www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 5 Issues 6 June 2016, Page No. 17001-17005

Database Management System And Information Retrival

Shubham S. Tade[1], Tejashri A. Kapse[2]

[1]S.G.B.A University, P. R. Pote College of Engg. & Mang.,
Amravati, Maharashtra, India *shubham.tade@gmail.com*

[2] S.G.B.A University, P. R. Pote College of Engg. & Mang.,
Amravati, Maharashtra, India *teju.kapse1992@gmail.com*

Abstract: ***A database management system (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data. It allows organizations to conveniently develop databases for various applications by database administrators (DBAs) and other specialists. Information retrieval (IR) is the activity of obtaining information resources relevant to an information need from a collection of information resources. Searches can be based on or on full-text (or other contentbased) indexing. 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. 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 IRS, DBMIRS, scalability**.

1. Introduction

New applications like office information systems need interfaces to data bases which integrate classical data manipulation with management and retrieval of textual ("unformatted") data. The relational data model is widely accepted as a high level interface to classical ("formatted") data management. It turns out, however, to be inconvenient for handling even simple data structures as commonly used in information retrieval systems. To attack this shortcoming we propose an extension of the relational model by allowing Non First Normal Form (NF^2) relations. We summarize extensions of the relational algebra, with main emphasis on the new "nest" and "unnest" operations which transform between first normal form relations and the NF^2 ones. In the past, database management systems (DBMS) and information retrieval systems (IRS) were separated in research and development and different products have been developed for either purpose. At present there is a trend towards a single integrated system for data base management and information retrieval called DBMIRS -because of the following reasons: Many applications need a DBMIRS. Examples are patients' data within hospital information Systems, laboratory document administration Pharmaceutical data bases, and library Information systems, and with growing awareness office information systems. A characteristically feature of these applications is the fact that it is necessary to combine text management and retrieval with usual formatted data manipulation. Therefore a single user interface is necessary. 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.. Reliability becomes also a problem. The possibility of corrupting the whole index during a system crash is much higher than losing 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 mirror-ing techniques is becoming an important issue to large scale search engine projects such as Nutch [5].We propose *using current DBMS as backend to existing full text search engines* as opposed to either reimplementing full text search engine functionality into DBMS or re-implementing core DBMS features into search engines. As a case study, we use the opensource *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. spectrum of basic infrastructural facilities offered by DBMS The rest of the paper is organized as follows.

DOI: 10.18535/ijecs/v5i6.35

2. BACKGROUND ON FULL TEXT SEARCH ENGINES

2.1Typical Features

Full text search engines do not care about the source of the data or its format as long as it is con- verted 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 dis- tance 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 seman- tically 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 wellknown *in- verted 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 cri- teria during the traversal and the score of the hit.

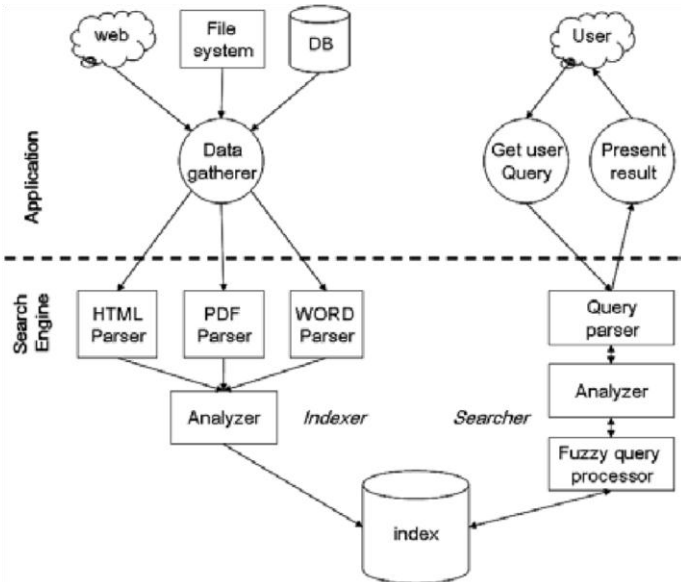

Figure 1: Architecture of a full text search engine

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 index- ing*. It is not supported by all search engines. Typi- cally, a worker thread of the application monitors the actual inventory of documents. In case of doc- ument 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 infor- mation. 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 illu- strated 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 present a simple yet elegant way of choosing between the different file storage media (file system, RAM files, or *database*).

DOI: 10.18535/

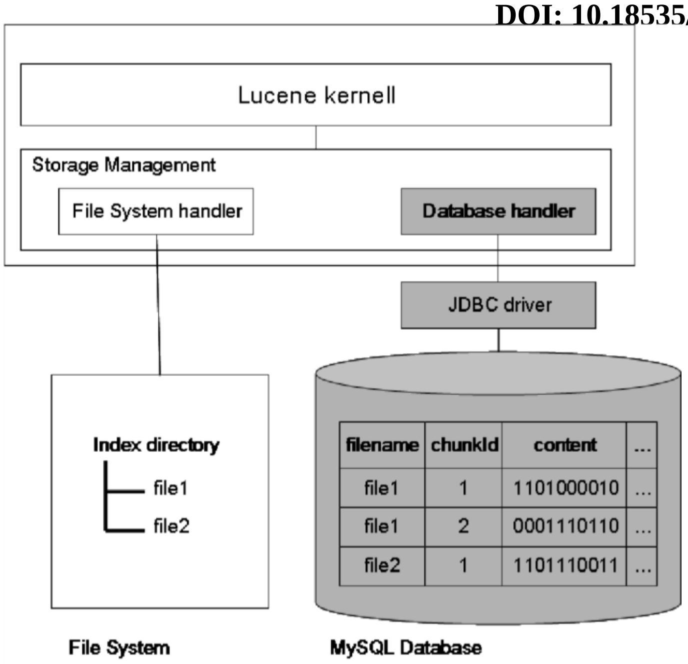

Figure 2: Integrating Lucene index in MySQL database

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

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 inRAM index (RAM Directory). 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 Input Stream object and creating a new file by returning a reference to a new instance of the Output Stream 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 RAMInputStream, we provide the database dependent implementation of the read Internal and seekInternal methods. Moreover, the DBOutputStream 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

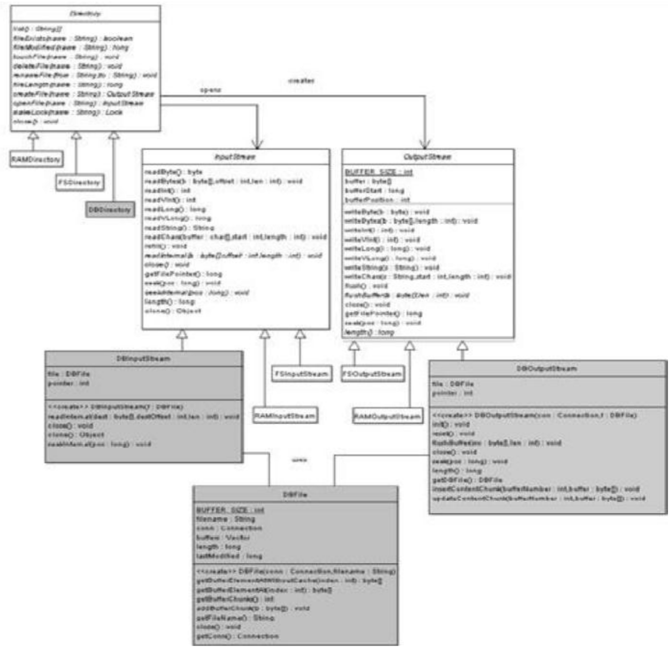

Figure 3: UML class diagram of the store package after modification.

adjusted to make use of the available free memory of the system. The class is responsible for guaranteeing the coherency of the cache.

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 elec- tronic marketplace. The index is build over the tex- tual description of more than one million products. Each product contains approximately 25 attributes varying from few characters to more than 1300 cha- racters 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 con- tent 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 con- tent coming from the product update simulator. The performance of the system is monitored using the *performance monitor* unit.

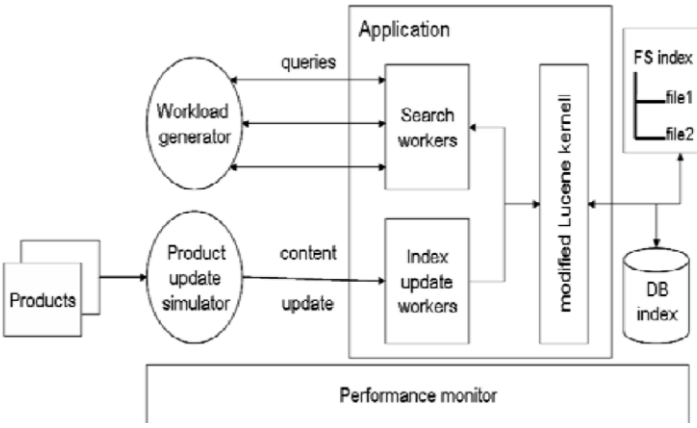

Figure 4: Components of the performance evalua- tion toolkit.

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 specifyic 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 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 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, illu- strated in Fig. 5, and the query response time, illu- strated in Fig. 6, for both index storage techniques.

We find that the performance indices are en- hanced 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.

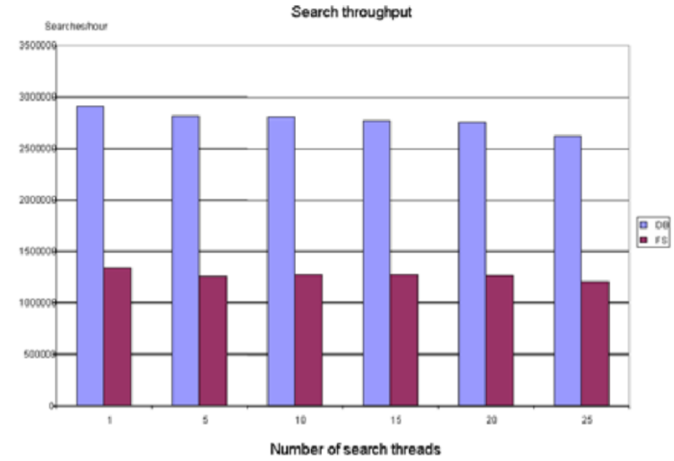

Figure 5: Search throughput in an update free environment.

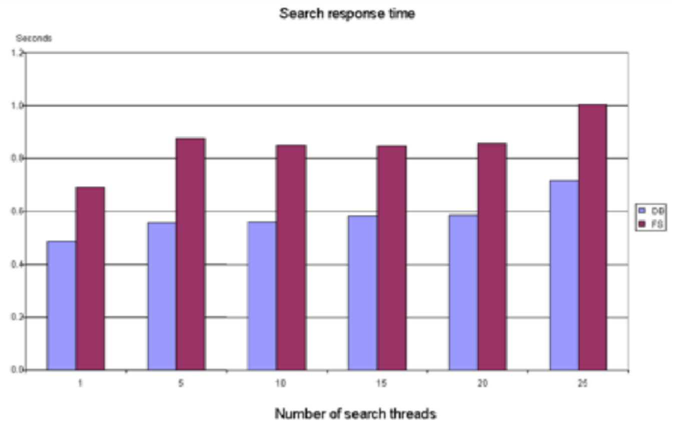

Figure 6: Search response time in an update free environment.

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] Apache Lucene – Index File Formats, http://lucene.apache.org/java/docs/fileformats.html
[2] Oracle Text. An Oracle Technical White Paper, http://www.oracle.com/technology/product/text/pdf/10gR2text_twp_f.pdf (2005)
[3] Apache Lucene, http://lucene.apache.org/java/docs/index,html
[4] Nutch home page, http://lucene.apache.org/nutch/
[5] D. Cutting, J. Pedersen : Space Optimizations for Total Ranking, Proceedings of RIAO (1997)
[6] D. Cutting, J. Pedersen : Optimizations for Dynamic Inverted Index Maintenance, Proceedings of SIGIR (1990)

Author Profile

Shubham S. Tade Studing B.E. Computer Science Engg. AT P. R. Pote College of Engg. & Mang., Amravati

Tejashri A. Kapse Studing B.E. Computer Science Engg. AT P. R. Pote College of Engg. & Mang., Amravati