DATA REPRESENTATION AND RETRIEVAL USING SOFT COMPUTING

Bindu Sharma[1]
Associate Professor
IIMT-IET, Meerut

Ritu Rathi[2]
Assistant Professor
IIMT-IET, Meerut

Amit Kr. Chaudhary[3]
Associate Professor
IIMT-IET, Meerut

Abstract—

In data base management system an information retrieval is emerged as independent research area for more than a decade ago. It was the requirement due to the increasing functional needs that modern search engines have to meet. Current DBMS are not efficient to supporting such flexibility. To manage the huge amount of data, some methods are used. However, with the increasing of data as per demand to be indexed and retrieved and increasing heavy work load, modern search engines suffers from various factors like scalability, reliability, distribution, performance and result orientation performance problems. The DBMS is popular for its long traditional in coping with these challenges. Instead of traditional approach, we propose using current DBMS as backend to existing full text search engines. We present a new and simple method for integration and compare the performance of our system to fetch the efficient data representation of DBMS and information Retrievals.

1.Introduction:

Data base management systems commercially used to store large amount of data in specific manner. The cost of storage devices growth is based on relational presentation, graphical presentation and textual presentation. Many commercial data base management systems are organized with basic phonetic full text search functionality or operationability. To prove this, DBMS example called Oracle; has a level called Oracle text[8]. Although seeking to add much more functionality and intelligence for data representation and to their skills, many commercial database application use another party specialized full text search engines instead. For this purpose various commercial products are available in the market. Lucene[2] is the very famous open source product at this moment, as it based on refined searching qualities for the Eclipse IDE[3], the Encyclopedia Britannica CD-ROM/DVD, FedEx, new scientist magazine, Epiphany, MIT's open-courseware[6] and many more.

An index of the data to be retrieved in user queries as all search engines is based on this. The index is usually stored in the file system on disk and can be called or loaded at startup of the memory to fetch faster querying. However, this is not applicable easily for large indices due to the memory size limitations. So basically the standard data storage usually remains the file system of the disk. However, many search engines are suffer from scalability problems due to the increase of data to be indexed and retrieved under heavy workloads of user queries, to provide adequate response time for their users and keeping good complete system throughput. To manage with above problems, search engines should provide or must based on more intelligent techniques for accessing the disk. Another problem also arises i.e. reliability. To lost the data in a database after a similar crash to compare with whole index during a system crash is much higher the possibility of corrupting. To recall or restore the damaged index can also take lots of hours and situation become more complex even further. The search engine must manage its read & write locks by itself as well. The index is distributing at several sites and providing efficient

mirroring techniques is becoming an important issue to large scale search engine based projects like Nutch[7].
The database management systems are popular for its long traditional in coping with these challenges. Instead of traditional approach, we propose using current DBMS as backend to existing full text search engines. We present a new & simple method for integration. As a base of study, we use data extraction based on real world with an electronic area and concluded real world workload to show that the overall system throughput and query response time do not suffer with the use of DBMS as backend with their inherent overhead.

2. Introduction to full text search engines :

2.1Traditional Characteristics:

Full text search engines are very easy to understand as it do not care about the base of data or its formation as it is converted to plain text. Text is logically grouped into a set as documents. The user program creates the user query which is collected to the search engine. The result of the query execution is a list of documents identification numbers which satisfy the predicates described in the query.
The results are usually stored according to an internal scoring mechanism using fuzzy query processing technique [4][5]. The score is an indication of the relevance of the document which can be affected by many factors. Some fields are boosted so that hits within the 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.

For example: searching for "Sieman Hakines", a document containing "Sieman" at its starting and "Hakines" at its end. Except this, search terms can be easily augmented by searches with synonyms. For example, searching for "Tiger" retrieves documents with the term 'Animal' or "Zero' as well. This give the large area for ontological searches and other semantically richer similarly searches.

2.2 Architecture :

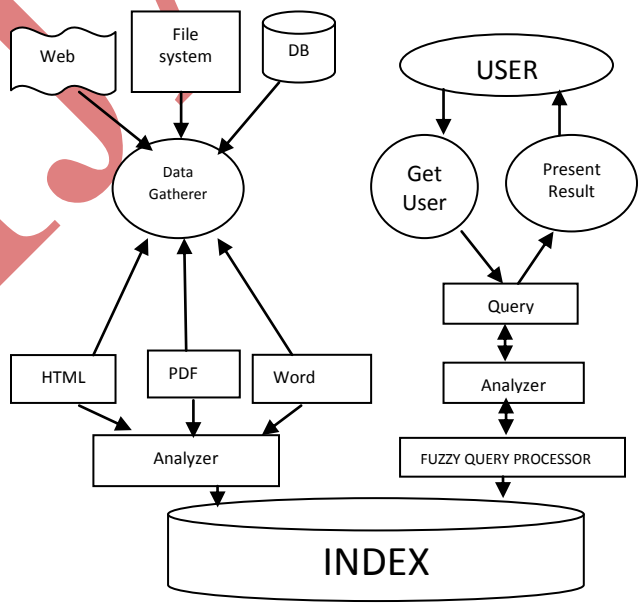

Figure 1 : Architecture of a Full text search Engine

An index is highly efficient cross-reference lookup data structure. Figure1: shows at the heart of a search engine resides an index. In most of the search engines, a variety of the well known **inverted index** structure is used[5]. An inverted index is an inside out organization of documents such that terms take centre stage. Each term refers to a set of documents.

Generally, 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 token could be lowercased before indexing. It is also desirable to remove all step words. In addition, it is common to reduce them to their roots to enable phonetic and grammatical similarly searches. The search process begins with parsing the user query. The token 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.

2.3 Traditional Operations:

2.3.1 Creation of Complete Index:

Only once this operation is used. The complete set of documents is parsed and searched in order to create the index from scratch. This operation is taking long hours to complete.

2.3.2 About Full text search:

This operation merging processing the query and returning page hits in the form of a list of documents IDs stored according to their relevance.

2.3.3 About Index Update:

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

3. Proposed System Integration :

3.1 Introduction to Architecture Proposal:

Lucene divides its index into various small modules. The data in each module is spread across several files. Each index file keeping a certain type of information. The exact number of files that constitute a Lucene index and the exact number of modules change from one index to another and based on number of

fields the index contains. The internal structure of the index file is public and is platform independent. This ensures its portability. We are using the index file as our basic building block and store it in the MySQL database as shown in figure 2. The set of files; i.e. the logical directory, is mapped to one database relation. Due to the high range of changes in file size, we divide each file into multiple frame of fixed length. Each frame is stored in a individual tuple in the relation.

This leads to better outcomes than storing the whole tuple is the F_Name and frame 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 described mapping. Using this simple mapping, we do not violate the public index file format and present a simple yet efficient way to select between the various file storage media likewise file system, RAM files or database.

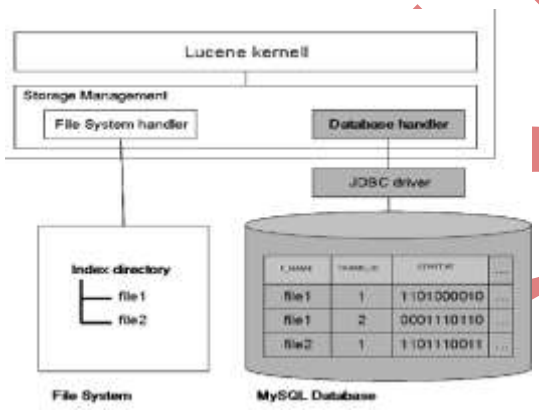

Figure 2: Integrating Lucene index in MySQL Database

3.2 System Design:

To design the system we used the UML class diagram of the store package of Lucene. Only relevant classes are including for the processing. The newly introduced class is grayed. Director is an abstract class that acts as a container for the index files. Lucene is based on two implementation for the file system directory also called FS directory and in-RAM index also called RAM-Directory. It is used to declare all basic files operations such as listing all file names, checking its time stamp etc. it is also responsible for opening files by returning an input stream object and creating a new file by returning a reference to a new instance of the output stream class. We introduce a database implementation, DB directory, which maps these operations to SQL operations on the data base.

Input stream & *output stream* are abstract classes that mimic the functionality of the java.io. counter parts. Default, 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 *FSInput* Stream and *RAMInput* Stream, we provide the database dependent implementation of the *readInternal* and *seekinternal* methods. Moreover, the *DBOutputStream* provides the data base specific flushing of the file buffer after the different write operations. Other buffer management operations are also implemented.

Both *DBInputstream* and *DBOutputStream* use the central class *DBFile*. A *DBFil*e object provides access to 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 for coherency of the cache.

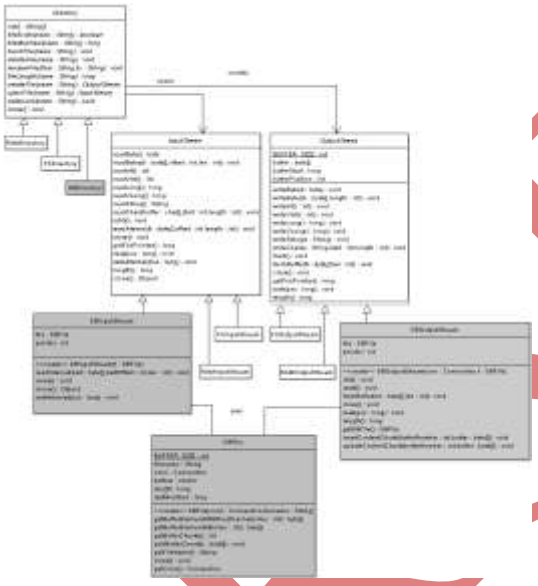

Figure 3: UML class diagram of the store package after modification

4. Performance Evaluation :

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 market place. It is build over the textual description of more than one million products and each product contains around 25 fields varying from few characters to more than 1300 characters each. We create a performance evaluation toolkit around the search engine shown in figure 4.

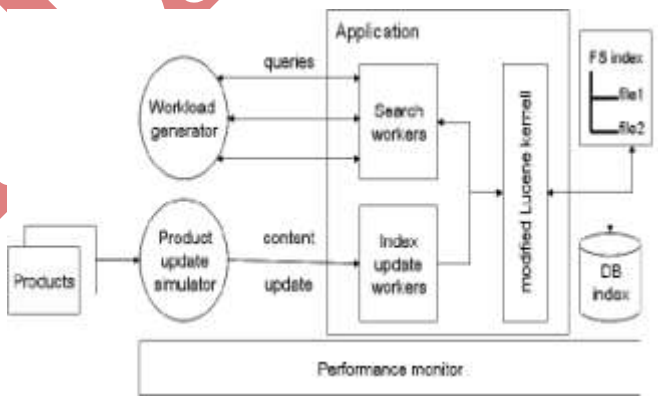

Figure 4: Components of performance the evaluation toolkit

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 vary-ing from 20 to 100 milliseconds between successive index update requests to simulate the format specif-ic 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 re-ceiving the result.

4.2 System Configuration Specification:

In our experiments we use a dual core Intel Pentium 5.4 GHz processor, 4 GB RAM 867 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 Findings

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 up-date* as product data change. The experiments are conducted for the file system index and the data-base 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 Creation of Complete index:

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 data-base management systems on top in our proposed system.

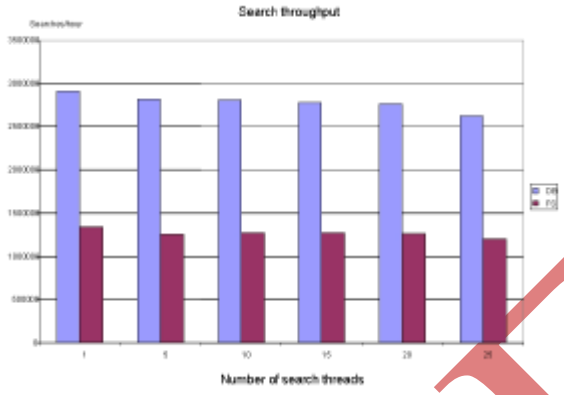

Figure 5: Search throughput in an update free envi-ronment

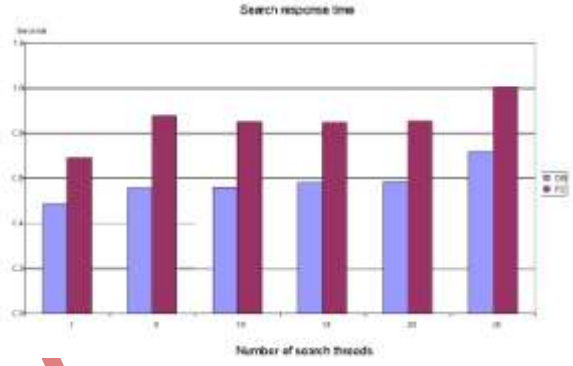

Figure 6: Search response time in an update free environment

4.3.3 Index update approach:

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.

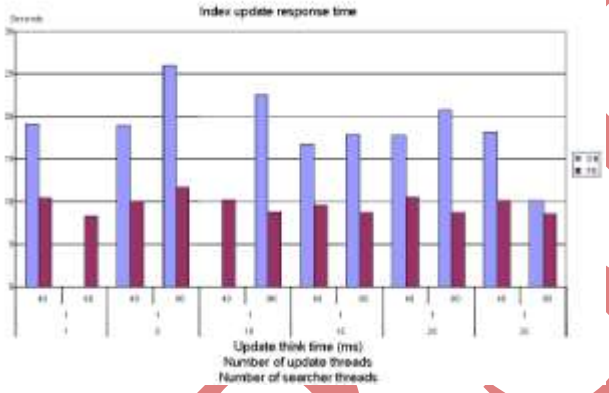

Figure 7: Index update throughput

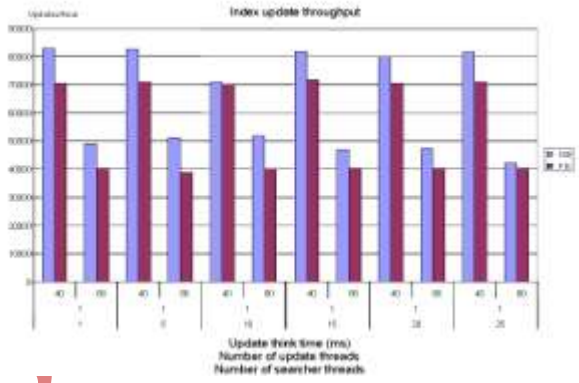

Figure 8: Index update response time

The search performance of our proposed sys-tem becomes very comparable to the original file system based implementation in an environment suffering from a high rate of index updates.

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. 8 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.

5 CONCLUSIONS 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 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 to work on several sites of a distributed database.

REFERENCES:

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

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

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

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

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

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

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

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