An Indexing Algorithm for Text Retrieval*

Maxim Martynov, Boris Novikov
University of St.-Petersburg, Russia
E-mail: mart@gamma.niimm.spb.su, boris@orl.usr.pu.ru

Abstract

The rapid growth of world-wide information systems results in new requirements for text indexing and retrieval. In this paper we propose an algorithm for query evaluation in text retrieval systems based on well-known inverted lists augmented with additional data structure and estimate expected performance gains. In addition to improved performance, this data structure is able to support dynamic indexing, which is especially important for environments where documents are changed frequently.

1 Introduction

The importance of text retrieval systems has grown dramatically during recent years due to very rapid increase of available storage capacity, increased performance of all types of processors and exponential growth of the global networks that provide an enormous source of different documents.

One of the factors that a critical for usability of text retrieval systems is the performance of search engine and underlying indexing techniques [5].

The process of query evaluation and indexing in high performance text retrieval systems usually consists of several steps. The following steps are typically included:

- query pre-processing, e.g. some kinds of linguistic normalization of words used in the query, extension of query with synonyms checking against thesaurus etc.
- launching of search engine based on pre-build indices
- post-processing of candidate documents to filter out the set of documents relevant to a query
- refinement of the query based on user's feedback, and re-evaluation of the query.

All these steps are extremely important for quality of the result, but in this paper we concentrate only on the second step of the above listed, namely, on indexing and search engine.

*This work was partially supported by Russian Foundation for Basic Research under grant 95-01-00636, and UrbanSoft Ltd. under contract 35/96.

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS'96). Moscow, September 10–13, 1996.
Moscow: MEPhI, 1996.

During past years two major classes of indices for text retrieval were proposed, investigated and extensively used: a technique based on inverted lists and superimposed coding scheme proposed in [4] and further enhanced by several authors, e.g. [6]. Both structures have certain scalability problems of different nature.

The inverted lists provide reasonably good performance for single-keys searches (logarithmic on the database size, which actually means few disk accesses per search), but their performance rapidly degrades when the query size increases, which is of major importance for text retrieval. Typical queries may contain several index terms. Moreover, initial user queries may be augmented with several additional terms (e.g. synonyms), making the query size significantly larger.

The performance of another extreme, superimposed coding, does not depend on the query size, but depends linearly on database size, which potentially implies hard scalability problems for huge text collections. In addition, this kind of indices inevitably returns significant portion of noise (that is, documents that are not relevant to the specified query), which implies expensive extra filtering on the subsequent phases of query evaluation.

Recent improvements of compressed inverted lists [8, 9] significantly enhanced their performance in terms of storage requirements (and hence access times), so that these indices outperform superimposed coding in most cases. In this paper we propose a variation of this scheme with improved performance especially for queries with large number of index terms.

The model of data structure for this indexing scheme is represented on fig. 1.

The dictionary contains all index terms, with the expected size about few megabytes [8]. The authors of [8] suggest to store it in the main memory. Although this is probably reasonable for English, the number of word forms to be stored in the dictionary may be significantly larger for other languages, e.g. Russian and Finnish [7]. However, we assume that at least significant part of the dictionary resides in the main memory.

The inverted lists are stored in compressed form on the disk. The compression algorithm allows one-pass compression and decompression, providing the possibility to process the lists "on the fly", without storing decompressed data at all. However, the technique described in [8] suggests only sequential scans of the lists, which may cause extra disk accesses during processing of large inverted lists.

We assume that documents (or paragraphs of documents) are sequentially numbered and are accessible via addressing table.

The query is represented (for the step of index lookup) as a set of index terms that should (or should not) appear in the same document (or same paragraph), and combinations of such conditions. The query evaluation consists of retrieval of all terms mentioned in the query from the dictionary, and then merge (intersection, union,

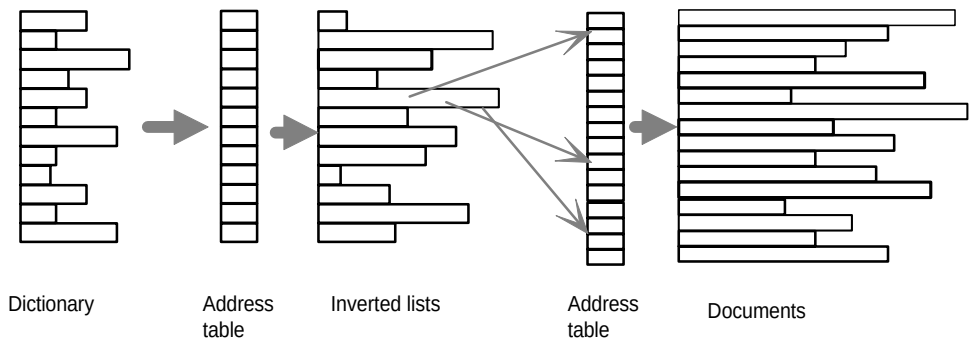

Figure 1: The general index structure

or difference) of corresponding inverted list, to obtain a candidate list of relevant documents.

The candidate list is passed to subsequent steps of query evaluation, which may involve document scans to check the conditions that cannot be checked with inverted lists only.

The most expensive (time-consuming) step of the query evaluation is the merge of inverted lists.

In this paper we introduce additional index structure which allows random access to inverted lists, thus decreasing the number of required disk accesses during the merge phase. In addition, this structure simplifies modification of the inverted list, which may be important for indexing of dynamic document collections. We further propose a merging algorithm that utilizes this additional data structure and estimate the expected performance improvement.

The paper is organized as follows. In section 2 index structure that supports random access to inverted lists is described. An algorithm for processing inverted lists is presented and discussed in section 3. Then in section 4 we estimate the performance improvement that can be achieved with our algorithm. Section 5 concludes the paper.

2 The Index for Inverted Lists

In this section we introduce an auxiliary data structure (actually a variation of B-tree) which is used for text retrieval in the algorithm described in the next section.

For every word (index Term) that occurs in a stored document, the inverted list contains an entry that includes a document number and optionally position information. For better search speed and storage utilization the inverted lists are ordered to enable run-length encoding and then compressed using one-pass compression algorithm. This technique is described in detail in [8] and reduces the total size of inverted lists to few percents of the database size, if positional information is not stored.

To find all texts relevant to a query (that is, containing specified words and possibly their synonyms) we need to retrieve appropriate inverted lists for processing. The reasonable strategy for retrieval of inverted lists is to order them by their length and process strings from the shortest ones. This strategy keeps the size of the intermediate results relatively small.

Problem arises when some lists have very big length and traversing becomes inefficient. To overcome this we introduce an index similar to B-tree for long lists.

To be suitable for search speed-up, the index structure must satisfy the following properties:

- It should combine run-length encoding and compression with tree structure so that it does not occupy much more space than the purely sequential lists without additional indices.

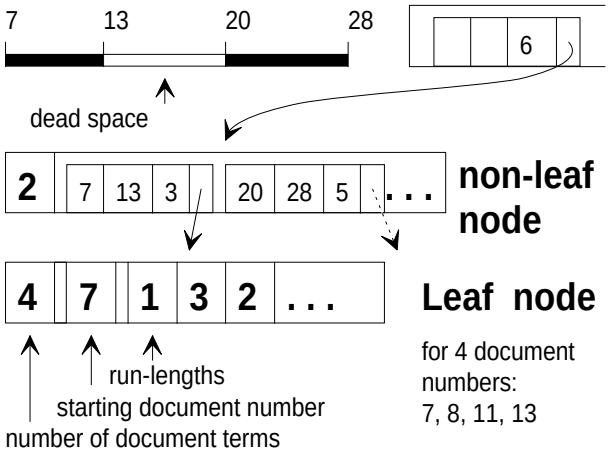

Figure 2: SB-tree structure example

- It must have possibility to represent *dead space* in every node so that eliminate unnecessary traversing of the tree.

- It should store information about size of the dead space of children nodes in every non-leaf node. This information is used in the proposed algorithm.

Thus, our B-tree should have the following structure. Every non-leaf node contains entries of the form (min, max, ds, ref), where min and max is a minimal and maximal text numbers of the sub-tree pointed by ref, and ds is a dead space of the child node. Dead space of a node is defined as a sum of lengths of extents from node extent which is not covered by children node extents. Node extent is a minimal extent containing all subtree text numbers.

Leaf nodes contain lowest text number and encoded differences for other text numbers sorted in increasing order.

Structure of the tree is shown on fig. 2.

This tree structure may be viewed as one-dimensional R-tree[1] with additional information and special form of leaf nodes. Manipulation of such trees is similar to spatial (multi-dimensional) case with some simplifications (or similar to B-trees with additional features). For example, application of *forced reinsert algorithm* [1] is desirable and provides for better distribution of the document numbers and more adequate representation of clustering structure of the inverted lists.

Dead space information is maintained by additional actions during tree insertion procedure. Initially tree has only one leaf node and no dead space information. When we insert new text value, insertion path is found first. Then starting from the leaf node we

insert new entry and evaluate parent node entry with dead space information, if split is performed then we compose two new nodes and also evaluate parent node entries for them. Then we insert evaluated entries to the parent node in the same way going upwards the tree.

Another feature of the insertion algorithm of our tree is managing encoded information in leaves. For them we should use insertion algorithm for run-length encoded lists [8].

Using such B-trees for long lists will cause some additional disk space consumption, but it will speed up search and update of the inverted lists.

3 The Algorithm

In this section we will describe the algorithm for merge of inverted lists which is based on the index structure outlined in the previous section. We denote this special variant of B-tree as SB-tree in following.

Let $P(w,t)$ be a predicate: word w is contained in a document t. We consider queries of the following form: find all documents t, where $\wedge_{i=1}^{n} \vee_{j=1}^{m_i} P(w_{ij},t)$.

Let $Q = \cup_{i=1}^{n} \cup_{j=1}^{m_i} w_{ij}$ be a set of all terms that occurs in the query.

For query evaluation we use $|Q|$ stacks which elements are SB-tree entries contained in increasing order of text numbers. The entries of the leaf level nodes are placed into stacks in decoded form: $(elem, elem, 0, NULL)$, where $elem$ is a text number, dead space is zero and pointer to child node is null. The stacks are referred in the algorithm in two forms: $Stack(i) \quad i = 1,\ldots,|Q|$ and $Stack(w_{ij})$, so that if w_{ij} is k-th element in Q then $Stack(k)$ and $Stack(w_{ij})$ denote the same stack. Besides, we use one additional stack $PosZones$ for storing extents showing zones of possible resulting document numbers.

Now we describe the merge algorithm.

1. **Initialization.**
   (a) For $i = 1,\ldots,|Q|$ read root node of the SB-tree for i-th word in Q and place its entries in $Stack(i)$. If there is no SB-tree for particular word its list numbers must be decoded and placed in stack as well as leaf entries.
   (b) Initialize $PosZones$ to be actual.
2. **Normalization.**
   (a) Delete all entries from word stacks which intersects no $PosZones$ extents.
   (b) If for text number t all word stacks have entry of the form $(t,t,0,NULL)$ then add t to the result set of the query. Delete this entry from stacks and adjust $PosZones$ to be actual.
   (c) If all stacks are empty then end of the algorithm.
3. **Search and retrieval.**
   (a) Let (t_{min}, t_{max}) be the top entry of $PosZones$. Choose word stack which top entry extent (t_0,t_1) satisfy $t_0 < t_{min} < t_1$, and value of dead space is maximal. If there is no such stack choose from ones satisfying $t_0 \le t_{min} \le t_1$.
   (b) Read child node of the top entry of chosen stack and place its entries instead top entry into stack.
   (c) Adjust $PosZones$ to be actual and go to the step 2.

Actual state of $PosZones$ is defined in the following way. Let $F(Stack)$ be a union of all extents of $Stack$ entries. Then $PosZones$ is a minimal set of extents so that

$$F(PosZones) = \cap_{i=1}^{n} \cup_{j=1}^{m_i} F(Stack(w_{ij})),$$

where $F(PosZones)$ is defined as well as $F(Stack)$. To compute $PosZones$ we should for each OR group of word stacks evaluate respective possible zones by merging and uniting stack extents. Then possible zones of OR groups must be intersected also using merge to get result.

More detailed algorithm of $PosZones$ maintaining is described below. Besides $PosZones$ we use n additional stacks $OrPosZones(i)$ for every OR group and additional $m * n$ stack pointers for every word stack, so we can access these stacks without loosing data. We denote counterpart stacks as $EStack(w_{ij})$. Further an algorithm for computing $OrPosZones(i)$ is described.

1. Clear $OrPosZones(i)$, initialize $EStack(w_{ij})$ to be $Stack(w_{ij})$ for every j.
2. Find extent from tops of $EStack(w_{ij})$ with minimal left border. If stacks are empty then end of the algorithm.
3. Pop found extent form its $EStack$ and push it into $OrPosZones(i)$ if $OrPosZones(i)$ is empty or its top extent is not intersected with one added. If added extent do intersect top one than right border of top extent gets right border from added.

To initialize $PosZones$ we should evaluate $OrPosZones(i)$ for every OR group and then intersect them. In item 2.a of the main algorithm we need only delete entry (t,t) from all $OrPosZones$ and $PosZones$. In item 3.c to do $PosZones$ actual we should compute $OrPosZones$ for OR group of chosen stack and then evaluate PosZones.

The figure 3 demonstrate how $PosZones$ is defined. Tops of stacks are at the left side of the picture. Vertical line corresponds to t_{min}.

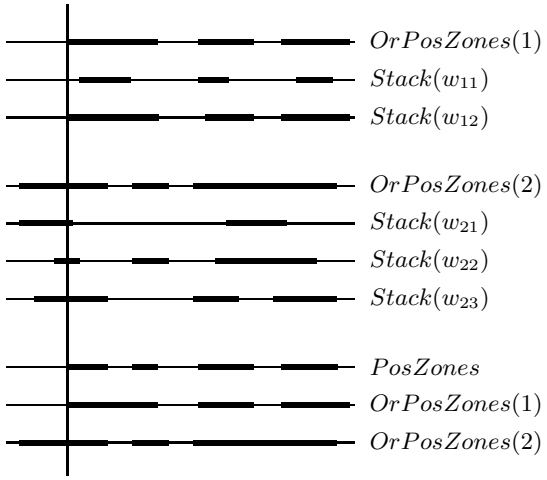

Figure 3: Example of defining $PosZones$.

Adjustment of $PosZones$ is more simple because it links with local changes of stacks and can be done more effective than initialization. Thus, after reading one stack entry only such extents of $PosZones$ may change which intersects extent of read entry.

The proposed algorithm has the following properties.

- Because of stack order of processing memory requirements are limited by

$$|Q| * (L * (E-1)+1) * (Ss + Sp) + C,$$

where L is a maximal number of levels of SB-trees participating in the query, E is maximal number of entries of SB-tree node, Ss and Sp are sizes of stack and $PosZones$ entries. C is size of memory for other temporary variables.

- Algorithm prevents most unnecessary page fetching choosing at every step appropriate read candidate.

Evaluation of more complex queries with negations can be done in the following way. Given query in the form: find all documents t, where $\wedge_{i=1}^{n} \vee_{j=1}^{m_i} A(w_{ij}, t)$. Here $A(w_{ij}, t)$ is $P(w_{ij}, t)$ or its negation. First we find all documents satisfying our query without OR groups where negations occur. Then we filter out irrelevant documents for initial query.

For very frequent terms SB-trees of *irrelevant* documents can be maintained, and can be used for negated predicates of such terms.

4 Performance Analysis

In this section we analyze possible performance gains of the proposed algorithm and provide results of some experimental measurements.

To estimate the performance we need certain assumptions about term frequencies in the documents and distribution of length of the inverted lists. It is well known that this distribution is very far from uniform and usually satisfies Zipf law.

Consequently, the vast majority of inverted list will be extremely small and will not require any additional index, and our algorithm will not help to process these terms.

However, we should assume that the frequencies of terms in queries are very close to that in documents (because otherwise the probability of any successful retrieval is very close to zero, which is not the case for actual systems). This means that practically each large query contains significant number of very frequent terms that have very long inverted lists.

For the query containing N terms we estimate the number of frequent terms as $N_f = N * P_{freq}$.

The number of disk accesses for frequent terms can be estimated as

$$\text{Cost}_{conv} = \frac{\text{AVGFreqListSize} * N_f}{\text{BlockSize}}$$

for conventional linear scan of the lists, and as

$$\text{Cost}_{optim} = \left(T * \log_m \frac{\text{AVGFreqListSize}}{\text{BlockSize}} \right) * N_f.$$

Where m is average number of tree entries in a node, and T is average length of merged inverted lists for non-frequent terms.

Thus the performance gain for the whole query can be estimated as

$$\frac{N - N_f + \text{Cost}_{optim}}{N - N_f + \text{Cost}_{conv}}$$

This formula was evaluated for following parameters:

Param	Value
BlockSize	1 KB

Figure 4 shows results of the evaluation.

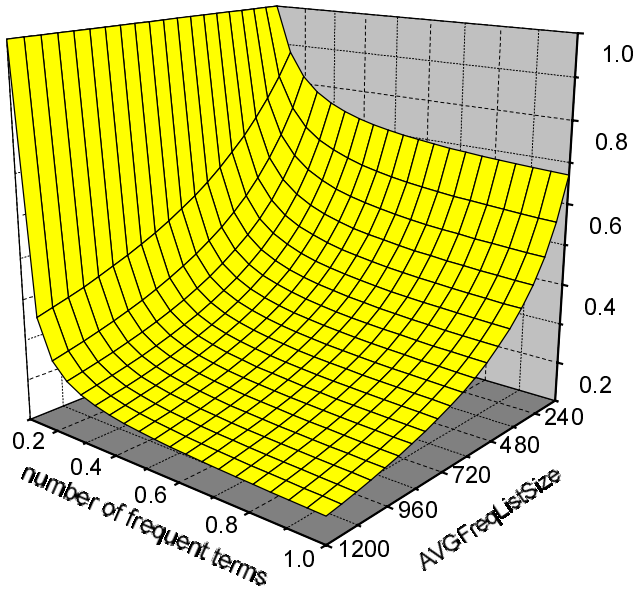

Figure 4: Performance gain estimation.

Really, performance gain will be higher because estimation of Cost_{optim} does not consider savings due to decreasing number of relevant non-frequent terms while processing SB-trees, and savings are gained by choosing appropriate read candidate from stacks. The latter saving may be high when all query terms are frequent.

In the presence of rather small inverted lists a simplified algorithm may be used. We can traverse SB-trees for frequent terms sequentially, decreasing the output set in every step.

5 Conclusions

In this paper we propose a technique for creation of dynamic indices for text retrieval systems and full text databases.

This techniques improves efficiency of retrieval and enables efficient update of the inverted lists, which is especially important for electronic documents that are subject to frequent updates.

The implementation of this structure should be build on the top of an object-oriented database system, which enables additional performance gains [3, 2], but this is a direction of further work. Another important direction might be an extension of the algorithm to support positional information in queries.

References

[1] N. Beekmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, 1990.

[2] E.W. Brown, J.P. Callan, W.B. Croft, and J.E.B. Moss. Supporting full-text information retrieval with a persistent object store,. In *Proc. Intnl.Conf. on EDBT.*, 1994.

[3] W.B. Croft and L.A. Smith. A loosely-coupled integration of a text retrieval system and an object-oriented database system. In *15th Annu. Int. ACM SIGIR Conf. Res. and Dev. Inf. Retriev.*, SIGIR Forum, pages 223–232, October 1991.

[4] C. Faloutsos and S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. on Database Systems*, 4(2):267–288, 1984.

[5] Gerald Huck, Frank Moser, and Erich J. Neuhold. Integration and handling of hypermedia information as a challenge for multimedia and federated database systems. In *Proc. of the Second Intnl. Workshop on Advances in Databases and Information Systems - ADBIS'95*, pages 183–194, Moscow, June 27–30 1995. Phasis.

[6] A. Kent, R. Sacks-Davies, and K. Ramamohanarao. A superimposed coding scheme based on multiple block descriptor files for indexing very large databases. In *Proc. 14 conf. VLDB*, pages 351–359, 1988.

[7] Tatu Ylonen. An algorith for full text indexing. Master's thesis, Helsinki University of Technology, 1992.

[8] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Efficient indexing technique for full-text database systems. In *Proc. 18th Intnl.Conf. on VLDB. Vancouver, British Columbia, Canada, 1992.*, pages 352–362, 1992.

[9] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proc. 19th Intnl.Conf. on VLDB. Dublin, Ireland, 1993.*, pages 290–301, 1992.