text. They are also used when searching. Because the search string has to be processed the same way that the indexed text is processed, it is crucial to use the same Analyzer for both indexing and searching. Not using the same Analyzer will result in invalid search results.

The Analyzer class is an abstract class, if there is a need to pre-process input text and queries in a way that is not provided by any of Lucene's Analyzers, one will need to implement a custom Analyzer.

2.8 Overview of Lucene Architecture and Lucene Applications

It is possible to add indexing and searching capabilities to any application using Lucene. It is possible to index and make searchable any data that can be converted to a text format. Lucene is independent of the source of data, its format, and even its language as long as it can be converted to text. This means one can use Lucene to index and search data stored in files, web pages, on remote web servers, documents stored in local file systems, simple text files, Microsoft Word documents, HTML or PDF files, or any other format from which one can extract textual information. Similarly, with Lucene one can index data stored in databases, giving users full-text search capabilities that are absent in many databases. The basic architecture of a Lucene application can be visualized as shown in figure 2.2

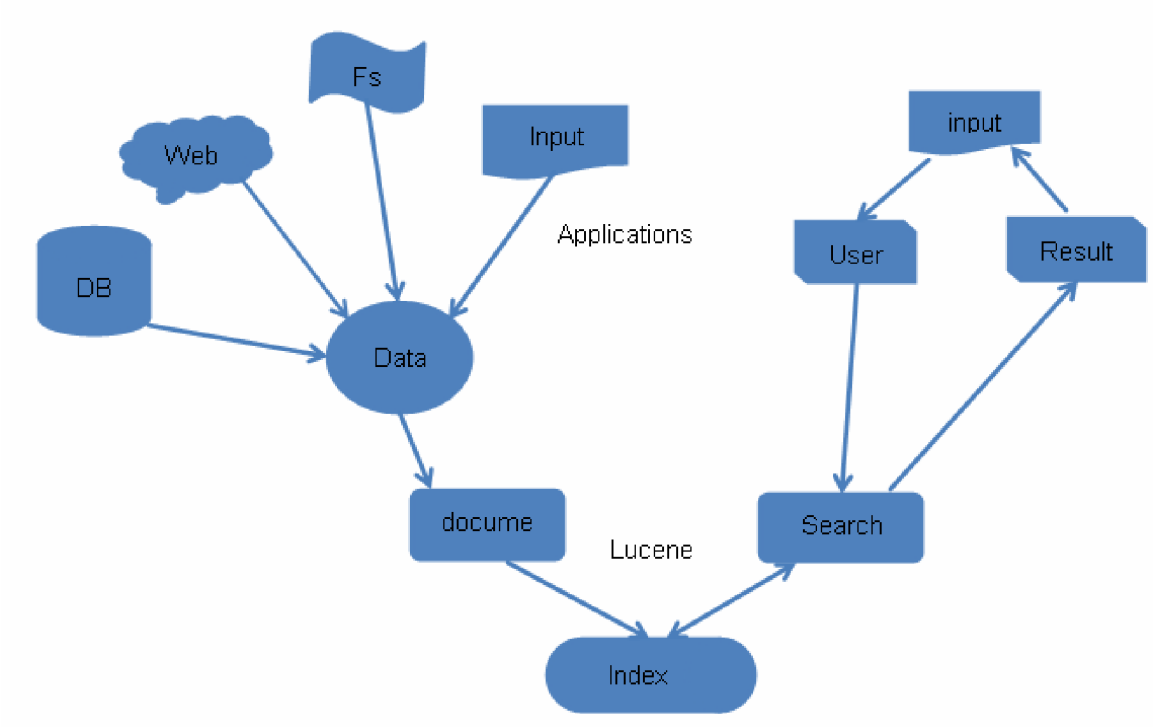

Figure 2.2 Overview of Lucene applications

2.9 Lucene in action

Lucene is an information retrieval (IR) library written purely in Java. Lucene provides the core API's for adding full text indexing and searching functionalities for a given application; Hence Lucene is not a complete framework for implementing a search engine, As a matter of fact Lucene only helps the user with indexing and searching functions, more over to index any type of given data the data has to be converted into text format as Lucene can only index text data and search with given user queries.

In the older search engines indexing is done by keywords and it is represented by text pairs determined by the user. Due to this type of design one has to use Boolean queries for creating pairs (AND, OR, NOT). The main flaw in this kind of design is the amount of time consumed in creating the pairs. But as technology advanced, the new mathematical models functionalities in full-text search engines reached a new horizon. Lucene has grown to support the ranking feature in which the result is the file that consists of the most number of user queries. Lucene also supports Boolean queries term, range, prefix, phrase, wildcard and fuzzy queries.

2.9.1 Indexing

Indexing is the heart of Lucene, during indexing the original data is processed into a highly efficient cross reference lookup. This is done in order to search at a faster rate. Indexing is done by analyzers. The unusable texts such as the stop words, word suffixes or prefixes are discarded at the analyzer stage. At the end of an indexing stage an index is created.

A Lucene index consists of Lucene document class instances which defines the index documents. Each document contains a pair consisting of a Field name and a Field value.

2.9.2 A Lucene index

A Lucene index is an inverted index. An inverted index means that the content of the documents that are analyzed has their important terms indexed as a pair consisting of a field name and a field value. A field contains many terms that point to the corresponding documents. To summarize, an inverted index makes the process of retrieving documents from a system a breeze. Finally, the documents are searched in the fields and in their values.

A Lucene index consists of many segments. A segment is created every time when new documents are created and indexed. Hence each segment has many documents stored in it. The documents consist of indexed Fields. An indexed field is a pair consisting of a field name and a field value pairs. Fields are used for calculating weights and ranking search results.

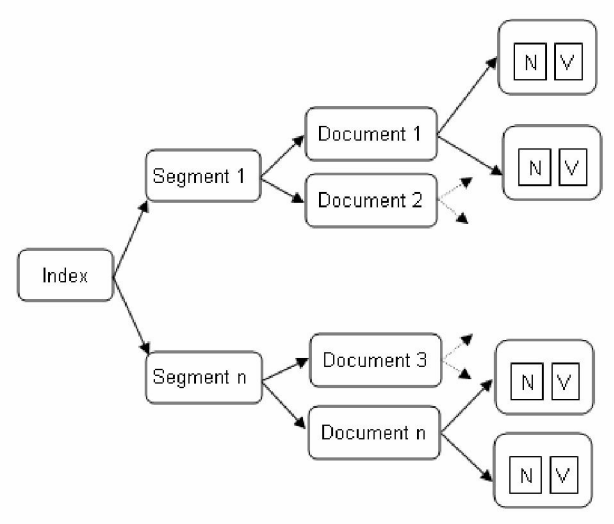

Figure: 2.3 A Lucene index

An actual Lucene index screen capture is shown in figure2.4, it is the index of a PDF document, as it can be seen in the figure it consists of three fields and it is the index of only one PDF document.

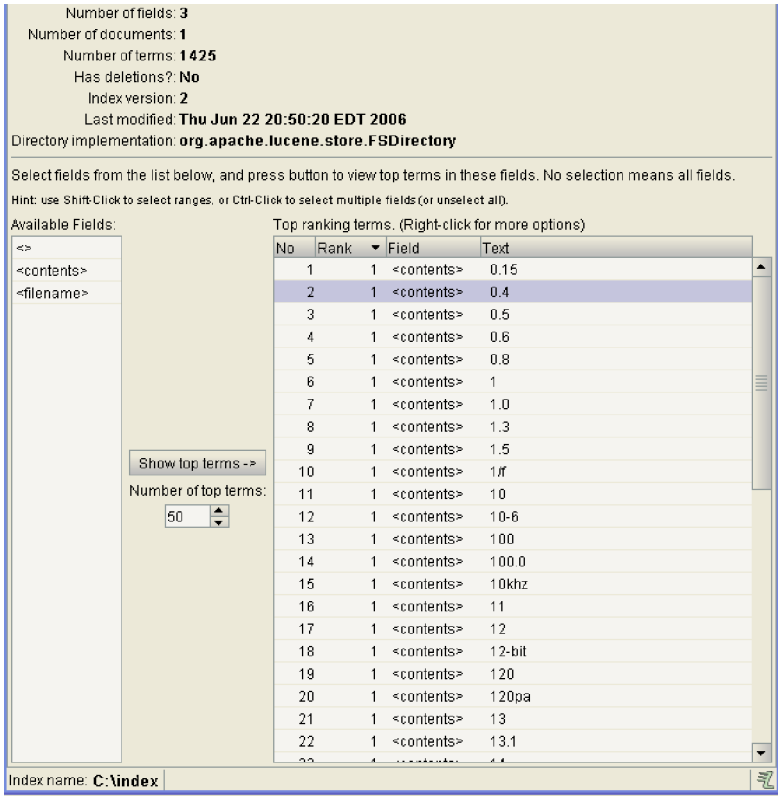

Figure 2.4: Lucene index screen capture

3. Application Design

To support management and utilization of tens of thousands of training courses, my system facilitates the user with six aspects of functions. They are listed below.

1. System configuration. The user can set up the database connection and table information.
2. Collecting structured information including the course properties and short texts such as the title, abstract, description and audience. The user's inputs are inserted into a table of the database.
3. Collecting the unstructured information, mainly the course documents. These documents can be in plain text, PDF, HTML, or Word.
4. Creating indexes for the unstructured documents. This is a critical step in providing keyword search with satisfactory response time.
5. Facilitating course search. The search takes account of three groups of information: (1) course properties such as course length, course form ("instructor lead" or "asynchronous online"); (2) keywords in the title, abstract, description and audience fields; (3) keywords in the course contexts.
6. Course property management. The user can setup or modify course property fields according to specific needs without dealing with database tables.

3.1 System configuration

This function is implemented as the system configuration GUI that provides the user an interface to enter the login credentials to get access to a particular database. Apart from the connections, the user can also select a particular table which contains the properties (the Meta data) of each of the courses, the document of which will be indexed. A text console is present in the GUI which would report any errors encountered while establishing a connection to the database. The system configuration information in the text fields including the table that is selected for querying is written to a CONFIG file that is used by other components of our system.

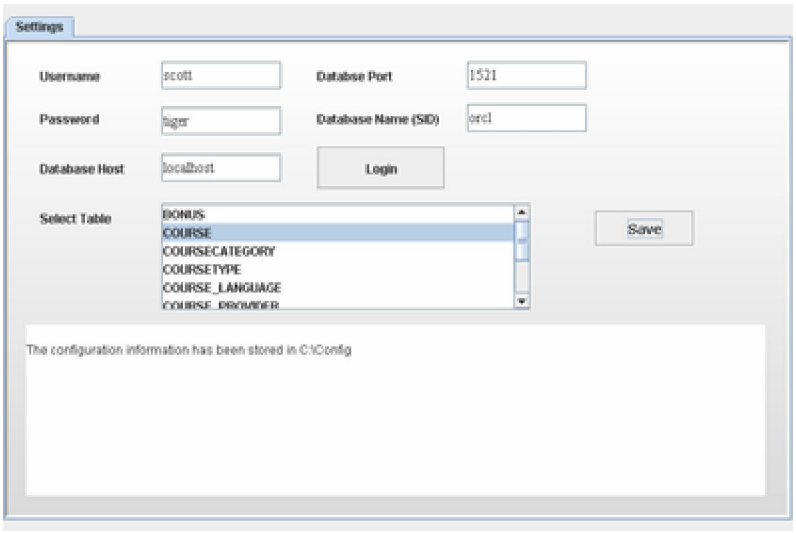

Figure 3.1: System configuration GUI

3.2 Information collection

The course property information is comprehensive and tedious. Practically, it is loaded into our system from the data files provided by the course vendor companies. We have used the Oracle database's SQL Loader to collect the course property information from plain textual files. Loading such data in other popular formats such as Excel or XML will be a straightforward programming task.

In daily maintenance, we also need to input or modify such information in the interactive mode. Figure 3.2 shows my GUI.

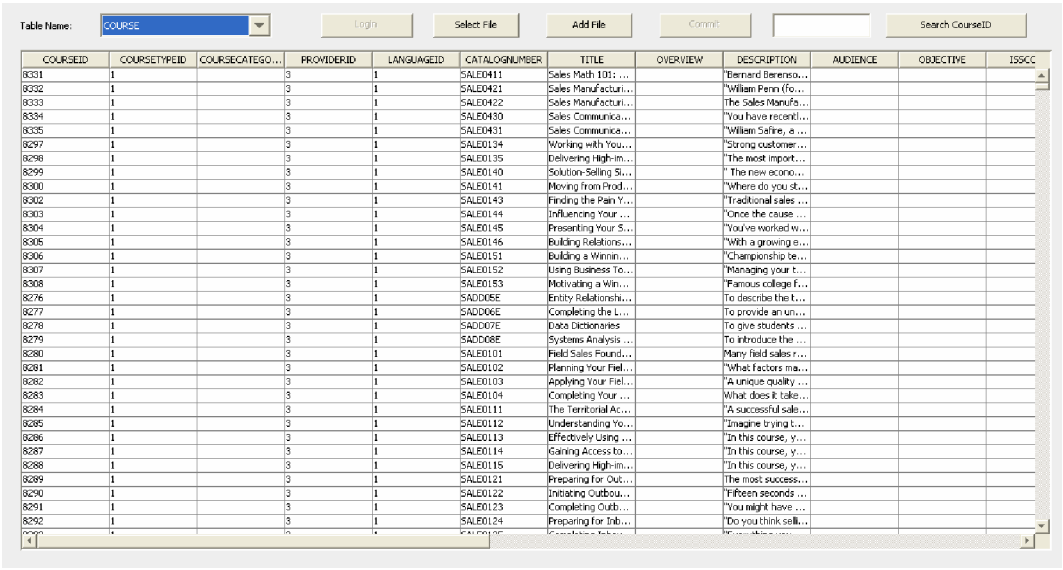

Figure 3.2: Add/edit course properties information

The buttons at the top are “Add course”, “Search course ID”, and “Add document”. The entire table is editable. The “Add course” button will add a new row to the table. “Search course ID” help the user locate a specific course, with which editing can be done. The “Add document” button allows the user to associate a document file with the current selected course [Figure 3.3].

Figure 3.3: Add/edit GUI Buttons

3.3 Database table management

The course property information can be modified in the interactive mode, each course property can be modified individually and saved by clicking the Save button which is provided for each individual course as shown in my GUI below Figure 3.4.

COURSEID	TITLE	CATALOGNUMBER	DESCRIPTION	INSERTDATE	ISACTIVE	Commit
7559	Organizational In...	HR0143	"""We tried diversity, and it didn"t wor...	18-MAY-05	1	Save
7560	Corporate Cultur...	HR0144	"Has your company tried implementing...	18-MAY-05	1	Save
7561	Management Skill...	HR0145	Just how do managers handle diversit...	18-MAY-05	1	Save
7562	Communication a...	HR0146	"Companies can recruit and hire emplo...	18-MAY-05	1	Save
7563	Workplace Haras...	HR0151	"This course addresses issues of work...	18-MAY-05	1	Save
7564	Diversity in the W...	HR0152	Did you know that: 1) Hispanics are th...	18-MAY-05	1	Save
7565	Business Ethics	HR0153	"Think about the last difficult decision ...	18-MAY-05	1	Save
7566	Family and Medic...	HR0154	Have you or a family member ever be...	18-MAY-05	1	Save

Figure 3.4: Database management

When the user clicks the Save button a SQL insert statement is generated to insert the modified values into the database Figure xx shows a SQL statement generated by my program.

```
INSERT INTO COURSE (COURSEID, COURSETYPEID, PROVIDERID, LANGUAGEID, CATALOGNUMBER,
TITLE, INSERTDATE, ISACTIVE) VALUES (75313, 1, 3, 1, 'HAR0001', 'Coaching For
Results', '18-MAY-05', '1')
```

The above SQL statement is generated in JAVA using Vectors and Strings. When the user clicks the save button the individual course property is converted to a Vector and a SQL statement is generated as shown in my code below.

```
String stmt = "INSERT INTO " + tableName + " VALUES(";
for (int i=0; i<cTypes.length; i++) {
    if (i > 0) stmt = stmt + ", ";
    if (cTypes[i] == java.sql.Types.NUMERIC)
        stmt = stmt + dataRow.elementAt(i);
    else
        stmt = stmt + "'" + dataRow.elementAt(i) + "'";
}
stmt = stmt + ")";
```

3.4 Indexing

The main motive behind this thesis was to build a successful indexer that is capable of indexing a variety of documents. The Indexer GUI is used to index a directory of documents that contains Rich Text Format Documents, PDF Documents, Word Documents and Text Documents. The indexer indexes all the sub directories recursively in the user specified directory. In my GUI the user can enter the source directory either manually or by using the file selector which is available for both the text fields (Destination Directory and Source Directory), the destination directory stores the indices for all the indexed documents, indexing is based on Lucene indexing as stated in the previous chapters. All the details entered in the text field either manually or by using the file selector are automatically written to the CONFIG file. The indexer also has an output console that can report any errors encountered during the indexing of the documents as shown in Figure 3.5.

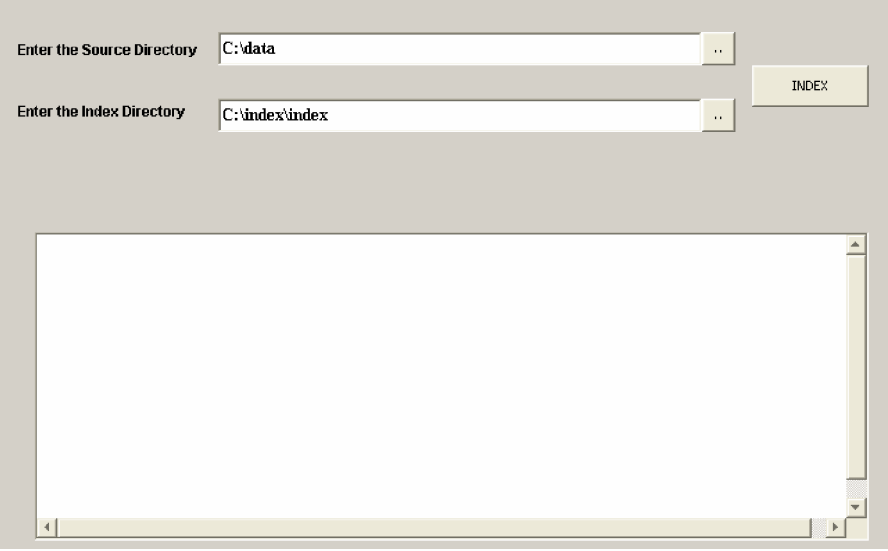

Figure 3.5: Indexing GUI

3.5 Searching

The relationship between constraints by the course properties and the keyword search results can be best explained by considering the Searcher GUI in Figure 3.6.

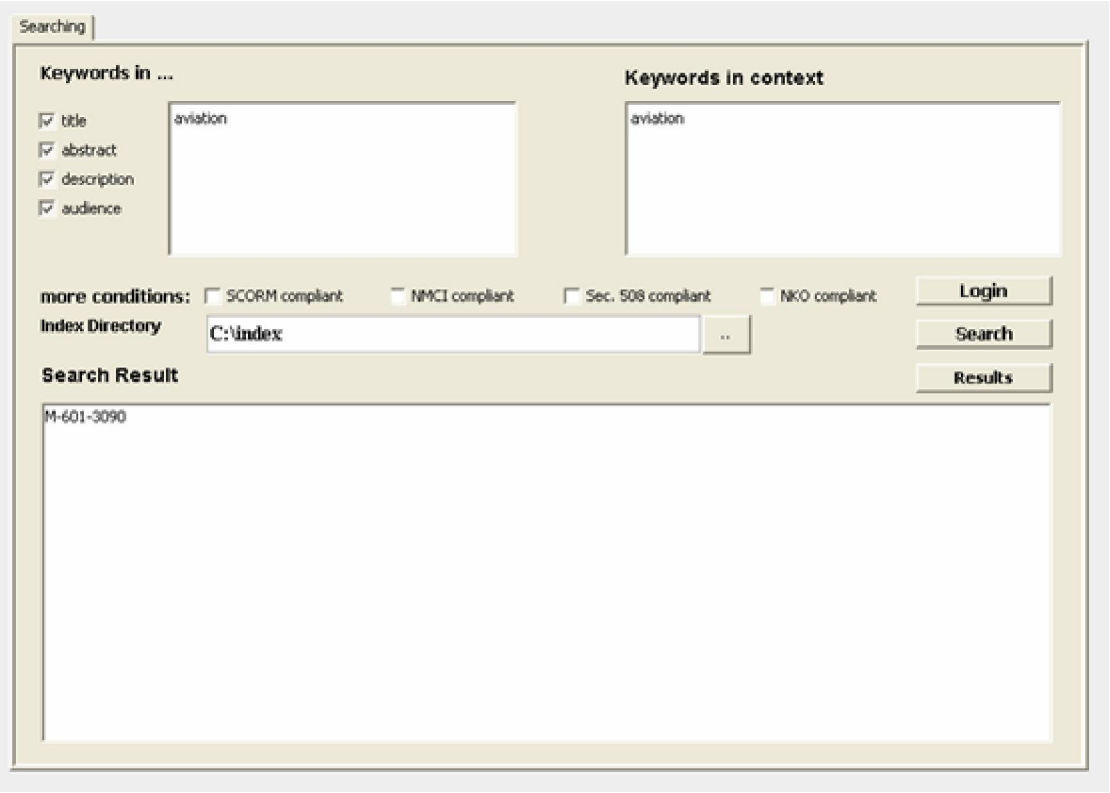

Figure 3.6: Searcher GUI

The upper left part of the GUI lists the choice of desired text fields namely title, abstract, description and audience. The logical relation between different fields is "OR". In the example shown in Figure 3.y, since all four fields (title, abstract, description and audience) are checked, as long as the keyword "aviation" appears in any one of these four fields, the course will be a candidate selection. In addition, the four checkboxes "SCORN compliant", "NMCI compliant", "Sec 508 compliant", and "NCO compliant" are four Boolean conditions regarding the standard compliance of the course (See Fig 3.7). Note, all these eight pieces of information are managed by the database. Thus, these constraints will be enforced by a SQL query.

Figure 3.7: Searcher GUI Boolean Conditions

Although there is only one word (“aviation”) shown in the left-hand side keyword area, my GUI and SQL query are capable of processing any number of keywords given in the text area. If the user does not give any keyword in the left-hand side keyword area, then the title, abstract, description, and audience checkboxes will be ignored. On the other hand, if the user gives at least one keyword in the left-hand side keyword area but does not select any checkbox for title, abstract, description, and audience, then all the four checkboxes will be automatically selected.

The upper right part of the GUI is the text area for “Keywords in context”. If no word is given in this area, then the Lucene search engine will not be used. Giving at least one word in this area will kick off Lucene search engine which return course identifiers (course catalog codes) that contains the given words based on the given index. Furthermore, Lucene search engine will also generate an HTML document with keywords highlighted for each returned course concurrently.

When multiple words can be given, the user should specify the logical operations, “OR”, “AND”, “NOT” between words. Lucene searcher can interpret Boolean queries effectively and perform logical operations on a set of key words. For examples, “aviation **OR** army” will return the documents which contains either word “aviation” or “army”. Similarly, “aviation **AND** army” will return the documents containing both words “aviation” and “army”. If “aviation NOT army” is given, then the documents which contain the word “aviation” but not the word “army” will be returned. Logical operations can also be grouped. For example, “(Aviation **OR** naval) **AND** army” can be used as the searching criterion effectively. In addition to the logical operations in the searching words, Lucene can also perform wild card searches. For example, “av*” returns all the documents that contain words starting with “av”.

Lucene supports fuzzy searches based on the Edit Distance algorithm [Levenshtein]. To do a fuzzy search one has to use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search “**`roam~`**”. This search will find terms like foam and roams. More advanced proximity search by using the tilde at the end of a phrase. For example to search for a "apache" and "jakarta" within 10 words of each other in a document use the search:

`"jakarta apache"~10`

It is critical to integrate the query results returned by the database with the searching results returned by Lucene. Technically, the final results should be the logical intersection of both results. My program design is explained in the next section.

3.6 Integrating keyword search with course property query

Logically, the database query and the Lucene search engine run independently. In a sense, the database query computation serves as a filter that eliminates the courses selected by Lucene but failed in satisfying the course property conditions. A critical step is to compute the intersection between the results produced by Lucene and the result returned by the database query as the final result. Figure 3.8 illustrates the architecture of my system.

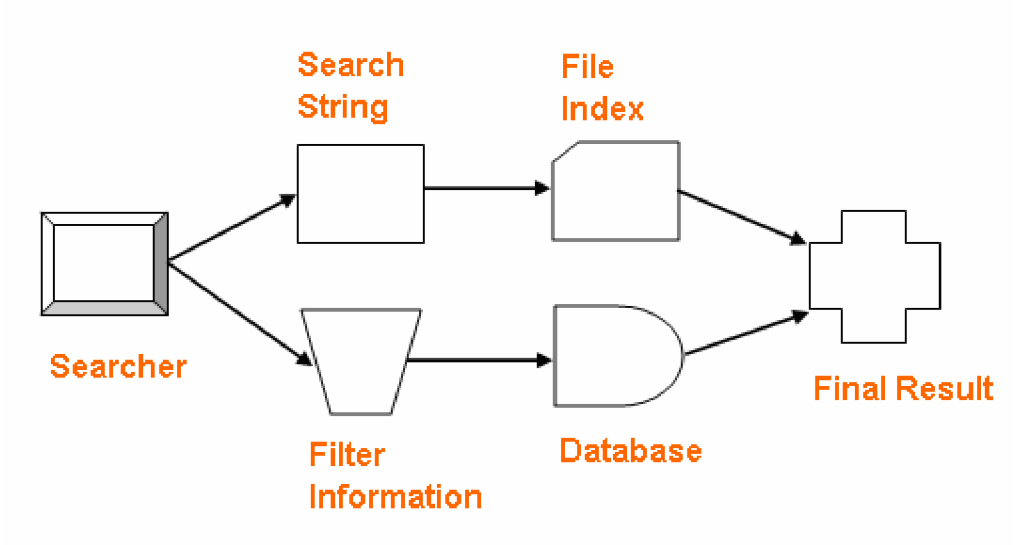

Figure 3.8 Searcher architecture

4. System Implementation

In my project, I have learned and utilized a number of technologies including Eclipse, JDBC, Lucene indexer, and Lucene searcher. An overview of my system is shown in Figure 4.1. Lucene can index a variety of documents. All the documents that are to be indexed are stored in the same directory and location of the source documents folder has to be specified to the program prior to indexing.

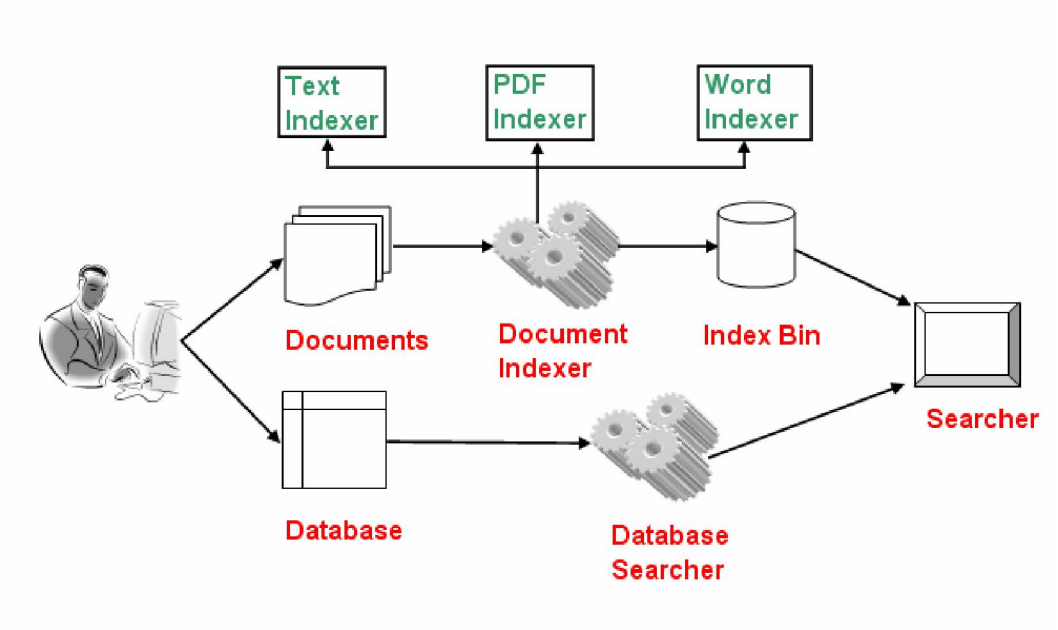

Figure 4.1: System Overview

4.1 Database queries

In my implementation, I need only two tables to accommodate the course properties and the document locations associated with the courses. They are course Table and course_document. Table course_document has two fields only, (course_ID, document). Table course has its schemas shown in Figure 4.2.

Columns

Name	Datatype	Size	Scale	Nulls?	Default Value
COURSEID	NUMBER	9	0		
COURSETYPEID	NUMBER	9	0	✔	
COURSECATEGORYID	NUMBER	9	0	✔	
PROVIDERID	NUMBER	9	0		
LANGUAGEID	NUMBER	9	0	✔	
CATALOGNUMBER	VARCHAR2	50		✔	
TITLE	VARCHAR2	255			
OVERVIEW	VARCHAR2	4000		✔	
DESCRIPTION	VARCHAR2	4000		✔	
AUDIENCE	VARCHAR2	4000		✔	
OBJECTIVE	VARCHAR2	4000		✔	
ISSCORM	CHAR	1		✔	
ISNMCI	CHAR	1		✔	
ISSECTION508	CHAR	1		✔	
NKO_BOOL	CHAR	1		✔	
URL	VARCHAR2	255		✔	
ACCESSINSTRUCTIONS	VARCHAR2	500		✔	
CREATEDATE	DATE			✔	
UPDATEDATE	DATE			✔	
RETIREDATE	DATE			✔	
INSERTDATE	DATE				
DURATION	VARCHAR2	50		✔	
COST	NUMBER	6	2	✔	
ISACTIVE	CHAR	1			1

Figure 4.2: Table Schema

A challenge in the JDBC programming is to generate the query that searches multiple words in up to four fields (title, abstract, description and audience). I used the following Java code to solve this problem.

```
// Split the filter query input with space as delimiter
String fq = fquer.getText();
String[] fqry = fq.split(" ");

// Append OR inbetween the String
for (int col = 0; col < selcount; col++) {
    for (int qterm = 0; qterm < fqry.length; qterm++) {
        if ((qterm > 0) || (col > 0)) {
            genqry = genqry + "OR ";
        }
        genqry = genqry + "LOWER(" + selitems[col] + ")" + " like "
        + "'%" + fqry[qterm] + "%' ";
    }
}
```

In my code I initially get the fields to search the keywords and then the keywords are split into separate words using space as the delimiter. An example of the query generated by my code is shown below.

```
SELECT CATALOGNUMBER, TITLE FROM COURSE WHERE (LOWER (TITLE) LIKE '%Aviation%' OR LOWER (OVERVIEW) LIKE '%Aviation%' OR LOWER (DESCRIPTION) LIKE '%Aviation%' OR LOWER (AUDIENCE) LIKE '%Aviation%')
```

4.2 Indexer

The document indexer indexes all the source documents. Indexing of multiple formats is handled in a very simple manner using an If – then – else syntax trying to understand the file extension, and calling the right method for a particular extension. The index of each document is appended to the previous index to make the entire process efficient. Documents can be present in multiple directories and the indexer can index all the files and the subdirectories within the user specified directory. Moreover the documents need not be sorted by their file types all the file types can co-exist in the same directory and the indexer can sort the various file types automatically. To utilize Lucene indexer, I had to study Lucene index classes. I used the following Java code to handle multiple formats.

```
File[] files = dir.listFiles();

for (int i = 0; i < files.length; i++) {
    File f = files[i];
    if (f.isDirectory()) {
        indexDirectory(writer, f);  // recurse
    } if (f.getName().endsWith(".txt")) {
        indexFile(writer, f);
    } else if (f.getName().endsWith(".pdf")) {
        indexpdf(writer, f);
    } else if (f.getName().endsWith(".doc")) {
        indexdoc(writer, f);
    } else if (f.getName().endsWith(".rtf")) {
        indexrtf(writer, f);
    }

}
```

4.2.1 Lucene Index Classes

During the indexing process there are five basic Lucene classes, they are the IndexWriter, Analyzer, Directory, Document, and Field

IndexWriter

An IndexWriter creates and maintains an index. The IndexWriter takes an argument that determines whether a new index is created, or whether an existing index is opened for the addition of new documents.

Opening an IndexWriter creates a lock file for the directory in use. When we try to open another IndexWriter on the same directory will lead to an IOException. As a matter of fact IndexWriter is the only class that has write-access to the index and using its methods one can add documents to the index for searching purposes.

Analyzer

An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The analyzer discards text that is not useful for a searching application. Lucene has a number of analyzers some of them are BrazilianAnalyzer, ChineseAnalyzer, CJKAnalyzer, CzechAnalyzer, SimpleAnalyzer, SnowballAnalyzer, StandardAnalyzer, StopAnalyzer and a WhitespaceAnalyzer. The most generic and important analyzers are explained below

SimpleAnalyzer

A SimpleAnalyzer uses a Tokenizer that converts all of the input to lower case.

StopAnalyzer

A StopAnalyzer includes the lower-case filter, and also has a filter that drops out any "stop words", words examples of stop words include a, an, the, etc. A stop word can be defined as a word that occurs commonly in a given language. A stop word is analogous to noise in an electrical circuit. A StopAnalyzer always comes with a preset of stop words, but one can always add or remove words from the preset list.

StandardAnalyzer

A StandardAnalyzer does both lower case and stop word filtering, and in addition it also tries to do some basic clean-up of words, for example taking out apostrophes (') and removes periods. The current application which is designed uses the StandardAnalyzer.

To summarize the three analyzers, the output when given a particular string is tabulated below

Analyzer	**Analyzer Result**
	[lucene] [is] [a] [gem] [in] [the] [open]
Simple Analyzer	[source] [world]
Stop Analyzer	[lucene] [gem] [open] [source] [world]
Standard Analyzer	[lucene] [gem] [open] [source] [world]

Directory

The Directory class represents the location of the index. It consists of FSDirectory and RAMDirectory classes. FSDirectory class stores the index in the file system while RAMDirectory class stores the index in the memory. Because RAMDirectory does not write anything to the disk, it is faster than FSDirectory. However, since computers usually come with less RAM than hard disk space, RAMDirectory is not suitable for very large indices.

Document

Documents are the unit of indexing and search. A Document is a set of fields. Each field has a name and a textual value. A field may be stored with the document, in which case it is returned with search hits on the document. Thus each document typically contains one or more stored fields which uniquely identify it.

Field

A field is a section of a Document. Each field has two parts, a name and a value. Values are free text, provided as a String or as a Reader, or they are atomic keywords, which are not further processed. Such keywords are used to represent dates, urls, etc. Fields are optionally stored in the index, so they are returned with hits on the document. The different types of fields, their characteristics and common usage are listed in the table below.

Field Type	Stored	Indexed	Analyzed	Example
Keyword	Y	Y	N	Empid,Empname
UnIndexed	Y	N	N	Database primary key,Url
UnStored	N	Y	Y	Web page title /contents
Text	Y	Y	Y	Contents

Finally, the entire indexing procedure can be summarized as shown in figure 4.3

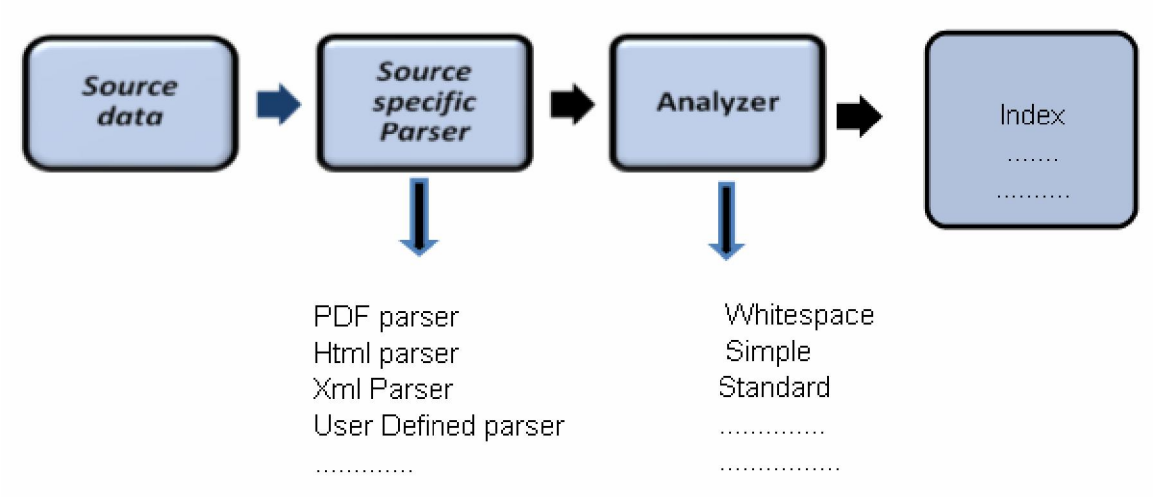

Figure 4.3: Lucene indexing architecture

4.2.2 Indexing PDF files

Parsers

In order to index various documents like PDF, Word documents, HTML etc using Lucene one has to convert it into textual format. Lucene does not have built in parsers like a search engine library. Therefore to index various kinds of data one has to use external tools that can be easily integrated with Lucene. An example is using PDFBox to index PDF files.

PDF (Portable Document Format) documents were originally created by Adobe, today the PDF document has become a standard format among documents. PDF documents are virtually used

everywhere in the electronic media. Hence it is important to have a good information retrieval system for PDF documents.

PDFBox is used to extract text from a PDF document; this is done as Lucene can accept data that is only in textual format. PDFBox is a free open source library that can be downloaded from their website. One of the main features of PDFBox is its ability to quickly and accurately extract text from a variety of PDF documents. This functionality is encapsulated in the PDFTextStripper class which can extract text from the given PDF document. PDF files can be indexed using two methods; both involve the usage of PDFBox.

Simple text extraction and Indexing

This method makes use of the PDFTextStripper class. This class takes a PDF document and strips out all of the text and ignores the formatting and such. The output of the PDFTextStripper class is written to a text document and the document is passed as such to the Lucene indexer. A snippet using this method is shown below

```
fis = new FileInputStream(f);
PDFParser parser = new PDFParser(fis);
parser.parse();
PDDocument pdfDocument = parser.getPDDocument();
PDFTextStripper stripper = new PDFTextStripper();
String pdftext = stripper.getText(pdfDocument);
```

This writeText class takes a PDF document and writes the text of that document to the print writer. The entire procedure can be summarized as shown in figure 4.4

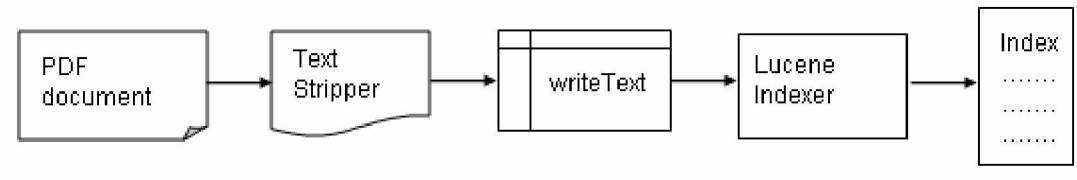

Figure 4.4: Indexing PDF files

Lucene Integration

The main advantage of using PDFBox is the ease at which it can be integrated with Lucene. PDFBox provides a simple approach for adding PDF documents into a Lucene index.

```
Document luceneDocument = LucenePDFDocument.getDocument( ... );
```

Once a Lucene Document object is created, it can be added to the Lucene index as though it had been created from a text or a HTML file. The LucenePDFDocument automatically extracts a variety of metadata fields from the PDF to be added to the index, the details of those fields are tabulated below.

Lucene Field Name	Description
Path	File system path if loaded from a file
url	URL to PDF document
Contents	Entire contents of PDF document, indexed but not stored
Summary	First 500 characters of content
Modified	The modified date/time according to the url or path
Uid	A unique identifier for the Lucene document.
CreationDate	From PDF meta-data if available
Creator	From PDF meta-data if available
Keywords	From PDF meta-data if available
ModificationDate	From PDF meta-data if available
Producer	From PDF meta-data if available
Subject	From PDF meta-data if available
Trapped	From PDF meta-data if available

Advanced Text Extraction

It is possible to perform advanced text extraction using PDFBox. One can utilize or extend the PDFTextStripper class to perform the same.

Limiting the Extracted Text

When using PDFBox it is possible to limit the text that is extracted during the extraction process. The simplest way is to specify the range of pages in which the text needs to be extracted. The snippet shown below extracts text between page 7 and 8

```
PDFTextStripper stripper = new PDFTextStripper();
stripper.setStartPage( 7 );
stripper.setEndPage( 8 );
stripper.writeText( ... );
```

4.2.3 Indexing Microsoft Word and Rich Text Format (RTF) Documents

Indexing Microsoft Word Documents

Microsoft word documents can be indexed using Lucene when the text from the word document is extracted. There are a number of packages that are available to extract text from a word document. In this thesis the text mining package which is available with the Lucene sand box is used to extract the text from the document. This package is capable of extracting text from Microsoft Word 6.0/95/97/2000/XP

The text can be extracted very easily from the word document as shown below

```
bodyText = new WordExtractor().extractText(new FileInputStream(f));
```

The document should be sent in as a File Input Stream for the word extractor to function properly.

Indexing Rich Text Format (RTF) Documents

RTF documents can be indexed with the help of the RTFEditorkit supplied with the JAVA SDK. The text from the RTF document is read by using the read method of the RTFEditorkit and the text is extracted by using the getText method of the tool kit. This is demonstrated in the code snippet shown below

```
new RTFEditorKit().read(new FileInputStream(f), styledDoc, 0);
bodyText = styledDoc.getText(0, styledDoc.getLength());
```

The read position can also be specified when reading and extracting text from a RTF document.

4.2.4 Using Lucene Index Classes

Once the text has been extracted from a document using any of the methods explained in the previous sections, the text content has to be indexed. Lucene can accept only text to index. The following code is used to index the text content.

```
//Create a New Document
Document doc = new Document();
//Add the Contents and the Filename
doc.add(Field.Text("contents", pdftext));
doc.add(Field.Keyword("filename", f.getCanonicalPath()));
//adds the Document to Indexer
writer.addDocument(doc);
```

A new document has to be created; the contents and the filename are added to a specific field type. There are four types of fields Text, Keyword, UnIndexed and UnStored.

4.3 Searcher

Searching is the process that follows indexing. Once the required documents are indexed a search method has to be implanted. A search method takes the user queries and they are parsed using the searcher parser. The results of a search method consist of hits from the index.

4.3.1 Search Classes in Lucene

The main search classes in Lucene are IndexSearcher, Query, Term, and Hits.

IndexSearcher

An IndexSearcher searches a document from an index. An index is opened in read only mode and uses its methods to return the search results. The final results can be printed or sorted or listed.

Query

A query class is used for defining user queries. There are a number of query types in Lucene, the various query types are BooleanQuery, FilteredQuery, MultiTermQuery, PhrasePrefixQuery, PhraseQuery, PrefixQuery, RangeQuery, SpanQuery and TermQuery. The QueryParser class can automatically understand which type the user query belongs to. The Backus-Naur form (BNF) grammar of a Lucene query is

```
Query ::= ( Clause )*

Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")"
```

<TERM> specifies the index field in which the terms are searched

["+", "-"] '+' specifies the query parser to include the clause in the search criteria while '-' specifies the query parser to remove the clause from the search criteria.

Term

The term class represents the text in a document while searching. The term class takes two parameters, (a field and a text) the field is one in which the text will be searched. The term class is used for constructing the user query.

Hits

After the construction of a query, the IndexSearcher class searches the documents from the index. The results of the IndexSearcher class are pointed by the Hits class.

Scoring

Lucene scoring is fast and it hides almost all of the complexity from the user. Lucene scoring uses a combination of the Vector Space Model (VSM) of Information Retrieval and the Boolean model to determine how relevant a given Document is to a User's query. In VSM the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. It uses the Boolean model to first narrow down the documents that need to be scored based on the use of Boolean logic in the Query specification. Lucene also adds some capabilities and refinements onto this

model to support Boolean and fuzzy searching, but it essentially remains a VSM based system at the heart. In Lucene scoring is very much dependent on the way documents are indexed; the objects that are scored are the Documents and a Document is a collection of Fields. Each Field has semantics about how it is created and stored (i.e. tokenized, untokenized, raw data, compressed, etc.).Lucene scoring works on Fields and then combines the results to return Documents. This is important because two Documents with the exact same content, but one having the content in two Fields and the other in one Field will return different scores for the same query due to length normalization (assuming the DefaultSimilarity on the Fields).

The Lucene Scoring Formula

Lucene's scoring formula computes the score of one document *d* for a given query *q* across each term *t* that occurs in q. The score attempts to measure relevance, so the higher the score, the more relevant document *d* is to the query *q*.

```
score (q,d) = sum  t in q( tf (t in d) * idf (t)^2 * getboost(t in q) *
getBoost (t.field in d) * lengthNorm (t.field in d) ) * coord (q,d) *
queryNorm (sumOfSquaredWeights)

where

sumOfSquaredWeights = sumt in q( idf (t) * getBoost (t in q) )^2
```

This scoring formula is mostly implemented in the TermScorer class, where it makes calls to the Similarity class to retrieve values for the following. tf(t in d) - Term Frequency - The number of times the term *t* appears in the current document *d* being scored. Documents that have more occurrences of a given term receive a higher score.

1. idf(t) - Inverse Document Frequency - One divided by the number of documents in which the term *t* appears. This means rarer terms give higher contribution to the total score.
2. getBoost(t in q) - The boost, specified in the query by the user, that should be applied to this term. A boost over 1.0 will increase the importance of this term; a boost under 1.0 will decrease its importance. A boost of 1.0 (the default boost) has no effect.

3. lengthNorm(t.field in q) - The factor to apply to account for differing lengths in the fields that are being searched. Typically longer fields return a smaller value. This means matches against shorter fields receive a higher score than matches against longer fields.

4. coord(q, d) - Score factor based on how many terms the specified document has in common with the query. Typically, a document that contains more of the query's terms will receive a higher score than another document with fewer query terms.

5. queryNorm(sumOfSquaredWeights) - Factor used to make scores between queries comparable.

4.3.2 Searching PDF documents

As described in Section 4.2.2, PDF documents are indexed. A search application has to be written to search index for the given query. The search application written for a PDF document is similar to that of a normal text search. The searcher application uses the IndexSearcher and the FSDirectory classes to open the index for Searching, the IndexSearcher constructor takes the index directory as a parameter and the Query Parser takes the human readable query into Lucene's Query class, and a standard analyzer is used to index the PDF document; hence the same analyzer is used to search the Lucene index also. Searching returns hits in the form of hits object. I have used the following code to search my Lucene index.

```
        //open the Index
fsDir = FSDirectory.getDirectory(indexDir, false);
IndexSearcher is = new IndexSearcher(fsDir);
        //Parse the given Query
QueryParser parser = new QueryParser("contents",
        new StandardAnalyzer());
Query query = parser.parse(q);
        //Search the Index
hits = is.search(query);
scorer = new QueryScorer(query);

String[] IndexSearchResult1 = new String[hits.length()];
for (int i = 0; i < hits.length(); i++) {
           // Retrieves the matching document
    Document doc = hits.doc(i);
             //Displays the filename
    String f = doc.get("filename");

    File fi = new File(f);
    String fii = fi.getName();

    int fext = fii.lastIndexOf(".");
    if (fext != 1)
        fii = fii.substring(0, fext);
    IndexSearchResult1[i] = fii;
```

4.3.3 Search Results

Once Lucene completes a search, the results are displayed in a very easy to use manner. A HTML page is generated with all the search results. The search results page consists of highlighting the search string and also extracting the best fragments. This is all achieved by using the highlighter class supplied with Lucene.

The highlighter class is used to markup highlighted terms found in the best sections of a text, using configurable Fragmenter, Scorer, Formatter, Encoder and Tokenizers. The search results consist of a small fragment of the document containing the search string. The fragment that is extracted has the maximum number of occurrences of the search string and this is achieved by using the getBestFragment method. The getBestFragment method extracts the most relevant sections and the document text is analyzed in chunks to record hit statistics across the document.

Formatter

After the best fragments are obtained they need to be formatted. The formatter processes terms found in the original text, typically by applying some form of mark-up to highlight terms in HTML search results pages There are two types of formatter available with this API. The GradientFormatter and the SimpleHTMLFormatter. A simple HTML formatter is used in this application, the simple formatter highlights terms by applying a pre tag and a post tag.

Results formatting

Results are available for the search queries in a very friendly manner, there are two ways how the results is displayed. A short summary of each document is presented when the user clicks the Results button on the GUI or a List of results without the summary is automatically displayed in the GUI.

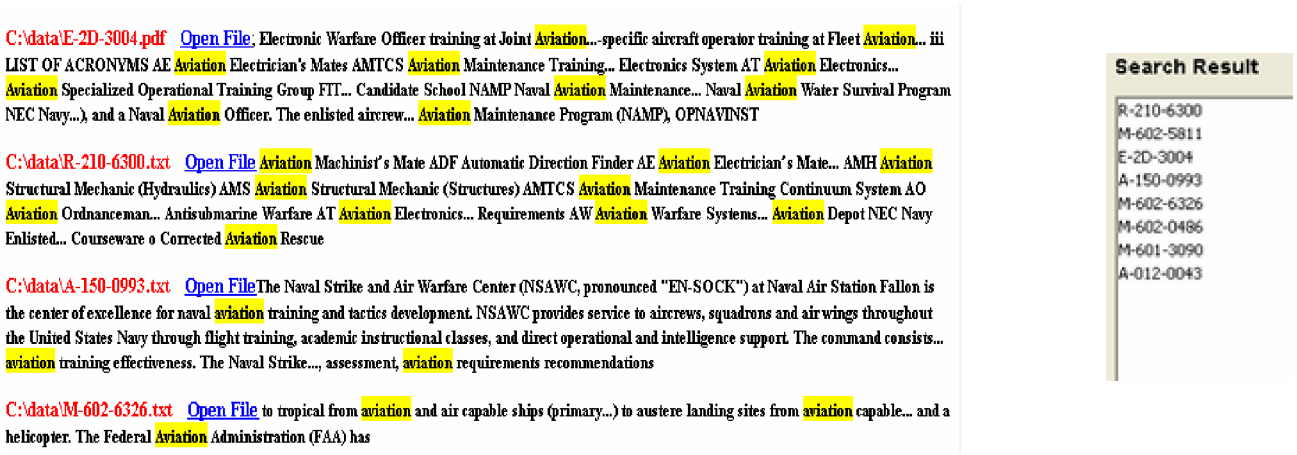

Figure 4.5: **Formatted Results**

When the user clicks on a document from the list, the user can view the entire document in the output window which disables the summary view.

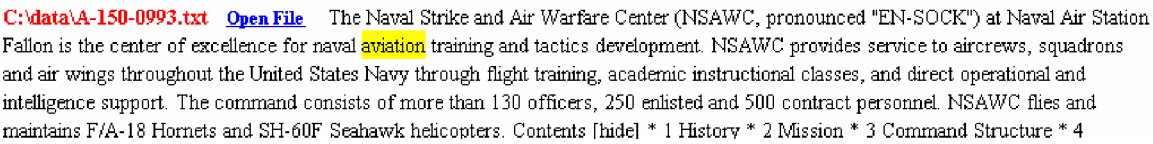

Figure 4.6: Result File View

It can be seen from the above figure all the results have the search query highlighted in yellow color. This is made possible by using the Lucene highlighter class. The results that are displayed in the result list or in the summary page are also ranked, meaning the documents with the maximum number of hits occupy a higher position in the list or in the display of summary results. Hits mean the documents that contain the maximum number of the search string.

5. Results

A document indexer and a searcher that combines one of the IR techniques – keyword searching with the database techniques through a filtering process were implemented successfully.
The indexer is capable of indexing a variety of document formats and a Graphical User Interface based on the implementation of Lucene was built successfully as shown in the screen shot below.

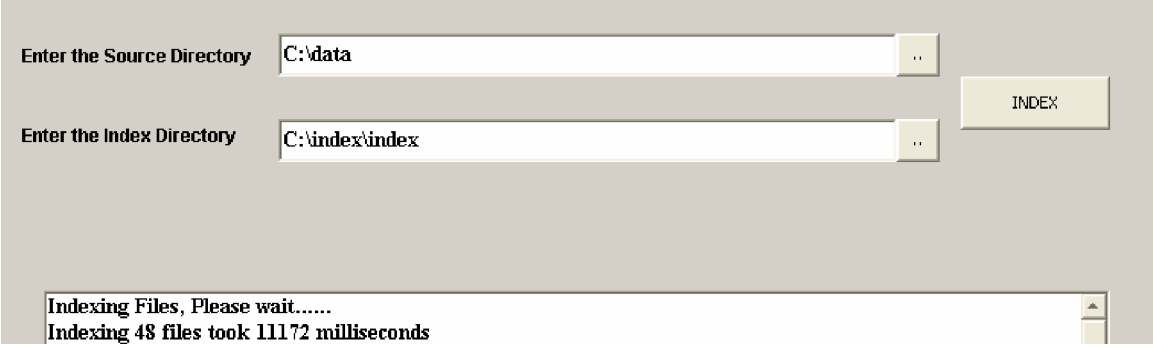

Figure 5.1: Indexer Results (GUI)

A Lucene searcher capable of searching the Lucene document indices with filtering processes and displaying the results in HTML format were implemented. The formatted results have the search terms highlighted.

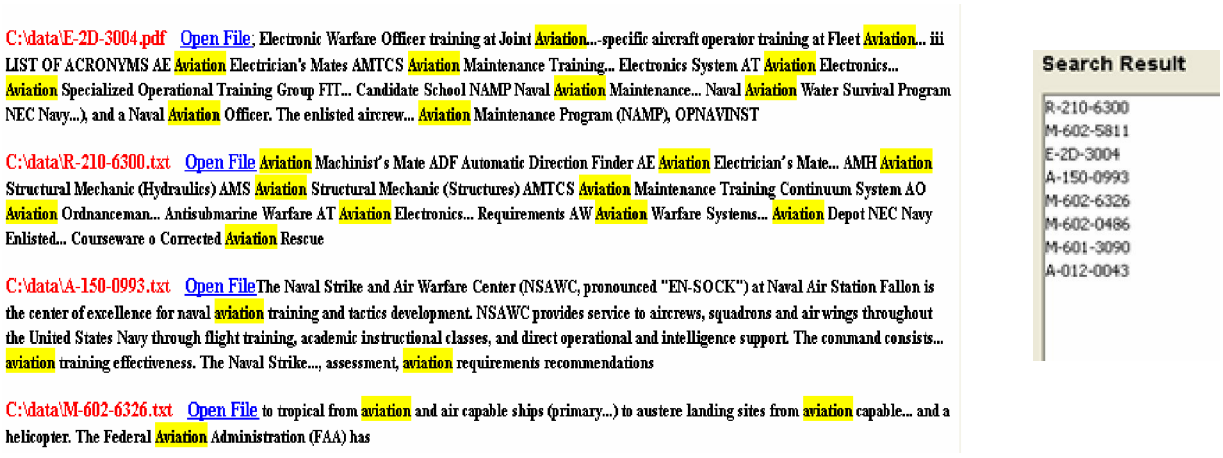

Figure 5.2: Formatted Results

A System configuration GUI where the user can set up the database connection and table information was built successfully. This GUI provides the user an interface to enter the login credentials to get access to a particular database.

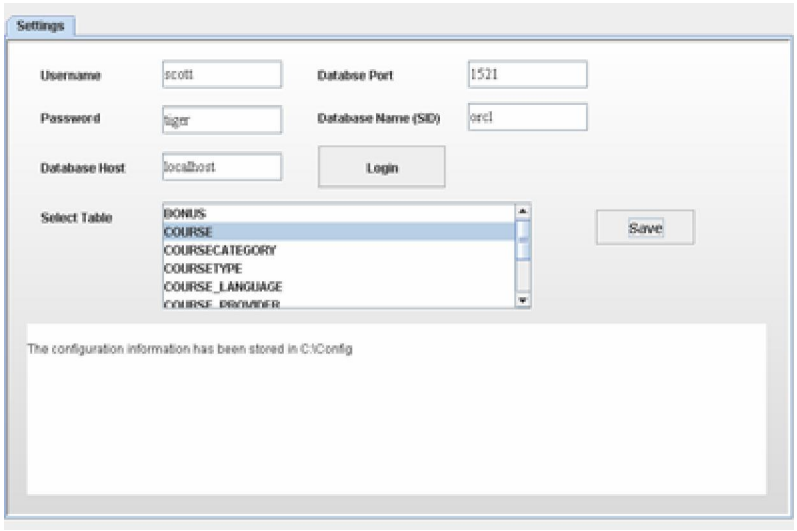

Figure 5.3: Configuration GUI

A Graphical User Interface for Course property management was built where the user can setup or modify course property fields according to specific needs without dealing with database tables. This GUI is capable of database management.

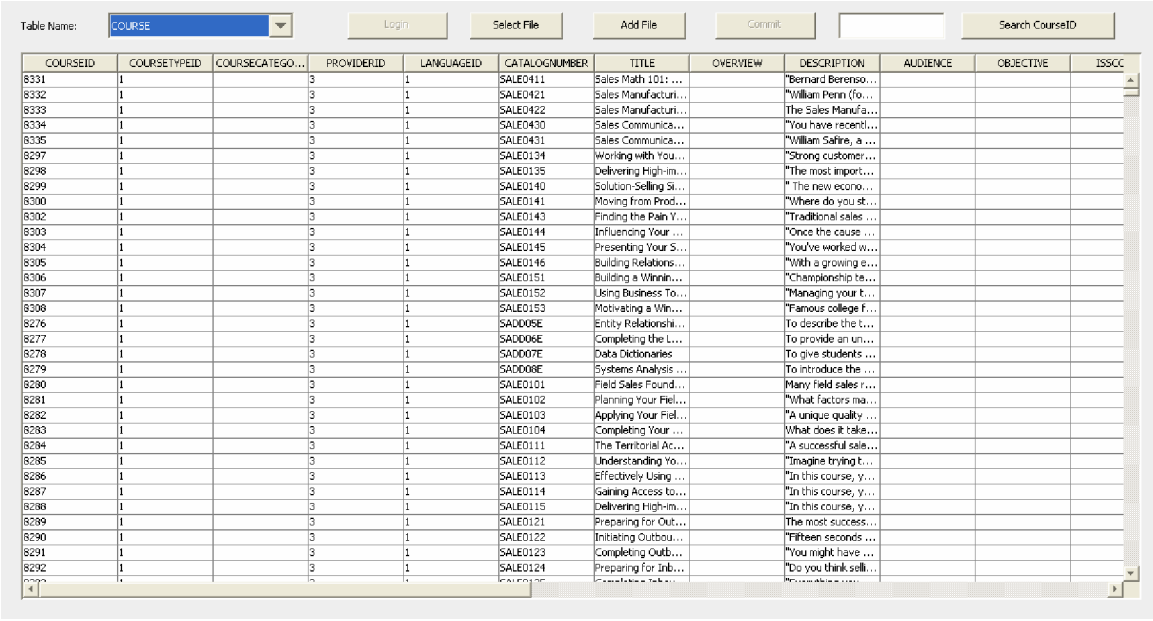

Figure 5.4: Table Modifier GUI

6. CONCLUSION / FUTURE WORK

A successful indexing and searching application to support the management of training courses from tens of thousands of entries owned by a military branch where the users need to search for courses against specific training needs was built using Lucene. This application integrated information retrieval from unstructured documents and traditional database querying for the structured information such as the courses' properties.

For the unstructured documents, my system can index a variety of file formats and search in a variety of applications. By taking advantage of Lucene's portability and scalability, my system is highly customizable. It supports a variety of keyword search queries such as wild card searches, Boolean queries, and fuzzy searches. While Lucene is a highly sophisticated search engine, it is not possible to automatically search documents using Lucene. Lucene provides a framework for writing a customized search application. Although Lucene only supports simple text, I used a number of third-party software that can convert HTML, XML, Word and PDF documents into simple text. Once converted, the keyword-search based information retrieval system was built. Additional plug-ins is available to expand the capabilities of search libraries and the ability to fetch data from different sources.

For the structured information, I applied the relational database techniques especially the string search features in SQL. Upon my Java GUI components collecting search constraints from the user, the application automatically forms very sophisticated SQL queries. Experiments showed that my database querying part performed cause little delay.

The complexity of my design and implementation came from the integration of Lucene indexing engine and the database querying system. My approach was to filter the Lucene search results with the database query results, which has been effective in my experiments.

The application implemented in this project effectively fulfilled the core backend-server functions. The results produced by my system are ready for Web delivery. In the future my application can be embedded into a Web application or Web services.

7. REFERENCES

[Crane] G. Crane and A. Jones, “Text, Information, Knowledge and the Evolving Record of Humanity”, D-Lib Magazine, 12:3, March 2006.
http://www.dlib.org/dlib/march06/jones/03jones.html

[IBM] IBM AlphaWorks, Unstructured Information Management Architecture SDK, http://www.alphaworks.ibm.com/tech/uima

[Moens] M-F Moens, “Combining Structured and Unstructured Information in a Retrieval Model for Accessing Legislation”, Proceedings of the 10th International Conference on Artificial Intelligence and Law, Bologna, Italy, pp 131-145, June, 2005.

[Carnegie]Carnegie Mellon Software Engineering Institute, COTS-based systems initiative, http://www.sei.cmu.edu/cbs/

[Brownsword] Lisa Brownsword, Forward, J. Dean, A. Gravel (Eds.), *COTS-Based Software Systems*, (*Proceedings of the First International Conference on COTS-Based Software Systems* (ICCBSS 2002), Orlando, FL, USA, February 4-6, 2002), LNCS 2255, Springer-Verlag, Heidelberg, Germany, 2002.

“Lucene in Action” - Erik Hatcher, Otis Gospodnetic Manning Publications (December 28, 2004)

“The Apache Software Foundation” – www.apache.org

“Apache Lucene” - http://lucene.apache.org/

“Information Retrieval” - http://en.wikipedia.org/wiki/Information_retrieval

“PDFBox – JAVA PDF Library” – www.pdfbox.org

“Text mining and natural language processing” – www.textmining.org

“Jigloo GUI builder” - http://www.cloudgarden.com/jigloo/index.html

“Lucene Wikipedia indexer” - http://schmidt.devlib.org/software/lucene-wikipedia.html

“Luke – Lucene Index Toolbox” – http://www.getopt.org/luke

VITA

Sridevi Addagada was born in Hyderabad, India and received her B.Tech degree in Computer Science Engineering from Jawaharlal Nehru Technology University. She completed her undergraduate final project on Digital Signatures. She was admitted to the graduate school of University of New Orleans, New Orleans in January 2004 in the Department of Computer Science Engineering. Her graduate studies were concentrated on Systems and Networks.