MJRET
Open Access
Multidisciplinary Journal of Research in Engineering and Technology
Journal homepage: www.mjret.in
ISSN:2348 - 6953

BEST SEARCH AND RETRIEVAL PERFORMANCE EVALUATION WITH LUCENE INDEXING

Sonam Baban Borhade, Prof. Pankaj Agarkar
Department of Computer Engineering
Dr. D.Y.Patil School of Engineering,Lohegaon
Pune, Maharashtra, India.
borhade.sonam@gmail.com , pmagarkar@gmail.com

Abstract: *Transaction processing in organizations commonly use relational database, but big part of database operations need select operation. As data increases with few gigabytes selection operation required more time to process whole transaction. There is one approach get frequently used to build indexes in database on columns which is selection. If number of table's (which is general case)get used then selection takes more time. Another approach we can use which is searching framework for searching records. Apache Lucene is a high performance full featured text search engine library written entirely in java which allows simultaneous update and searching. In this paper, we are using these capabilities of Lucene to improve performance of data retrieval. We are using Lucene indexing types to reduce load on database for selection operations. So leveraging advantages of Lucene search for our need, we propose and implement search history module in Lucene as a plugin so that it can be added at any stage in search framework. So that the search history maintenance for each user will be easy as well as can give good performance for large number of users. This paper surveys Lucene indexing and searching.*

Keywords: *Information Search, Information Retrieval, Performance Evaluation, Indexing.*

1. INTRODUCTION

1.1 Lucene Technology

Lucene is NOT a crawler, not an application, and mainly not a library to do Google PageRank and other link analysis algorithms. Lucene is actually a library which gives text based search. A Lucene Index is a collection of Documents, A Document is a collection of Fields, A Field is content along with metadata describing the content. Field can have several attributes:

1. Tokenized - Analyze the content, extracting tokens and adding them to the inverted index
2. Stored - Keep the content in a storage data structure for use by application

However, as confirmation of Lucene's popularity, there are numerous projects that integrate with or build on Lucene, and that could be a good fit for your application. let see concept of searching and indexing in detail.

1.1.1. Lucene architecture Component

Lucene s primary goal is to facilitate information retrieval. The emphasis on retrieval is important. Indexing and searching steps have to be supplemented with parsing and analysis in order to achieve the best search results. The whole process of information retrieval can be divided into several sequential steps. Lucene act as component for search application it does not provide all functionality of searching. Lucene mainly focus on indexing, searching and analysis. Figure below shows all architectural components of lucene required for searching which will discussed in reminder sections.

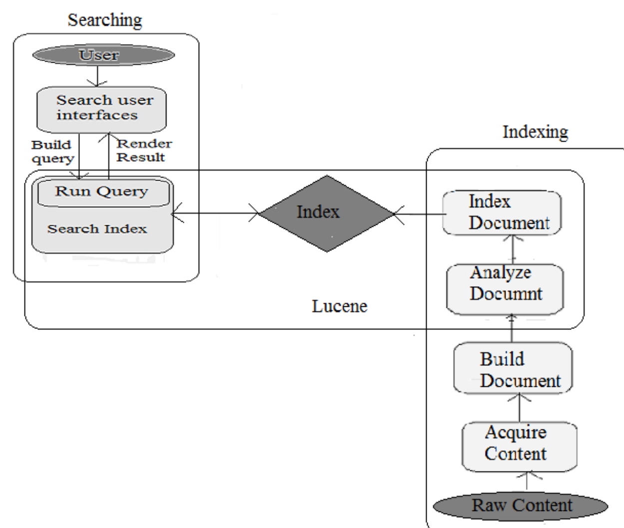

Fig.1: Architectural components of Lucene search application

1.2. Indexing

Process of preparing and adding text to Lucene and Optimized for searching. Key Point of Lucene is Lucene only indexes Strings but it not means that Lucene have not care about XML, Word, PDF, etc. There are available good open source extractors available and It's our job to convert it in file format which we want. Indexing factors as below:

- Lucene indexes Documents into memory
- On certain occasions, memory is flushed to the index representation (called a segment)
- Segments are periodically merged
- Internal Lucene models are changing and (drastically) improving performance

1.2.1. Indexing process

As discussed in previous section there are, to index a document few methods of Lucene's public API necessary. And as a result, indexing with Lucene looks like a spuriously simple and intractable operation from the outside,. But this simple API actually relay on quite different and comparatively complex operations. All those operations get break down into three major and operatively discrete groups, which is depicted in following figure 1.1

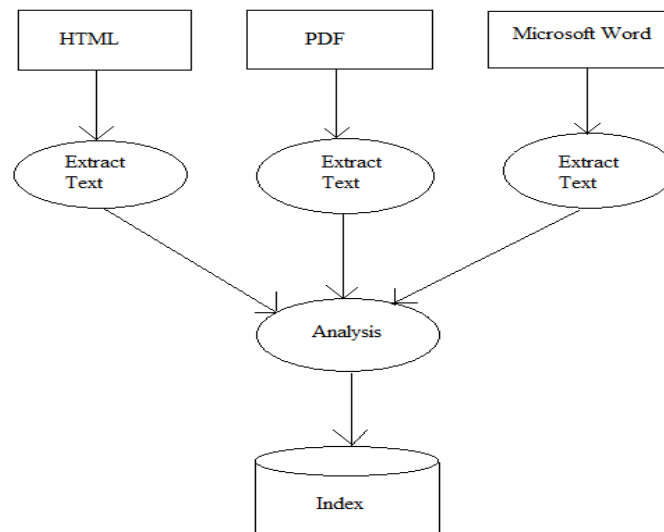

Fig.2: Indexing with Lucene [1]

1. Extracting text from source.
2. Analyzing it,
3. Saving it in Index,

In indexing step, the text first get extracted from the original content. Then extracted text used to create a Document instance. Document instance contains Field instances which

holds content. Then text in the fields get analyzed which produces a tokens stream. At end, tokens are combined to the index in a segmented architecture. Let focus on extraction first.

1.2.1.1 Extracting text and creating the document

To index data with Lucene, we have to extract plain text from it, and the format that Lucene can bare, and afterwards create Lucene document. Suppose you want to index some documents in PDF format. To create such documents for indexing, you first need to use method which will extract the information in form of text from the PDF manuals and after that that extracted text get used to create documents in Lucene and their fields. Similarly if you want to apply index on Microsoft Word documents or any other document which is not in plain text form. And also if you're handling with XML or HTML documents, which use plain-text characters, you need proper skill about preparing the data for indexing. Once you got the text that you want to index, and created a document with all fields that you want to index, all text should be analyzed then

1.2.1.2. Analysis

Analysis is the process of converting raw text into indexable Tokens. In Lucene, this is done by the Analyzer, Tokenizer and Token Filter classes The Tokenizer is responsible for chunking the input into Tokens. Token Filters can further modify the Tokens produced by the Tokenizer, including: Removing it, Stemming, and Other. Analysis is easy to add your own, done on both the content to be indexed and the query.

Once we created Lucene documents with fields, then can call Index- Writer's add Document method and pass data to Lucene to index. After that, Lucene first analyzes the text, and then textual data get splited into a tokens stream, and then performs a number of operations on tokens. Use Lucene's Lowercase Filter to make searches case insensitive. The combination of an main source of tokens, with number of filters which modify the tokens created by that original source, buildup up analyzer. We can build our own analyzer by adding Lucene's token sources and filters in chaining together, or by own customized ways.

This important step, covered under the Analyze Document step in Fig.1, is called analysis. The analysis process then convert stream of tokens to written into the files in index.

1.2.1.3. Adding to the index

After analyzing input, now it is ready to add to the index. Lucene uses data structure known as an inverted index to stores the input. Inverted index makes proper use of disk space and allowing quick keyword lookups. It is Inverted structure because is it uses tokens which are extracted from input documents in form of lookup keys. It doesn't treat documents as the central entities. Means we can directly search word instead of whole document.

1.2.2. Index Segments

Lucene having a rich and detailed index file format which has been carefully and properly optimized with time. Lucene index have more than one segments, as shown in figure 3 below.

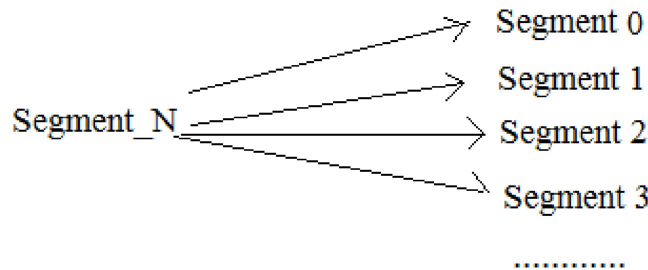

Fig.3: Lucene inverted index segments.

Each segment holds a subset all documents and document's in indexed format. At the time At the time of creation of new segment writer will flush buffered pending deletions and added documents into the directory. And at search time, each and every segment will be visited separately and then results are combined. Every segment consists of multiple files, in the form of _X. <ext>, segment's name is X here and to identify which part of the index extension <ext>is used. Index Writer select segments then merge them into a single new segment after that removes old segments. There are a separate Merge Policy for selection of segments to be merged. Merge Scheduler is then used to execute selected merges.

1.2.3. Core classes to index

For simplest indexing procedure we need following classes [2]:

1. Index Writer
2. Directory
3. Analyzer
4. Document
5. Field

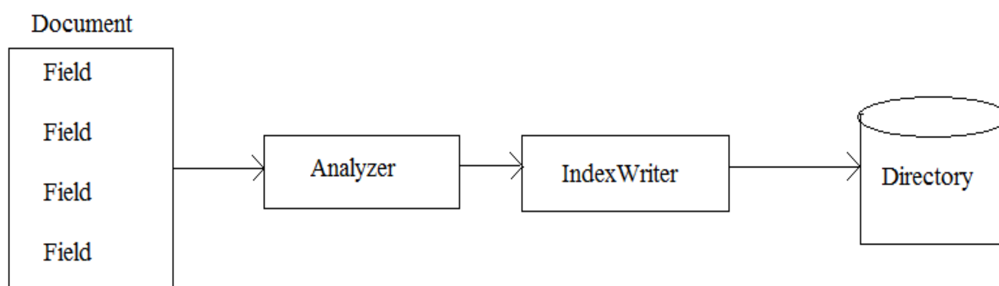

Fig.4: Indexing documents with Lucene classes.

Fig.4 [2] shows how classes used in the indexing process. Details are as below

1.2.3.1. IndexWriter [2]

Index Writer is the main component of the indexing process. Index writer class used to make new index also it can open an existing index. This class can adds, delete, or modify documents in the index.

1.2.3.2. Directory [2]

The Directory class is an abstract class which represents the location of a Lucene index. In directory class it permit subclasses to store the index as it fit in application. Example, FSDirectory.

1.2.3.3. Analyzer [2]

Before indexing text, have to pass it to analyzer. Index Writer constructor used to specify analyzer, out of text tokens get extracted by this class. And this token should be indexed and eliminate those are not useful. Before indexing you should first extract plain text from content if it is in not textual form. Analyzer is important class of Lucene and used as more than simple filtering input.

1.2.3.4. Document [2]

The Document class is a collection of fields. Fields contain metadata which belong to document and document is just an carrier of filed. The real source of document is not relevant to Lucen. It is like text extracted from for example binary documents, and gets added as Field instance. Metadata is in form like author name, title, date, subject etc. this kind of metadata separately stored and indexed as fields of a document. Text and numbers handled only by Lucene. In Lucene its document and its fields are designed such that it will fulfill exact requirement of your content sources and application.

1.4.5. Field [2]

Field holds the textual content. Each indexed document contains one or more named fields. Each field has a name and value, and a plenty of options, used to control how Lucene able to index the field’s value. A document can have one field with the same name number of time. If such case noticed, the values of the fields are used to identify field in order in which they added to document in indexing.

1.3. SEARCHING

1.3.1. Search Concepts

User inputs one or more keywords along with some operators and expects to get back a ranked list of documents relevant to the keywords.

Models are used for searching like Boolean Model, Vector Space Model, Probabilistic Model. Vector Space Model is probably the most common and is generally fast Making Content Searchable Search engines generally[2]:Extract Tokens from Content, Optionally transform said tokens depending on needs,Stemming,Expand with synonyms (usually done at query time),Remove token (stopword),Add metadata,Store tokens and related metadata (position, etc.) in a data structured optimized for searching,Called an Inverted Index,Lucene Query Parser converts strings into Java objects that can be used for searching Query objects can also be constructed programmatically

1.3.2. Core Searching Classes

Few classes are needed to perform the basic search operation:

1. IndexSearcher
2. Term
3. Query
4. TermQuery
5. TopDocs

1.3.2.1. IndexSearcher

IndexSearcher searches content indexed by IndexWriter is. And it uses Directory instance to search indexed content, and then apply a number of search methods, methods implemented in its abstract parent class Searcher. A syntax as follows [5]:

Directory name = FSDirectory.open(path));[5]

Searching class= new searchingclass(directory name)[5]

Query = new TermQuery(new Term("contents", "lucene"));[5]

TopDocs hits = searcher.search(query object value);[5]

searcher.close();[5]

1.3.1.2. Term

Term is searching fundamental unit. like Field object, follows syntax as: pair of (field name, word (text value) of that field). We can use Term objects in the indexing process also.

1.3.1.2.Query

Lucenehave number of concrete Query subclasses.First Lucene Query is TermQuery. Second BooleanQuery, expect that it uses PhraseQuery, then PrefixQuery, TermRangeQuery, NumericRangeQuery, FilteredQuery, etc. Query is abstract parent class with several utility methods.

1.3.1.3. TermQuery

TermQuery is having primitive query type. This is an basic query. And to match documents which contain fields with specific values use primitive query type.

1.3.1.4. TopDocs

The TopDocs class having pointers.With this pointer it point,s to N ranked search results which are topN of documents that match a given query.And for every top N results, TopDocs records int docID also float score.

2. RELATED WORK

Generally lucene is used to internally store indexing in form of documents. Documents internally having fields. Every field have specific field name and every field name have value stored with it. Document can have multiple fields. Because of this feature we can search on a field of documents when searching accomplished. Here we need to treat every row as a document, and every column as fields to enter database entries from mysql database. Document is stored by its unique document ID. When searching lucene only provides document IDs. By using document IDs we can retrieve documents. Here document IDs are unique not permanent. That's why IDs for deleted documents do remain unused.For reuse purpose Document IDs of deleted documents get collected and to do that method of compression of index performed by lucene, but here in this case the document IDs change, due to that we can't depend on lucene document Ids [3]. Another framework is there which is apache solr , which is open source enterprise search platform provided by apache lucene project mainly used for web searching purpose. Solr uses lucene internally for searching and indexing then builds the server above that. This scenario provides scaling to searching which achieved by index replication on multiple servers. Where centralized server enters the indexing entries and then the server replicates the indexes. To perform searching any replication servers can be used.Lucene having indexing types like RAM Index,NIO Index,Simple Index. We can use SimpleFSDirectory having JAVA IO API which store index on hard disk, NIOFSDirectory having JAVA NIO API which store index on hard disk, and RAMDirectory having physical memory which store index. For small indexes RAMDirectory gives best performance; but having limitation of physical memory available on system [1].Lucene provide search results relvantly, which gives advantage that more relevant results shown first. On documents automatic indexing is performed, document vector contains words of documents and the weight of the word [4]. Precision and recall of the term search get calculated then. But here in our case relevance is not required because each and every record is mandatory for transaction processing. To apply search on multiple fields lucene search queries helpful. By using pre-classified documents set and using machine learning, automatic text classification achieved. But it produces vectors which are not interpreted by human. To classify documents and produce classification which human can interpret, genetic algorithms suggested by author [5]. we can provide fuzzy search [6] in Lucene in

which small error in input can be tolerated [7].In lucene every input string divided in tokens (single word) and multiple word input is divided into tokens to search. On same column to search can use multiple keywords. Complex identifier indexing is now in active research area [8], but it is not supported by lucene.Numerous approaches for type-ahead search get studied in 2011, Guoliang Li et al [9]. When users enters query, new query generated by every keystroke. To mitigate minor errors fuzzy search used. A tree with inverted lists at leaf nodes used as data structures. After this approach, new approach get analyzed to use full-text indexing for map-reduce framework which optimize selection operations on records text field, In 2011, Jimmy Lin et al. [10]. This paper concluded on some results. These results actually focus on moderate improvement in query processing time and at worker nodes it saves processing time. In 2003, James Abello et al. [11] focus on indexing mechanisms which can used for graph databases. James states in this paper hierarchical two-level indexing schema that is gkd*-tree, and this schema is an combination of first-level kd-tree index with second-level of redundant R*-tree index which indexes leaf pages of gkd-tree. Maayan Geffet et al. In 2001 [12] gave some glossary on Web project and uses hierarchical index to which entries are linked. In which hierarchy of results of relevant topics have been shown. In 2005,a effective B+-tree based method of indexing for k-nearest neighbour search in high-dimensional metric space get proposed by H. V. Jagadish et al. [13]. This indexing method concluded that data partitions are flattened into single dimensional value for indexing and KNN-search performed using range search.At the same time In 2005, two compressed data structures for full-text indexing get proposed by Paolo Ferragina et al. [14] this approach added new invention that so while searching, decompression of data would not require and data storage will be less also overall processing required for searching will be less. Rushdi Shams et al. [15] stated in 2012 that text denoising method can used to extracts denoised text, in this approach indexer performs better than full-text trained indexer. Text denoising gives advantage over original size that it reduce text size up to 30% of original size.Jose E. Moreira et al. [16] In 2007, focused on analysis of performance and scalability of nutch configurations.

3. PROPOSED WORK

In this paper, we are evaluating the performance of record retrieval from apache lucene indexes with relational database. Using lucene one can search records faster so, by using lucene one can offload record retrieval from database to lucene which give free database resources for different transactions.

The search history maintenance is implemented which can be used by anyone who wants to maintain user search history. Per user search history will allow to show search history of particular user. This will enhance the lucene searching feature.

This system evaluates the use of lucene searching framework to perform data searching and retrieval from big databases. The searching performance evaluation is done on mysql database and different lucene indexes. The performance is compared and analyzed to see that which arrangement gives best performance.

In first phase the raw input data is inserted into mysql database and the lucene indexes are created. Then the searching is performed on both mysql database and different lucene indexes. The performance evaluation is done. Different architectures are used to get searched data, as data is directly stored in lucene indexes, or only index can be created with database primary key stored, and when search results are collected then the data is retrieved from database. The search history maintenance module is added to the searching module which uses lucene searching. The searching module stores and retrieves the user search history from history maintenance module. Figure below shows the proposed system architecture flow.

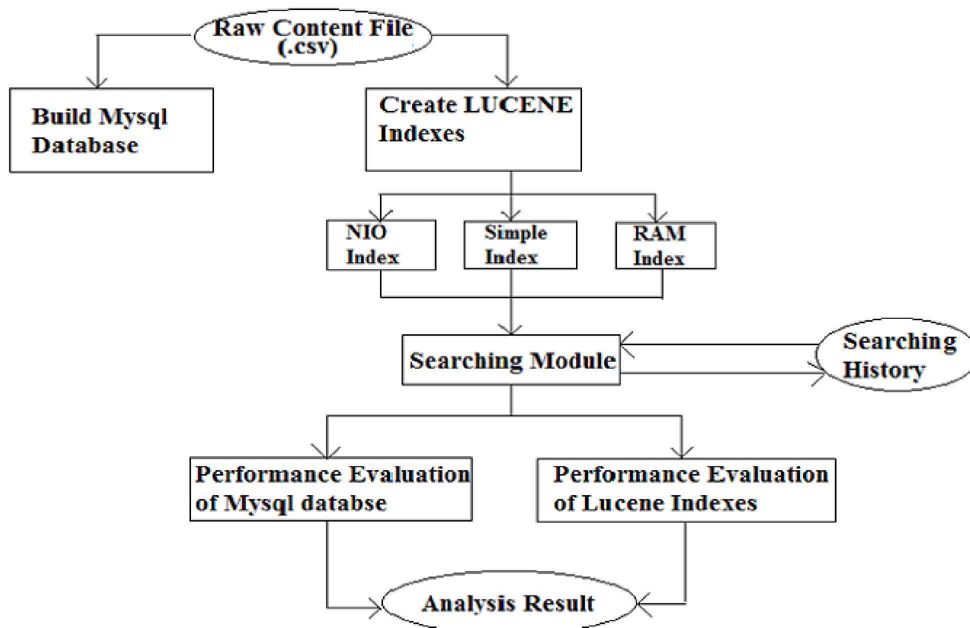

Fig.5: Proposed System Workflow

4. CONCLUSION

The main objectives of this paper were to study the Lucene technology. Initially, background knowledge of architecture on search applications was presented, and also core Lucene knowledge. This paper clear that Lucene is an information retrieval library, not a ready-to-use standalone product, and that it does not contain a data parser, document filter, or a search user interface. The introductory section provided a general overview of information retrieval and plus point of lucene is that lucene having very fast searching capabilities. So it helps to retrieve records from large databases. Observation says that index with data stored

give less performance compare to index without data stored. In this paper our focus on to retrieve data from database so no need of data storage. So can achieve faster performance.

ACKNOWLEDGEMENT

I would like to articulate our deep gratitude to our Project Guide and PG Cordinator Prof. Pankaj Agarkar, who has always been a source of motivation and firm support for carrying out the paper. I would also like to convey our sincerest gratitude and indebtedness to Prof. Soumitra Das, Head of Computer Engineering Department and all other faculty members of Department of Computer Engineering, Dr. D .Y Patil School of Engineering, Pune, who bestowed their great effort and guidance at appropriate times without which it would have been very difficult on our Paper Work.

REFERENCES

[1] Deng Peng Zhou, "Delve inside the Lucene indexing mechanism", Online At http://www.ibm.com/developerworks/library/wa-lucene/ (as of 14 January 2014)

[2] Michael McCandless, Erik Hatcher, Otis Gospodnetic, "Lucene in Action" 2nd edition, Manning Publications Co, 2010

[3] C. T. YU, G. Salton, "Precision Weighting - An effective Automatic indexing method", journal of Association for Computing Machinery, Vol. 23, No 1, January 1976.

[4] Laurence Hirsch, Robin Hirsch, Masoud Saeedi, "Evolving Lucene Search Queries for Text Classification", GECCO'07, July 7-11, 2007, London, England, United Kingdom. Copyright 2007 ACM 978-1-59593-697-4/07/0007.

[5] Amol Sonawane, "Using Apache Lucene to search text" Online At "http://www.ibm.com/developerworks/opensource/library/ os-apache-lucenesearch/" (as of 11 December 2013)

[6] Shengyue Ji, Guoliang Li, Chen Li, Jianhua Feng, "Efficient Interactive Fuzzy Keyword Search", Apr 20-24, 2009, ACM 978-1-60558-487-4/09/04.

[7] Gerard Salton, "Automatic text indexing using complex identifiers", 1988 ACM 0-89791-2918.

[8] Guaoliang Li, Shenguye Ji, Chen Li, Jianhua Feng, "Efficient fuzzy full-text type-ahead search", The VLDB Journal (2011) 20:617–640 DOI 10.1007/s00778-011-0218-x

[9] Jimmy Lin, Dmitriy Ryaboy, Kevin Weil, "Full-text indexing for optimizing selection operations in large-scale data analytics", ACM, San Jose, California, USA, 978-1-4503-0700-0/11/06, June, 2011.

[10] James Abello, Yannis Kotidis, "Hierarchical graph indexing", ACM, 1581137230/03/0011, Nov 2003.

[11] Maayan Geffet, Dror G. Feitelson, "Hierarchical indexing and document matching in BoW", ACM 1-58113-345-6/01/0006, June 2001.

[12] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, Rui Zhang, "iDistance: an adaptive B+- tree based indexing method for nearest neighbor search", ACM Transactions on Database Systems, Vol. 30, No. 2, June 2005.

[13] Paolo Ferragina, Giovanni Manzini, "Indexing compressed text", ACM, Vol. 52, No. 4, July 2005.

[14] Rushdi Shams, Robert E. Mercer, "Investigating keyphrase indexing with text Denoising", ACM 978-1-4503-1154-0/12/06, June 2012.

[15] Jose E. Moreira, Dilma Da Silva, Parijat Dube, Maged M. Michael, Doron Shiloach, Li Zhang, "Scalability of nutch search engine", ACM 978-1-59593-768-1/07/0006, June 2007.

[16] Michael McCandless, Erik Hatcher, Otis Gospodnetić Lucene in Action, Second Edition. Manning, 2010, 475 pages.

[17] David A. Grossman, Ophir Frieder Information retrieval: algorithms and heuristics. Springer, 2004, 332 pages.

[19] Ayşe Göker, John Davies Information Retrieval: Searching in the 21st Century.John Wiley and Sons, 2009, 295 pages.