COMPARING THE PERFORMANCE OF INDEX BASED SEARCHING AND ONTOLOGY BASED APPLICATION OF OLAP FOR INFORMATION RETRIEVAL

Fisnik DALIPI
Department of IT, Faculty of Math-Natural Sciences, Tetovo State University
e-mail: fisnik.dalipi@unite.edu.mk
Ilia NINKA
Department of IT, Faculty of Natural Sciences, University of Tirana
e-mail: ilia.ninka@fshn.edu.al

Abstract
Analyzing the data types and structures in a business environment where data grows rapidly has become a serious challenge for companies. As most business data are unstructured and modelled into complex models, we need a solution to perform multidimensional searching for achieving sustainable results. In this paper, we present the idea of incorporating index searching as part of a standard information retrieval system. We test our concepts by using business documents from Knauf Radika AD, a leading Macedonian plasterboard manufacturer located in Diber. Further, we describe and propose a novel architecture which is including an ontology-based approach by integrating OLAP and information extraction attributes to access structured and unstructured data, mainly organized in form of documents. In our first demonstration, the query performance was reduced as more documents were added to the index, and consequently the growth factor becomes very low. While at the second case, when a user performs navigation throughout the OLAP report, it is possible to track the user context information which can be used for searching other relevant documents.

1. Introduction

Considering the fact that most business data is highly unstructured and mainly organized as file based, the need for access to structured data is increasing. In order to reduce the cost for finding information and achieve relevant results there is a need to build a very complex query which indeed is a serious challenge.

Taking into account the continuous rise of world's information resources, it is undoubtedly necessary for enterprises to obtain aggregate, process and utilize them in an appropriate manner in order to maximize the efficiency of business activities.

Since the document libraries (or a number of sources to filter from) are becoming bigger and bigger, it is crucial to provide a trusted system which would be able to find the resources relevant to user's needs. This is a main goal of information retrieval (IR) systems [1].

Following the definition in [2] an IR model is a quadruple [D,Q,F, sim], where:

- D is a set of (logical representations of) documents.
- Q is a set of (logical representations of) queries.
- F is a framework for modeling documents, queries, and their relationships.
- sim: Q × D → U is a ranking function that defines an association between queries and documents, where U is a totally ordered set (commonly [0,1], or P, or a subset thereof). This ranking and the total order in U define an order in the set of documents, for a fixed query.

2. Vector space model

In the vector space model text is represented by a vector of terms [3]. The definition of a term is not inherent in the model, but terms are typically words and phrases. If words are chosen as terms, then every word in the vocabulary becomes an independent dimension in a very high dimensional vector space. Any text can then be represented by a vector in this high dimensional space. If a term belongs to a text, it gets a non-zero value in the text-vector along the dimension corresponding to the term. Since any text contains a limited set of terms (the vocabulary can be millions of terms), most text vectors are very sparse. Most vector based systems operate in the positive quadrant of the vector space, i.e., no term is assigned a negative value [4].

Following the presented previous notation, we define the vector space model as below:

- D: documents are represented by a vector of words or index terms occurring in the document. Each term in the document – or, for that matter, each pair (t_i, d_j) – has a positive, non-binary associated weight $w_{i,j}$.
- Q: queries are represented as a vector of words or index terms occurring in the query. Each term in the query– or, for that matter, each pair (t_i, q) – has a positive, non-binary associated weight $w_{i,q}$.
- F is an algebraic model over vectors in a t-dimensional space.
- sim estimates the degree of similarity of a document d_j to a query q as the correlation between the vectors d_j and q. This correlation can be quantified, for instance, by the cosine of the angle between the two vectors:

$$\circ \quad \mathrm{sim}(\vec{q}, \vec{d}_j) = \cos(\vec{q}, \vec{d}_j) = \frac{\vec{q} \cdot \vec{d_j}}{|\vec{q}| \times |\vec{d_j}|} = \frac{\sum_{i=1}^{t} w_{i,q} \times w_{i,j}}{\sqrt{\sum_{i=1}^{t} {w_{i,q}}^2} \times \sqrt{\sum_{i=1}^{t} {w_{i,j}}^2}}$$

Since $w_{i,j} > 0$ and $w_{i,q} > 0$, sim(q, d_j) varies from 0 to 1, i.e. 1.0 for identical vectors and 0.0 for orthogonal vectors. Thus, instead of attempting to predict whether a document is relevant or not, the vector model ranks the documents according to their degree of similarity to the query.

3. Implementing the index-based searching

In our work we were granted access to business documents at Knauf. To be able to evaluate text mining techniques we implement a test framework based on Lucene[1], which is a high-performance, fully-featured text search engine library written entirely in Java. It is an open source search engine API released by the Apache Software Foundation. According to [2] the vector space model is based on assumption that the same terms occur

[1] http://lucene.apache.org/core/

in both the query and the relevant document. The Lucene API offers a simple way to extract the term-documented frequency matrices for each indexed field. Before we go further, we start this section by explaining the meaning of an index. An index is identical to an index at the back of the book, where we can find the search terms and their corresponding pages in a book. Similarly, when we create an index based on documents, we can query the index to discover what documents fit our search terms. We will both create an index and make searches against the index. However, these are conceptually two different tasks. The structure of the demonstrated project is shown here. We have a directory called "fajlla_ne_Index" that is composed of text files that we are aiming to index. We also have a directory called "Skedari_Index". This will contain the index that we create. Our project is using the default lucene-core jar file with the class named LuceneKnauf.

We are going to index two text files in the "fajlla_ne_Index" directory. Within the first one, gypsum1.txt, there is information about gypsum material. This file contains these words: gur, gips, miniera, qymyr, kristal. The second text file, gypsum2.txt, contains some other gypsum related material. It contains the following words: suva, sediment, shkemb, miniera, renaturim.

Firstly, we have to create an index through its method *createIndex()*. We also make use of an *IndexWriter* object to create and update the index whereby a directory name can be passed to the *IndexWriter* constructor. *IndexWriter* has several constructors. We used a constructor that takes three arguments. The first argument represents the location of the directory where the index files should appear. The second argument is an *AnalizuesiStandard* object. An analyzer symbolizes the principles or the policy for extracting index terms from text. The third argument *rikrijoIndexNeseEkziston* is a boolean parameter set to be true, which notifies the IndexWriter to rebuild the index from the beginning if it already exists.

```
public class LuceneKnauf {

        public static final String FILES_TO_INDEX_DIRECTORY = "fajlla_ne_Index";
        public static final String INDEX_DIRECTORY = "Skedari_Index";

        public static final String FIELD_PATH = "path";
        public static final String FIELD_CONTENTS = "contents";

        public static void main(String[] args) throws Exception {

                createIndex();
                searchIndex("qymyr");
                searchIndex("miniera");
                searchIndex("miniera AND renaturim");
                searchIndex("miniera and renaturim");
                searchIndex("kalcium OR renaturim");

        }
        public static void createIndex() throws CorruptIndexException, LockObtainFailedException, IOException {
                Analyzer analyzer = new AnalizuesiStandard();
                boolean rikrijoIndexNeseEkziston = true;
                IndexWriter indexWriter = new IndexWriter(INDEX_DIRECTORY, analyzer,
rikrijoIndexNeseEkziston);
                File dir = new File(FILES_TO_INDEX_DIRECTORY);
                File[] files = dir.listFiles();
                for (File file : files) {
                        Document document = new Document();
                        String path = file.getCanonicalPath();
                        document.add(new Field(FIELD_PATH, path, Field.Store.YES,
Field.Index.UN_TOKENIZED));
                        Reader reader = new FileReader(file);
                        document.add(new Field(FIELD_CONTENTS, reader));
                        indexWriter.addDocument(document);
                }
                indexWriter.optimize();
                indexWriter.close();
        }
public static void searchIndex(String kerkoString) throws IOException, ParseException {
                System.out.println("Kerkimi per '" + kerkoString + "'");
```

```
                    Directory directory = FSDirectory.getDirectory(INDEX_DIRECTORY);
                    IndexReader indexReader = IndexReader.open(directory);
                    IndexSearcher indexSearcher = new IndexSearcher(indexReader);
                    Analyzer analyzer = new AnalizuesiStandard();
                    QueryParser queryParser = new QueryParser(FIELD_CONTENTS, analyzer);
                    Query query = queryParser.parse(kerkoString);
                    Hits hits = indexSearcher.search(query);
                    System.out.println("Eshte gjetur: " + hits.length() + "here");
                    Iterator<Hit> it = hits.iterator();
                    while (it.hasNext()) {
                              Hit hit = it.next();
                              Document document = hit.getDocument();
                              String path = document.get(FIELD_PATH);
                              System.out.println("Lokacioni: " + path);
                    }
}
```

Further, we go via the files in the "fajlla_ne_Index" directory. For all particular files, we create a Lucene document object, which is made of a set of fields that can be the content, metadata, and other data related to the actual document. We then create two fields and accompany them to the document. The first field is used to save and copy the canonical path to each text file in the index. We specify to copy it in the index through the Field.Store.YES argument. We further define and make sure to not let the Analyzer tokenize the path through the Field.Index.UN_TOKENIZED. This is why the path stays whole and integral and doesn't get fragmentized up by the Analyzer in the index. The second field represents the contents of the file. The contents get tokenized and indexed, but they will not be stored in the index. The reason for this to happen is because as an argument we used a Reader to the Field constructor. If we'd like to save the all contents in the index, we need to use a String instead of Reader. The process adding all of the documents or file attachments to the index is performed through the method addDocument(). Afterwards, the index is then optimized and then closed. Now, our index has been created and now we can search it. But for searching to take place, we need to take our kerkoString and then create a query object. In order to create the query, we create a QueryParser, which will specify that the target of searching will be within the contents field. We use a AnalizuesiStandard to analyze the text in kerkoString for search terms. We get the Query object by calling the parse() method on our QueryParser object with the kerkoString as an argument. Now, after completing the aforementioned operations, we can perform our search. In order to start searching we need the IndexSearcher's method search(). We saw that in the IndexSearcher the Query object is the argument. The searching results are associated with the Hits object. After performing several iterations over the Hits, as an output we get each individual Hit object. Along with the number of text occurrence, we obtain the path field from the document showing the location of that file. We performed several searches using the following queries: "qymyr", "miniera", "miniera AND renaturim", "miniera and renaturim", and "kalcium OR renaturim". The console output from executing LuceneKnauf is the following:

```
Kerkimi per 'qymyr'
Eshte gjetur: 2
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum1.txt
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum2.txt
Kerkimi per 'miniera'
Eshte gjetur: 1
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum2.txt
Kerkimi per 'miniera AND renaturim'
Eshte gjetur: 0
Kerkimi per 'miniera and renaturim'
Eshte gjetur: 2
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum1.txt
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum2.txt\
Kerkimi per 'kalcium OR renaturim'
Eshte gjetur: 1
Hit:
C:\projekte\workspace\demo\fajlla_ne_Index\gypsum1.txt
```

According to the search results, we both have "qymyr" listed but only we have "miniera". Additionally, notice the use of "AND" versus "and". The uppercase "AND" results in a boolean query, which requires "miniera" and "renaturim" to both be in a document in order for it to be a hit. The lowercase "and" is treated as an irrelevant word, so "miniera and renaturim" is like Kerkimi per "miniera renaturim", which returns 2 hits since "miniera" is found in one text file and "renaturim" is found in the other text file. An examination of the STOP_WORDS of AnalizuesiStandard shows that it uses that StopAnalyzer class' ENGLISH_STOP_WORDS. As you can see below, "and" is listed as one of the stop words, which explains why it was ignored in the "miniera and renaturim" query.

```
public static final String[] ENGLISH_STOP_WORDS = {
    "a", "an", "and", "are", "as", "at", "be", "but", "by",
    "for", "if", "in", "into", "is", "it",
    "no", "not", "of", "on", "or", "such",
    "that", "the", "their", "then", "there", "these",
    "they", "this", "to", "was", "will", "with"
  };
```

When we examined our project after LuceneKnauf has executed, we can see that index files have been created in the "Skedari_Index" directory.

4. Semantic information retrieval

Recently, ontologies have been used in Information Retrieval to improve recall and precision [5]. Its principal use is related to query expansion, which consists in looking for the terms in the ontology more related to the query terms, to use them as a part of the query. Much ontology has been designed for the purposes of managing and extracting semantic knowledge from online literature and databases.

IR systems that use semantic technologies for enhancing different parts of IR are called semantic search systems. Searching for the online ontologies, fact extraction from the ontologies, question answering, filtering and ranking retrieved information are usually put under the wing of semantic search. The introduction of ontologies to move beyond the capabilities of current search technologies has been an often portrayed scenario in the area of semantic-based technologies since the late nineties [6].

Based on literature review, below we provide the categories of semantic search engines. [7,8] [15,16].

Fig 1. Classification of semantic search systems

In spite of the fact that that these concepts have shown several enhancements compared to classic keyword searching methodology, it is not clear among researchers that these techniques could be suitable to deal with large scale information sources.

5. Integrating OLAP to enhance the searching functionality

The approach introduced in this paper is based on the principle of a Web-based portal system, incorporating three information sources: an OLAP system for accessing the data warehouse, a document management system (DMS) to make use of the unstructured data in form of documents, and native portal content such as business documents. The main reason for the usage of the OLAP system, with its own individual metadata set, is to achieve the process of metadata integration. For this purpose we also propose to enhance the standard data warehouse ETL (extract, transform, and load). As regards the DMS, however, we presume that the system supports RDF-based metadata storage. For the system components to understand and communicate with each other, we propose a global ontology to be used, which will furnish the needed information for mapping between different metadata constructs.

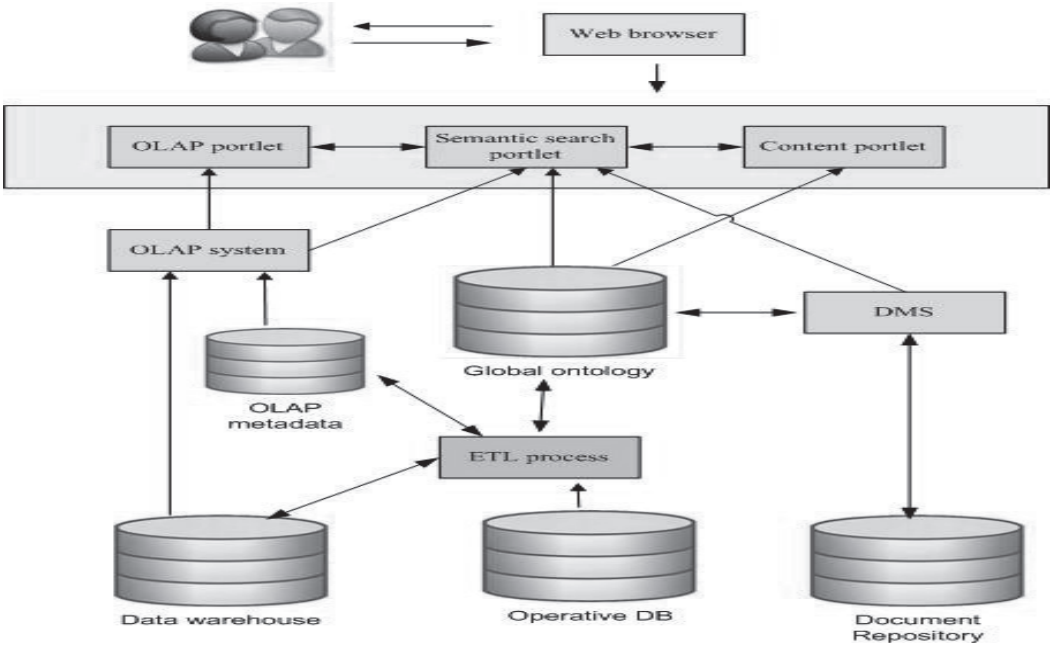

Fig 2. The architecture of the proposed system

Figure 2 shows the global architecture of the proposed system. The enhanced ETL process supplies the data warehouse and refreshes the ontology by keeping it updated. The user interface can be represented by many different portal components. For this reason, we propose some pluggable software components so-called portlets to be responsible for the user interface. Portlets can provide the same functionality as an ordinary web application, but they run inside a portal server, which means that they can be arranged on different portal pages and access to them can be controlled by the server. They can be built by any web framework such as Java Server Faces, Apache Wicket and so on. We assume that the proposed system is made of three such portlets: semantic search portlet, OLAP portlet and content portlet. Firstly, the semantic search portlet here represents the search user interface and the primary role of it is to query the metadata. Secondly, the OLAP portlet includes the interface with common OLAP operations, such as slice and dice, drill down, roll up and pivot. Thirdly, the content portlet is proposed to be used for displaying all other contents related to the native portal. For example, as content can be the news article.

The OLAP portlet can access the OLAP metadata through the OLAP system. On the other hand, the primary role of DMS will be for controlling the access to the document set; it can have an RDF compatible interface. The global ontology will supply the DMS with metadata.

The heart of the architecture is the global ontology. The overall data model, the resource metadata and the mapping information can be included here. Moreover, the global ontology can include instances for business objects, such as the customers, products, time, location, and so on, which are of great importance during the document searching. These business objects and the metadata for OLAP reports can be generated also during the ETL process. They also can serve as data model to represent the OLAP dimensions. When a user will perform navigation throughout the OLAP report, we can track the user context information which can be used for searching other relevant documents.

6. Conclusions

The main objective of this paper was to present the idea of incorporating index searching as part of a standard information retrieval system. We provided our concepts and they were tested by using real business documents.

Besides this, we described and proposed a novel architecture which is including an ontology-based approach by integrating OLAP and information extraction attributes to access structured and unstructured data, mainly organized in form of documents. We propose the OLAP integration because its tools allow a breakdown structure of the data where we start with a single piece of data and we can dissect it into a series of levels looking at the data for something interesting.

As a result, in our first demonstration, the query performance was reduced as more documents were added to the index, and consequently the growth factor becomes very low. While at the second case, when a user will perform navigation throughout the OLAP report, due to OLAP breakdown structure, it is possible to track the user context information which can be used for searching other relevant documents.

7. References

M. C. Daconta, L. J. Obrst, K. T. Smith, The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management, John Wiley & Sons, 2003

Baeza-Yates, R. & Ribeiro-Neto, B. (1999), Modern Information Retrieval. Addison-Wesley. ISBN 0-201-39829-X

D'Aquin, M., Gridinoc, L., Sabou, M., Angeletou, S., & Motta, E. (2007). Characterizing Knowledge on the Semantic Web with Watson. 5th International EON Workshop at International Semantic Web Conference (ISWC'07). Busan, Korea
D'Aquin, M., Gridinoc, L., Sabou, M., Angeletou, S., & Motta, E. (2007). Characterizing Knowledge on the Semantic Web with Watson. 5th International EON Workshop at International Semantic Web Conference (ISWC'07). Busan, Korea
T. Andreasen, J. Nilsson, and H. Thomsen.Ontology-based querying. In Proceedings of the Fourth International Conference on Flexible Query-Answering Systems, pages 15–26, Warsaw, Poland, Agosto 2000.
Luke, S., Spector, L., & Rager, D. (1996). Ontology-Based Knowledge Discovery on the World-Wide Web. Internet-Based Information Systems: Papers from the AAAI Workshop. AAAI, (pp. 96-102). Menlo Park, California.
Mangold, C. (2007) "A survey and classification of semantic search approaches", Int. J. Metadata, Semantics and Ontology, Vol. 2, No. 1, pp.23–34
Esmaili, K.S. and Abolhassani, H. (2006) "A Categorization Scheme for Semantic Web Search Engines", In Proc. of IEEE Conf. on Computer Systems and Applications, pp 171-178.