International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:11, 2007

Dynamic Inverted Index Maintenance

Leo Galambos
Department of Software Engineering
Charles University in Prague, Czech republic
Email: leo.galambos@mff.cuni.cz

***Abstract*— The majority of today's IR systems base the IR task on two main processes: indexing and searching. There exists a special group of dynamic IR systems where both processes (indexing and searching) happen simultaneously; such a system discards obsolete information, simultaneously dealing with the insertion of new information, while still answering user queries. In these dynamic, time critical text document databases, it is often important to modify index structures quickly, as documents arrive. This paper presents a method for dynamization which may be used for this task. Experimental results show that the dynamization process is possible and that it guarantees the response time for the query operation and index actualization.**

***Keywords*— Search engine, inverted file, index management.**

I. Introduction

Many algorithms for actualization of indices in place often disable new additions unless the actualization is finished. Unfortunately, this lock may be active for several minutes, for a small collection of documents, and in the case of terabyte-sized collections, it may be active for hours. Therefore, our goal is the algorithm which will minimize this delay, while keeping the data structure (index) in good shape. What is meant by "good shape" is that the data structure is not significantly slower than the fully-optimized index structure.

A. This paper

The method of this paper addresses how to incrementally update an index in place while still guaranteeing the response time for querying and index actualization. Second, the method uses a native solution which can be easily implemented without complex data structures or extra data space. Therefore, the method can be used in cases where it is not possible to use new advanced methods (see Chapter 2).

This paper is organized as follows. The current methods are summarized in Chapter 2. The actualization algorithm, based on the dynamization, is presented and discussed in Chapter 3. Chapter 4 presents the extension for batch processing. The impact on searching phase is discussed in Chapter 5. Issues related to the Web are discussed in Chapter 6.

B. Theory, Background

The results presented in this paper were achieved with the EGOTHOR engine [8]. At its core, it uses the vector model with implementation of inverted lists [7].

In the Vector space model [15], [16], the query (Q) and document (D) are represented as vectors. The vectors are indexed by terms (t_i), or rather, by their numbers ($i = 1 \ldots terms$). A document vector is defined as $\overrightarrow{D_i} = (w_{i,j})_{j=1\ldots m}$, where $w_{i,j}$ is the weight of the j-th term in the i-th document and m is the number of different terms in a corpus. Generally, terms that are absent from a document are given zero weight. A query vector is defined in the same way ($\overrightarrow{Q} = (q_j)_{j=1\ldots m} = (w_{q,j})_{j=1\ldots m}$). The document-term matrix $DT = (w_{i,j})_{i,j}$ represents the index. The columns of that matrix (without zero-cells) represent the inverted lists. The number of rows (number of indexed documents) is also termed "the size of the index". The index may contain other values, for instance, positions of words (tokens) in documents. For purposes of this paper an item in an inverted list is termed a "tuple".

We will assume that inverted lists are stored in one file (inverted file) in an order that reflects the order of their terms. That is, the inverted list of term t is stored before the lists of terms $t'_1 \ldots t'_s$, if and only if the term t is (alphabetically) lower than any of $t'_1 \ldots t'_s$ terms. This format ensures that two inverted files T and U can be merged by reading them sequentially. Assuming that the inverted file X is built up for the document collection C_X, and its length is L_X, the merge operation then produces a new inverted file V, and it holds: $C_V = C_T \cup C_U$, $L_V = L_T + L_U$, with the operation taking $O(L_V)$ instructions. When a set of inverted files is merged, the process uses $O(L_V\ .\ .\ \ u)$ instructions, where u is the number of merged files.

Index actualization can be reduced to inverted file actualization. Therefore (for simplicity's sake) the inverted file represents the index in this paper.

II. Existing solutions

In this section, the methods for modification of inverted indices are discussed briefly. We will assume that the indexed collection of documents has the same properties as WWW, where the number of inserted or deleted documents is smaller than the number of modified documents [9]. Moreover, the number of changes is rather small when held against the whole collection of documents.

Rebuild. This method replaces an obsolete index with a new one which is built from scratch. Obviously, this way is not effective because it always re-scans all documents in a collection. The method could be improved by distributed processing, but it does not change the fact that this method is wasteful for the document collection assumed herein.

Delta change-append only. Some improvement is achieved when just the modified documents are processed. This can be implemented effectively using the standard indexing technique (known as merging [2], pp 198). For this, the index is merged with an index built for new documents. When a document should be removed, it is only denoted as "deleted". Modification of a document is then realized via Delete+Insert operations. This implementation is very popular and can be found, for instance, in the Lucene engine [1].

Unfortunately, one serious drawback exists - if we execute many Delete operations (also part of the Modification operation), the index structure stops working effectively, and what is more important, it is not possible to easily detect this situation. Therefore, from time to time one must optimize the index. Such an optimize phase may take a lot of time, which again makes this method less effective. The issue was studied by many researchers, for instance [5]. Unfortunately, their solution was demonstrated with simplified conditions; it would be interesting to see whether these conditions hold in a heavy load system.

Forward index. Another method was proposed by Brin and Page for the Google search engine [3]. This method constructs an auxiliary data structure (forward index) which speeds up the modifications in the main (inverted) index. The term "modifications" means the real changes of the existing values stored in the index. Other modifications (insertion or removal of values to/from the index) are realized using other approaches, e.g. Btree [6].

Landmark. Research in this area continues and new approaches are still developed, for instance, a landmark method [10] which introduces special marks in the index. The marks are used as initial points from which some values in the index are derived. Therefore, if one modifies the mark, all values dependent on the mark are also shifted. It was shown in the cited paper that such a case often happens in an index built for WWW and, as a result, the landmark method was faster than the forward index.

Other approaches. Our summary misses some methods, which were developed in past decade, i.e. [4], [17].

This paper. We will try a straightforward way that is based on the dynamization of a static index structure. The method solves the issue (where the index structure can stop working effectively) of the "Delta change-append only" approach. Moreover, the method does not need extra data space (as does a "Forward index"), and even works when the inverted lists are compressed and not based on the index structure supposed for the "landmark" method, i.e., when positions of words in a document are not stored. Last, but not least, the method saves a file system structure, because it may keep all the index files defragmented.

III. Dynamization

The dynamization we use is inspired by Mehlhorn's algorithm [12]–[14] for common data structures, e.g., hash tables. The algorithm will be modified and reformulated for an IR system. Let us agree upon the following terminology: *a Barrel is an autonomous index that is often static (it represents the inverted file above); a Tanker is an index that is decomposed to smaller barrels, and it disposes of dynamization as the actualization algorithm.* The term (Barrel) was used in many previous papers, i.e. [3], but it is used in a different meaning in this paper.

Example: One can build a simple index (of size 1) for any document. These indices are always barrels, because the index is static (for a given document). Moreover, the index can be searched, therefore it is also autonomous. Later we will show how these simple barrels are organized into tankers, and how the larger barrels are constructed.

We already know that a document can be easily transformed to a simple barrel, thus barrels are considered instead of documents in this paper. Firstly, let us agree upon the following notation:

Notation 1: Let B be the barrel that is built for the collection of documents C_B. The barrel B offers the query operation $search_B(q, C_B)$ (for a query q). It does not matter how the search is implemented. If n is the number of documents in the collection C_B, then the size of barrel B is $B.size() = n$; the time we need to build B over $|C_B|$ is $Ti_B(n)$; the space we need for the structure B is $Sp_B(n)$; the time we need to compute $search_B(q, C_B)$ is denoted $Q_B(n)$.

The document removal operation is implemented using a bit array which sets the bit related to the document to 1, if and only if the document is removed from the collection. Obviously, such documents must be also filtered out of the hit lists prepared by $search_B$. This operation can be easily implemented, however.

Notation 2: Let B be the barrel. The number of documents denoted as removed in the barrel is $B.deleted()$. The number of live documents in the barrel is $|B| = B.size() - B.deleted()$.

Similarly, we define the notation in the case of tanker T – $Ti_T(n)$, $Sp_T(n)$ and $Q_T(n)$. Moreover, we must require a condition which is described in the following definition:

Definition 1: Let T be the tanker containing barrels $B_0, B_1, \ldots B_r$ (also accessed as $T\ i$ in the algorithm below). Next, let C_i be a short form of C_{B_i}. We request that $\forall i \neq j : C_i \cap C_j = \emptyset$. If and only if a position j is not occupied in the tanker, we define: $B_j = \emptyset$, $C_j = \emptyset$. When the position j is not occupied, we define $search_{B_j}(q, C_j) = \emptyset$.

Let us denote the tanker collection $C_T = {}_{i=0}^{r} C_i$. Tanker T is able to solve the query operation (for a query q) as $search_T(q, C_T) = search_{B_0}(q, C_0) \oplus search_{B_1}(q, C_1) \oplus \cdots \oplus search_{B_r}(q, C_r)$, where $\oplus$ denotes the composition of partial results. This operation is computable in constant time because we only compute hit lists of limited length.

Moreover, we request:

$$\forall i : B_i \neq \emptyset : 2^{i-3} < |B_i| \leq 2^i \ and \ 1 \leq |B_i| \qquad (1)$$

This condition ensures that no part of a tanker is degenerated after a number of *Delete* operations. If such a situation did happen, we would execute a reorganization which would repair the tanker structure (see below). After this operation, the condition would again hold.

The actualization in a tanker after barrel insertion can be achieved by Alg. 1. The algorithm ensures that the tanker will reorganize its inner structure and that the condition (1) always holds.

Obviously, it is not effective to work with small barrels on a disk. That is why the EGOTHOR engine introduces barrels implemented in memory. They can be used as a replacement

Algorithm 1 Algorithm of dynamization – Tanker, routine insert(B:barrel)

```
$k := \ {}_{..}\ \{x; |B| + \ {}_{i=0}^{x} |B_i| \leq 2^x\};$
$S := \{B_i; i < k \wedge B_i \neq \emptyset\} \cup \{B\};$
$\forall i < k : B_i := \emptyset;$
$B_k := Merge(S);$ {Merge barrels of set S; values of removed docs are left out}
```

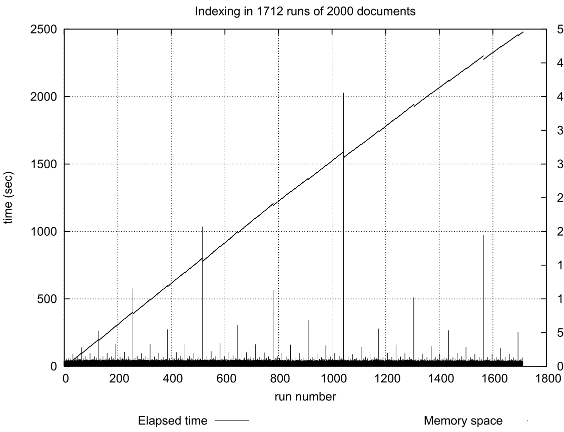

Fig. 1. Indexing (Redhat 7.3, IBM JDK 1.3, SMP 2xPIII/700MHz, SCSI).

for barrels which are used in the lower cells of a tanker. More often, we can implement the memory barrel as a dynamic barrel that can absorb other small barrels without use of the dynamization algorithm (it is obvious that in memory we can simply append to any inverted list, for instance when the inverted lists are implemented as growing arrays).

This "catch-them" barrel can be used as a cache in a tanker and in this way the engine can speed up indexing. In the tanker that was used in tests (see below), a cache barrel of size 64 was active. Because of this the tanker does not use dynamization to build up barrels in cells $0 \ldots 6$.

The experiment is organized as follows: 3.5M HTML documents (on the `ac.uk` domain) were divided to n groups each containing 2000 documents. The collection is indexed in n runs and for each input group a Java program adds new documents to already processed documents; processing time Ti_i of the i-th run, and size Mem_i of the index after the i-th run are measured.

The two properties ("Elapsed time" Ti_i, "Memory space" Mem_i) are summarized in Fig. 1 ("Memory space" is drawn by points which form the diagonal lines).

When studying the test, it is known that the higher values (jumps) in the presented graphs reflect the situation when a large barrel is built. If the run contained more documents, it could also signal that a lot of these barrels were built. Therefore, runs of 2000 documents were presumed to be quite enough to get relevant results. The tests then revealed that we can assume that $Ti_B(2n) = 2Ti_B(n) = Ti_B(n) + Ti_B(\frac{n}{2}) + Ti_B(\frac{n}{4}) + \cdots + Ti_B(1)$.

It can also be seen, that the presented method works as fast as merging with a merge factor equal to 2. On the other hand, in a real system the index must also reflect other modifications than simple insertions, for instance, when a document is changed or removed.

IV. Batch processing

Until now we assumed that the index is modified after each insertion. This chapter introduces batch processing, when a set of insertions or removals is executed at once. For clarity's sake we will present a solution to a case where the number of removals is almost the same as number of insertions. All these changes come from modifications of documents, because we use Delete+Insert approach. It is also assumed that the index is already huge and the number of removals or insertions is rather small (less than 30%).

Algorithm 2 (Algorithm of dynamization) Tanker, routine **modify(D:documents)**.

```
{step1: remove all obsolete data}
mark existing documents $D$ as deleted/invalid in the index using bit vector
{r is the number of cells in this tanker}
for all i=0...r do
  if $|b\ i\ | == 0$ then
    n[i]=empty
  else
    if $|b\ i\ | > 2^{i-3} > 0$ then
      n[i]=b[i]
    else if i==0 then
      n[i]=b[i]
    else if $|n\ i - 1\ | >= 2^{i-2}$ then
      n[i]=merge(n[i-1]);n[i-1]=merge(b[i])
    else
      y=merge(b[i],n[i-1])
      if $|y| >= 2^{i-2}$ then
        n[i]=y;n[i-1]=empty
      else
        n[i]=empty;n[i-1]=y
      end if
    end if
  end if
end for
{step2: if a barrel is still small then eliminate it}
$zombie = \{n\ i\ ; n\ i\ .size() > 0 \wedge |n\ i\ | \leq 2^{i-3}\}$
new=merge({D},zombie)
{step3: insert the new barrel}
while all positions for $new$ are occupied in $n$, merge the barrels with $new$
$new$ is saved into the array $n$
{step4: execute all merges we planned}
COMMIT: array $n$ becomes $b$, obsolete barrels are discarded
```

The algorithm 2 repairs a tanker which reflects changes in documents D. It transforms an existing tanker's structure b to a new one – n (both are arrays of barrels, so that B_i is represented as $b\ i$ or $n\ i$). All merge operations denoted by the $merge$ method are delayed until we really need the product of the merge. The empty barrel is denoted as $empty$.

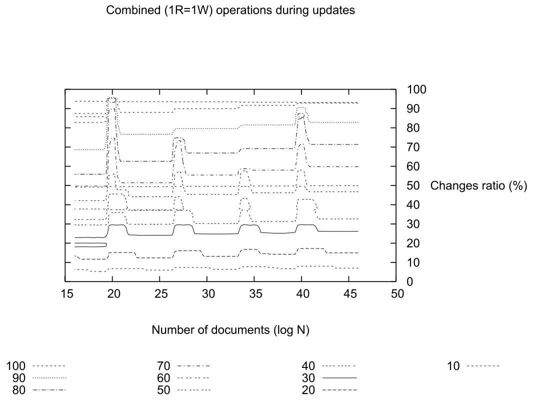

Fig. 2. Comparison: dynamization versus RFS.

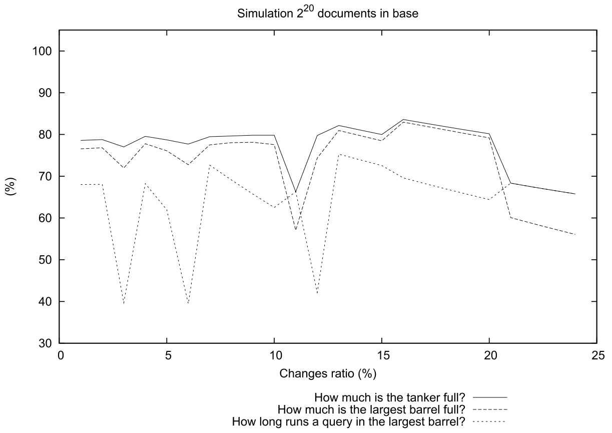

Fig. 3. Simulation of the dynamization algorithm.

We also note that the $merge(B_i)$ operation creates a copy, not containing any values of documents which are denoted as removed, of barrel B_i.

The main strategy of the algorithm 2 is summarized in four steps: 1) all obsolete documents are denoted as removed, and the tanker is repaired to better comply with the condition in Def. 1; 2) all new documents are indexed using the classic merge algorithm of merge factor m (we use $m = 100$). This way a barrel new is produced. If some tanker's barrels do not comply with the condition in Def. 1, they are merged with the barrel of new documents; 3) the new barrel is appended to the index; 4) all delayed $merge$ operations are executed, and the tanker is transformed to a new consistent state.

A. Experiment

We decided to describe the algorithm alternatively, using its simulation. For this, we utilize the formula $Ti_B(2n) = 2Ti_B(n)$, which holds in our real IR system. Then, we can estimate the time needed to realize the steps of the Alg. 2.

The existing methods introduced in the Chapter 2 are often compared with the method labeled as "rebuild from scratch" (RFS). We will apply the same approach.

The simulation is then organized as follows: the original tanker has 2^N documents and it consists of one barrel B without deleted documents $B.deleted() = 0$. The barrel is placed at position N. The value N is placed on the X-axis in the figure. The updater always removes and inserts chg percent of the index size (Y-axis). We measure the number of read and write operations S needed to realize the Alg. 2. The Z-axis then presents the ratio $\frac{S}{M}$, where M is the number of read and write operations needed by the merging RFS strategy of factor 100. The measured values are the average of 10000 reiterated runs. The 3-d figure is equivalently represented as a contour figure (see Fig. 2).

B. Discussion

The first step of Alg. 2 is the removal of values of all removed or modified documents in the collection. This can be done quickly, and if desired, the operation can be already done by a crawler that gathers the new or modified documents. Other steps are not so easy, but the simulation shows how they work over a long term (10000 reiterated runs). It should be noted that we can simulate the algorithm for collections which are almost unrealistic for evaluation. A collection of 2^{46} documents is not easy to obtain, and 10000 repetitions of the experiment are pretty unrealistic on the hardware available.

The simulation was verified on a collection of size 2^{21}. When the test was repeated with 10% of the documents randomly changed, it was saved about 86% of time comparing to the complete rebuild. Our theoretical assumption of 88% was not achieved, but the difference is small, such that it can be rooted in the fact that we had to measure only some parts of engine's routines, and in JAVA, we were not able to use accurate system timers.

V. Searching

The typical searching phase consists of two steps. First, we look up all terms in a dictionary to find offset positions of the respective inverted lists. Next, the inverted lists are read and evaluated. Obviously, if the dictionary is cached in a hash table in a memory (RAM), then the majority of time is consumed by the second step.

Due to the fact that a tanker $T(n)$ must have more than $\frac{T.size()}{8}$ live documents, we can claim that a query over the tanker needs time $Q_B(8n)$ to be solved. This holds for a single thread environment (STE), but the query could also run over each of the tanker's barrels concurrently. Then the time is rather limited by the construction size of the largest barrel in the tanker.

Ee are also interested what happens in a long term. It could happen that the tanker almost always contains just a few deleted documents and the limit case (the tanker has up to $7/8$ of deleted documents) occures only time to time and infrequently. Obviously, the opposite case could also happen.

A. Experiments

The answer to this is given by Figure 3 describing the experiment. The tanker T is an index over 2^N documents and it consists of one barrel B without deleted documents $B.deleted() = 0$. The barrel is placed at position N in the tanker ($T\ N\ = B$). The updater always removes and inserts chg percent of the index size (X-axis). We measure the ratio $\frac{T.size()-T.deleted()}{T.size()}$ and the Y-axis then presents the average value of the ratio after 10000 updates. It should be also noted that we simulated the algorithm for collections which are almost unrealistic for evaluation ($15 < N < 47$), but the collection size has not any impact on the results, so we decided not to include this parameter in the experiments presented herein.

The figure shows that we lost about 25% (in a long term) when using the dynamization for an index that is updated concurrently and the number of changes is less than 10% – a typical situation for a live web search engine. We may also lost about 35% in several concrete cases. Fortunately, our algorithm could be enhanced with a routine that checks whether we are in such a bad case. If so, the routine could shorten or lengthen the period of updates and move us to a better case.

The presented figure is related to the situation when the barrels of a tanker are evaluated sequentially. How is this changed when the barrels are evaluated in a multi-threaded environment (MTE) and they operate over a shared result list? The answer is given by the ratio of the construction size of the largest barrel and the number of live documents in the tanker (as explained above). The simulation is presented in Figure 3. It shows that we could save 30%-70% of time (comparing to a fully optimized index) when the number of changes is within a reasonable boundary (less than 30%).

Now, another point is interesting as well – how many live documents are stored in the largest barrel? The answer to this shows us how effectively we can evaluate a query in MTE.

The respective simulation is presented in Figure 3 and we could again claim that only about 25% of performance is lost due to the deleted documents (in a long term).

It was still told about the average case, or rather, about the long-term effectiveness. However, a real threat was not discussed yet – what will happen in the worst case when the algorithm only guarantees that the response is given (up to) eight-times slower than with the fully optimized index? Could it be improved this parameter? Can the speed be improved when the barrels are ordered by a page rank?

VI. The Web, practical notes

Our algorithm could run eight-times slower than the fully optimized index in the querying phase. However, according to [11], we could still save more than 60% of time with skipping of $L = 100$ (L – skip factor [11]). All we need to do is to implement our bit array (storing the 1-bits for deleted documents) as an inverted list and include it into the queries. Since the bit array is (can be) fully kept in RAM, we can save more than was described in the cited paper – the authors assumed that all inverted lists are read from a hard disk, while we have one of them in RAM.

Moreover, if the inverted lists are sorted by a document rank (the highest first), then just a few first blocks are read, because all other hits would be computed for less-important pages, so they would not be included amongst top-N.

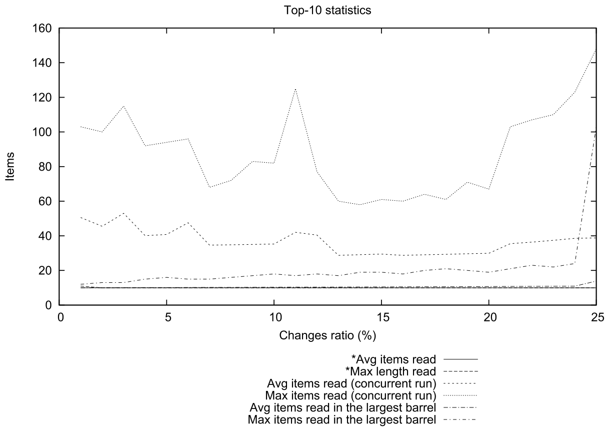

Fig. 4. Length of the inverted lists for top-10 evaluation.

We must only find a correct starting point of an inverted list (disk seek operation) and start to read sequentially. In fact the most expensive routine is the first seek operation. According to our tests with the EGOTHOR system [8] we can claim, that a disk block (4kB) is able to save about 500-1000 tuples of an inverted list. Obviously, such a block is often quite enough to generate top-10. Using the dynamization, we may have to generate top-80 (up to 70 documents could be still denoted as deleted), but this could be still covered by the tuples of the first block implying no extra I/O operations.

The following experiment shows what happens with the index, when the index is ordered by page rank. Our setup is: only chg documents are changing; the probability that a document is modified since the last update is chg (changes ratio); top-10 is constructed; the index is updated 10000 times; if a document was not modified in last 1000 updates, it is supposed to be static.

Since the engine operates on the Web, we assume that the top-10 hits are (primarily) looked for in the last barrels constructed with Alg. 2. It is based on the fact, that the most wanted documents are probably saved in the most up-to-date documents of high page ranks.

However, if our query algorithm cannot adopt the evaluation (where the last barrels have higher priority), all barrels must be asked for top-N lists and the final top-N is their joint. Then one might be interested in the maximum length of an inverted list we must scan in any of the barrels (they are asked for the hit lists concurrently). Next, the total number of tuples read in all barrels is an interesting value as well. Both values (including their means after 10000 updates) are presented for top-10 queries in Figure 4. The figure also presents the number of tuples read in a situation when we can adopt the special evaluation as mentioned above. These values are denoted by an asterix.

For instance, when 5% documents are changing since last update, then we must read 10.15 tuples from the largest barrel in an average case (the maximum value is 16). It implies, if the querying runs concurrently, the queries run 1.6x slower

TABLE I
SUMMARY

	Rebuild	Delta change	Dynamization
#Barrels	1	$M \log_M MN$	$log_2 8N$
Indexing	$N \log_M N$	$\Delta \log_M MN$	$\Delta \log_M 8N$
Search (STE)	N	$N \sim MN$	$N \sim 8N$
Search (MTE)	N	$\frac{N}{M} \sim MN$	$\frac{N}{2} \sim 8N$

than over a compact index.

The ”1.6x” factor could imply more I/O operations for high N. Fortunately, if we prepare the hit lists for “top-N queries” where N is in reasonable boundaries (for instance <100), the higher demand for read operations is not significant. This is based on the fact that the inverted lists are always read by disk blocks and it does not matter whether we read 1 byte or the full block. If one tuple was stored using 5 bytes then one disk block of 4kB could save about 819 tuples and the factor of 1.6 would not represent any serious threat.

The experiments confirm that the new method has a real practical impact for search engines operating on the Web.

Finally, the summarization of some features is presented in Table I. The first feature – number of barrels – is one of the most important attributes for a real search system. When the system operates on more barrels, it needs more file handles to read data. Since the maximum of file handles is limited at a given moment, the system might not be able to serve high load. Other features observed are: time that is needed to apply changes of Δ documents amongst N (using merge factor M); time of a query evaluation when the index is not sorted and the query has just one term.

Our method could not be directly compared with other high-performance methods, i.e. “Landmarks”, due to unavailability of source code of the other algorithms. Therefore, the experiments were aligned with the rebuild-from-scratch approach which is also used as a base-line by other researchers [10].

Although we believe that our simulation is good and it accurately reflects reality some of the experiments were verified on a collection of size 2^{21}. When the test was repeated with 10% of the documents randomly changed, it was lost about 25% of time on querying in STE mode (the test computed full hit lists, not only top-10). Our theoretical assumption of 21% was not achieved, but the difference is small, such that it can be rooted in the fact that we had to measure only some parts of engine’s routines, and in JAVA, we were not able to use accurate system timers.

The same test was also repeated in MTE. For this, we borrowed a strong RAID-0 array that could represent a real hardware for a search engine. JDK 1.5 and Java NIO were used on 2xAMD Opteron and the result surpassed our theoretical simulation by five percent (62% comparing to 57%).

The next experiment was pointed to top-10 calculation. Unfortunately, it was not possible to measure any significant difference, because all the differencies were neglible. This is also confirmed by the emulation presented in this paper.

VII. CONCLUSION

We have developed a method for updating index structures in place, especially in dynamic, time critical document databases. The method can be used in configurations where one cannot easily update values stored in inverted lists, and it makes the modern techniques (i.e. landmarks) unusable.

It was shown that the method can work much more effectively comparing to ”rebuild from scratch”. On the other hand, the trade-off is based on the fact that we are satisfied with slower querying – up to eight times comparing to a fully optimized index. Fortunately, it was shown that the factor can be further lowered. Our preliminary experiments and simulations show that the extra operations are almost fully compensated by the hardware architecture of current computers, where it does not matter whether a program reads one byte or hundred bytes from a disk.

REFERENCES

[1] Apache, Jakarta project: Lucene. `http://jakarta.apache.org`.

[2] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Chapter 8. ACM Press 1999.

[3] Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems 30, 1–7, 107–117, 1998.

[4] Brown, E.W.: Execution Performance Issues in Full-Text Information Retrieval. Computer Science Department, University of Massachusetts at Amherst, Technical Report 95-81, October 1995.

[5] Clarke, C., Cormack, G.: Dynamic Inverted Indexes for a Distributed Full-Text Retrieval System. TechRep MT-95-01, University of Waterloo, February 1995.

[6] Cutting, D., Pedersen, J.: Optimizations for dynamic inverted index maintenance. Proceedings of SIGIR, 405-411, 1990.

[7] Fox, E.A., Harman, D.K., Baeza-Yates, R., Lee, W.C.: Inverted files. In Information Retrieval: Data Structures and Algorithms, Prentice-Hall, pp 28-43.

[8] EGOTHOR, JAVA IR system. `http://www.egothor.org/`.

[9] Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.: Characterizing Web Document Change, LNCS 2118, 133–146, 2001.

[10] Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.: Dynamic Maintenance of Web Indexes Using Landmarks. Proc. of the 12th W3 Conference, 2003.

[11] Moffat, A., Zobel, J.: Self-Indexing Inverted Files for Fast Text Retrival. ACM TIS, 349–379, October 1996, Volume 14, Number 4.

[12] Mehlhorn, K.: Data Structures and Efficient Algorithms, Springer Verlag, EATCS Monographs, 1984.

[13] Mehlhorn, K., Overmars, M.H.: Optimal Dynamization of Decomposable Searching Problems. IPL 12, 93–98, 1981.

[14] Mehlhorn, K.: Lower Bounds on the Efficiency of Transforming Static Data Structures into Dynamic Data Structures. Math. Systems Theory 15, 1–16, 1981.

[15] Salton, G., Lesk, M.E.: Computer evaluation of indexing and text processing. Journal of the ACM, 15(1):8-36, January 1968.

[16] Salton, G.: The SMART Retrieval System - Experiments in Automatic Document Processing. Prentice Hall Inc., Englewood Cliffs, 1971.

[17] Tomasic, A., Garcia-Molina, H., Shoens, K.: Incremental Updates of Inverted Lists for Text Document Retrieval. Short Version of Stanford University Computer Science Technical Note STAN-CS-TN-93-1, December, 1993.