Indexing for Information Retrieval System supported with Relational Database [⋆]

Rastislav Lencses

Ústav informatiky, Prírodovedná fakulta,
Univerzita P. J. Šafárika v Košiciach,
Jesenná 5, 040 01, Košice,
Slovenská republika,
lencses@science.upjs.sk

Abstract. We deal with indexing of document collection for information retrieval engine which use relational database as data store. We will present an algorithm for index generation, compare it with other algorithms in this area and estimate its time complexity. We created a distributed version of this algorithm and discuss about its effectiveness.

Keywords: information retrieval, relation database, indexing, inverted list, distributed computing

1 Introduction

An employing relational database model supported with SQL for needs of information retrieval can be found in early works written in the time of first commercial relational database : [8] - Macleod in the year 1978. More recently contribution is [7], for example. First step in these works is to set up (at least) three relational tables (fig. 1). These tables are used to information retrieval by SQL.

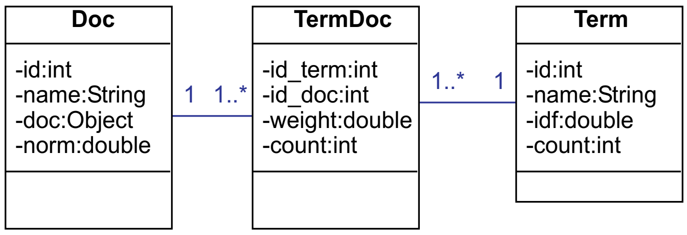


Fig. 1. Database relations used in information retrieval

The table *doc* contains name of the document, its content and norm of the vector, which represents this document. The table *term* contains name of the

[⋆] This research was supported by the grant VEGA 1/0385/03 term, its inverse document frequency, a document frequency and total count of this term in whole collection. Finally, *termDoc* contains foreign keys *id_ doc* and *id_ term*, number and weight of term *id_ term* in document *id_ doc*.

We will define definitions used in information retrieval in the second chapter and an inverted list in the third chapter. Generation of the inverted list is described in the fourth chapter, distributed version in the fifth chapter. Sixth chapter deals with time complexity of sequential algorithm and effectiveness of distributed algorithm.

2 Definitions

Information retrieval engine (IRS, fulltext engine) implements two basic operations - a creation of index over document collection and querying this collection with support of index to get documents relevant to user query. Documents have no structure, they contains words (terms).

Collection of documents is progression D_i

$$D_i, i \in [1, n]$$

where n is number of documents. Every item of progression D_i is progression

$$s_k^i, k \in [1, m_{D_i}]$$

where m_{D_i} is number of words in the document D_i. Terms from all documents are in progression T

$$T = t_j, j \in [1, m_T]$$

where m_T is number of unique terms in all documents. T can be also defined with help of set S_T

$$S_T = \bigcup_{i=1}^{n} \bigcup_{k=1}^{m_{D_i}} \{s_k^i\}$$

Items of S_T are just items of T (ordered lexicographically, for example).

There are several theoretical models used to design document representation and define function, which assign relevant documents to user query. Vector space model ([11]) defines documents as vectors in the space with dimension equals to number of unique terms (m_T). Every item of this vector contains the weight which expresses power of term contribution to the document. We define inversion document frequency first

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

where df_j is number of documents (document frequency) containing term t_j. We define (sparse) matrix A with dimensions $m_T \times n$, with items defined as follows

$$w_{ij} = \mathrm{tf}_{ij} \ . \ \mathrm{idf}_j$$

where tf_{ij} is number of occurrences (term frequency) of term t_j in document D_i.

We can see, that w_{ij} is weight of term t_j in document d_i.

3 Index

Documents relevant to user query can be found by searching full text directly. There are several algorithms capable of search efficiently (Knuth-Moris-Pratt algorithm, Boyer-Moore algorithm and the others [1]). This approach is suitable only for a small document collection. Information retrieval engine need to create external data structure (index) to speed up searching.

There are three types of typically used indexes: inverted list, signature and suffix trees. Inverted list have the least time and space complexity ([1]).

Inverted list contains list of terms and for every term records occurrences this term in documents. Inverted list can be binary (existence of term occurrence in document is recorded), extended (number of term occurrences in document is recorded) and full (position of term occurrence in document is recorded). In the next we will use extended inverted list.

More formally, inverted list I is

$$(t_j, Z_j), j \in [1, m_T]$$

For extended inverted list we can define Z_j as follows

$$Z_j = (i, k_l^i), l \in [1, m_{D_i}]\}$$

where k_l^i is number of term occurrences t_j in document D_i.

We will only record basic morphemes in inverted list. The semantics will decrease only a little, but length of inverted list will decreases significantly. We used free open-source program *ispell* [4] to transform words of documents to basic morphemes. In contrast to English language, where the Porter algorithm is typically used, *ispell* use dictionary (Slovak language is morphematically much more rich than English).

4 Generation of Inverted List

We can create inverted list straightforward in the memory with sorted array of terms. This array contains unique terms from whole document collection (progression t_j). For every term we record array of its occurrences in documents (progression Z_j). Occurrences of the term are in the (sorted) array, where is recorded occurrence of the term in all documents. Each term of document is binary searched in the sorted array of terms during the process of document indexation. In the case the term exists in the sorted array, we search occurrences and add/update occurrence of the term. If the term was not found, it is added to sorted array of terms and occurrence of the term is recorded for this term.

This basic algorithm will fail in the case of larger document collection where inverted list does not fit in the primary (operational, main) memory. As the operation system is not able swap/page memory effectively without good knowledge of the algorithm, it is necessary to use the persistence (in the secondary memory - disk) directed by the algorithm.

In [1] is mentioned an algorithm, which build inverted list with algorithm described above until the primary memory is exhausted. When no memory is available, the partial inverted list (with occurrences) is written to the secondary memory and the primary memory is erased. This process continues until whole collection is proceed. Finally, we have a number of partial inverted lists. These lists can be merged hierarchically. The merging of two inverted lists consists of merging the sorted arrays of terms and if same term appears in both arrays, merging occurrences from both arrays of terms. The phase of merging spent 20-30% of the overall time.

Another algorithm ([2]) was designed directly for Information Retrieval System supported with a relational database. The algorithm builds inverted list for every document. Inverted list is transformed into INSERT commands into table *termDoc* and one INSERT into table *doc* (see fig. 1). After document processing is inverted list erased, so there is no lack of memory. Inverted list of unique terms is generated in table *term* with help of special SQL command. It is also necessary to compute term frequency and inverse document frequency.

We can use ideas from [2] and [1] to develop original algorithm with the name Index(D, M). The algorithm in [2] use relational database and do not use available primary memory, we can name it internally as Index(D). The algorithm in [1] do not use database and use available memory, so we can name it as Index(M). Algorithm Index(M) must merge inverted lists (and Index(D, M) does not). Algorithm Index(M) must compute global weights and table *term* (and Index(D, M) does not). The process of writing of term occurrences to table consume significantly more time in algorithm Index(D) than Index(D, M). The reason is that writing of term name takes more time than writing of simple number identifier of the term. [1]

We analyze structure of inverted list. This structure consists of list of terms and every term is linked with list of term occurrences. The list of terms alone is not very large and it grows sublinearly (according to Heaps law - [3]). Basically, it is constant for the used language (there is a limited number of words in every language). In contrast, lists of occurrences grow linearly (and depended on size of collection).

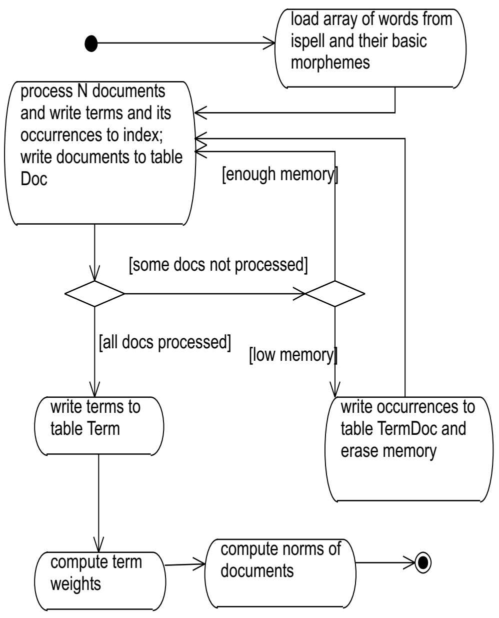
At the beginning of our algorithm Index(D, M) we load to primary memory sorted list of every known basic and derived word morphemes and known by *ispell*. We are able to determine basic morpheme of the term parsed from document with help of this list, if it contained in this list. We use basic algorithm until primary memory is exhausted. Then we transmit term occurrences to the database and erase it from primary memory. It is important, that list of terms is in memory all the time. At the end we load list of terms to the database, compute weights of terms and norms of documents. Weights can be computed with SQL command

[1] Algorithm Index(D) had in tests about 20% speed penalty comparing with algorithm Index(D,M) especially due to writing of term occurrences to table *termDoc*

```
INSERT into TermDoc (id_term, id_doc, weight, count)
SELECT TD.id_term, TD.id_doc, TD.count*Term.idf, TD.count
from Term, Temp TD
where TD.id_term=Term.id
```

where table *Temp* consists of columns *id_ term*, *id_ doc* and *count* and it contains data from inverted list (which was transmitted to the database with *LOAD* command). Norms can be computed with similar SQL command.

We separate terms and term occurrences, so we do not merge partial inverted lists as Index(M). We record term frequency and inverse document frequency during the running of algorithm, so we do not compute it as Index(D). Algorithm Index(D, M) is in fig. 2. Parts introduced in basic algorithm are leaved out.



load array of words from ispell and their basic morphemes
process N documents and write terms and its occurrences to index; write documents to table Doc
[enough memory]
[some docs not processed]
[all docs processed]
[low memory]
write terms to table Term
write occurrences to table TermDoc and erase memory
compute term weights
compute norms of documents

Fig. 2. Generation of inverted list - algorithm Index(D, M)

5 Distributed Indexing

We can speed up computation of inverted list by distribution of indexing to more computers. The main idea consists of dividing of computing to four phases. Two of them can be distributed. We used SIMD architecture (see [12]). Most of the time is taken by computation, sending data over network take negligible time. We will refer our algorithm by name *DIndex(D, M).*

Phase A concerns with document parsing. Document collection is divided regularly amongst q computers. Computer i $(1 \leq i \leq q)$ creates table Doc_i, $termDoc_i$ and $term_i$. Table $term_i$ does not contain *id* as key identifier. The key is just the name of the term (*id* is not useful as the table *term* is distributed). At the end of the phase tables $term_i$ will be sent to the central computer.

In phase B partial tables $term_i$ are joined together in central computer. It can be done with SQL command

```
INSERT into Term (name, df, idf)                      (1)
select name, sum(df), log(n/sum(df))
from
select name, df from Term1 union all
select name, df from Term2 union all ...
group by name
```

where n is number of documents. Primary key of table *Term* will have *auto_ increment* type.

Phase C begins with sending of table *Term* to every computer. Term weights are computed by each computer with following SQL command

```
INSERT into TermDoci (id_term, id_doc, weight, count)
SELECT TD.id_term, TD.id_doc, TD.count*Term.idf, TD.count
from Term, Tempi TD
where TD.name=Term.name
```

Norms of documents can be also computed in phase C(every computer has necessary data).

Last phase D consists of joining tables *termDoc* and *doc* in central computer similar to phase D. This phase is not necessary in the case of distributed querying, which can speed up simple querying. In this case documents are distributed amongst computers, term weights are computed consistently.

6 Test Results of Indexing

We used collection of news articles of Slovak daily SME. This collection has size 53 MB, time of indexing was about 10 minutes. Computer configuration consists of processor 1,2 GHz and 512 MB of primary memory. We used database MySQL.

We carried out the tests also with the bigger collection LATimes [5] (English news articles published in Los Angeles Times, 130000 documents, 390 MB). The whole process of indexing took about 60 minutes.

6.1 Estimation of Time Complexity of Indexing

We determine relation between collection size n_{MB} (in megabytes, for example) and number of all (non-unique) terms in whole collection. Such terms are in progression $s^i_k, k \in [1, m_{D_i}]$, we define m_K as a number of these terms. Our tests show linear relation between n_{MB} and m_K.

Relation between number of terms m_K and number of unique terms m_T for specified collection can be expressed exponentially according to Heaps law [3]

$$m_T = O((m_K)^\beta)$$

where $0 < \beta < 1$ is constant depended on collection. The value of this constant (obtained experimentally) is $\beta_{SME} = 0.77$ for the collection SME and $\beta_{LATimes} = 0.7$ for the collection LATimes. Since m_K linearly depended on n_{MB}, we can write

$$m_T = O((m_{MB})^\beta)$$

Algorithm Index(D, M) consists of several phases:

– A: Create/load ispell dictionary
– B: Parse documents
– C: Write terms to database
– D: Write term occurrences to database
– E: Compute term weights
– F: Compute document norms

Phase A take constant time, we will refer it as t_A.

Phase B depends on number of terms m_K (all must be processed). The algorithm search in sorted array of morphemes (from *ispell* dictionary) insert terms into sorted array of unique terms (progression $t_j, j \in [1, m_T]$). Searching/inserting in sorted array have logarithmic time complexity. Since both arrays are in primary memory, speed of work with these array are much bigger the work with documents in the secondary memory. Moreover, array of morphemes have constant size, that means searching take constant time also. Next array of unique terms have much lesser size than array of all terms ($m_T << m_K$). Time complexity of this phase can be expressed

$$t_B = O(m_K \cdot \log(m_T)) \approx O(m_K) = O(m_{MB})$$

That means

$$t_B = k_B \cdot n_{MB}$$

Our experiments validate this equation. Value of constant k_B can be seen in fig. 3. This linear time complexity $O(n_{MB})$ is confirmed with other works too, for example [10].

Phases A and B were fully under our control and we can exactly estimate time complexity for bigger collections too. Phases C, D, E and F use database. Algorithms use by database can vary and studying them is not our goal (we used SQL commands only). Experimental results range from linear complexity $O(n_{MB})$ to complexity close to quadratic $O((n_{MB})^2)$. It depends on memory available for database cache.

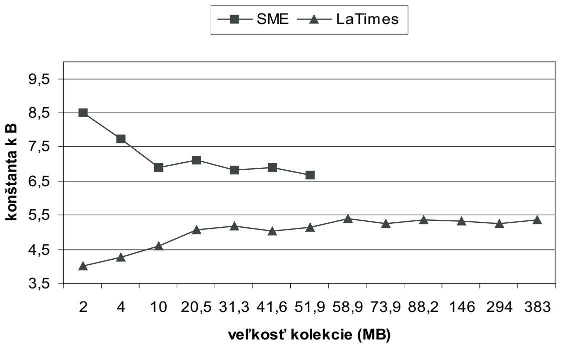


Fig. 3. Values of the constant k_B in dependence on collection size, for SME and LA-Times

6.2 Estimation of Time Complexity of Distributed Indexing

Speedup of computation can be measured by effectiveness, which can be computed as follows (see [12])

$$E_K(n, q) = \frac{S_K(n)}{T_K(n, q) \cdot q}$$

where $S_K(n)$ is time to solve task K with input of size n by best known sequential algorithm and $T_K(n, q)$ is time to solve task K with input of size n by q computers. If $E_K(n, k)$ equals to 1 then distributed algorithm is optimal distributed. If $E_K(n, k)$ equals to some constant $c < 1$ for every q then distributed algorithm is linearly speeded up.

Comparison of sequential algorithm against distributed algorithm can be seen in fig. 4 for collection SME. We can see effectiveness of computation, which descends with increasing number of computers. It causes non-distributed phases B and D of algorithm *DIndex(M, D)*.

Three main activities are executed in the phase A. First of them - parsing of documents - take about 70% of overall time of sequential algorithm *Index(D, M)* for our collection. [2] So we can find out a best acquisition of distributing in this activity. Parsing of documents has linear time complexity, so it will be speed up optimal, as we can see in fig. 5.

Second activity of phase A is loading term occurrences to database. This activity has linearly effectiveness, but not optimal (see fig. 5). The reason follows: table $docTerm_i$ contains string identifier of term (its name) instead of its numeric identifier. A string is longer than a number and creation of database index for

[2] This number will decrease with size of collection, since loading of term occurrences and weight computation have complexity close to quadratic for bigger collection. Parsing of collection LATimes take about 60% of overall time.

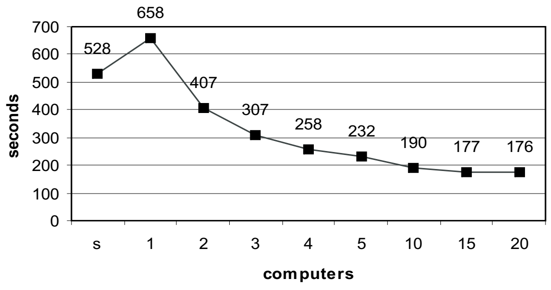


Fig. 4. Distributing of indexing, *s* means sequential algorithm

string is much more difficult. Third activity is generation of local table $term_i$. The effectiveness is linear (see fig. 5). We cannot speak about optimality, since this activity has no opposite in sequential algorithm. In other side, in sequential algorithm are activities, which have no opposite in distributed algorithm (loading unique terms to database, for example).

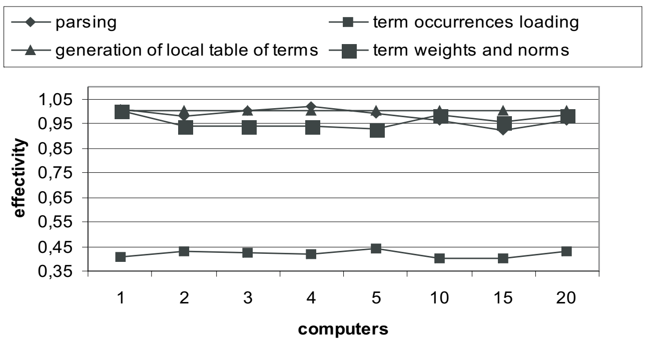


Fig. 5. Effectiveness of several activities of distributed indexing

Phase *B* concerns with generation of global table *term*. If number of computers increases, command (1) will be more complex (one table will be added for every computer). This command executes joining of tables while their whole size is constant (for specified collection). The more tables we need to join (even sum of their sizes is constant), more complicated SQL command will be. Syntactic parsing of this command, initial works for every table and other overhead cause linearly (and not constant) time complexity of this phase, as follows

$$t_G(q,n) = t^1{}_G(n) + k_G \cdot q$$

$t^1{}_G(n)$ is time to execute of phase B on one computer and $k_G \cdot q$ expresses time penalty for a lot of tables in SQL command (1). The value of the constant k_G equals to 0,5s in our configuration.

Phase C - computation of term weights and norms - has optimal linearly effectiveness (see fig. 5). Time complexity of phase D (joining of tables $termDoc_i$ and doc_i) is similar to phase B and can be similar expressed

$$t_J(q,n) = t^1{}_J(n) + k_J \cdot q$$

7 Conclusion

Both indexing and distributive indexing of document collection were successfully implemented in Java, we choose database MySQL (in principle any database compliance SQL99 and supported Java can be used). Time complexity of generation of inverted list is linear. Time complexity of loading of this list to the database and computing of the term weights is between linear and quadratic complexity, which depends on size of available memory and used database. Distributed indexing enables speed up indexing rapidly, although effectiveness is not optimal, therefore number of useful computers is limited.

References

1. *Baeza-Yates, R., Ribeiro-Neto, B.* : Modern information retrieval. Addison Wesley (1999), 197-198
2. *Grossman, D. A., Frieder, O.* : Information Retrieval: Algoritms and heuristics. Kluwer Academic Publishers (2000) 169-170
3. *Heaps, J.* : Information Retrieval - Computational and Theoretical aspects. Academic Press, 1978
4. http://spell.linux.sk
5. http://trec.nist.gov
6. http://jakarta.apache.org/lucene
7. *Lundquist, C.* : Relational information retrieval: Using Relevance Feedback and Parallelism to Improve Accuracy and Performance. PhD thesis, George Mason University (1997)
8. *Macleod, I.* : A relational approach to modular information retrieval systems design. In Proceedings of the ASIS Annual Meeting (1978) 83-85
9. http://www.ibm.com/notes
10. *Ribeiro-Neto, B., Moura, E. S., Neubert, M. S., Ziviani, N.* : Efficient distributed algorithms to buildinverted files, In 22th ACM Conf. on R&D in Information Retrieval, 1999
11. *Salton, G., Buckley, C.* : Term-weighting approaches in automatic text retrieval. Information Processing and Management 24, 513-523 (1988)
12. Tvrdlík, P.: Parallel Systems and Algorithms, ČVUT, 1997.