Efficient On-line Index Maintenance for Dynamic Text Collections by Using Dynamic Balancing Tree

Ruijie Guo
Institute of Computing Technology, CAS
ruijieguo@software.ict.ac.cn

Xueqi Cheng
Institute of Computing Technology, CAS
cxq@ict.ac.cn

Hongbo Xu
Institute of Computing Technology, CAS
hbxu@ict.ac.cn

Bin Wang
Institute of Computing Technology, CAS
wangbin@ict.ac.cn

ABSTRACT

Previous on-line index maintenance strategies are mainly designed for document insertions without considering document deletions. In a truly dynamic search environment, however, documents may be added to and removed from the collection at any point in time. In this paper, we examine issues of on-line index maintenance with support for instantaneous document deletions and insertions. We present a *DBT* Merge strategy that can dynamically adjust the sequence of sub-index merge operations during index construction, and offers better query processing performance than previous methods, while providing an equivalent level of index maintenance performance when document insertions and deletions exist in parallel. Using experiments on 426 GB of web data we demonstrate the efficiency of our method in practice, showing that on-line index construction for dynamic text collections can be performed efficiently and almost as fast as for growing text collections.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information Retrieval—*Indexing Methods,Search Process*; H.3.4 [**Systems and Software**]: Performance evaluation

General Terms

Experimentation, Performance

Keywords

Information Retrieval, Index Maintenance, On-line Index, Garbage Collection, Dynamic Text Collections

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'07, November 6–8, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-803-9/07/0011 ...$5.00.

High-performance inverted indices are used in modern information retrieval systems to provide efficient text query evaluation [13]. They are employed on scales ranging from desktop search facilities up to file system search, and ranging from search in digital libraries up to search on collections of entire Web. Generally, an inverted index stores terms, document numbers, term frequency and term positions, which can be used for ranked queries or phrase queries. Witten et al. [13] provide an introduction to inverted indexes, and to some query modes.

The efficiency of inverted index construction by off-line approaches has improved dramatically in the past decade [13, 14]. Witten et al. [13] give an overview of several off-line algorithms. Heinz and Zobel [6] present a mechanism with improved performance. Off-line index construction algorithms are efficient and suitable for static collections in which documents are rare updated and deleted. In such off-line mechanism, queries can be only processed at the completion of the construction.

There are some search environments, such as news search, products search, and file system search, where documents arrive at a high rate, and the documents must be indexed for search as soon as they are inserted. In this case, off-line index construction algorithms are prohibitively expensive and may be unacceptable. Lester [8] and Büttcher [3] present an on-line index construction mechanism that builds an inverted index when the underlying data must be continuously queryable. On the other hand, documents are also removed from the collection. Chiueh and Huang [5] and Büttcher [4] give a discussion of index maintenance strategies in the presence of document deletions.

On-line index construction which performs on dynamic text collections includes query processing and two types of index update operations: document insertions and document deletions. Indexing performance of off-line approaches is for creation of a complete index on a collection of documents. In contrast, the performance of on-line index maintenance actually refers to index updating performance and query processing performance. In principle, insertion or deletion of a single document involves modifying a few bytes of the posting list of every term in the document. A document may contain hundreds of thousands of distinct terms, and each posting list may contain millions of entries. Therefore, update operations are potentially costly in disk accesses and complex in space management [9]. At the same point in

time, queries must be processed over the index, and efficient query processing requires each list to be stored contiguously on disk, which brings more difficult problems for update operations.

In this paper, we examine issues of on-line index maintenance with support for instantaneous document deletions and insertions. We present a DBT Merge strategy that can dynamically adjust the sequence of sub-index merge operations, and aims to achieve a good indexing and query processing performance by always merging indices with similar sizes. The key idea behind our new approach is to dispatch sub-indices to the appropriate layer of a tree in which each node corresponds to one sub-index. The placing of sub-indexes in the tree depends on the number of documents contained in the sub-indices, so that each merging takes place among similarly sized sub-indexes. Our new method, along with an efficient garbage collection mechanism, offers better query processing performance than previous methods, while providing an equivalent level of index maintenance performance when document insertions and deletions exist in parallel. Since all existing merge-based index maintenance strategies are a particular case of our method, it provides a general and flexible framework for merge-based sub-indices merge strategies.

The remainder of this paper is organized as follows: The next section gives a brief overview of related work in the area of both off-line and on-line index construction, including in-place, merge-based and hybrid on-line index update strategies. In section 3, we present our DBT Merge strategy to index maintenance, along with an efficient garbage collection method. An experimental evaluation of our new approach and existing merge-based maintenance strategies is presented in section 4. Finally, section 5 gives the conclusions.

2. RELATED WORK

Inverted files are the most efficient data structures for full-text search [15]. An inverted index, which maps from terms to documents, contains two main parts: a vocabulary of searchable terms, and posting lists of terms in the collection. A term's posting list indicates the presence of that term in all documents, and may contain some ancillary information, such as within-document frequencies and positions for all occurrences. Typically, each posting list stored on disk in a single contiguous extent for efficient query processing.

Off-line index construction approaches for static text collection usually divide the collection into partitions, and construct an index for each partition. After all partitions have been indexed, all these sub-indices are brought together through a multi-way merge process, resulting in the final index for querying.

It is clear that if search queries can be processed during index construction, the off-line approach described above can be transformed into an on-line approach. The simplest method is to process queries by concatenating posting lists retrieved from the on-disk sub-indices created so far and in-memory index [4]. Query processing performance of this mechanism may be very poor, because posting list of each term spreads across a number of fragments, implying a great number of disk seek operations for each term.

In-place Index Maintenance

For in-place update strategy, documents are accumulated in main memory. Then, once main memory is exhausted, the existing on-disk index is combined with the in-memory index by appending each posting list from the in-memory index to the corresponding list in the on-disk index. To make this approach more efficient, it is necessary to allocate more space than actually needed. Whenever a posting list needs to be written to disk, it is transferred into the over-allocated space. If the free space is insufficient, relocation is needed in order to create more room for new postings at the end of the list. Shoens et al. [11], Shieh and Chung [10] propose a predictive over-allocation strategy, which can reduce the number of times a posting list has to be relocated.

In general, there are two types of strategies to manage the posting lists of in-place mechanism: those that require each on-disk posting list to be contiguous [9] and those that allow posting lists to be spread across a number of fragments [12]. It can maximize query processing performance to keep posting lists in a contiguous form, but requires frequent relocations of most lists in the index, and introduces a free-space management problem [9]. Allowing posting lists to be non-contiguous, on the other hand, can avoid data movements during each merging operation, but needs some additional bits to link all the fragments. This scheme therefore increases index maintenance performance, but decreases query processing performance, as a greater number of disk seek operations are needed for each term.

Lester et al. [9] present some refinements to in-place update that have the potential to dramatically reduce costs. Cost analysis of in-place update is also given by them in [8].

Merge-Based Index Maintenance

A shared element between merge-based update and in-place update is that new documents are accumulated in main memory to amortize costs. Differing form in-place update, the merge-based approaches merge the in-memory index and on-disk index, resulting in a new on-disk index. The most popular form of merge update is the Immediate Merge strategy: Once main memory is exhausted, a merge event is trigged, and the entire on-disk index is read sequentially and merged with the in-memory index, then, written to a new location to form a new on-disk index that immediately replaces the old one. Since, at any given time, there is at most one active on-disk inverted files, this strategy maximized the query processing performance. The disadvantage is that for every merge event the entire index must be processed.

The second form of merge-based update is the No Merge strategy which does not perform any merge operations. Whenever the main memory is exhausted, in-memory index is transferred to disk, allowing a number of on-disk sub-indices to exist in parallel. Although, this strategy can maximize index maintenance performance, the query processing performance is very poor, as posting list of each term may spread across a great number of sub-indices.

Lester et al. [8] and Büttcher [4] analyze the costs of these update mechanisms. Lester et al. also evaluate these mechanisms [9], they experiments show that the Immediate Merge strategy outperforms the in-place update scheme in almost all practical scenarios.

Recently, It has been studied how allowing a controlled number of on-disk indices to exist in parallel increases in-

dexing performance, without allowing query processing performance to decrease excessively. Lester et al. [8] propose a scheme that breaks the index into tightly controlled number of partitions. They introduce a key parameter r (usually, $r = 2$ or $r = 3$ is chosen). If main memory can hold b postings, then the kth partition contains not more than $(r-1)r^{k-1}b$ postings. In addition, at level k the partition is either empty, or contain at least $r^{k-1}b$ postings. Whenever the creation of a new on-disk index leads to a situation where there are more than $(r-1)^{k-1}b$ postings in kth partition, they are merged into a new index and dispatched to the appropriate partition. This strategy is referred to as Geometric Partitioning. Lester et al. [8] give a cost analysis of this strategy.

Büttcher and Clarke [4] propose a strategy which makes use of the concept of index *generation*. An on-disk index that was created directly from in-memory index is of generation 0. An index that was the result of a merge operation is of generation $g+1$, where g is the highest generation of any of the indices involved in the merge operation. Whenever the creation of a new on-disk index leads to a situation where there are more than one on-disk index of the same generation g, all these indices are merged, resulting in a new on-disk index of generation $g+1$. This is repeated until there are no more such collisions. This strategy is referred to as Logarithmic Merge. Cost analysis of this scheme is described in [4].

Geometric Partitioning and Logarithmic Merge strategies are designed for growing text collections and offer better index maintenance performance than Immediate Merge. The disadvantage is that query processing performance is worse, as posting list of each term may spread across a number of on-disk sub-indices, and that document deletions are not considered.

Hybrid Index Maintenance

Büttcher and Clarke [3] present a family of hybrid index maintenance strategies which are combinations of merge-based and in-place index maintenance methods. The motivation of these strategies is to avoid the data movements of long posting lists. The key idea is to distinguish between short and long posting lists. Short posting lists are updated using a merge strategy, while long posting lists are updated in-place.

The hybrid approaches are also designed for growing text collections and achieve better indexing performance than in-place or merge-based approaches, but query processing performance is slightly reduced, as there are internal fragments in the on-disk posting lists.

Garbage Collection

Chiueh and Huang [5] present a lazy invalidation approach that uses a delete table, which is organized as a search tree, to identify all deleted documents. For every query, a post-processing operation is performed, checking against the delete table and removing deleted documents. Büttcher and Clarke [4] present an approach that integrates the check operation into the query processing. They also argued that if there are a great number of deleted documents in indices, the query processing performance will be reduced, and the deleted documents should be removed form indices. They introduced a threshold-based garbage collection strategy that integrated into Logarithmic Merge strategy, without considering the impact of garbage collection on the merge strategy.

Removing deleted documents from posting lists helps avoid the decompression of deleted documents during query processing. It therefore increases query processing performance, but deteriorates index maintenance performance, as during garbage collection all posting lists have to be read and decompressed, and, after filtering out postings belong to deleted documents, have to be re-constructed to form new posting lists. Keeping deleted documents in posting lists, on the other hand, avoids expensive decompression and reconstruction of posting lists, but requires extra space to store deleted documents and wastes time on copying of those deleted postings during merge operation, and reduces query processing performance, caused by the reading and decompressing of deleted documents.

3. ON-LINE INDEX MAINTENANCE USING DYNAMIC BALANCING TREE

We have discussed index update strategies for continuously growing text collections in the previous section. These index update strategies are designed for documents insertions, without considering document deletions. In a truly dynamic environment, however, documents may be added to and removed from the collection at any point in time. For document insertions, the postings of newly added documents can simply be appended to existing in-memory posting lists (If memory is exhausted, the in-memory index is transferred to disk, resulting in a new on-disk sub-index, and the space is freed), without modifying the other on-disk sub-indices. Therefore, sub-index merging process can be done very efficiently if there is no document deletion, as posting lists do not have to be decompressed, and can simply be copied in their compressed form, avoiding expensive posting decompressing and re-compressing. This is not true for document deletions, as they can potentially affect the all sub-indices (include in-memory index). Their impact makes a large number of posting lists of terms have to be decompressed and re-constructed, which adds considerably to the complexity of document deletions. Therefore, the situation of index maintenance is more complicated for dynamic text collections than for growing text collections.

In this section, we first present a dynamic layout of sub-indices, and then give a definition of *Dynamic Balancing Tree* and propose a new on-line index maintenance strategy based on this tree that is designed for both document insertions and deletions.

3.1 Dynamic Layout Definition for Sub-Indices

Many index construction approaches for a given text collections divide the collection into partitions, whose size is determined by the amount of available memory. Documents are accumulated in main memory, and, once main memory is exhausted, the in-memory index is transferred to disk, forming a new sub-index. At any given point, the index is the concatenation of an in-memory index and a set of on-disk sub-indices, each of which contains a vocabulary component and a posting lists component, and can be seen as a partial index for a contiguous subset of the documents in the collection.

Suppose there is a set of N sub-indices I which is defined as:

$$I = \{i_0, i_1, \cdots, i_j, \cdots, i_{N-1}\}$$

Where $|i_p| \geqslant |i_q|$ if $p > q$ ($|i_p|$ is the size of sub-index i_p), and the order of elements in I is strictly defined. Previous methods merge sub-indices i_p and i_q only if $p = q + 1$ or $q = p + 1$. This scheme is inflexible and therefore makes it impossible to adopt some flexible index maintenance strategies. Moreover, when document deletions are existed during index construction, this scheme may be destroyed, as $|i_p|$ may be less than $|i_q|$ when $p > q$.

To amend this problem, we redefine I as:

$$I = \{i_{b_0}, i_{b_1}, \cdots, i_{b_j}, \cdots, i_{b_{N-1}}\}$$

Where $b_j (0 \leqslant j < N)$ is the radix of *global numbers* of i_{b_j}, each of which is the unified identifier of the document of sub-index i_{b_j}. We therefore assume

$$b_0 = 0 \; and \; b_{j+1} = b_j + |i_{b_j}| \tag{1}$$

We can see that the elements in I are ordered by global number, as the global numbers of the p-th sub-index are greater than the q-th sub-index if $p > q$. In addition, we assign a *local number* to each document in the sub-indices. If the local number of j-th document in p-th sub-index is $l_{p,j}$, and the global number is $g_{p,j}$, we let

$$g_{p,j} = l_{p,j} + b_p \tag{2}$$

We now use these local numbers to identify documents in the posting lists of sub-index. The local numbers of a sub-index keep unchanged, while the global numbers can be changed freely by changing the value of b_j (using equation 1 and 2). It means that, if required, the layout of sub-indices can be dynamically adjusted. This scheme has the benefit that the order of the sub-index sequence can be changed just by the re-assignment of b_j. A further benefit is that any two sub-indices can be merged in a merge operation by adjusting their global number radixes to let the global numbers of them be continuous (merge operation requires the global numbers of sub-indices that involved to be continuous). As each sub-index is an inverted index of a partition of the document collection, this adjustment actually changes the indexing order of partitions. The order of sub-indices can be changed freely means that merging events can be handled rather more flexibly than before.

3.2 Merging Based On Dynamic Balancing Tree

The key issue related to merge-based strategies is how best to manage the sequence of merge operations so as to minimize the total merging cost, without allowing the number of sub-indices to grow excessively. In order to make merge operation to be relatively efficient, the two sub-indices should not differ significantly in size [8]. This is a basic principle for efficiently merging of sub-indices.

From section 2 we know that the index maintenance strategies presented so far assume that the size of the sub-index which created from a merge operation is the sum of the size of the sub-indices involved in the merging process. Whereas, removing deleted documents from posting lists during a sub-index merging process may lead to a situation where the size of an index is smaller than it was. On the other hand, for these strategies, the order of sub-indices is strictly defined (see section 3.1), which means that a merge operation can only be performed on two adjacent sub-indices. Therefore, the two sub-indices involved in a merge operation may differ significantly in size, breaking the basic principle described above.

In order to offer a better support for document deletions and address this problem, we propose a new index maintenance strategy based on *Dynamic Balancing Tree*. A central idea underpinning the technique is to use a Dynamic Balancing Tree to manage and dynamically adjust the sequence of sub-index merge operations. During index construction, there must continue to be merging events, but they are handled by a predefined Dynamic Balancing Tree more strategically than before.

Dynamic Balancing Tree Merge

A *Dynamic Balancing Tree* (*DBT* for short) is an m-way tree, each node of which is a sub-index. We divide, from bottom to top, the nodes of the tree into H layers. At layer k, the number of nodes is either zero, or is less than $m (m \geqslant 2)$. We introduce a key parameter $c (c \geqslant m)$ to define the capacity of the node in each layer. Let $e_{k,j}$ be the capacity of node $j (0 \leqslant j < m-1)$ at layer k $(0 \leqslant k < H)$, a constraint of the number of documents in one node of layer k is given by

$$c^k \leqslant \frac{e_{k,j}}{s} < c^{k+1} \tag{3}$$

Where $s (s \geqslant 0)$ is the scale factor of tree node. Specially, when $s = 0$, the size of each leaf node of the merge tree is 1, in spite of the size of sub-index, and the size of non-leaf node is the number of child nodes it contains.

From equation 3 we know that the node in layer $k + 1$ is roughly c times bigger than the node in layer k, and that the any two nodes in each layer are not differ significantly in size. When the size of each node in layer k satisfy equation 3, the tree is balanced. If a DBT is balanced and a sub-index merge operation is only performed on one layer, the efficiency of merging process can be guaranteed (it follows the basic principle described in previous section).

The new strategy starts like any of the merge-based strategies described in section 2. During the indexing process, however, whenever main memory is exhausted and a sub-index will be written to disk, the sub-index, suppose i_{mem} $(e_{k,j} = |i_{mem}|)$, as a tree node, is dispatched to the layer k of DBT. k is determined by equation 3, that is

$$c^k \leqslant \frac{|i_{mem}|}{s} < c^{k+1} (s \neq 0)$$

$$\Rightarrow k \leqslant \log_c \frac{|i_{mem}|}{s} < k + 1 (s \neq 0)$$

Hence, we have

$$k = \lfloor \log_c \frac{|i_{mem}|}{s} \rfloor (s \neq 0) \tag{4}$$

If the dispatch leads to a situation where the number of nodes in the layer k is equal to or greater than m, a merge event will be triggered, resulting in a new sub-index. If there is no document deletion during index construction, the newly created sub-index will be placed into layer $k + 1$ or k (depends on equation 3). However, when a part of documents are deleted, the sub-index may be smaller than it was and makes the tree to be unbalanced (it dissatisfies equation 3). Therefore, for guaranteeing the efficiency of merging of its following sub-indices, it has to be pushed down to a lower layer to make the tree to be balanced. We refer this operation as *Dynamic Balancing Operation*, and refer this node (or sub-index) as *Push Down Node* (*PDN*). After a Dynamic

Balancing Operation, the global number radix (see section 3.1) of each existing sub-index has to be updated to let the global numbers of two adjacent sub-indices be continuous, which is required for a sub-index merge operation. Then, if, after the new sub-index has been dispatched, a new merge event is triggered again, it is combined with the previous merge event. This is repeated until there is no collision in each layer of DBT. Dynamic Balancing Operation, which is performed after a merge operation with garbage collection, reduces the potential for merge operations in which the sub-indices that involved are differ significantly in size. It therefore makes merge events always follow the basic principle for efficiently merging of sub-indices. When $s = 0$, the Dynamic Balancing Operation is turned off, as the size of sub-index only depends on the number of sub-indices that involved in the merge operation of that sub-index, without considering the actual number of documents contained in that sub-index.

The merge strategy based on *DBT* is referred to as *Dynamic Balancing Tree Merge* (*DBT Merge* for short). The garbage collection mechanism included in DBT Merge strategy is described in the following section.

Garbage Collection in DBT

For document deletions, the approach, which uses a bitmap to identify the deleted documents and filters out those deleted documents at query processing time, can be taken. In general, there are two different methods to integrate this approach into the merge-based index update strategies described in the previous section: those that keep deleted documents in the posting lists of the new merged index and those that do not. If deleted documents are kept, all bitmaps of sub-indices have to be combined to create a new bitmap for the new created index. Performing garbage collection during merging process, on the other hand, needs to filter out postings that belong to deleted documents and recompress and write those postings that belong to remained documents to the output index.

A straightforward approach to garbage collection is to remove deleted documents whenever there is a sub-index merge event. Since garbage collection requires expensive decompression and re-construction of posting lists, it dramatically reduces sub-index merging performance. It is therefore not desirable to perform garbage collection for every merge event. Like previous garbage collection methods, we keep track of the number of documents that have been deleted in each sub-index, and define a threshold value ρ ($\rho \in (0, 1]$, $\rho = 1.0$ means that no garbage collection is performed). For every merge event, two checks are performed to determine whether a garbage collection is integrated into the merging process or not:

1. Check the relative number of documents that have been deleted in each sub-index:

$$r' = \frac{\#deleted\ documents\ in\ sub-index}{\#total\ documents\ in\ sub-index}$$

   The sub-index whose relative number r' is greater than ρ is added to the set of input indices of a merge operation.

2. Check the relative number of documents that have been deleted in all input indices of the merge operation:

$$r = \frac{\sum_{j=0}^{k-1} \#deleted\ doucments\ in\ index\ i_{b_j}}{\sum_{j=0}^{k-1} \#total\ doucments\ in\ index\ i_{b_j}}$$

   where $i_{b_0}, i_{b_1}, \cdots, i_{b_{k-1}}$ are the input indices of the merge operation. The garbage collection is integrated into a merging process if $r > \rho$

Algorithm MergeIndexWithGC(*I, V*)
Input: a set of sub-indices $I=\{i_0, i_1,\ldots i_k\}$, a set of bitmaps $V=\{v_0,v_1,\ldots v_k\}$, which identify deleted documents of sub-indices *I*.
Output: a new merged sub-index
Method:

```
1.  Initialize priority queue Q of terms In I
2.  Create an empty on-disk sub-index i_merged
3.  WHILE Q is not empty
4.      t_j { t_j∈i_j } = pop(Q)
5.      W←{ t_j }
6.      WHILE (top(Q) = t_j)
7.          W← pop(Q)
8.      END WHILE
9.      FOR each entry t_j {t_j∈i_j } of W
10.         u←upper bound of document number in posting
                list p of t_j
11.         IF u < lower bound of v_j THEN
12.             copy p to i_merged
13.         ELSE
14.             decompress p
15.             filter out documents marked in v_j
16.             re-construct p and add it to i_merged
17.         END IF
18.     END FOR
19.     FOR each entry t_j {t_j∈i_j } of W
20.         put(Q, next term of i_j)
21.         clear W
22.     END FOR
23. END WHILE
24. return i_merged
```

Figure 1: Algorithm for merging sub-indices with garbage collection

Since in a merge operation which integrates with garbage collection, a part of terms of the input indices may not be contained in documents that have been deleted, it is not necessary to decompress the posting lists of those terms. Therefore, for every term of sub-indices, a further check is performed to see whether it is contained in deleted documents or not. If a term is not contained in deleted documents, the posting lists of that term are only copied to the new sub-index, avoiding expensive decompression; otherwise, the posting lists are decompressed, and, after filtering out postings belong to deleted documents, re-constructed to form a new posting lists. The algorithm of the merge operation which integrates with garbage collection is depicted in Figure 1.

This strategy is similar to the threshold-based method proposed by Büttcher [4], but our method offers better performance, due to the avoidance of decompression of posting lists that only belong to non-deleted documents.

Analysis by Example

For DBT Merge, the index construction can be seen as the construction of a DBT. Suppose, for example, that $m = c = 3$, $s = 0$ and $\rho = 1.0$, and each sub-index generated from in-memory index is of size 1. The first and the second sub-indices are written to disk and placed into layer 0. The

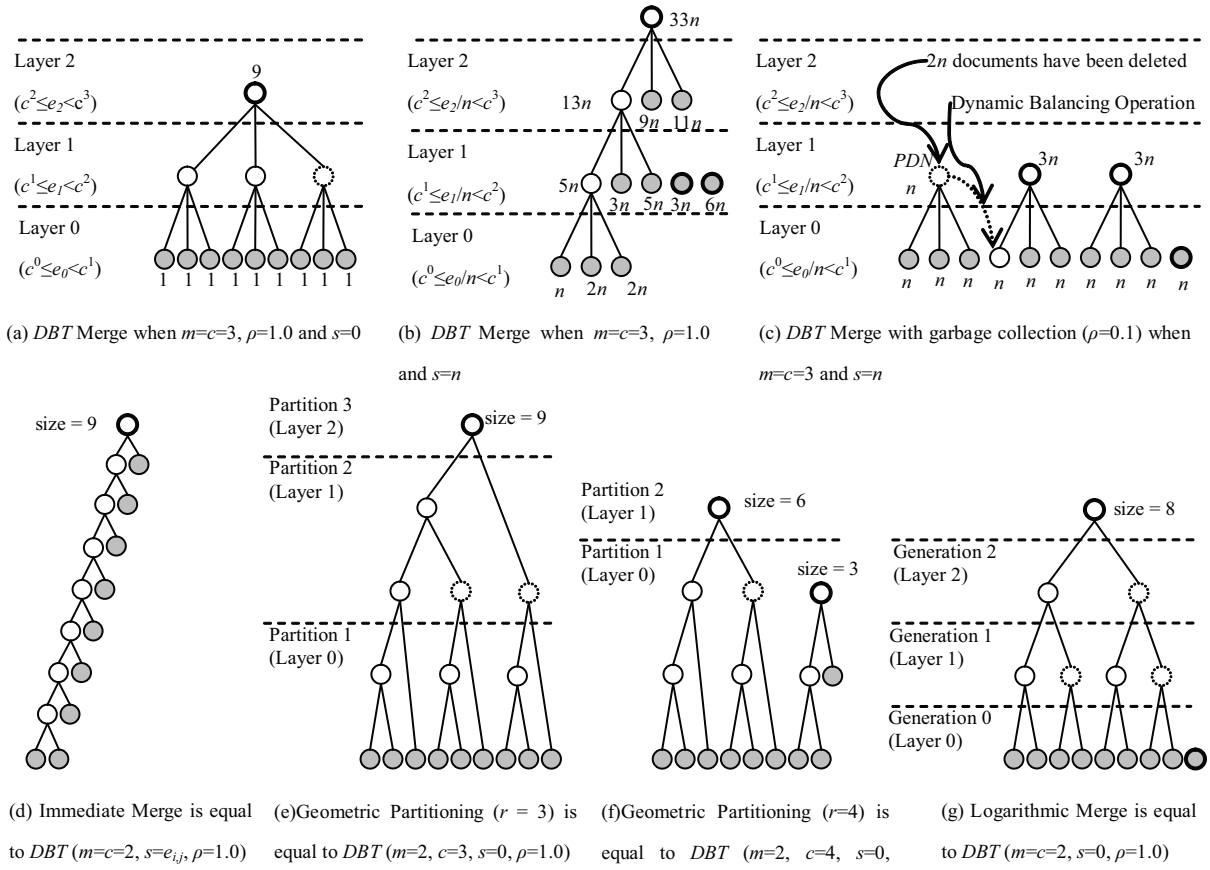

Figure 2: The Dynamic Balancing Trees for sub-index merging process. The dark leaf node is the sub-index which generated by in-memory index (from left to right). The dash-line node is the virtual sub-index which is not created actually. The thick-line node is the final sub-index.

third sub-index is merged with the first and the second sub-indices, creating a sub-index of size 3, which is placed into layer 1. Similar operations are performed when the fourth to eighth sub-indices are generated. Then, the ninth sub-index joins it and triggers a five-way merge, to make a sub-index of size 9. The merge process is depicted in Figure 2(a). As can be seen, the merging process is the construction of a complete three-way tree. Figure 2(b) shows a situation where $s = n$ and the sub-indices generated by in-memory indices are not of the same size. In this example, the first to third sub-indices are placed into layer 0, and the fourth to seventh sub-indices are placed into layer 1, and the eighth and the ninth sub-indices are placed into layer 2. The generations of the third, the fifth and the ninth sub-indices trigger three three-way merge. Finally, there are three on-disk sub-indices which exist in parallel. If an isolated node is seen as the root node, without children, of a m-way tree, then the merging process is the construction of three three-way tree. From the two examples, we know that, if $m = c$, the DBT consists of one or more m-way trees, and that, if $s = 0$, the m-way is actually a complete m-way. If no garbage collection is performed($\rho = 1.0$), there is no Push Down Node exist in the DBT.

In Figure 2(c), we show a more complex situation where $s = n$, $\rho = 0.1$ and $2n$ documents have been deleted before the generation of third sub-index. The third sub-index triggers a three-way merge, to make a new sub-index, which is placed into layer 1. As there are $2n$ documents have been deleted and the garbage collection threshold $\rho < \frac{2n}{3n} = \frac{2}{3}$, a garbage collection is performed. Therefore, the new sub-index created from merge operation is of size n. As $\frac{n}{s} < c^1(c = 3, s = n)$, the new sub-index is a Push Down Node. Hence, it is immediately pushed down to layer 0 by a Dynamic Balancing Operation.

We note that, for the existing merge-based strategies, the merging process can also be seen as a construction of DBT without Dynamic Balancing Operation. For example, when $r = 3$, Geometric Partitioning is equal to DBT Merge with parameter settings of $c = 3$, $m = 2$, $s = 0$ and $\rho = 1.0$ (depicted in Figure 2 (e)), and when $r = 4$, Geometric Partitioning is equal to DBT Merge with parameter settings of $c = 4$, $m = 2$, $s = 0$ and $\rho = 1.0$(Figure 2 (f)). Logarithmic Merge is a DBT Merge with parameter settings of $m = c = 2$, $s = 0$ and $\rho = 1.0$ (Figure 2 (g)). When $m = c = 2$, $s = e_{i,j}$ and $\rho = 1.0$, DBT Merge is equal to Immediate Merge (Figure 2 (d)). Therefore, DBT Merge provides a general and flexible framework for merge-based index maintenance strategies.

4. EXPERIMENTS

In this section, we construct several experiments to illus-

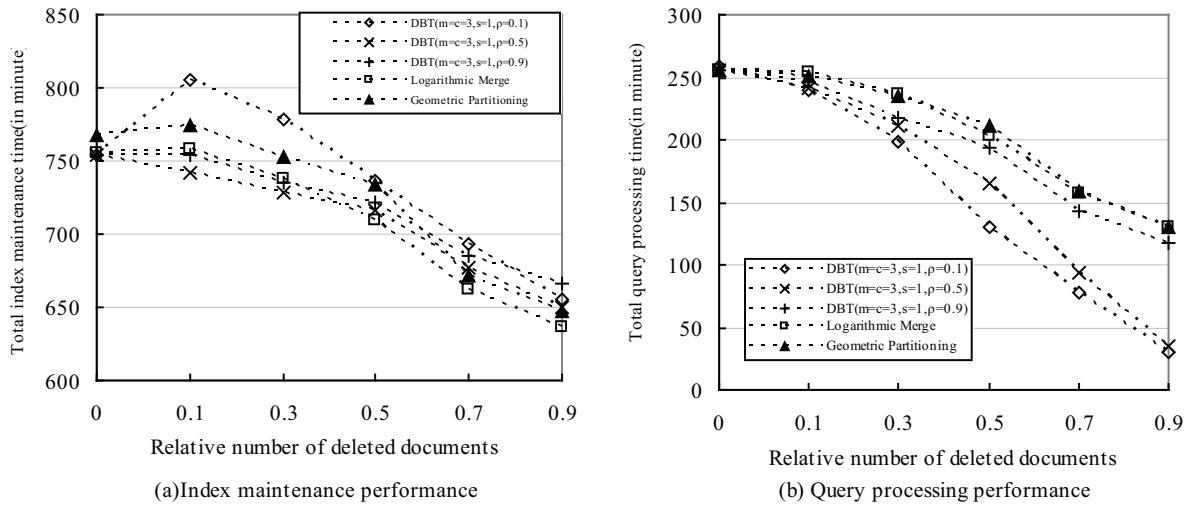

Figure 3: Index maintenance and query processing performance for different strategies with various parameter settings.

trate the efficiency of our DBT based index maintenance strategy. We implement two most popular merge-based online index maintenance strategies—Geometric Partitioning [8] and Logarithmic Merge strategies [4], and compare them with DBT Merge strategy with different values of parameter settings—DBT parameter m and c, scale factor s, garbage collection threshold ρ, relative number of deleted documents R, paying attention to both index maintenance and query processing performance. In-place and hybrid index maintenance strategies are not implemented here, because they introduce a rather complicated space management problem when document insertions and deletions are both considered.

4.1 Experimental Setup

For our experiments, we employed the GOV2 text collection used in the TREC Terabyte track[1]. GOV2 consists of 25.2 million documents with compressed size of about 81 GB (uncompressed size is 426 GB). The collection was indexed in a compressed form, using 512MB of main memory for the in-memory index. Times of index construction are elapsed times, including all reading, decompressing, parsing, indexing, and sub-indices merging phases. The query set employed here are 50,000 queries used in the efficiency task of the 2005 TREC Terabyte track.

For each experiment, we used a mixed update/search sequence consisting of 25,200 document insertions, 25,200 document deletions and 25,200 search operations, simulating a truly dynamic environment. For each search operation, 100 queries are processed. So, there are 2,520,000 queries (repeatedly taken from the query set described above) processed in each experiment in total. In every document insertion, 10,000 documents are inserted and indexed. For each document deletion, we vary the number of deleted documents. During the index construction, all words in the collection are changed to their lower case, no stop word is removed, no stemming algorithm is employed, and full positional information for all terms in the collection is recorded. Search queries are processed by using document-at-a-time (DAAT) algorithm [1, 2, 7], and no caching is performed by the information retrieval system itself.

The experiments were run on a Linux PC based on 4 AMD Opteron2 2.2GHZ CPU with 8 GB of main memory, 15,000-rpm SCSI hard drivers and 7,200-rpm IDE hard drives. The input documents were read from a 15000-rpm SCSI hard driver. Indices were written to a 7,200-rpm IDE hard driver.

The experiments described in this paper have been implemented in the FirteX information retrieval system, written in C++ code and is available under a GPL license.[2]

4.2 Results

In our experiments, we use our retrieval system to process the mixed update/search sequence, building an index for GOV2 and concurrently processing 2,520,000 search queries and 25,200 document deletions. We allow the system to use 512 MB of main memory for the in-memory index, and 512 MB of main memory for reading, decompressing, and storing posting lists during query processing. We vary the relative number of deleted documents R for the three different merge strategies and vary the parameters of our method to analyze index maintenance and query processing performance. For Geometric Partitioning strategy, the parameter $r = 3$ is chosen.

Figure 3(a) shows the index maintenance time for various relative numbers of deleted documents. As can be seen, when a part of documents are deleted and when the threshold value $\rho > 0$, the index maintenance performance is reduced slightly, due to the impact of garbage collection on merge operations. When $m = c = 3$, $s = 1$, $\rho = 0.1$ and the relative number of deleted documents $R = 0.1$, compared to Geometric Partitioning strategy, the index maintenance performance of DBT Merge drops by 4%. As R increases, the index maintenance performance of DBT Merge strategies ($\rho > 0$)improves, and is very close to or even better than

[1]http://www-nlpir.nist.gov/projects/terabyte/

[2]Available from http://sourceforge.net/projects/firtex or http://www.firtex.org

other strategies which not integrates with garbage collection (Logarithmic Merge and Geometric Partitioning). For example, When $m = c = 3$, $s = 1$ and $\rho = 0.5$, compared to Geometric Merge strategy, the index maintenance time of DBT Merge drops by 2% on average. It indicates that, for DBT Merge strategy, the index construction for dynamic text collections is as fast as for growing text collections. On the other hand, the query time in Figure 3(b) shows that our strategy exhibits significantly better query processing performance than other strategies when there are more than 30% documents are deleted during index construction. For instance, when $m = c = 3$, $s = 1$, $\rho = 0.1$ and $R = 0.9$, DBT Merge processes the queries 77% faster than Geometric Partitioning or Logarithmic Merge strategy, and index maintenance performance only drops by 1.2% (Logarithmic Merge:3%). The results suggest that postings that belong to deleted documents really should be removed in order to improve query processing performance when more than 30% documents are existed in indices. For DBT strategy, greater ρ values mean less garbage collect operations and thus better indexing performance. When $\rho = 0.5$, it improves query processing performance by 20%, while achieving a acceptable indexing performance. The results are consistent with earlier findings [4] and show that, for most update loads, $\rho = 0.5$ is a safe choice, leading to an acceptable index maintenance performance and a better query processing performance.

For a monotonically growing text collection, DBT strategy improves the system's performance only marginally, indicating that the Geometric Partitioning and Logarithmic Merge strategies are suited very well for growing text collections. We note that, compared to Geometric Partitioning strategy, Logarithmic Merge strategy offers a slightly better index index maintenance performance.

In Figure 4, we add some more parameter settings for DBT Merge strategy and show the pure index construction time during index maintenance phrase, which includes the time for building in-memory index and the time for merging of sub-indices. The results depicted in Figure 4 show that, the integration of garbage collection into the merge process deteriorates the indexing performance. For the garbage collection threshold $\rho = 0.1$, the indexing performance of DBT Merge drops by roughly 20% when the relative number of deleted documents is varied from 0 to 0.1. Similar behavior is also present in Figure 3(a). For DBT strategy, greater ρ values mean less garbage collect operations and thus better indexing performance. Table 1 shows the extract reduction percentage of indexing performance. The results indicates that, although the cost of processing of document deletions is rather more expensive than that of document insertion, the indexing performance of DBT Merge strategy reduces only insignificantly, and that the garbage collection mechanism in DBT Merge strategy is efficient.

Figure 4 and 3(b) also show that, compared to DBT's parameter settings of $m = 2, c = 3$ and $s = 0$, the parameter setting of $m = c = 3$, and $s = 1$ offers better indexing performance, while query processing performance keeps unchanged. For instance, when $\rho = 0.1$, average indexing performance improves by 10% (from Figure 3).

5. CONCLUSIONS

In this paper we examine issues of on-line index construction for dynamic text collections that supports instantaneous document insertions and deletions. We propose a

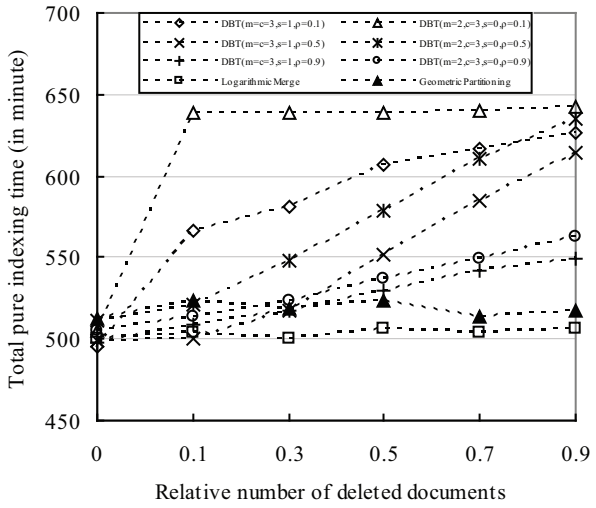

Figure 4: Pure index construction time during index maintenance phrase, which includes the time for building in-memory index and the time for merging of sub-indices.

Table 1: Exact reduction percentage of indexing performance for the results shown in Figure 4 (compare to Logarithmic Merge and Geometric Partitioning strategy). The reduction is caused by garbage collection.

	Log. Merge	Geo. Part.
DBT Merge (m=c=3,s=1,ρ=0.1)	15.5%	12.3%
DBT Merge (m=2,c=3,s=0,ρ=0.1)	19.1%	19.1%
DBT Merge (m=c=3,s=1,ρ=0.5)	5.1%	5.1%
DBT Merge (m=2,c=3,s=0,ρ=0.5)	12.7%	9.5%
DBT Merge (m=c=3,s=1,ρ=0.9)	4.0%	1.1%
DBT Merge (m=2,c=3,s=0,ρ=0.9)	5.6%	2.6%

DBT Merge strategy that can dynamically adjust the sequence of sub-index merge operations during index construction. Our strategy, along with an efficient garbage collection mechanism, offers better query processing performance than existing merge-based strategies, while providing an equivalent level of index maintenance performance when document insertions and deletions exist in parallel. Since existing merge-based strategies are a particular case of our method, it also provides a general and flexible framework for merge-based index maintenance strategies.

In our experiments, using optimal parameter settings, our method can process queries roughly 30 percent faster than previous methods on average, and index maintenance time only increases by 2 percent. This demonstrates that on-line index construction for dynamic text collections, in which document insertions, document deletions and search operations are interleaved, can be performed efficiently and almost as fast as for growing text collections.

The main disadvantage of the new strategy is that the cost of document deletions is still very expensive, due to the decompression and re-construction of posting lists that partially belong to deleted documents. The main focus on our future work in this area will be to investigate other opti-

mization technology that achieves a further improvement of both index maintenance and query processing performance.

6. ACKNOWLEDGMENTS

This work was mainly supported by special fund of Chinese Academy of Sciences, "Research on Opinion Mining of Web Text", under grant number 0704021000 and two projects, i.e., 2007CB311100 and 2004CB318109.

7. REFERENCES

[1] S. Brin and L. Page. The anatomy of large-scale hypertextual web search engine. In *Proc. 7th International WWW Conference*, pages 107–117, 1998.

[2] A. Broder, D. Carmel, M. Herscovichi, A.Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *In Twelfth International Conference on Information and Knowledge Management (CIKM 2003)*, pages 426–434. New Orleans, LA, USA, November 2003.

[3] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th ACM Conference on Research and Development on Information Retrieval (SIGIR 2006)*. Seattle, USA, August 2006.

[4] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*. Bremen, Germany, November 2005.

[5] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. *Technical report,Stony Brook*, August 1998.

[6] S. Heinz and J. Zobel. Efficient single-pass index construction for text database. *Jour. Of the American Society for Information Science and Technology*, 54(8):731–729, 2003.

[7] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. acm transactions on information systems. *ACM Transactions on Information Systems*, 17(4):406–439, October 1999.

[8] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the ACM CIKM Conference on Information and Knowledge Management*, pages 776–783, November 2005.

[9] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for text retrieval systems. *Information Processing and Management*, 42(4):916–933, July 2006.

[10] W. Shieh and C. Chung. A statistics-based approach to incrementally update inverted files. In *Proc. Int. Conf. on Information and Knowledge Engineering*, pages 38–43. Las Vegas, Nevada, June 2003.

[11] K. Shoens, A. Tomasic, and H. Garia-Molina. Synthetic workload performance analysis of incremental updates. In *Proc. International ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 329–338. Dublin, Ireland, July 1994.

[12] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 289–300. New York, USA, 1994.

[13] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, second edition.* Morgan Kaufmann Publishing, San Francisco, California, 1999.

[14] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), June 2006.

[15] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.