

Timothy A. Ross. Searching without SQL: Re-engineering a database-centric web application with open-source information retrieval software. A Master's Paper for the M.S. in IS degree. November, 2008. 34 pages. Advisor: Gary Marchionini.

This paper seeks to describe the process by which a database-centric web application was redesigned and rewritten to take advantage of Apache's Lucene - an open-source information retrieval software library written in the Java programming language. After the implementation of a Lucene-based text index of "semi-structured data", a college radio station's card catalog application was able to deliver higher-quality search results in significantly less time than it was able to do using just a relational database alone. Additionally, the dramatic improvements in speed and performance even allowed the search results interface to be redesigned and enhanced with an improved pagination system and new features such as faceted search/filtering.

Headings:

Information Systems - Design

Database Management Systems – Evaluation

Information Retrieval – Evaluation

Computer Software - Development

Open Source Software

SEARCHING WITHOUT SQL:
RE-ENGINEERING A DATABASE-CENTRIC WEB APPLICATION
WITH OPEN-SOURCE INFORMATION RETRIEVAL SOFTWARE.

by
Timothy A. Ross

A Master's paper submitted to the faculty
of the School of Information and Library Science
of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements
for the degree of Master of Science in
Information Science.

Chapel Hill, North Carolina

November 2008

Approved by

Gary Marchionini

Table of Contents

Introduction.....	2
Background and Description of the Existing System	3
Technical Details and the Data Model.....	7
Problems with the Existing System	9
Implementing Lucene	14
Results and Comparative Analysis	22
Additional Enhancements and Challenges.....	27
Conclusions and Future Directions.....	31

Introduction

A solid understanding of relational database principles and the ability to write queries using the Structured Query Language (SQL) are two of the most fundamental skills in the toolboxes of most server-side developers and web programmers. The clear and human-comprehensible syntax of a SQL SELECT statement make it an elegant and almost irresistible choice for a web developer who is asked to take unknown input from a user and dynamically generate a web page full of specifically tailored results derived from existing application data.

Despite the simplicity and ubiquity of SQL, however, it is not always the best solution to a data-retrieval need – even for a system in which there is already a database involved. Depending on the type of data that is contained in an application, the range of possible user inputs, and the extent to which multi-word pieces of text are present within the data, a full-text search solution with ranked search results may be a much better option. The field of information retrieval may be the bedrock of Internet search engines and other complex systems that manage large quantities of text-filled documents, but the specific methods and techniques developed in the information retrieval world can also offer great benefit to all types of applications large and small, even those in which the data is more structured than it might typically be in a system containing journal articles or blog postings. Many relational databases do not offer much, if any, support when it comes to full-text search. By using a hybrid approach that supplements a persistent

relational data store with an additional text-based index, developers can use the best of both worlds to optimize both the performance and the usability of their applications.

This paper seeks to describe the process by which a database-centric web application was redesigned and rewritten to take advantage of Apache's Lucene - an open-source information retrieval software library written in the Java programming language. After the implementation of a Lucene-based text index of "semi-structured data", a college radio station's card catalog application was able to deliver higher-quality search results in significantly less time than it was able to do using just a relational database alone. Additionally, the dramatic improvements in speed and performance even allowed the search results interface to be redesigned and enhanced with an improved pagination system and new features such as faceted search/filtering.

Background and Description of Existing System

WXYC 89.3 FM is the college radio station for the University of North Carolina at Chapel Hill. The station has a huge music library consisting of well over 50,000 records and CDs. For decades, the station had used a paper-based card catalog index to store information about its collection, but in 2002, a card catalog application that I designed and developed became the digital replacement for the aging paper-based system. In its initial 2002 version, this card catalog application was mainly a web interface that allowed DJs to search and view limited amounts of data about all of the various artists and releases in the station's music library. Special WXYC-specific "library codes" helped DJs know where to locate artists and releases within the physical music

library, which was divided up by both format and genre across several different rooms throughout the station.

An administrative interface to the system allowed music librarians and other authorized WXYC personnel to add new artists and releases to the card catalog whenever the station acquired new records and CDs. These music librarians/catalog administrators could also use this interface to modify and/or delete artist and release data as necessary. The central component of the online card catalog application, however, was its search interface. Search, after all, was the core functionality that allowed DJs to see what releases the station had for a given artist, whether or not the station had a specific album, and exactly where to find these artists and releases within the station's physical record library.

The card catalog application had two search interfaces. One was a basic “simple search” interface (see Figure 1) not too unlike the no-frills textbox that one finds on the Google homepage. This interface would simply attempt to search the station's catalog of releases using all of the default search options.

Search the **WXYC** Library:

This is a basic text search using default options. [Click here for Advanced Search](#)

Figure 1: Basic Search Interface

A second “advanced search” interface (see Figure 2) allowed DJs to tweak the search options however they saw fit. DJs could search for either artists or releases, and results could be limited by format, by genre, by specific portions of the library code, or alphabetically. Additional search metadata allowed DJs to specify how they wanted their search string to be handled by the system. These extra parameters specified what fields the system should search against (artist, title, or both), as well as whether the system should attempt to match any of the search words, all of the search words, or simply the exact phrase in the order that it was typed.

[Click here for Simple Search](#) **Search the Card Catalog**

Bring back results as:	<input checked="" type="radio"/> releases <input type="radio"/> artists
Bring back results in blocks of:	<input type="text" value="100"/>
Search the field(s):	<input checked="" type="radio"/> both artist name and title of release <input type="radio"/> title only <input type="radio"/> artist name only
for the word(s):	<input type="text"/>
match:	<input checked="" type="radio"/> any of the words <input type="radio"/> all of the words <input type="radio"/> exact phrase
limit search by format:	<input checked="" type="radio"/> all formats <input type="radio"/> cd <input type="radio"/> vinyl <input type="radio"/> --- or choose a specific format / size --- <input type="text"/>
limit search by genre:	<input type="text" value="All Genres"/> <input type="text"/>
limit search by library code:	Call Letters: <input type="text" value="----"/> <input type="text"/> Artist Number: <input type="text"/> Release Number: <input type="text"/>
limit search alphabetically:	<input checked="" type="radio"/> All Letters (A-Z + V/A) <input type="radio"/> Artists Starting With <input type="text" value="A"/> <input type="text"/>
	<input type="button" value="Search!"/> <input type="button" value="Reset values"/>

Figure 2: Advanced Search Interface

After a user submitted a search via either of the two search interfaces, search results would be presented on either an “artist results” page or a “release results” page. (Figure 3 shows a portion of a “release results” page for the query “library science”). Over the years, release searches have been far more common in the system, and these searches pose a more interesting problem from an information retrieval point of view. As a result, I have focused primarily on release searches in this paper.










Simple Search		Search Results			Advanced Search	
Currently Displaying: 1 - 52		Sort these results by: <input type="text" value="Format of Release"/> in <input type="text" value="Ascending"/> order <input style="background-color: #d4d4d4; border: 1px solid #ccc;" type="button" value="Sort!"/>			Number of releases found: 52	
Do another search						
Comment(s)	Format	Library Code	Artist Name	Title Of Release		
	↑ ↓	↑ ↓	↑ ↓	↑ ↓		
	cd	Asia V/A- 79	Various Artists - Asia	World Library of Folk and Primitive Music (vol. 7): India		
	cd	Hiphop BE 1/ 23	Beastie Boys	Sounds of Science sampler		
	vinyl	Hiphop BL 18/ 4	Blackalicious	Passion 12" feat. Rakaa Iriscience and Dj Babu		
	cd	Hiphop BL 36/ 1	Blockhead	Downtown Science		
	cd	Hiphop BO 7/ 1	B.O.X.	Beyond Ordinary Science		
	cd	Hiphop DE 19/ 1	Deep Dish	Junk Science		
	vinyl	Hiphop DO 7/ 1	Downtown Science	Downtown Science		
	vinyl	Hiphop DO 7/ 2	Downtown Science	Room to Breathe 12"		
	cd	Hiphop LI 23/ 1	Library Science	High Life Honey		

Figure 3: Portion of search results page for the query “library science”.

Technical Details and the Data Model

I designed and developed this card catalog application in Java, using the standard server-side Java/J2EE technologies of the 2001-2002 era (servlets, JDBC, Java Server Pages) and employing the fairly common “Model 2” version of the popular “model-view-controller” pattern/architecture. The web application was deployed to a Tomcat application server that supported all of the aforementioned Java technologies. All data was persisted to a MySQL database. Of the approximately ten tables in the MySQL database, four are directly queried by the application's original searching routine: FORMAT, GENRE, LIBRARY_CODE, and LIBRARY_RELEASE. (See Figure 4 for an ER diagram of the entities and relationships represented in these four database tables.)

The FORMAT and GENRE tables basically acted as “lookup” tables that stored all of the various release formats and genres found in the station's music library. The data in these two tables did not change much at all, and each table only had a limited number of categories that could be used to classify artists and releases. The FORMAT table contained details about 22 specific formats, all of them different variations of the two major format types – CD and vinyl. The GENRE table was even smaller. It had 14 entries that represented the 14 genre sections of the WXYC music library: Africa, Asia, Blues, Classical, Comedy, Hip-hop (a section that also includes Electronic/Techno), Jazz, Latin, OCS (a station-specific moniker for traditional/folk music), Reggae, Rock, Soundtracks, Spoken Word, and Xmas.

The LIBRARY_CODE table is somewhat confusingly named and is better thought of as being an artist table that includes specific library code attributes for each artist. This table contains an artist's alphabetical and presentation names - with a band

these are usually identical, but with an individual performer, these tend to be different (e.g. “Springsteen, Bruce” and “Bruce Springsteen”). The LIBRARY_CODE table also contains the library code letters/numbers used to identify the artist and all associated releases within a certain section of the music library. A foreign key to the GENRE table signifies the genre section in which this happens to be. As of November 2008, the LIBRARY_CODE table contained well over 18,000 artist records.

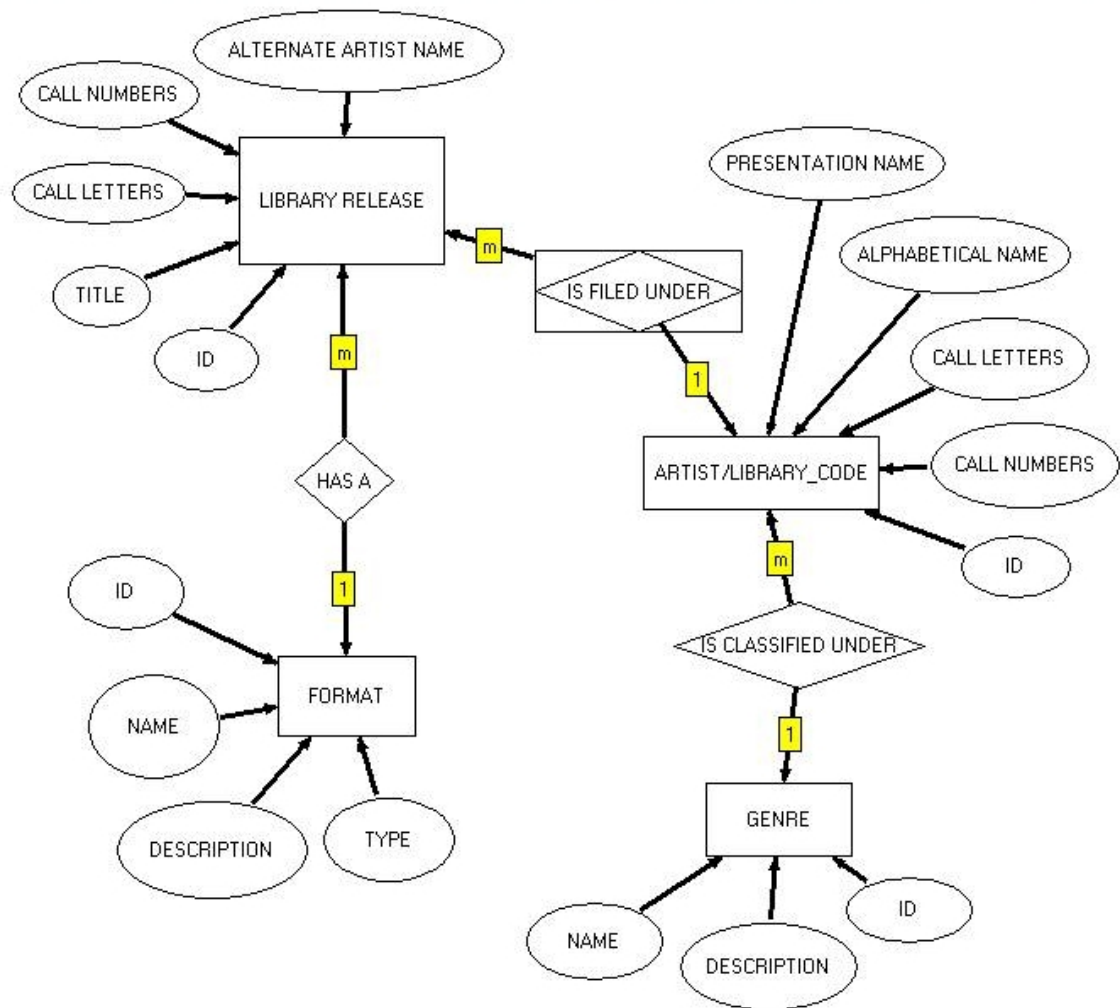


Figure 4: Entity-Relationship Diagram for WXYC Card Catalog Data

The LIBRARY_RELEASE table contains information about a specific library release. Basic library release fields include title, library call letters, and library call numbers. A foreign key to the FORMAT table specifies the format of the release, and a foreign key to the LIBRARY_CODE table identifies the artist/library code under which the release is filed. An optional ALTERNATE_ARTIST_NAME field also allows for the storage of the release's actual artist listing (e.g., "Bruce Springsteen and the E-Street Band") in the event that this information differs from the artist name associated with the release's library code. As of November 2008, the LIBRARY_RELEASE table contained well over 50,000 library release records.

Problems With The Existing System

I had a few years of Java development experience under my belt before beginning work on this application in 2001, but this was my first attempt at building a MySQL database from scratch and writing all of the queries that would be needed by the application. Looking back, I can honestly say that in my eagerness to flex some of my new MySQL/database skills, I implemented much of the application's search functionality in a fairly crude way. Upon receiving a search request from the user, the application would first read in the search string and all of the extra search parameters specifying how to process the search string, what additional search requirements/limits to use, and exactly how to sort/present search results. This information was then passed to a Java object that would go through all of it and dynamically create a giant SQL statement that reflected the specified search logic. If a search string happened to contain multiple words and both the "any of the words" match option and "both artist name and title of

release” field-specification option had been selected, the SQL-creating code would generate numerous text-comparison “LIKE” clauses (one for each combination of field and search term) and “OR” these SQL fragments together into a convoluted WHERE clause that would attempt to match any of the supplied search words with either of the artist or title fields. See Figure 5 for an example of how the aforementioned query for “library science” was translated into SQL.

```

SELECT LR.ID, LR.LIBRARY_CODE_ID, LR.CALL_NUMBERS, LR.CALL_LETTERS,
LR.TITLE, LR.ALTERNATE_ARTIST_NAME, LR.FORMAT_ID,
LR.TIME_LAST_MODIFIED, LR.TIME_CREATED, LC.GENRE_ID, LC.CALL_LETTERS,
LC.CALL_NUMBERS, LC.ARTIST_ID, LC.TIME_LAST_MODIFIED, LC.TIME_CREATED,
LC.PRESENTATION_NAME, LC.ALPHABETICAL_NAME, G.REFERENCE_NAME,
F.REFERENCE_NAME
FROM LIBRARY_RELEASE LR, LIBRARY_CODE LC, FORMAT F, GENRE G
WHERE LR.LIBRARY_CODE_ID = LC.ID
AND LR.FORMAT_ID = F.ID AND LC.GENRE_ID = G.ID AND
(LC.PRESENTATION_NAME like '%library%' OR LR.TITLE like '%library%' OR
LC.PRESENTATION_NAME like '%science%' OR LR.TITLE like '%science%')
ORDER BY LC.GENRE_ID ASC, LC.CALL_LETTERS ASC, LC.CALL_NUMBERS ASC;

```

Figure 5: SQL query generated by the existing system for the user query “library science”.

This brute-force approach to matching text fragments with SQL may have been sufficient for the system's initially modest goal of getting library data out of the paper-based system and into a digital form where it could be searched more easily, but from an information retrieval standpoint, this search system had numerous flaws. First and foremost, search results were not ranked or scored in any fashion. In this “pure Boolean” model of search, every result that came back from a database query was considered to be

an equal match, and everything else that did not come back from the database was not a match at all. If the number of search results was high, a user might have to wade through pages and pages of results in order to find the desired release. If the user instead tried to narrow the number of search results by selecting the “match all” or “exact phrase” options, there was a risk that the unforgiving system would exclude the desired release from search results because of inevitable differences between the user's search string and the specific text fields of the release that it was trying to match. To put this issue in information retrieval terms, users were often forced to oscillate between extremely low precision and extremely low recall.

A second major drawback to the existing search system was that searching was often relatively slow. Using Java code to precisely calculate and log query times over a two week period in November 2008, I discovered that 1143 release queries were executed in an average of 0.816 milliseconds each. This by itself seemed far from ideal, but even worse, I discovered that many of the “default option” queries took three to five seconds each and that a couple of queries required at least 10 seconds to execute. This poor performance often correlated directly with the number and/or size of search terms used in an all-encompassing “match any” query. In the “match any” scenario, the system would create a SQL query in which two distinct VARCHAR fields in two separate tables would be used in text-comparing LIKE clauses for each of the supplied search terms. Steps had already been taken to optimize both the database tables where possible – for instance, indexes were placed on all foreign keys used for table-joining as well as any other field that might be directly queried against. But database indexes are typically most efficient when a single numeric field is involved. A basic index on a VARCHAR field does a good

job at supporting lexicographical sorting, but it has inherent limitations that prevent it from being very efficient in cases where *all* portions of a VARCHAR field (and not merely the beginning few letters) must be examined for a possible match. An SQL query that contains numerous text-comparing LIKE fragments (each effectively requiring a full table scan) is bound to perform more slowly than a similar query with either fewer text-comparisons and/or an additional WHERE clause fragment that utilizes a numeric index.

To make things worse, the problems with search query performance were often compounded by users trying to work around the issue of not having prioritized/ranked results. An examination of user queries indicates that users often addressed the low precision issue by first trying to sort the results and then paging through them until they found a specific release. Sorting had been implemented by simply resubmitting the original query with a different “ORDER BY” clause – if a sort happened to be on one of the text-based fields (artist, title), this would potentially add some time to the execution of the original SQL query. Pagination was also implemented via re-submission of the original SQL query along with a different set of “start display” and “end display” numbers that would determine the specific page of results to display. Depending on how many pages a user had to browse through, a sub-optimal query might be executed several times in a row.

Finally, it was clear that the existing search system was not very extensible and would have a difficult time growing to support future functionality and additional search needs. The optional “ALTERNATE_ARTIST_NAME” field in the LIBRARY_RELEASE table had not originally been included among the text fields searched by the dynamically created SQL statements. Adding it into the mix at this stage

might have exacerbated some of the existing problems with query performance.

Additionally, song titles had never been part of the card catalog system at all, simply because the station did not have this data in digital form. But if the station decided to take the step of entering song/track data and/or capturing portions of track-listings from an existing music database, this would be another major text field to search. Finally, some radio station personnel were interested in giving DJs the ability to “tag” releases in various ways and add their own comments to the library catalog. This data would only be useful if it was searchable in some fashion.

Before discussing the alternative search solution that I researched and implemented, it should be noted that MySQL does have limited support for full-text indexing/searching. Database tables that use the MyISAM storage engine can use a special full-text index that allows character-based content to be broken down into terms and searched with a special SQL syntax. Full-text indexes can only exist on columns within a single table, however, so in order to simultaneously search both artist and release information, I would have to effectively de-normalize my database schema so that those fields could be part of the same database table. This seemed unacceptable, especially since future search needs would only require more full-text indexes and/or additional database schema changes. I decided to apply the “separation of concerns” principle to this situation, and turn to a separate information retrieval utility for the application's full-text search needs. MySQL could be left to do what it does best: serve as the persistent data store for all of the application's data.

Implementing Lucene

Lucene is an open-source information retrieval software library written entirely in Java. Lucene is not a standalone application or an actual search engine by itself - it is simply a powerful software component that can be used inside other Java applications as a means of implementing search. I had heard many good things about Lucene over the years, and after downloading the latest Lucene release and trying it out, I quickly started to understand why Lucene has become the leading information retrieval library for Java applications. The software's simple yet flexible application programming interface (API) provides a very clean abstraction of the indexing and searching process.

Underneath its fairly straightforward API, Lucene implements some sophisticated and well-established information retrieval techniques. Lucene uses the highly efficient “inverted index” format to store details about the documents that it is indexing. This data structure is built around individual terms and not documents, so that term-based indexing and term-lookup searching can both be done as quickly as possible. Lucene experts often compare this index format to the alphabetical index in the back of a book, but as a programmer I prefer to think of it as a giant hashtable that provides fast access to data/objects/documents based on a supplied key.

Lucene implements the commonly used TF/IDF method as a means to weight terms when indexing documents. This method calculates a given term's weight/importance to a document by combining the term's frequency within a document (TF) with its “inverse document frequency” (IDF) – the inverse of the total number of documents in which the term appears. Individual documents can be represented as a set of terms and their associated term weights, and this information can then be mapped out

in an n-dimensional vector space, so that each document is represented as a “term vector”. Queries can also be translated into term vectors in this same n-dimensional space, and a document's relevance score for any given query can be calculated by comparing the document's term vector with the query's term vector. In theory, the most relevant search results are the documents with term vectors closest to the term vector of the query. This “vector space model” is the approach that Lucene uses in implementing its search algorithms.

The inverted index, TF/IDF, and the vector space model are all fundamental concepts within the field of information retrieval, and yet a Java developer need not know much about any of these things in order to build a search application using Lucene. Indexing with Lucene can be done by simply learning how to use a few fairly basic classes. The **IndexWriter** class allows write-access to a new or existing Lucene-based index that can be stored on the local filesystem. A variety of **Analyzer** classes can be used by the **IndexWriter** class to analyze text and break this text down into terms during the index-writing process. The **Document** class is an abstract representation of an actual “document” or other text-based object that an application might want to index for searching purposes. The **Field** class represents a portion of a document that has a name and some associated content/text associated with it. For indexing purposes, one could think of a document as basically a collection of named fields/sections with associated values, and similarly, a Lucene **Document** object acts as a logical container for as many **Field** instances that are necessary to represent the object.

Many of the decisions that a developer must make when implementing Lucene within a Java application all revolve around the same general question - exactly how

should a system's searchable/retrievable content be represented as fields and documents within the Lucene world?

The first part of this decision process involves determining what specific logical entities/concepts should be represented as documents in the Lucene index. This may be fairly easy to determine in a system involving actual documents (e.g. journal papers, web pages, email messages), but for other systems with slightly more complex and/or hierarchical object models, this sometimes becomes a question of granularity – should the most coarse-grained entity be translated into a single document with numerous fields representing the more fine-grained details? Or should the entity be broken down into individual documents representing distinct components? Or perhaps both? Lucene actually allows heterogeneous document types to exist within the same index, so for maximum flexibility a developer might choose to create overlapping documents for related entities.

For the WXYC card catalog application, I decided to create a single document for each distinct library release. Releases are both the central entities in the card catalog data model and the primary objects that users want to find. Artist-only search has occasionally been used in the existing system, but I have come to suspect that many of those uses are attempts to get around and/or lessen the performance problems and low-precision problems presented by artist-and-title search. I could always come back and create artist-based documents at a later date, so I decided that release-based documents would be the best choice for my initial Lucene index. Artist-related information could be placed in specific fields within each release-based document.

The next step in the process was determining exactly what fields I would use to make up each document. Additionally, for each specific field that I chose, some indexing/storage decisions would have to be made. When a **Field** object is added to a **Document** instance in Lucene, four pieces of information must be provided: the name of the field (examples: “artist”, “title”, “abstract”, “ID”); the value of the field (the text content that is stored and/or indexed); metadata detailing if/how the field should be stored; and metadata detailing if/how the field should be indexed. Name and value are fairly self-explanatory pieces of information, but the storage and indexing specifics required a little more thought. This metadata would determine much about the indexing process, what pieces of information would be searchable, what pieces of information would be available as part of search results, and even what pieces of information could be used to sort search results.

If the data value of a given field needed to be presented to the user as part of the displayed search results, that field would be marked as stored so that it could be retrieved directly from the Lucene index at search time, thus saving the application an additional round trip back to the database. If a field's data value did not need to be presented to the user and was only being added for indexing purposes, the field was not stored.

Just as some fields needed to be indexed for searching purposes but not actually stored, some of the stored fields did not need to be indexed. For those fields that did need to be indexed, I would need to specify whether or not the field should be analyzed during the indexing process. This analysis process is where text is tokenized and broken down into parts, punctuation marks are removed, and common stop words like “the” and “a” can be discarded. For some pieces of data such as unique document ID numbers or

library-specific letter codes, analysis is not necessary or advisable. Additionally, fields used for sorting should not be analyzed so as to preserve the lexicographical order of the original text data. For most text data that needs to be searched, however, analysis is a necessary process that transforms raw unstructured text into distinct terms that can be stored in an index.

In constructing the makeup of the release-based document, I found myself creating fields not to represent specific pieces of data, but rather to address specific application needs such as searching, sorting, and search result presentation. These fields sometimes overlapped in the actual data that they contained – clearly a no-no in the world of relational databases but not a concern here. In order to facilitate a basic default search that would attempt to search all of the artist and title information about a particular release, I designated an aggregate field named “releaseContents” that would contain all of the text from three different database fields (TITLE and ALTERNATE_ARTIST_NAME from the LIBRARY_RELEASE table, PRESENTATION_NAME from the LIBRARY_CODE table). Individual fields named “title” and “artist” were also created, so that field-specific queries could be performed directly against that data. The table in Figure 6 lists all of the fields that were added to each release-based document as well as information about how the fields were indexed and/or stored.

Once this document/field structure had been determined, I wrote some code that took a **LibraryRelease** object and converted it into **Field** and **Document** objects that were ready for indexing. This “mapping” operation turned out to be quite similar to the object-relational mapping process that must occur when data retrieved from relational database tables is mapped into in-memory Java objects that have a very different

Field Name	Index?	Store?	Field Purpose
docID	NOT_ANALYZED	No	unique document locator
libraryReleaseID	NOT_ANALYZED	Yes	for display (value in hyperlink)
libraryCodeID	NOT_ANALYZED	Yes	for display (value in hyperlink)
releaseContents	ANALYZED	No	for searching
title	ANALYZED	Yes	field-specific searching, display
artist	ANALYZED	Yes	field-specific searching, display
format	NO	Yes	display
cdVinyl	ANALYZED	No	faceted search
genre	ANALYZED	Yes	faceted search, display
libraryCode	NO	Yes	display
releaseCL	NOT_ANALYZED	No	possible field-specific search
releaseCN	NOT_ANALYZED	No	possible field-specific search
artistCL	NOT_ANALYZED	No	possible field-specific search
artistCN	NOT_ANALYZED	No	possible field-specific search
artistSort	NOT_ANALYZED	No	sorting by artist
releaseSort	NOT_ANALYZED	No	sorting by release

Figure 6: Table showing fields and how they were stored or indexed

structure. I then wrote a simple batch method to create the initial Lucene index. After retrieving all of the library release data from the database in the form of **LibraryRelease** objects, this batch method iterated through these **LibraryRelease** objects, converted each one into a **Document**, and then added the collection of **Document** objects to the index using the **IndexWriter**. For this initial index, I created the **IndexWriter** with an instance of the **StandardAnalyzer** class. Different **Analyzer** subclasses treated text in different ways, but the **StandardAnalyzer** seemed most suitable for analyzing music release metadata involving lots of proper names and titles that did not require word stemming or support for synonyms.

My initial indexing routine went through 50,000+ library releases and generated 10.4 megabytes worth of Lucene index files. Creating this initial index would not be sufficient by itself, however, as the index would be out of date as soon as either a new

library release was added to the catalog or an existing library release was modified. To address this issue, I created some additional code that would keep the Lucene index in sync with the permanent data store. Code that added a new library release to the database would now also create and index a corresponding document for that new release. Code that updated or deleted library release information would now retrieve the existing document from the index and update it or delete it accordingly. Because a document may need to be updated and/or deleted from the index at some point, it is especially crucial that all documents contain a unique-ID-based field that allows them to be uniquely singled out for updating/deletion without any of the other indexed documents being impacted.

This need to keep the index in sync with the database does demonstrate a fairly minor downside to the hybrid Lucene/MySQL approach. Since data is no longer stored in only one place, there is an extra step that must be taken by the application whenever data is created, updated, or deleted. This is likely to be a very slight and potentially unnoticeable drag on data-modification performance, however. In the case of the WXYC card catalog application, only the administrative operations performed by music librarians/admins would ever incur this extra cost. Given how infrequently these operations are performed in comparison to the more widely used search functionality, the tradeoff makes a lot of sense if it results in faster and superior search capabilities.

With the index created, I started developing the searching code. The **Query** class is the heart of Lucene's searching API, and the numerous subclasses of **Query** (**RangeQuery**, **BooleanQuery**, **TermQuery**, **FuzzyQuery**) offer developers a variety of ways to represent specific search logic in a form that can be used to programmatically

query a Lucene index. But despite the wealth of **Query** subclass options, I found that almost all of the search functionality required by the card catalog application could be obtained by using the extremely powerful **QueryParser** class to automatically generate **Query** objects representing a search string entered by the user. Like the **IndexWriter** class, **QueryParser** also uses an **Analyzer** instance to analyze the query terms, but before that analysis stage takes place, **QueryParser** automatically detects Boolean operators like AND and OR and NOT, wildcard characters like the asterisk(*), a tilde(~) used to signify “fuzzy querying”, and even special field constraints (“artist:_____”). I found this amazing out-of-the-box support for user-query parsing to be one of Lucene's greatest strengths. As a developer, I would no longer need to write code to interpret double quotes to mean “exact phrase”, and the logic that had been written to interpret “any term”/“all terms” options could be discarded in favor of Lucene's support for commonly understood Boolean operators. By using **QueryParser**, I would be able to offer the system's users a lot more power and flexibility; the exact syntax and specifics of the new querying possibilities would just have to be communicated and taught to users via training and/or a “helpful search tips” section of the search interface.

Once a **Query** object had been generated by the **QueryParser** class, querying was a breeze. The **IndexSearcher** class provided an interface to the actual Lucene index on the filesystem. A **Query** object could be passed in, and results would be returned in a **Hits** object that served as a collection of **Document** objects. I again wrote a simple mapping procedure that would translate between the returned **Document** objects and a data-transfer-type object representing a library release. The application's original **LibraryRelease** class could have been used here, but in order to keep things cleaner

and more distinct from the database-generated transfer object, I created a new **LibrarySearchResult** class that simply encapsulated all of the various stored properties that had been stored in the index. The existing “release search results” web page was then modified to expect a collection of **LibrarySearchResult** objects instead of the collection of **LibraryRelease** objects that it had displayed previously.

Results and Comparative Analysis

With the Lucene implementation complete, I began the testing and comparative analysis process. It only took a few trial runs of various text queries for me to discover that searching via the Lucene index was *significantly* faster than searching the MySQL database via dynamically constructed SQL statements. Using Java code that precisely calculated and logged the exact time of all search operations, I discovered that most Lucene searches were completed in 35 milliseconds or less – a tremendous improvement over the half-second, one-second, two-second, and five-second searches that had been all too common in the MySQL-based system.

I initially questioned whether part of these performance improvements might be attributable to the fact that the Lucene index existed on the same filesystem as the Tomcat server itself, whereas the MySQL database was hosted on a separate database server requiring a network hop. However, this possibility was discounted by examining and timing other database queries that the application was making. Queries that actually took advantage of integer-based indexes usually completed in less than 50 milliseconds. It was only the queries that overutilized text-comparison clauses that demonstrated such poor performance. Neither network latency nor database overhead costs were to blame.

Additionally, I found that the search results generated by Lucene were of much higher quality than the unranked results obtained from querying the MySQL database. In the Lucene-based system, a query for “library science” brought back an album by the group Library Science as the number one match (see Figure 7). When this same “library science” query had been run using the MySQL-based search routine (Figure 3), the Library Science album was just one of 52 equivalent matches returned from the database. (It should be noted that the only reason the Library Science album appeared as high as it did in Figure 3 was due to the fact that the results were alphabetized by Genre and

Search WXYC Library!
[Search Tips](#)

Click on "Artist" or "Title" column headings to sort on those fields.

Displaying 1 - 47 of 47 results matching text query **library science**

Library Code	Artist Name	Title Of Release	Format
Hiphop LI 23/1	Library Science	High Life Honey	cd
Rock ON 31/1	On-Air Library	On-Air Library	cd
Hiphop DO 7/1	Downtown Science	Downtown Science	vinyl
Rock BO 142/1	The Boys' Star Library	Sugar & Water	cd
Rock MI 91/1	Mind Science of the Mind	Mind Science of the Mind	vinyl
Hiphop DO 7/2	Downtown Science	Room to Breathe 12"	vinyl
Rock CU 22/7	Science Group	Red Science	cd
Jazz CU 10/7	Science Group	Science Group	cd
Rock SC 75/1	Science Kit	Seven Times Around	cd
Hiphop VA 686/203	various	Breakbeat Science	cd
Rock SH 14/2	Shriekback	Jam Science	vinyl
Rock SW 35/1-B	Swob	The Explosion of Science	vinyl - 7"

Figure 7: New search results page for the query “library science”.

Library Code, so all releases from the Hiphop section were displayed before any Jazz, Reggae, or Rock releases).

The Lucene API makes it easy to retrieve the actual score assigned to a document for any given query. While this level of detail is usually not necessary from an application development standpoint, I collected the scores for the top 12 “library science” matches in order to help illustrate the underlying information retrieval algorithms at work (see Figure 8). As the only release that contains both “library” and “science” amongst its document text, the Library Science album scores far higher than the next 11 search results. It should be noted that Lucene normalizes scores to a scale that runs from 0.0 to 1.0, so the Library Science album's 1.0 score is not indicative of a “perfect score” but rather the highest score amongst the entire set of documents.

Score	Artist (Library Code)	Artist (Library Release)	Title of Library Release
1.0000	Library Science	Library Science	High Life Honey
0.3463	On-Air Library	On-Air Library	On-Air Library
0.2661	Downtown Science	Downtown Science	Downtown Science
0.2356	The Boys' Star Library	The Boys' Star Library	Sugar & Water
0.2218	Mind Science of the Mind	Mind Science of the Mind	Mind Science of the Mind
0.2173	Downtown Science	Downtown Science	Room to Breathe 12"
0.2173	Chris Cutler	Science Group	Red Science
0.2173	Chris Cutler	Science Group	Science Group
0.2173	Science Kit	Science Kit	Seven Times Around
0.2049	various	various	Breakbeat Science
0.2049	Shriekback	Shriekback	Jam Science
0.2049	Swob	Swob	The Explosion of Science

Figure 8: Score of “library science” hits in Lucene-based system.

The order of the subsequent 11 search results can be easily explained by looking at both the term frequencies (TF) and inverse document frequency (IDF) values for the

terms “library” and “science”. Within the entire set of documents, the term “library” tends to appear less frequently than the term “science”, so an occurrence of the term “library” will weigh more for a given document than an occurrence of the term “science”, simply by virtue of the higher IDF value for “library”. Actual term frequency within a single document also plays a role. The self-titled On-Air Library album contains 3 occurrences of the term “library”, while the *Sugar & Water* release from The Boys' Star Library only contains 2 occurrences of “library”.

Lucene's ability to generate ranked search results were clearly a benefit, but in order to examine other potential differences between the MySQL-based and Lucene-based search routines, I closely compared the entire set of search results that the two systems had generated in response to the “library science” query. I found that the two sets of results were mostly, but not entirely, overlapping - 44 releases has been found by both systems. However, the Lucene-based search results also included three releases not found by the MySQL-based search routine, while the MySQL-based search results included eight releases that had not been found by the Lucene-based search routine.

Artist (Library Code)	Artist (Library Release)	Title of Library Release
The Album Leaf	The Album Leaf on!air!library!	A Lifetime or More
Big Heifer	Science Kit and Big Heifer	Split Tour EP
Tim Berne	Tim Berne and Science Friction	The Sublime And [live]

Figure 9: “library science” hits in Lucene-based system not found by MySQL-based system.

A close examination of these discrepancies helps to illustrate some key functional differences between the two systems. The three extra Lucene-derived results (see Figure 9) all had the word “library” or “science” in the “Artist (Library Release)” field but not in either of the “Artist (Library Code)” or “Title” fields. The two artist fields are only different in cases where the `ALTERNATIVE_ARTIST_NAME` database field is present, indicating that a release has a different artist name than the artist under which it is filed. Because this field was empty for a majority of releases, it was not being queried against by the old MySQL-based system. The three particular results shown in Figure 9 may not have been the closest matches to the “library science” query, but one can easily imagine a case where users would be unable to find relevant documents if this information was not being factored into the searching process. For instance, in the old MySQL-based system, a search for the band “Science Kit” would not bring back the *Split Tour EP* by Science Kit and Big Heifer, simply because the release had been filed under Big Heifer and not Science Kit.

Artist Name	Title of Library Release
Blackalicious	Passion 12" feat. Rakaa Iriscience and Dj Babu
Omniscience	Amazin 12"
Majek Fashek	Prisoner of Conscience
Cry Before Dawn	Crimes of Conscience
DJ C	Conscience a Heng Dem
Homescience	End The Year [ep]
Homescience	Small Music [ep]
Swans	Omniscience [live]

Figure 10: “library science” hits in MySQL-based system not found in Lucene-based system.

The eight extra MySQL-derived results (see Figure 10) all contained words in which “science” was a fragment of a larger word or name: “Conscience”, “Omniscience”,

“Homescience”, “Iriscience”. Because its stored text is not indexed by individual terms, the MySQL system is forced to using a brute-force approach in its text-matching – search terms are wrapped up in percentage signs so that the entire content of a VARCHAR field can be examined for any occurrence of the supplied string. This method succeeds in bringing back many of the releases that actually contain the word “science”, but it also brings back these extra eight documents that are clearly not relevant for the supplied query. The fact that the Lucene-based system does not find these irrelevant documents can be seen as yet another plus.

Additional Enhancements and Challenges

The major boost in query execution speed allowed for the creation of an extremely snappy and responsive search results display interface. In the old searching system, both sorting and pagination had been implemented via subsequent/repeat queries to the database. These additional requests were often no faster than the first, but given the stateless nature of web applications, this had been the most straightforward solution. For the new Lucene-based system, I kept this subsequent/repeat-query model, only now all sorting operations and subsequent page requests would be just as *fast* as the original query - typically less than 50 milliseconds. This allowed the new system to have the extremely responsive feel of a desktop application.

Once I saw the usability/performance benefits that Lucene-based searching could bring to the catalog-searching system, I started designing an additional “faceted search” feature that would not have been feasible in the old system. Faceted search has become a very popular feature on e-commerce sites over the last few years – users can select

various categories/facets as a way of narrowing down search results to specific subsets that more closely reflect the items for which they are searching. There were two obvious facets in the music library data - format and genre.

For my initial implementation of faceted search, I chose CD and vinyl to be the only two format facets. Many of the releases in the card catalog database have additional size designations such as “7-inch vinyl” and “2-CD set”, but this level of detail is not consistently stored for all releases, so I felt like the more general facets of CD and vinyl would be the most useful. For genre, the facets were the 14 distinct genres in the WXYC music library: Africa, Asia, Blues, Classical, Comedy, Hiphop, Jazz, Latin, OCS, Reggae, Rock, Soundtracks, Spoken Word, and Xmas. Although genre is a more fluid and multi-valued construct in many music classification systems, within the WXYC music library each release is assigned one specific genre that designates the physical section of the library where it can be found.

I decided to present these facets as a sidebar beside search result data, as this would offer users a way of narrowing down search results by selecting a given facet. In order to dynamically generate this sidebar, I revisited the code where Lucene search results are iterated through and converted into **LibrarySearchResult** transfer objects. I added a section of code that harvested individual facet values as they appeared in search results and placed facet information into a data structure that would summarize the individual facet values and how many times they appeared in the search results.

When a user selected a specific facet value link from the facet sidebar (see Figure 11), the original query would be re-submitted and the extra facet parameter would be sent along with the request. I decided not to actually add the facet detail to the original Lucene

query, but simply to use the facet as a post-query filter when iterating through search results and deciding what results would be sent back to the user for display. This allowed the system to keep the entire list of facets available within an identical sidebar menu, instead of forcing a user to first remove one facet filter before selecting another mutually exclusive facet value.

library science [Search Tips](#)

Click on "Artist" or "Title" column headings to sort on those fields.

Displaying 1 - 7 of 7 results matching text query **library science** and **(format = vinyl)** and **(genre = Rock)**

All Results (47)

Narrow by...

Genre

- [Asia](#)
- [Hip-hop](#)
- [Jazz](#)
- [OCS](#)
- [Reggae](#)
- **Rock**

Format

- [cd](#)
- **vinyl**

	Library Code	Artist Name	Title Of Release	Format
Rock	MI 91/1	Mind Science of the Mind	Mind Science of the Mind	vinyl
Rock	SH 14/2	Shriekback	Jam Science	vinyl
Rock	SW 35/1-B	Swob	The Explosion of Science	vinyl - 7"
Rock	SH 161/1	Dead Science	Sholi/Dead Science Split 7"	vinyl - 7"
Rock	AN 8/1	Laurie Anderson	Big Science	vinyl
Rock	DO 8/2	Thomas Dolby	Blinded by Science	vinyl
Rock	EN 2/3	Brian Eno	Before and After Science	vinyl

Figure 11: Search results page for query “library science” with “Rock” and “vinyl” facets selected.

During my initial testing of Lucene-based sorting, I occasionally encountered memory-related exceptions while making repeated calls to sort a large set of search results alphabetically by artist or title. In order to prevent these memory-related exceptions from occurring, I implemented code that limits the total number of search results that are obtained for any given search. This type of result-limiting behavior is strongly encouraged by the latest version of Lucene (version 2.4.0), in which the `Hits` class is deprecated in favor of `TopDocCollector` with its numerical results ceiling that

it forces the developer to set. This API change is intended to help address performance issues caused by code that attempts to iterate through all search results.

This technical challenge serves to illustrate another key difference between the Lucene-based search system and its MySQL-based predecessor. In the MySQL-based “pure Boolean”-type system, all search results were equal in the sense that no one result was considered to be any more relevant than any other. As such, all results needed to be accessible to the user, so that the user could find the desired results, either through paging or sorting. With a Lucene-based search system, there is not as much need to obtain *all* of the matches for a given query in the event that the result set happens to be large. If a user's query were to result in more than 500 hits, chances are pretty strong that the user will find what he/she is looking for simply by looking through the first 100 hits, since these have been ranked the most relevant according to the user's query. Rather than page or sort through an excessive number of ranked search results, a user is much more likely to either refine the text of their query in hopes of getting a different ranked order or choose additional facets that will help narrow the results to an even smaller (but still ranked) set. The option to sort a large collection of search results becomes a lot less valuable once the results are already sorted by relevance in the first place. As a developer accustomed to the set-oriented nature of relational database theory, I found myself having to undergo a slight shift in thinking as I started writing code that operated on the *top* matches for a given query instead of *all* results meeting specific criteria.

Conclusions and Future Directions

Clearly, the Lucene-based redesign of the WXYC card catalog application offers many substantial improvements over the original MySQL-only version. By leveraging a high-performance information retrieval software library instead of merely relying on text-comparing SQL queries, the redesigned application is able to deliver ranked search results of a much higher quality than those delivered by the old system. Additionally, the redesigned application shows a dramatic improvement in query execution speed. This performance boost noticeably improved the overall usability of the card catalog application, while also allowing for the implementation of an additional faceted search feature.

Some Lucene-based systems may only use the software to index full-text documents that are not already stored in a relational database. But the redesigned card catalog application demonstrates that a hybrid IR/RDBMS approach can also work quite well. The exact same pieces of data can be stored in different data structures/formats that have distinctly different purposes: a Lucene index optimized for information retrieval/search purposes, and a relational data store that offers long-term persistence, data integrity/type checking, transactional support, and numerous other benefits. By implementing a hybrid approach, a developer might need to implement extra code to manage additions, updates, and deletes to the index whenever there are corresponding changes to the underlying database. In my experience with the WXYC card catalog application, these extra operations were well the overwhelming benefits provided by a superior search solution.

As I get ready to deploy the redesigned version of the WXYC card catalog application, I have already begun thinking of future enhancements and extensions to the current system. Cross-references (“See also...”) that once existed in the old paper-based system could potentially be implemented by including the referring artist amidst the indexed fields of a referred-to document. User-generated tags might allow for additional facets on which to filter. And ultimately, song data could be added to the index, greatly increasing the amount of searchable text included in release-based documents. These changes would have been difficult to implement efficiently in the old MySQL-based system, but the search functionality of the new Lucene-based system is a lot more extensible.

Finally, I intend to explore the possibilities of using Apache Solr for my next information-retrieval-related programming project. Solr is a Lucene-based search server that harnesses the many strengths of Lucene while also adding support for faceted search, hit highlighting, and additional enterprise-level features such as caching and replication. Solr seems to be very well-suited for searching structured and “semi-structured” data and I initially considered using it for the redesign of the WXYC card catalog application. Solr’s enterprise-level server features seemed a bit unnecessary for the size of this project, however, and I felt like Lucene’s mature Java API would be a much more sensible fit with the WXYC application’s existing Java codebase. Now that I’ve successfully completed this Lucene implementation and developed a solid understanding of the core Lucene functionality, I plan on investigating Solr more closely in order to see what extra benefits it has to offer.