BRINGING INFORMATION RETRIEVAL BACK TO DATABASE MANAGEMENT SYSTEMS

Khaled Nagi

Dept. of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Egypt.
khaled.nagi@eng.alex.edu.eg

ABSTRACT

Information retrieval emerged as independent research area from traditional database management system more than a decade ago. This was driven by the increasing functional requirements that modern full text search engines have to meet. Current database management systems (DBMS) are not capable of supporting such flexibility. However, with the increase of data to be indexed and retrieved and the increasing heavy workloads, modern search engines suffer from scalability, reliability, distribution and performance problems. The DBMS have a long tradition in coping with these challenges. Instead of reinventing the wheel, we propose using current DBMS as backend to existing full text search engines. This way, we bring back both worlds together. We present a new and simple way for integration and compare the performance of our system to the current implementations based on storing the full text index directly on the file system.

Keywords: Full text search engines, DBMS, Lucene, performance evaluation, scalability.

1 INTRODUCTION

Most commercial database management systems offer basic phonetic full text search functionality. For example, Oracle has a module called Oracle Text [1]. Yet, seeking to add more functionality and intelligence to their search capabilities, many commercial applications use third party specialized full text search engines instead. There are several commercial products on the market. But certainly Lucene [2] is the most popular open-source product at the moment. It provides searching capabilities for the Eclipse IDE [3], the Encyclopedia Britannica CD-ROM/DVD, FedEx, New Scientist magazine, Epiphany, MIT's Open-Courseware [4] and so on.

All search engines build an *index* of the data to be retrieved in user queries. The index is always stored in the file system on disk and can be loaded at startup in the memory (optional in Lucene) for faster querying. However, this is not feasible for large indices due to memory size limitations. So, the standard storage usually remains the file system of the disk.

However, with the increase of data to be indexed and retrieved under heavy workloads of user queries, search engines suffer from scalability problems both in providing adequate response times for their users and keeping good overall system throughput. To cope with these problems, search engines should provide more intelligent techniques for accessing the disk. Reliability becomes also a problem. The possibility of corrupting the whole index during a system crash is much higher than loosing the data in a database after a similar crash. Restoring a defected index might also take several hours thus complicating the situation even further. The search engine must manage its read and write locks by itself as well. Distributing the index among several sites and providing efficient mirroring techniques is becoming an important issue to large scale search engine projects such as Nutch [5].

The database management systems have a long tradition in coping with these challenges. Instead of reinventing the wheel, we try to bring both world together again in a new way. We propose *using current DBMS as backend to existing full text search engines* as opposed to either re-implementing full text search engine functionality into DBMS or re-implementing core DBMS features into search engines. As a case study, we use the open-source *Lucene* and *MySQL* without loss of generality. We use real world data extracted from an electronic marketplace and simulate real world workload traces in order to demonstrate that the overall system throughput and query response time do not suffer with the introduction of DBMS as a backend with their inherent overhead. In some cases, some performance indices are also improved which paves the way to using the whole spectrum of basic infrastructural facilities offered by DBMS such as recovery, automatic replication, distribution, and segmentation.

The rest of the paper is organized as follows.

Section 2 provides a background on full text search engines. Our proposed system integration is presented in Section 3. Section 4 contains the results of our performance evaluation and Section 5 concludes the paper.

2 BACKGROUND ON FULL TEXT SEARCH ENGINES

2.1 Typical Features

Full text search engines do not care about the source of the data or its format as long as it is converted to *plain text*. Text is logically grouped into a set of *documents*. The user application constructs the user query which is submitted to the search engine. The result of the query execution is a list of document IDs which satisfy the predicate described in the query.

The results are usually sorted according to an internal scoring mechanism using fuzzy query processing techniques [6]. The score is an indication of the relevance of the document which can be affected by many factors. The phonetic difference between the search term and the hit is one of the most important factors. Some fields are boosted so that hits within these fields are more relevant to the search result as hits in other fields. Also, the distance between query terms found in a document can play a role in determining its relevance. E.g., searching for "John Smith", a document containing "John Smith" has a higher score than a document containing "John" at its beginning and "Smith" at its end. Furthermore, search terms can be easily augmented by searches with synonyms. E.g., searching for "car" retrieves documents with the term "vehicle" or "automobile" as well. This opens the door for ontological searches and other semantically richer similarity searches.

2.2 Architecture

As illustrated in Fig. 1, at the heart of a search engine resides an *index*. An index is highly efficient cross-reference lookup data structure. In most search engines, a variation of the well-known *inverted index* structure is used [7]. An inverted index is an inside-out arrangement of documents such that terms take center stage. Each term refers to a set of documents. Usually, a B+-tree is used to speed up traversing the index structure.

The *indexing* process begins with collecting the available set of documents by the *data gatherer*. The *parser* converts them to a stream of plain text. For each document format, a parser has to be implemented. In the *analysis* phase, the stream of data is tokenized according to predefined delimiters and a number of operations are performed on the tokens. For example, the tokens could be lowercased before indexing. It is also desirable to remove all stop words. Additionally, it is common to reduce them to their roots to enable phonetic and grammatical

similarity searches.

The *search* process begins with *parsing* the user query. The tokens and the Boolean operators are extracted. The tokens have to be analyzed by the same *analyzer* used for indexing. Then, the index is traversed for possible matches in order to return an ordered collection of hits. The *fuzzy query processor* is responsible for defining the match criteria during the traversal and the score of the hit.
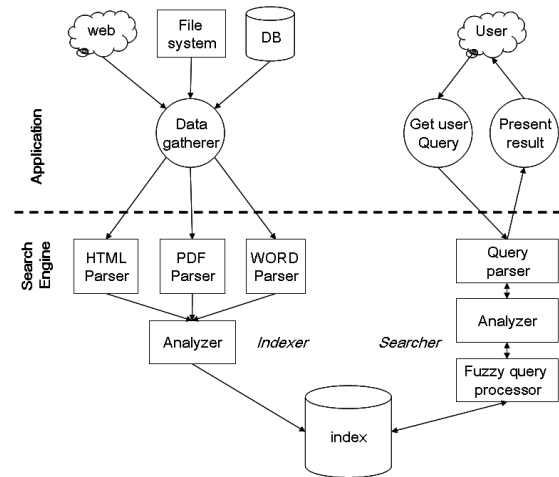

Figure 1: Architecture of a full text search engine

2.3 Typical Operations

2.3.1 Complete index creation

This operation occurs usually once. The whole set of documents is parsed and analyzed in order to create the index from scratch. This operation can take several hours to complete.

2.3.2 Full text search

This operation includes processing the query and returning page hits as a list of document IDs sorted according to their relevance.

2.3.3 Index update

This operation is also called *incremental indexing*. It is not supported by all search engines. Typically, a worker thread of the application monitors the actual inventory of documents. In case of document insertion, update, or deletion, the index is changed on the spot and its content is immediately made searchable. Lucene supports this operation.

3 PROPOSED SYSTEM INTEGRATION

3.1 Architecture

Lucene divides its index into several *segments*. The data in each segment is spread across several files. Each index file carries a certain type of information. The exact number of files that constitute a Lucene index and the exact number of segments vary from one index to another and depend on the

number of fields the index contains. The internal structure of the index file is public and is platform independent [8]. This ensures its portability.

We take the index file as our basic building block and store it in the MySQL database as illustrated in Fig. 2. The set of files, i.e. the logical directory, is mapped to one database relation. Due to the huge variation in file sizes, we divide each file into multiple chunks of fixed length. Each chunk is stored in a separate tuple in the relation. This leads to better performance than storing the whole file as CLOB in the database. The primary key of the tuple is the filename and the chunk id. Other normal file attributes such as its size and timestamp of last change are stored in the tuple next to the content. We provide standard random file access operations based on the above mentioned mapping. Using this simple mapping, we do not violate the public index file format and present a simple yet elegant way of choosing between the different file storage media (file system, RAM files, or *database*).
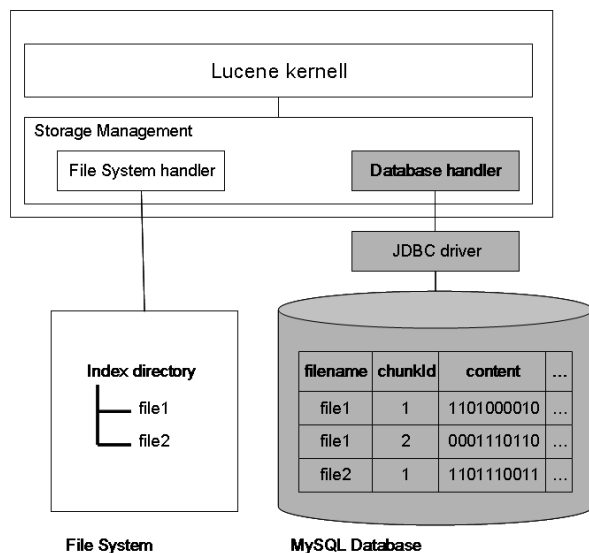

Figure 2: Integrating Lucene index in MySQL database

3.2 System Design

Fig. 3 illustrates the UML class diagram of the store package of Lucene. We only include the relevant classes. The newly introduced classes are grayed. `Directory` is an abstract class that acts as a container for the index files. Lucene comes with two implementations for file system directory (`FSDirectory`) and in-RAM index (`RAMDirectory`). It provides the declaration of all basic file operations such as listing all file names, checking the existence of a file, returning its length, changing its timestamp, etc. It is also responsible for opening files by returning an `InputStream` object and creating a new file by returning a refer-

ence to a new instance of the `OutputStream` class. We provide a database specific implementation, `DBDirectory`, which maps these operations to SQL operations on the database.

Both `InputStream` and `OutputStream` are abstract classes that mimic the functionality of their `java.io` counterparts. Basically, they implement the transformation of the file contents into a stream of basic data types, such as integer, long, byte, etc., according to the file standardized internal format [8]. Actual reading and writing from the file buffer remain as abstract method to decouple the classes from their physical storing mechanism. Similar to `FSInputStream` and `RAMInput-Stream`, we provide the database dependent implementation of the `readInternal` and `seekInternal` methods. Moreover, the `DBOutput-Stream` provides the database specific flushing of the file buffer after the different write operations. Other buffer management operations are also implemented.

Both `DBInputStream` and `DBOutput-Stream` use the central class `DBFile`. A `DBFile` object provides access to the correct file chunk stored in a separate tuple in the database. It also provides a clever caching mechanism for keeping recently used file chunks *in memory*. The size of the cache is dynamically adjusted to make use of the available free memory of the system. The class is responsible for guaranteeing the coherency of the cache.
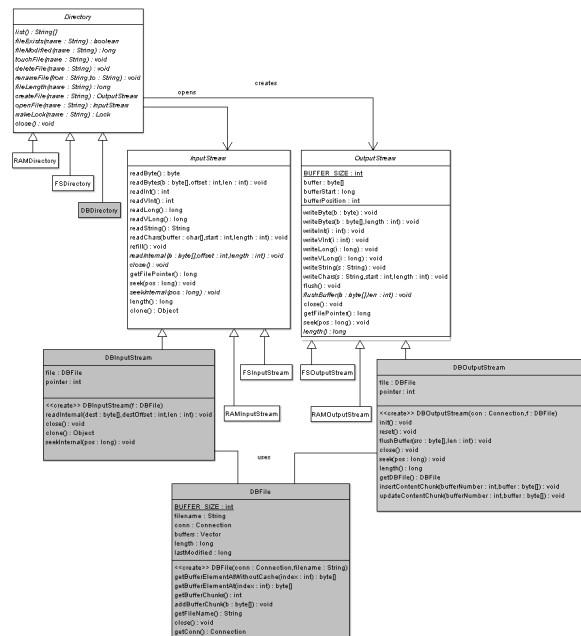

Figure 3: UML class diagram of the store package after modification

4 PERFORMANCE EVALUATION

In our order to evaluate the performance of our proposed system, we build a full text search engine on the data of a neutralized version of a real electronic marketplace. The index is build over the textual description of more than one million products. Each product contains approximately 25 attributes varying from few characters to more than 1300 characters each. We develop a performance evaluation toolkit around the search engine as illustrated in Fig. 4.

The *workload generator* composes queries of single terms, which are randomly extracted from the product description. It submits them in parallel to the application. The *product update simulator* mimics product changes and submits the new content to the application in order to update the Lucene index. The application consists of the *modified Lucene kernel* supporting both *file system* and *database storage* options of the full text index. The application under test manages two pools of worker threads. The first pool consists of *searcher* threads that process the search queries coming from the workload generator. The second pool consists of *index updater* threads that process the updated content coming from the product update simulator. The performance of the system is monitored using the *performance monitor* unit.
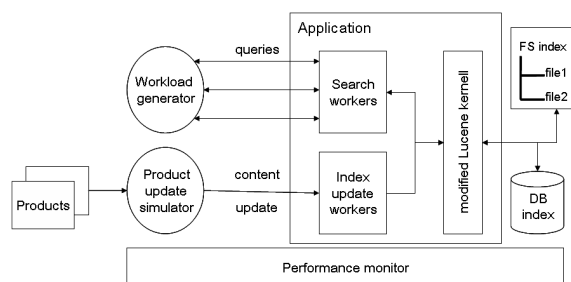
Figure 4: Components of the performance evaluation toolkit.

4.1 Input Parameters and Performance Metrics

We choose the maximum number of fetched hits to be 20 documents. This is a reasonable assumption taking into consideration that no more than 20 hits are usually displayed on a web page. The number of search threads is varied from 1 to 25 enabling the concurrent processing of 25 search queries. Due locking restrictions inherent in Lucene, we restrict our experiments to maximum one index update thread. We also introduce a think time varying from 20 to 100 milliseconds between successive index update requests to simulate the format specific parsing of the updated products.

In all our experiments, we monitor the overall system throughput in terms of conducted:
- *searches per second*, and
- *index updates per second.*

We also monitor the response time of:
- the *searches*, and
- the *index updates*

from the moment of submitting the request till receiving the result.

4.2 System Configuration

In our experiments we use a dual core Intel Pentium 3.4 GHz processor, 2 GB RAM 667 MHz and one hard disk having 7200 RPM, access time of 13.2 ms, seek time of 8.9 ms and latency of 4 ms. The operating system is Windows XP. We use JDK 1.4.2, MySQL version 5.0, JDBC mysql-connector version 3.1.12, and Lucene version 1.4.3.

4.3 Experiment Results

The performance evaluation considers the main operations: *complete index creation*, simultaneous *full text search* over single terms under various workloads, and - in parallel - performing *index update* as product data change. The experiments are conducted for the file system index and the database index. We drop the RAM directory from our consideration, since the index under investigation is too large to fit into the 1.5 GB heap size provided by Java under Windows.

4.3.1 Complete index creation

Building the complete index from scratch on the file system takes about 28 minutes. We find that the best way to create the complete index for the database is to first create a working copy on the file system and then to migrate the index from the file system to the database using a small utility that we developed to migrate the index from one storage to the other. This migration takes 3 minutes 19 seconds to complete. Thus, the overhead in this one time operation is less than 12%.

4.3.2 Full text search

In this set of experiments, we vary the number of search threads from 1 to 25 concurrent worker threads and compare the system throughput, illustrated in Fig. 5, and the query response time, illustrated in Fig. 6, for both index storage techniques.

We find that the performance indices are enhanced by a factor > 2. The search throughput jumps from round 1,250,000 searches per hour to almost 3,000,000 searches per hour in our proposed system. The query response time is lowered by 40% by decreasing from 0.8 second to 0.6 second in average. This is a very important result because it means that we increase the performance and take the robustness and scalability advantages of database management systems on top in our proposed system.
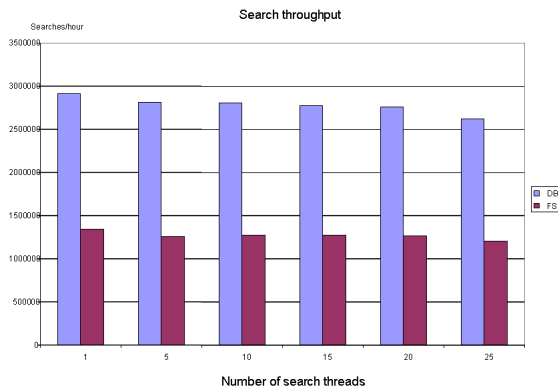
**Figure 5:** Search throughput in an update free environment
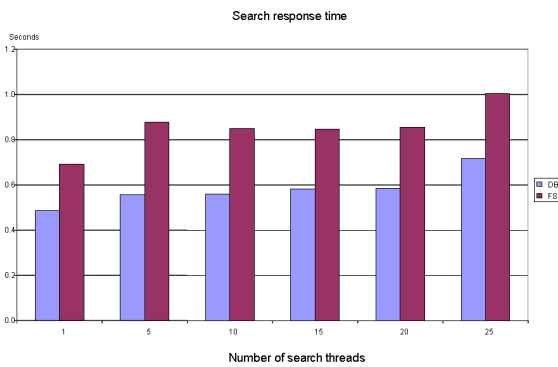
Figure 6: Search response time in an update free environment
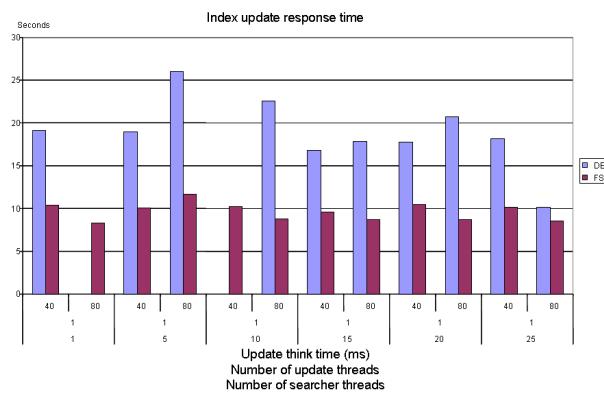
4.3.3 Index update

In this set of experiments, we enable the incremental indexing option and repeat the above mentioned experiments of Section 4.3.2. for different settings of think time between successive updates. In order to highlight the effect of incremental indexing, we choose very high index update rates by varying the think time from 20 to 100 milliseconds. For readability purposes, we only plot the results of the experiments having a think time of 40 and 80 milliseconds. In real life, we do not expect this exaggerated index update frequency.

Fig. 7 demonstrates that the throughput of the index update thread in our proposed system is slightly better than the file system based implementation. However, Fig. 8 shows that the response time of the index update operation in our system is worse than the original one. We attribute this to an inherent problem in Lucene. During index update, the whole index is exclusively locked by the index updater thread. This is *too* restrictive. In our implementation, we keep this exclusive lock although the database management system also keeps its own locking on the level of tuples which is less restrictive, which would allow for more than one index update thread and certainly more concurrent searches. The extra overhead of holding both locks lead to the increase in the system response time. The good news is that the response time always

remains under the absolute level of 25 seconds which is acceptable for most application taking into consideration the high update rate.
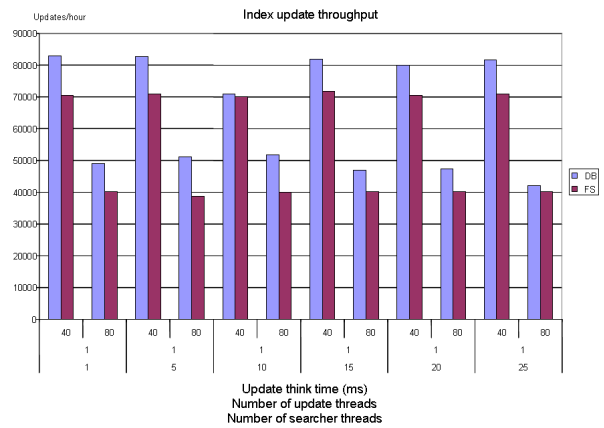

Figure 7: Index update throughput

Figure 8: Index update response time

The search performance of our proposed system becomes very comparable to the original file system based implementation in an environment suffering from a high rate of index updates. Fig. 9 shows that the search throughput of the proposed system is slightly better than the file system based implementation; whereas Fig. 10 shows that our database index suffer from a slightly higher response time than the original system.

Again, the effect of the exclusive lock over the whole index during index update is remarkable by comparing the performance indices of Fig. 5 and Fig. 6 to those of Fig. 9 and Fig. 10, respectively. The search throughput drops from 3,000,000 to round 1,100,000 searches per hour and the response time increases from 0.6 seconds to round 3 seconds.
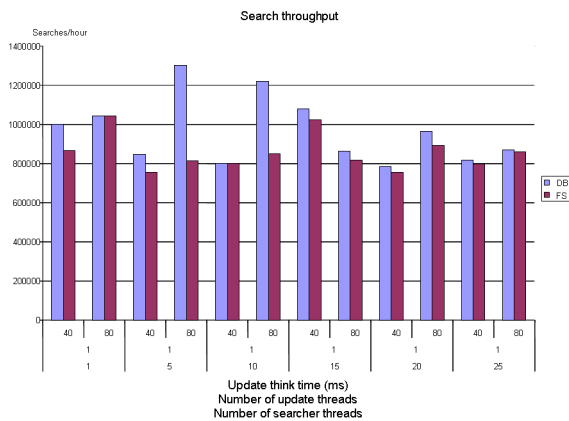
Figure 9: Search throughput in an environment with high update rate.
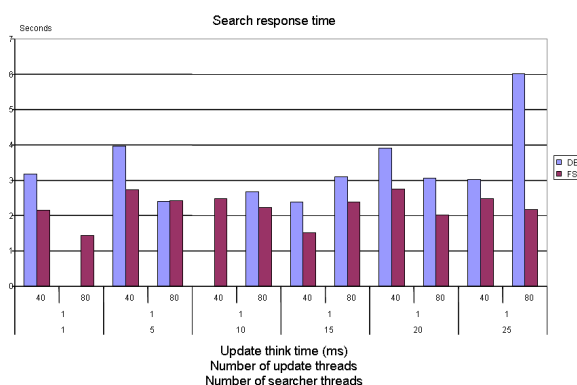
Figure 10. Search response time in an environment with high update rate.

5   CONCLUSION AND FUTURE WORK

In this paper, we attempt to bring information retrieval back to database management systems. We propose using commercial DBMS as backend to existing full text search engines. Achieving this, today's search engines directly gain more robustness, scalability, distribution and replication features provided by DBMS.

In our case study, we provide a simple system integration of Lucene and MySQL without loss of generality. We build a performance evaluation toolkit and conduct several experiments on real data of an electronic marketplace. The results show that we reach comparable system throughout and response times of typical full text search engine operations to the current implementation, which stores the index directly in the file system on the disk. In several cases, we even reach much better results which mean that we take the robustness and scalability of DBMS on top.

Yet, this is only the beginning. We plan on mapping the whole internal index structure into database logical schema instead of just taking the file chunk as the smallest building block. This will solve the restrictive locking problem inherent in Lucene and will definitely boost overall performance. We also plan on extending our performance evaluation toolkit to work on several sites of a distributed database.

REFERENCES

[1] Oracle Text. An Oracle Technical White Paper, http://www.oracle.com/technology/products/text/pdf/10gR2text_twp_f.pdf. (2005).

[2] Apache Lucene, http://lucene.apache.org/java/docs/index.html.

[3] B. Hermann, C. Müller, T. Schäfer, and M. Mezini: Search Browser: An efficient index based search feature for the Eclipse IDE, Eclipse Technology eXchange workshop (eTX) at ECOOP (2006).

[4] MIT OpenCourseWare, MIT Reports to the President (2003–2004).

[5] Nutch home page, http://lucene.apache.org/nutch/

[6] D. Cutting, J. Pedersen: Space Optimizations for Total Ranking, Proceedings of RIAO (1997).

[7] D. Cutting, J. Pedersen: Optimizations for Dynamic Inverted Index Maintenance, Proceedings of SIGIR (1990).

[8] Apache Lucene - Index File Formats, http://lucene.apache.org/java/docs/fileformats.html.