Fulltext indexing and querying with a support of relational database

Rastislav Lencses

Pavol Jozef Šafárik University, Košice, Slovak Republic
lencses@ics.upjs.sk

Abstract. We present an approach to fulltext indexing and querying over document collection which uses vector model. The document index represented by an inverted list stored in a relational database. Furthermore, we design and implement a tool JdbSearch which allows document indexing and query over documents based on our theoretical models.

1 Introduction and motivation

Fulltext indexing and querying is an typical problem solved in the area of Information Retrieval (IR). Usually, the information is stored and represented in the form of documents. User's query is evaluated and returned as a set of relevant documents. There are several possibilities how to deal and represent the documents and user's query and how to calculate the document relevance. The two most used theoretical models are boolean model and vector model (the latter in many variations), see [1].

Also, there are various approaches as how to store the document index. For example, the Apache Lucene [5] project stores its data in the self-mainained file in the filesystem. However, a relational database systems (which are nowadays the single most used data storage method) seem to have many advantages. Therefore we can consider them as a primary storage for our document index.

2 Methods Used

2.1 Document Index Storage

In order to have an efficient and fast access to the document index, a concept of *inverted list* is often used. This approach has the least time and space requirements amongts the other possibilities.

We can describe the inverted list as a simple array of terms which can appear in the collection of documents. Moreover, each term in this list is associated with an extra list of documents, in which the given term appears (see Figure 1).

Now we can propose the data model for our representation of the index in the relational database. There are three fundamental tables used, see Figure 2. The table *Doc* represents the concept of document. It contains an document identifier, name, actual document data – usually as an CLOB data – and an a norm of the

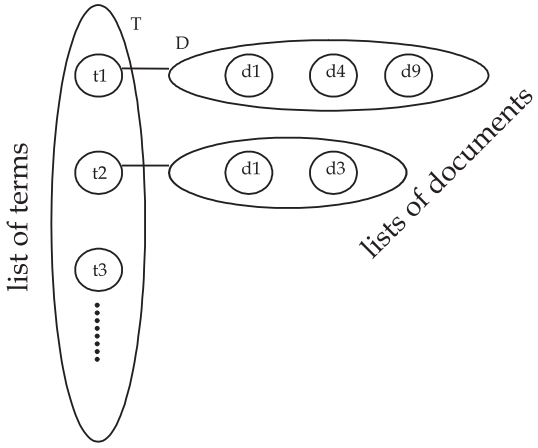

Fig. 1. Overview of an inverted list.

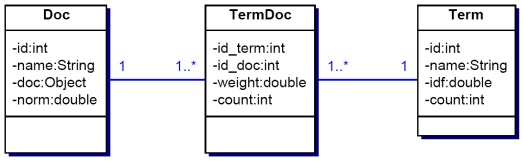

Fig. 2. Class Diagram representing data model.

vector, which represents this document. The table *Term* accomodates the term data – inverse document frequency, document frequency and the total count of occurence of the particular term in the whole collection. The third table, *TermDoc* connects the two aforementioned tables. It holds the identifiers of all terms in the given document with number and weight of term *id_term* in the document *id_doc*.

The process of indexation then fills the inverted list – and therefore the database tables.

3 Document Weight Calculation

We have mentioned various document weights. For our purpose we will calculate the weight using the inversion document frequency formula.

$$\mathrm{idf}_j = \log\left(\frac{n}{\mathrm{df}_j}\right),$$

where df_j is the number of documents which contain the term t_j.

Then we can define a matrix A with dimensions $m \times n$ (m is the number of terms, n the number of documents), with each item defined as

$$w_{ij} = \mathrm{tf}_{ij} \cdot \mathrm{idf}_j \,. \tag{1}$$

In this formula, the tf_{ij} denotes the number of occurences of term t_j in the document D_i. We can consider the w_{ij} as the weight of the term t_j in the document D_i.

4 Querying

User's query is typically a boolean question of two types: a *Any* query has terms delimited by *OR* conjunction, whereas a *All* query uses the *AND* conjunction.

The relational database, used as a backend for document index can be accessed by the SQL queries. However, we can fairly easily represent the user's queries in the SQL.

The *Any*-query can be mapped on the following SQL query.

```
SELECT distinct(TD.id_doc)
FROM Term
JOIN Term_Doc td ON Term.id=TD.id_term
WHERE Term.name='term1' OR Term.name='term2'
```

The *All*-query is, however, slightly more complicated. The analogue to the previous query would be very slow (due to number of joins equal to the number of terms in the boolean condition). However, we can use the following query.

```
SELECT TD.id_doc
FROM Query Q
JOIN Term_Doc TD ON Q.id = TD.id_term
WHERE GROUP BY TD.id_doc
HAVING COUNT(DISTINCT(TD.id_term))
  = (SELECT COUNT(*) FROM Query)
```

This query is based on an extra table *Query* which contains the identifiers of the query terms.

The use of the vector model has one huge advantage to the classical boolean model – it allows us to calculate the relevance of found documents. This can be computed also by using standard SQL resources, i. e. by following query:

```
SELECT TD.id_doc,
  SUM(TD.weight * Q.weight)
    /(Doc.norm*QNorm),
FROM Query Q
JOIN TermDoc TD oN Q.id = TD.id_term
JOIN Doc ON Doc.id = TD.id_doc
GROUP BY TD.id_doc
ORDER BY 2 DESC
```

In this query we use some additional tables: the *Query* contains terms of the query and their corresponding weight. The *DocQuery* holds documents, which satisfy the boolean condition, and their norm. The *QNorm* denotes the norm of the query vector.

5 Implementation

The methods described in the previous chapters were implemented in the form of a tool used in the project NAZOU[1].

The tool consists of two logical components: the *indexing component* and the *querying component*. Both components were implemented in Java and tested on the MySQL Relational Database.

The former component takes for an input a collection of documents (more precisely documents, which represent job offers) and creates the inverted list, which is stored in the aforementioned data structure.

The indexing is done in two phases. The very indexing part is done in-memory. However, our test have shown this process is very memory-intensive. Therefore we have implemented a procedure, in which the inverted list is periodically flushed to a disk file. When the indexing process is completed, the data from this disk file are directly loaded into the appropriate database tables.

The latter component reads from the input the query – which consisted of the list of words (query terms) and the query type (either *All* or *Any*). This specifies the conjunction which separates the query terms. The tool then retrieves documents, which fulfill the query, along with the degree of relevance. Furthermore, the tool is able to search for documents, which are similar to a given document. This can be achieved simply by transforming the input document into the query.

6 Experiments

We performed various experiments in the area of quality and efficiency of our tool. The figure 4 shows ration of precision of recall, which is a standard metric used to measure the quality of information retrieval systems. We have experimented with several weight functions, which has shown, that the best combination can be achieved by using weight functions from the formula (1).

The speed of our indexing algorithm with database support was compared to a ,,classical“ algorithm mentioned in [2]. The results are shown in the figure 5, and it displays, that our algorithm (denoted as Index(D, M)) requires less time to index a document collection.

[1] NAZOU = "Tools for acquisition, organization and maintenance of knowledge in an environment of heterogeneous information resources", homepage: http://nazou.fiit.stuba.sk/.

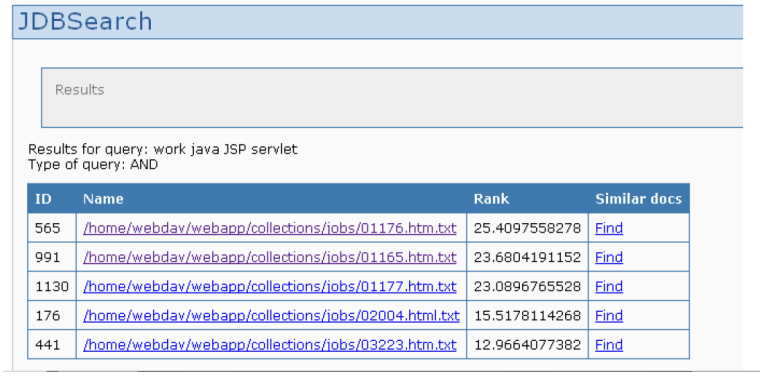

Fig. 3. Screenshot of the tool.

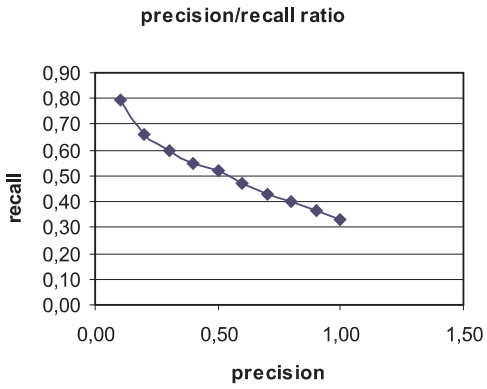

Fig. 4. Ratio of precision and recall.

7 Conclusion

We have formally designed and implemented a system for information retrieval based on the vector model, which leverages the use of the relational database.

The future work can be directed towards distributed and parallel indexing, thus speeding up the process.

References

1. Baeza-Yates, R., Ribeiro-Neto, B. A., Modern Information Retrieval. ACM Press / Addison-Wesley, 1999. ISBN 0-201-39829-X.
2. Grossman, D. A., Frieder, O., Information Retrieval: Algorithms and Heuristics, Kluwer Acamedic Publishers, 2000. ISBN 1-4020-3004-5

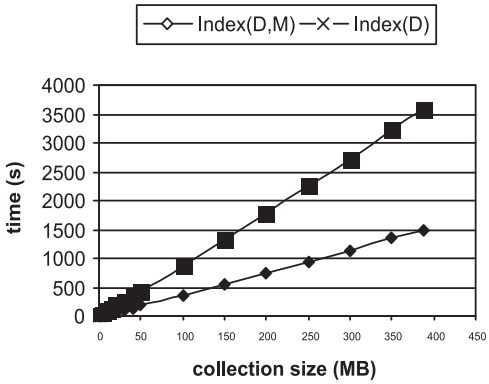

Fig. 5. Comparison between our and ,,classical“ algorithm

3. Lencses, R., Indexing for Information Retrieval System supported with Relational Database, Conference Sofsem 2005, Slovakia, January 2005, Proc. Vojtáš et al. (ed.): Sofsem 2005 Communications, Bratislava 2004, 81-90
4. Lencses, R., Dopytovanie v systéme zameranom na získavanie informácií s podporou relačnej databázy, Proceedings of Datakon 04, Brno, Czech Republic, Masaryk University, Brno, 2004, p. 271-280
5. Apache Lucene. A high-performance, full-featured Java text search engine. [online] URL: ⟨`http://lucene.apache.org`.⟩