On-line Index Maintenance Using Horizontal Partitioning

Sairam Gurajada Sreenivasa Kumar P

Indian Institute of Technology Madras
Chennai - 600036

{sairam,psk}@cse.iitm.ac.in

ABSTRACT

In this paper, we propose a new merge-based index maintenance strategy for Information Retrieval systems. The new model is based on partitioning of the inverted index across the terms in it. We exploit the query log to partition the on-disk inverted index into two types of sub-indexes. Inverted lists of the terms contained in the queries that are frequently posed to the Information Retrieval systems are kept in one partition, called frequent-term index and the other inverted lists form another partition, called infrequent-term index. We use a lazy-merge strategy for maintaining infrequent-term sub-indexes, and an active merge strategy for maintaining frequent-term sub-indexes. The sub-indexes are also similarly split into frequent and in-frequent parts. Experimental results show that the proposed method improves both index maintenance performance and query performance compared to the existing merge-based strategies.

Categories and Subject Descriptors: H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing—*Indexing Methods*; H.3.2 [Information Storage and Retrieval]: Information Storage—*File Organization*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*Search Process*

General Terms: Algorithms, Performance

Keywords: Inverted File, Inverted Index, Search Engine, Query Log

1. INTRODUCTION

Inverted Index is an important data structure used in many Information Retrieval (IR) systems. A list of terms, with their postings list comprises the inverted index. A term's postings list is a sequence of nodes, where each node contains the docID of the document containing the term, and a list of positions of the term in the document. In general, a query to an IR system is a list of terms along with constraints. The result is a list of documents obtained by applying the constraints given in the query, which is then ordered by a ranking method. While processing a query, the IR system has to retrieve the postings list from index for every query term. The length of the postings list retrieved can vary depending on how the documents are sorted. If they are sorted based on the increasing document IDs, the IR system has to retrieve entire postings list. If they are sorted based on the impact order, then only top-k postings of the term's posting list are retrieved. Witten et al. [17] gives an introduction to inverted lists, query modes, and ranking methods.

The off-line indexing approach deals with handling static document collection. During the indexing process, the document collection is unchanged and the queries are handled only after the entire index is built. Off-line indexing has been addressed in the past and efficient strategies were proposed [8].

On the other hand, the on-line indexing approach handles dynamic document collection. Any on-line indexing strategy has a capability to modify its index according to the documents that are being added to or deleted from document collection, and the queries are processed while the indexing is being performed. This type of indexing is needed for the cases where queries should be processed readily after a document is being indexed, eg., news search.

Many algorithms were proposed in the recent past to perform indexing dynamic document collections. Initially [2, 6, 16] addressed the problem of on-line indexing. A decade after, Lester et al. [14] readdressed the problem and gave a comparative study of three basic approaches: In-place, Re-Merge, and Re-Build, and showed that In-place and Remerge outperform Re-build in all cases. Lester et al. [9], and Büttcher et al. [4] proposed a new merge-based strategy where they trade-off query performance with index maintenance performance by having a controlled merging of on-disk sub-indexes. Tomasic et al. [16], and Büttcher et al. [4] proposed an hybrid approach and showed that performance of retrieval system is improved when the short length postings lists are handled using merge-based strategy and long length postings lists by In-place strategy. The methods for handling the document deletion in the on-line indexing environment are discussed in [3, 5, 7].

The systems that handle dynamic collections are called Dynamic Information Retrieval systems, where they efficiently process document insertions and deletions, and simultaneously serve query requests. These systems process every query as a new and independent one. In real world, the documents that are added to the document collection are not repeated, whereas the queries are repeated. Generally, queries that cover broad topics occur more frequently. For example, consider two queries *Information Retrieval* and *On-line Index*, the former query appears more frequently than later. And the individual terms *Information*, *Retrieval*, *On-line*, and *Index* are very frequently present in queries compared to the original queries. For a given query, the IR system's retrieval performance depends on its index partitioning strategy; i.e., the number of partitions to be read to build entire postings lists for the terms of the given query.

In this paper, we propose a new merge-based on-line indexing strategy, where the on-disk index is composed of frequent-term and infrequent-term sub-indexes. A frequent-term sub-index consists of high frequency query terms, and an infrequent-term sub-index consists of low frequency query terms. We employ different merge strategies for maintaining frequent and infrequent sub-indexes. A lazy merge strategy for infrequent sub-indexes improves index maintenance performance, and an active merge strategy for frequent sub-indexes improves query performance.

Alternatively, the query performance of an IR system can be improved by using an index cache that holds the postings lists of few frequently occurring query terms. Although such approach improves query performance, it incurs additional cost for cache index update, along with the main index update, whenever a new document containing high frequency terms is added to the IR system. The proposed merge-based on-line indexing strategy overcomes such overhead.

The rest of the paper is organized as follows: Section 2 gives background and related work on various indexing strategies, including details on merge-based and in-place strategies for on-line indexing. In Section 3, we explain our proposed Horizontal Partitioning approach for efficient on-line index maintenance. Section 4 gives a performance study of our approach and the state-of-the-art on-line indexing approaches. We conclude in Section 5.

2. BACKGROUND AND RELATED WORK

This section gives a brief background on the general method of off-line indexing approach, and an overview of work done in the recent past on dynamic information retrieval systems.

2.1 Off-line Indexing

As described in the previous section, the off-line indexing techniques are implemented to process static document collections. The indexing process starts by tokenizing the input documents forming a list of <term,doc> pairs. The list is sorted lexicographically across terms. All the pairs that have the same term are merged to form a list of docs for each term. All the process is carried out in main-memory if the document collection can be processed in main memory. If the document collection is too large to process in the main-memory, it is split into smaller collections of manageable sizes, which could then be easily processed in the main-memory. An inverted index is built for each smaller document collection. All these indexes (sub-indexes) are later merged to form the final index using a multi-way merge approach.

Query performance is the main motive behind merging a group of sub-indexes into single large index. In general, query to an index triggers an operation of preparing a contiguous postings list for each query term. But, if the term's postings are scattered across multiple locations, its retrieval requires many disk seeks, thus reducing the query performance. In this regard, off-line index provides a very high query performance by building a single large inverted index containing contiguous postings list for each term, but at the cost of having to build the index prior to handling queries.

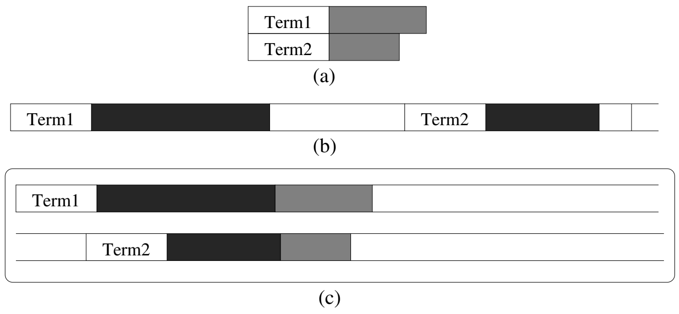


Figure 1: In-place update strategy. (a) represents the new postings lists of in-memory index. (b) represents state of on-disk inverted index. (c) represents in-place update with relocation for *Term2* and no relocation for *Term1*.

2.2 On-line Indexing

On-line indexing techniques are implemented to provide the capability of handling various index operations simultaneously. The operations include adding new documents to index, deleting existing documents from index, and serving query requests. On-line index consists of two parts: one resides in main-memory (also called *in-memory index*), and the other resides on disk. When a new document is added, the in-memory index is updated at first. And as the in-memory index grows, and eventually becomes too big to fit the specified size in the main memory, it will be moved to the disk using one of the three strategies: i) Re-build, ii) In-place, and iii) Re-merge.

Re-build

In Re-build indexing model, whenever the in-memory part of index exceeds the size limit, a new index is rebuilt from the scratch for the entire document collection. The existing index is used for processing queries, and then discarded once the new index is available for querying. Lester et al. [14] experimentally show that re-build strategy is less efficient than in-place and re-merge models in all non-trivial cases. Though the Re-build model is a very simple and inefficient on-line indexing approach, it is still used widely for its higher query performance.

In-place Update

While moving in-memory index on to the disk using in-place update technique, the postings of terms in in-memory index are appended to those of terms in on-disk index. Unlike re-build indexing model, this strategy avoids rebuilding of entire index from scratch. To avoid relocation of term's appended posting list, the on-disk index usually has a over-allocation of space for each term to accommodate future postings. If the term's over-allocation is exhausted, its in-memory postings list and on-disk postings list are concatenated, and then moved to a new location on the disk. Over-allocation strategy incurs some space overhead, but reduces the expensive frequent relocations of concatenated lists.

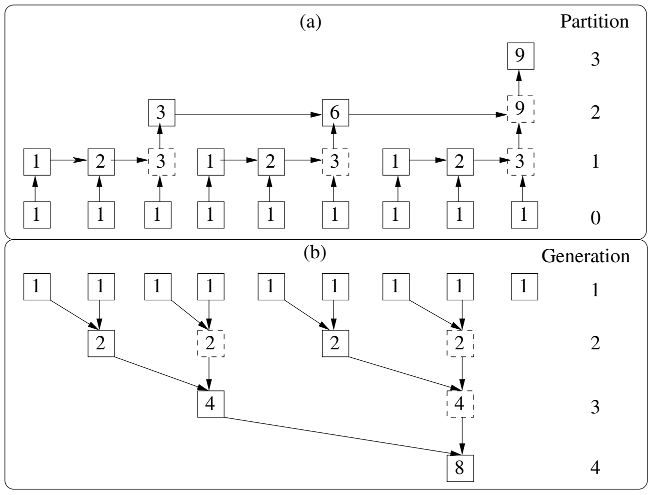


Figure 2: Multi-partition Strategies. (a) Overview of Geometric Partitioning with $r = 3$. (b) Overview of Logarithmic Merge ($k = 2$). The dashed nodes represent intermediate sub-indexes formed while merging.

An ineffective over-allocation strategy may result in poor update performance. For example, small over-allocations may cause frequent relocation of terms' concatenated lists, and large over-allocations may cause space overhead. To tackle this problem, Shoens et al. [13] (and also Shieh et al. [12]) proposed a predictive over-allocation strategy for improving the performance of In-place indexing model. And Lester et al.[10] proposed another solution by maintaining an inverted file for each term. However, the problem of fragmentation is left to the file system.

An illustration of In-place update is shown in Figure 1. Part(a) represents the new postings lists of in-memory index for *Term1* and *Term2*. And part (b) represents state of on-disk inverted index. When *Term1* and *Term2* are moved to disk, an attempt is made to fit them into over-allocation space. Here, *Term1's* new postings list can be easily fit into the over-allocation space, so it doesn't require relocation, whereas *Term2's* new postings list can't be fit, so it is relocated. Part (c) shows In-place update with relocation for *Term2* and no relocation for *Term1*.

Merge-based Update

Merge-based strategies maintain two indexes, one that is in main memory and the other resides on disk. When the in-memory index exceeds its limit, it is merged with on-disk index. The simplest form of merge-based update is *Immediate Merge*, described by Cutting et al. [6]. In this approach, only one on-disk inverted index exists. Here, the in-memory index is merged with existing on-disk inverted index forming a new index which replaces the existing one. *Immediate Merge* achieves high query performance because the on-disk index has contiguous postings list. But this approach requires high index maintenance cost as the entire on-disk index is read from the disk for each merge event. Results from Lester et al. [14] show that *Immediate Merge* outperforms in-place indexing technique in all practical scenarios, despite its high cost merge event.

Later, multi-partition merge strategy was proposed [9, 16], where more than one sub-index exist on the disk simultaneously. A sub-index is also an inverted index, but has only partial postings for the terms. A group of sub-indexes comprise an on-disk index. Since multi-partition strategies require few merge events, they achieve higher index maintenance performance compared to single partition strategies. But, the query performance of multi-partition strategies get degraded because of the scattering of postings list across multiple partitions.

In 2005, Lester et al. [9] proposed a merging strategy called *Geometric Partitioning* technique based on the multi-parti-tioning approach. Like other merge-based strategies, *Geometric Partitioning* employs a temporary in-memory index and persistent on-disk index. The in-memory index is moved on to the disk by cascading merges of it with the on-disk sub-indexes.

In *Geometric Partitioning*, each partition of the on-disk index contains a sub-index, whose size is less than maximum sub-index size defined for that partition. Here, the defined partitions form a geometric sequence over their maximum sub-index sizes, and the geometric ratio for this sequence is r. The geometric ratio r is defined such that, if the maximum sub-index size at k^{th} partition is S, then maximum sub-index size at $(k+1)^{th}$ partition is rS. Partition 0 is reserved for in-memory index block. When the in-memory index at partition 0 of size S is full, it's merged with the sub-index at partition 1. If the sub-index held at partition 1 reaches it maximum size of $(r-1)S$ then it is either moved to or merged with the sub-index at partition 2. The sub-index from partition 1 is merged with sub-index at partition 2 until the size of the sub-index at partition 2 reaches $(r-1)rS$, in which case the sub-index at partition 2 is moved to partition 3. To generalize, a sub-index at partition k is merged with sub-index at partition $k+1$ until the size of sub-index at partition k reaches its maximum size $(r-1) * r^{k-1}S$. At any instant, the size of the sub-index at partition k can be expressed as

$$ir^{k-1} \quad \text{for } 0 \leq i < r \tag{1}$$

The geometric merge operation is shown in Figure 2(a). The first in-memory sub-index of partition 0 is moved to partition 1. The next in-memory index from partition 0 is merged with sub-index at partition 1, now partition 1 holds the sub-index of size $2S$, where S is size of in-memory index. For the third in-memory index, it is merged with the sub-index at partition 1. Now the sub-index at partition 1 has exceeded its maximum limit $(r-1)S$, i.e $2S$. So, the merged sub-index is moved to partition 2. The handling of next in-memory blocks is shown in Figure 2(a).

Büttcher's *Logarithmic Merge* [3] is similar to Geometric partitioning technique. In *Logarithmic Merge*, every sub-index is given a unique number g, called generation number. A sub-index is said to be of generation $g+1$, if it is created by merging all the sub-indexes of generation g. In *Logarithmic Merge*, when the in-memory index is moved on to the disk, it is given a generation number 0. A merge event is triggered if more than one sub-index with the same generation number g form a sub-index of generation $g+1$. This leads to creation of sub-indexes whose sizes are exponentially increasing. At any instant, the maximum number of sub-indexes forming the inverted index for the collection is logarithmic of N, where N is number of in-memory blocks created so far. The similar

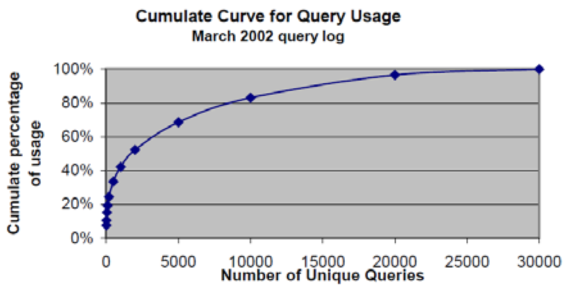


Figure 3: AOL Query log analysis: Cumulative Query Usage; obtained from technical report[2].

idea of logarithmic merge was implemented in LUCENE [1] by D. Cutting.

Figure 2(b) describes logarithmic merge. The first in-memory block, while moving to disk, is given generation number 1. When the next in-memory block arrives, it is also given generation number 1. Since two sub-indexes are of generation 1, they are merged to form a sub-index whose generation number is 2. When next in-memory block arrives, it is given generation number 1. Since there exist no sub-indexes of the same generation, no merge events occur. The process of handling the next few in-memory indexes are shown in Figure 2(b).

Strohman et al. [15] described an approach using splitting of terms into frequent and infrequent classes, based on their frequencies in the document collection. They proposed that keeping the frequent terms vocabulary information in main memory can decrease the latency of indexing documents. In contrast to this method, the approach proposed in this paper classifies terms into frequent and infrequent classes based on their frequency in typical query log. With this approach, the real-world queries can be handled more efficiently.

3. PROPOSED WORK

Merge-based strategies perform on-disk index updates by merging it with in-memory index. Whereas, the in-place strategy updates only those terms that have new postings in the in-memory index, with the cost of expensive relocations. Both of these on-line approaches trade-off between index maintenance performance and query performance. Generally, query consists of one or more terms, and the query performance is determined by the time taken to retrieve the postings list for each query term. We often observe that a small subset of queries occur frequently in a set of real-world queries. If the queries are broken down into individual terms, the frequency of terms will further increase. G Pass et al. [11] made an analysis on real-time AOL query log[1], as shown in Figure 3 (obtained from their technical report[2].) It presents the relationship between number of unique queries and cumulative query frequency. Here, the cumulative query frequency gives the percentage of the query log that gives rise to the specific set of unique queries. For example about $1/3^{rd}$ of queries comprise nearly 80% of the query log. Thus, the performance of IR system can be largely improved by tuning the index to handle frequent queries efficiently.

[1]http://research.aol.com, At present inactive.

[2]Query Log Analysis, www.ir.iit.edu/∼abdur/publications /QueryResearch.pdf

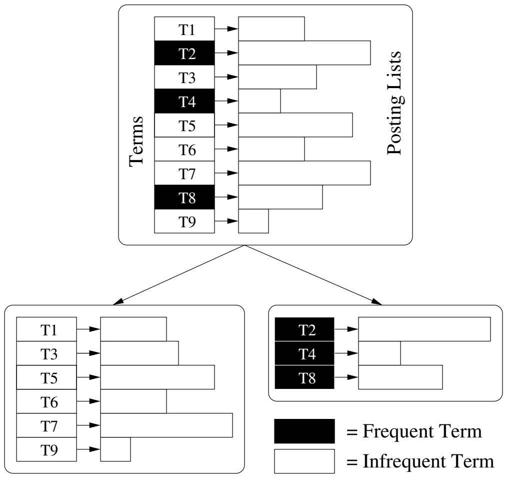


Figure 4: Horizontal Partitioning applied on an index to split into frequent and infrequent sub-indexes.

3.1 Horizontal Partitioning

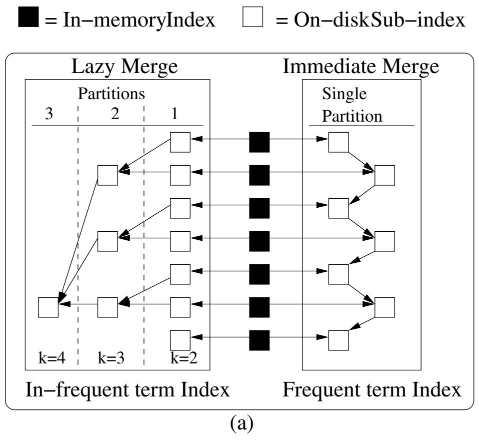
On-line indexing follows the general approach of Vertical Partitioning, where the terms' postings lists are split into multiple partitions, each containing a sub-index. Here, the terms get shared across the sub-indexes. Additionally, we propose Horizontal Partitioning approach, in which, the basic partitioning strategy is to split the index into two sub-indexes. In this approach, the terms do not get shared across the sub-indexes. In our work, we adopted a partitioning approach based on the frequency of terms derived from the query log. High-frequency terms form the frequent-term sub-index, and low-frequency terms form the infrequent-term sub-index. Figure 4 shows the basic Horizontal Partitioning applied on an index containing terms $T1$ to $T9$. The index is split into frequent-term sub-index having terms: *T2, T4,* & *T8*, and infrequent-term sub-index having terms: *T1, T3, T5, T6, T7,* & *T9.*

The main advantage of Horizontal Partitioning is that we can adopt more than one merge strategy for sub-indexes, and maintain them independently. The frequent-term sub-indexes are maintained with merge strategy that attains better query performance, for efficiently serving high-frequency queries. And the infrequent-term sub-indexes are maintained with merge strategy that attains better index update (maintenance) performance. By utilizing horizontal partitioning approach, we can achieve an overall higher index maintenance performance and higher query performance.

3.2 Index Maintenance

In our approach, the index is maintained by employing an *active merge* approach for frequent-term sub-indexes, and a *lazy merge* approach for infrequent-term sub-indexes. In active merge approach, the sub-indexes are frequently merged to reduce the number of on-disk sub-indexes to improve the query performance. But, in lazy merging approach, the sub-indexes are less frequently merged to save the cost of merging, which inturn gives a good update performance. An example of active merging is the geometric partitioning strategy: "varying r fixed p approach", mentioned in [9].

For lazy merging, we adopted a variant of logarithmic merge approach. The logarithmic merge proposed by Büttcher



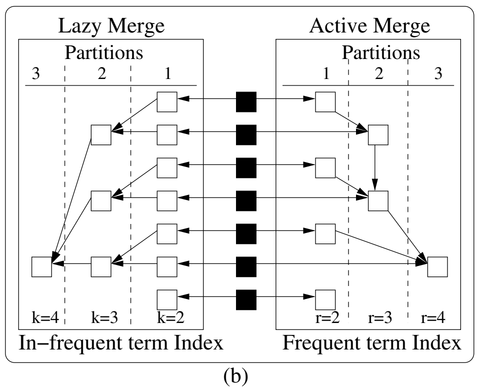


Figure 5: An overview of (a) Single-split Immediate Merge approach, and (b) Single-split Multi-partition approach.

et al. [4] constraints the number of sub-indexes, with same generation number, to be 2. Generalizing this, a logarithmic merge with constraint k allows k sub-indexes of the same generation number to exist simultaneously, before the merge event is triggered. A constraint k logarithmic merge does more number of merges than constraint $k + 1$ logarithmic merge. So, constraint $k + 1$ logarithmic merge is lazier and gives higher index maintenance performance compared to constraint k logarithmic merge. Inversely, the constraint k logarithmic merge results in lesser number of sub-indexes compared to constraint $k + 1$ logarithmic merge, thereby giving better query performance. Therefore, the lazier the merge approach is, the higher its index maintenance performance and lower its query performance.

Based on the idea of horizontal partitioning, we propose two indexing strategies: 1) Single-split indexing, and 2) Multi-split indexing approaches.

3.2.1 Single-split indexing approach

In this approach, we perform one-level splitting of the index into frequent-term sub-indexes and infrequent-term sub-indexes. As described in previous sections, we maintain these sub-indexes using different merge strategies.

The infrequent-term sub-indexes are spread across multiple partitions, and are maintained by lazy merge approach. Here, we adopted a variation of logarithmic merge with variable constraint k, where the value of k increases with the number of partitions, i.e., at partition p, we allow k sub-indexes of the same generation number to exist simultaneously, and we allow $k + 1$ sub-indexes at partition $p + 1$. For example, for partition 0, if we have two ($k = 2$) sub-indexes, then for partition 1, we will have three ($k = 3$) sub-indexes, and so on. By increasing k as partition number increases, we can achieve lazy merging of infrequent-term sub-indexes; thereby improving the overall index maintenance performance.

Here, we define two Single-split indexing approaches: 1) Single-split Immediate Merge, and 2) Single-split Multi-partition.

Single-split Immediate Merge

In *Single-split Immediate Merge*, the infrequent-term sub-indexes are maintained using *Lazy merge* approach, as described in the previous paragraph. The frequent-term sub-indexes are maintained in a single partition using *Immediate Merge* approach. This provides high query performance, but results in poor index maintenance performance. However, the index maintenance performance is better than the naïve *immediate merge* strategy performed without horizontal partitioning. Figure 5(a) represents the *Single-split Immediate Merge* approach. The in-memory index is split across terms, and merged with frequent-term and infrequent-term indexes located on the disk.

Single-split Multi-partition

The efficiency of *Single-split Immediate merge* approach degrades when the size of the frequent-term index increases, due to its high index maintenance cost. In such cases, the use of multi-partition strategy for frequent-term index is a better alternative. The multi-partition strategy provides better index maintenance performance with a slight degradation in query performance. A *Single-split Multi-partition* strategy uses a multi-partition active merge for maintaining frequent-term sub-indexes and a lazy merge approach for maintaining infrequent-term sub-indexes. An active merge approach employs geometric partitioning technique, where the value of r, geometric ratio, increases with the value of p, partition number. Figure 5(b) represents the Single-split multi-partition merge approach. Here, the frequent-term index is maintained using multi-partition active merge (*Geometric Partition*) strategy, and the in-frequent-term index is maintained using lazy merge.

3.2.2 Multi-split indexing

In Single-split indexing approach, the index is split into frequent-term and infrequent-term sub-indexes. However, as the size of these sub-indexes increases, the cost of index maintenance will also increase. So, we devised *Multi-split* indexing approach, where the sub-indexes are also similarly split into frequent and in-frequent parts. Frequent parts are maintained by an active merge approach, and in-frequent parts by a lazy merge approach. Alternatively, *Multi-split indexing* approach is a Single-split indexing approach applied at multiple levels of indexing process.

Index Tree

The recursive splitting of sub-indexes can be represented by a tree like data structure, called *Index Tree*. The index tree is a binary tree, where each node of the tree contains a set of partitions, and each partition contains a sub-index. For each

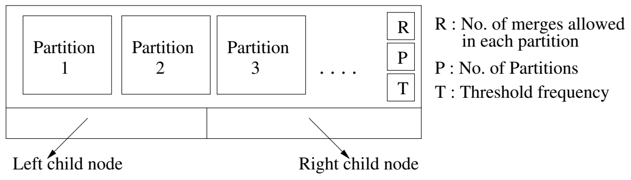


Figure 6: Internal details of a node in Index Tree.

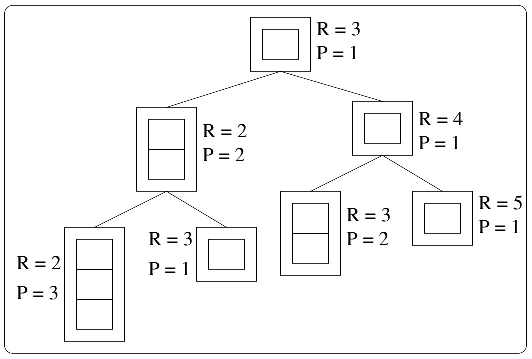


Figure 7: Index Tree with root node initialized to $R = 3$ and $P = 1$.

node there are three parameters defined: P, R, and T. Figure 6 shows a typical node structure of the *Index Tree*. Here, P and R define the merge policy to be followed at each node, where P specifies number of partitions, R specifies maximum number of merges allowed in each partition. T specifies the threshold value that is used to categorize frequent and infrequent terms. The values of P and R are initialized at the time of creation of the node, and are kept constant over the entire indexing process. For each partition in a node, we have a parameter r initialized to R, at the beginning. The parameter r specifies the number of merges allowed in the partition before moving the sub-index to next partition.

Building the Index Tree

Initially, when the index tree is empty, a root node is created with the values of R and P initialized to user defined values. For each partition in the root node, the value of parameter r is set to R. For every node, the minimum value of R is 2, and the minimum value of P is 1. This means that, each node has at least one partition ($P = 1$), and each partition has a sub-index that handles at least two-merge events ($R = 2$). Whenever the in-memory index is exhausted, it is moved on to the disk, and merged with the sub-index held at partition 0 of the root node. When r number of merges happened in partition 0, the sub-index is merged with the sub-index at the next partition present in the same node, or a simple move happens when the next partition is empty. If there is no additional partition in the node, the sub-index is split into frequent-term sub-index (positioned as right child) and infrequent-term sub-index (positioned as left child). The splitting strategy is determined by threshold T, which is computed based on the frequency of terms occurring in the query log. A node in an *Index Tree* is split after $R \times P$ merges.

For newly created child nodes, the values of R and P are initialized based on the values of R and P of their parent node. The left-child nodes hold infrequent-term sub-indexes, and right-child nodes hold frequent-term sub-indexes. As we move down the index tree, the value of R is decremented for left-child nodes, and incremented for right-child nodes. But, the value of P is incremented for left-child nodes, and decremented for right-child nodes. This index building strategy enables us to follow more active merge policy for right-child node and a lazier merge policy for left-child node than the merge policy of their parent node.

Calculation of Frequency Threshold T: Recall that, the index at each node is split based on the value of T, where all terms whose frequencies above T form frequent index, while all terms whose frequencies below T form infrequent-term index. The value of T is calculated by recursively partitioning the query log into multiple sub-logs. In each partition, a split point is defined as frequency value f such that, the sum of frequencies of terms above f constitute 80% of the total sum of frequencies for all terms in that partition. That is, a split point divides the partition into low-frequency and high-frequency parts. In short, the query log is recursively split into multiple sub-logs based on 80-20 principle. The value of T for a partition is equal to the value of the split point for that partition. For example, if the value of T for a node is assigned the frequency f, then the left-child's T value will be the value of split point of low-frequency partition, and right-child's T value will be the value of split point of high-frequency partition.

Figure 7 represents an instance of index tree. In this example, the root node has one partition ($P = 1$), and the sub-index in that partition is split after 3^{rd} merge ($R = 3$). For left-child node, the R value is decremented by 1 and P value is incremented by 1. So, we reduce the number of merges to reduce the cost of merge event, and increase the number of partitions to reduce the cost of node splitting. This strategy gives us better index maintenance performance for infrequent-term sub-indexes. For right-child nodes, we increment the value of R, allowing more number of merges to happen, and decrement P (minimum value of P is 1), to reduce the number of partitions at each node. This strategy gives better query performance for frequent-term sub-indexes due to more number of merges and fewer sub-indexes. For other nodes in the *Index Tree*, the values of R and P are assigned accordingly.

Example*: Step-by-Step Construction of Index tree*

Figure 8 gives an idea of multi-split indexing algorithm for the twelve successive in-memory block movements to disk, to finally build an *Index Tree* shown in Figure 7. At each node, the dark colored block represents the partition with the sub-index, while the light colored block represents an empty partition. At first, the root node is initialized with $R = 3$ and $P = 1$, and has an empty partition. The first in-memory index block is moved on to the root node's empty partition, as shown in step (1). In steps (2) and (3), the in-memory blocks are similarly merged with sub-index at root node. But in step (3), since $R = 3$, after 3^{rd} merge, the sub-index is split into infrequent and frequent parts. The left-child has $R = 2$ and $P = 2$, and the right-child has $R = 4$, $P = 1$. The infrequent part is moved to the 1^{st} partition of the newly created left-child, and the frequent part to the only partition in the newly created right-child. In steps (4) and (5), the root node is updated like in steps (1) and (2). In step (6), the sub-index at root node is split into frequent and infrequent parts. The infrequent part is

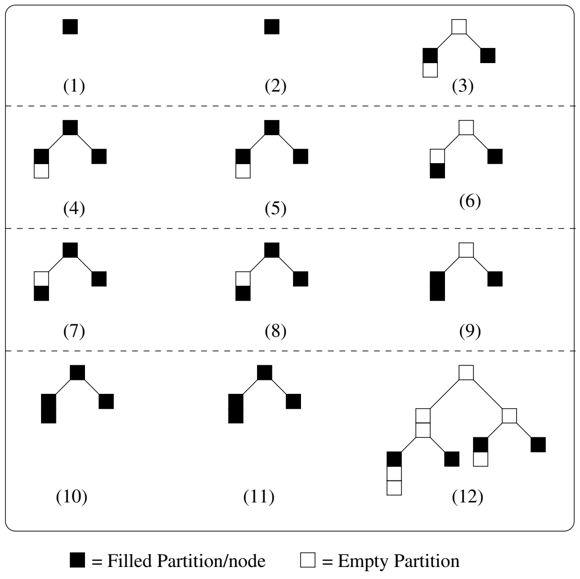


Figure 8: Multi-split indexing approach for twelve successive in-memory index block insertions.

merged with the sub-index at partition 1 of left child, since $R = 2$, the resulting sub-index is moved to partition 2. The frequent part is merged with the sub-index at partition 1, since $R = 4$, the resulting sub-index is placed at partition 1 of right child node replacing existing sub-index. In steps (7) and (8), follows the similar merging like in steps (1) and (2). In step (9), the sub-index at root node is split, and the infrequent part occupies the first empty partition of the left-child, and frequent part is merged with sub-index in the right-child. In steps (10) and (11), the new in-memory blocks are handled accordingly as in steps (1) and (2). In step (12), the sub-index at root node is split, and the infrequent part is merged with sub-index at 1^{st} partition of the left-child. Since R of the left-child is 2, the sub-index is merged and moved to the 2^{nd} partition. And after merging, as there is no new empty next partition available to hold the resulting sub-index, the sub-index is split, and new left-child ($R = 2$, $P = 3$) and right-child ($R = 3$, $P = 1$) are created.The new infrequent and frequent sub-indexes are moved to the 1^{st} partition of left-child and right-child respectively. Similar, the right-child of the root node exhausts its number of allowed merges. It is also then split into left-child ($R = 3$, $P = 2$) and right-child ($R = 5$, $P = 1$), and each child holds the new frequent and infrequent sub-indexes respectively in their first partitions.

Algorithm 1 gives the process of how an in-memory index block (*tmpIndex*) is moved to disk, and merged with on-disk index. When *root* is NULL, a new tree node is created, and the in-memory index block is moved to the first partition (*root.Partition.sIndex*) of the *root* node (lines 2–5). Otherwise, we initialize the variable *dbPartition* to 1^{st} partition of the *root* node. If parameter r of the partition is greater than 1, we merge the *tmpIndex* with *dbIndex*, where *dbIndex* is the sub-index at *dbPartition*, if *dbIndex* is non EMPTY (lines 15–17). Otherwise, the *tmpIndex* is just moved to the *dbIndex* (lines 11–13). The value of r of the *dbParition* is decremented in the both cases. If $r = 1$, and if there exists next partition (i.e., *dbParition.next* is not NULL),

Algorithm 1: InsBlock(root, tmpIndex)

```
1  begin
2      if $root = NULL$ then
3          $root \leftarrow$ getIndexTreeNode();
4          $root.Partition.sIndex \leftarrow tmpIndex$;
5          return $root$;
6      $dbPartition \leftarrow root.Partition$;
7      while $dbPartition \neq NULL$ do
8          $dbIndex \leftarrow dbPartition.sIndex$;
9          if $dbPartition.r > 1$ then
10             if $EMPTY(dbIndex)$ then
11                 $dbIndex \leftarrow tmpIndex$;
12                 $dbPartition.r \leftarrow dbPartition.r$ - 1;
13                 return $root$;
14             else
15                 $dbIndex \leftarrow$ Merge($dbIndex$,$tmpIndex$);
16                 $dbPartition.r \leftarrow dbPartition.r$ - 1;
17                 return $root$;
18         else if $dbPartition.next \neq NULL$ then
19             $tmpIndex \leftarrow$ Merge($dbIndex$, $tmpIndex$);
20             Reset($dbPartition$);
21             $dbPartition \leftarrow dbPartition.next$;
22         else
23             $fPart$, $ifPart \leftarrow$ splitMerge($dbIndex$, $tmpIndex$);
24             Reset($dbPartition$);
25             $root.left \leftarrow$ InsBlock($root.left$, $fPart$);
26             $root.right \leftarrow$ InsBlock($root.right$, $ifPart$);
27             return $root$;
28 end
```

then the *dbIndex* is merged with *tmpIndex*, and *dbParition* is assigned to *dbParition.next* (lines 19–21), and we continue with the next iteration. But, if there is no next partition, we merge *dbIndex* with *tmpIndex*, and split the new sub-index *tmpIndex* in to infrequent part (*ifPart*), and frequent part (*fPart*) (lines 23–27). For better performance, the splitting into frequent and infrequent parts is done while processing the merge event. The sub-indexes *ifPart*, *fPart* are recursively processed.

The function Merge() takes two sub-indexes as input, and returns merged sub-index. The function splitMerge() takes two sub-indexes as input, merges them, and splits the merged sub-index into frequent and infrequent sub-indexes. The function Reset() empties the partition, and resets parameter r to R.

4. PERFORMANCE STUDY

Data sets: For our experiments, we used the data set containing collections from Wikipedia. The collection is freely available[3], and is often used as data set in many IR related environments because of its comparable size to TREC GOV2[4]. The collection size used for our experiments is about 85GB (in uncompressed form), which contains about 7 million HTML documents. The average size of the document in the collection is around 12KB. We computed the term frequencies of all query terms from static AOL query log, which is then used to implement our Horizontal Partitioning strategy. The query log contains over 1.2 million unique query terms.

System setup: The experiments were conducted on Linux (Ubuntu 7.10) PC based on Intel Core 2 Duo (2.0GHz CPU)

[3]http://static.wikipedia.org

[4]Currently, due to unavailability of GOV2 data with us, we did not test our algorithms on TREC GOV2

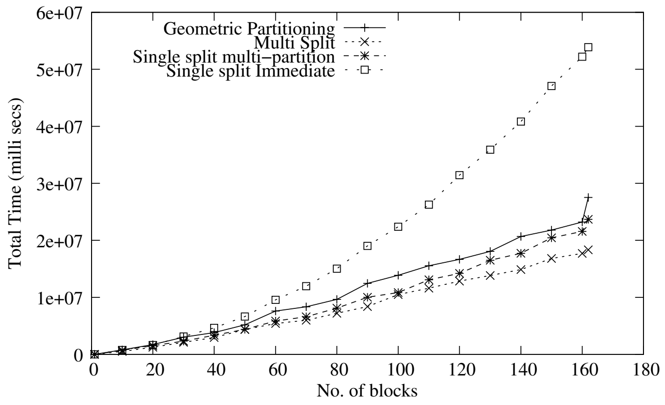


Figure 9: Index Maintenance performance for GP, Multi-split, and Single-split approaches. Times represent the total time taken to move in-memory blocks to disk

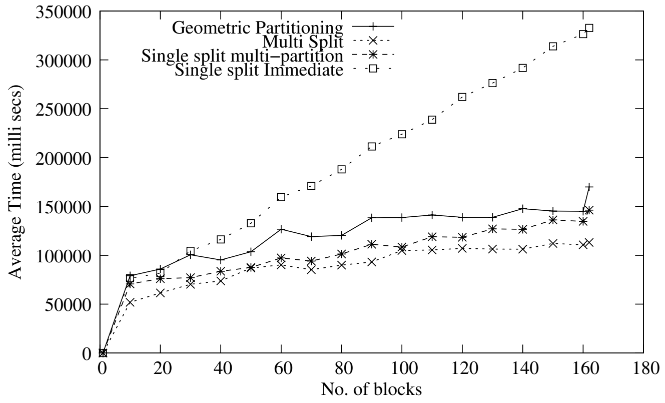


Figure 10: Index Maintenance performance for GP, Multi-split, and Single-split approaches. Times represent the average time taken to move an in-memory block to disk

with 1GB of RAM and 240 GB, 7200RPM SATA hard drive. The maximum size limit for in-memory index is set to 60MB.

We implemented four variants of on-line indexing approaches, and for all indexing approaches, the same document insertion and querying sequence was employed. The sequence contained 10000 queries interleaved with nearly 20000 document insertions. Document deletions are not handled in our experiments. Queries are drawn from the AOL query log. To simulate the real world scenario, we chose the sample 10000 queries whose frequency distribution is similar to the frequency distribution of queries in the AOL query-log. We observed a significant number of query terms in AOL query log are present in many documents in the collection taken. As of indexing approach, no caching is explicitly performed. We implemented our indexing approaches in LUCENE [1] Java 2.3.2.

Experimental Results

In our experiments, we evaluated the performance of the proposed Single-split, Multiple-split, and Geometric Partitioning. In Single-split approaches, the split point T, defined in Section 3, is evaluated based on 80-20% rule applied on query log. In Single-split Immediate merge approach, the frequent-term index is maintained using *Immediate merge* and infrequent-term index is maintained using Lazy approach, which is a variant of Logarithmic approach described in Section 3. Like *Single-split Immediate merge* approach, the infrequent-term index in Single-split multi-partition is maintained using Lazy approach, but the frequent-term index is maintained using an active merge approach, which is variant of variable r Geometric partitioning approach defined in Section 3. During the experiments, the value of r for *Single-split multi-partition* is initialized to 3. For *Multi-split* indexing approach, we initialized the values of parameters R, P to 3, 1. Here, we implemented the fixed r Geometric Partitioning strategy [9], by initializing r to 3.

The graphs in Figure 9 and 10 show that the approach of adopting different strategies for frequent-term and infrequent-term indexes gives higher index maintenance performance compared to the existing merge-based strategies. We observed that *Single-split Immediate* merge algorithm has poor index maintenance performance, because the in-memory block, when moved onto the disk, is immediately merged to the on-disk index. In the proposed *Single-split multi-partition* strategy, we adopted different merge strategies for sub-indexes; thus improving the overall index maintenance performance. Here, we observed a marginal 14% gain over *Geometric Partitioning*, and significant 56% gain over *Single-split Immediate* merge approach. The times mentioned in the graph are the absolute (in Figure 9) and average (in Figure 10) times taken to move in-memory blocks on to disk.

Multi-split merge strategy shows a better performance than *Single-split multi-partition* strategy and *Geometric Partitioning* strategy. This is because, as the frequent index becomes larger and larger, *Single-split multi-partition* and *Geometric Partitioning* incurs higher index maintenance cost. We observed that *Multi-split* approach shows higher index maintenance performance, of about 33% gain over *Geometric Partitioning*, and about 22.6% gain over *Single-split multi-partition* strategy.

For evaluating the query performance of each of the indexing approaches, we have taken a set of sample queries from the available AOL query log itself. These sample queries were further broken-down into individual terms, before performing the index lookup. We computed the time taken

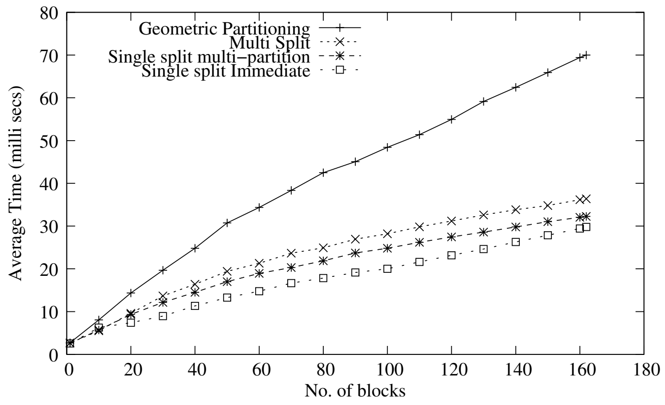


Figure 11: Query performance for GP, Multi-split, and Single-split approaches. Times (in ms) represent average time taken to evaluate a query after moving 'n' in-memory index blocks to disk

for each term to retrieve its postings from the index blocks. And we repeat this process for every new in-memory index block moved onto the disk. Figure 11 presents the query performance of the *Single-split*, *Multi-split* and *Geometric Partitioning* approaches. Here, we observed a significant 48% gain in query performance of our proposed *Multi-split* indexing approach over *Geometric Partitioning* approach. Similarly, there is 53% gain in query performance of our proposed *Single-split multi-partition* approach over *Geometric Partitioning*.

Although *Geometric Partitioning* was specifically designed for good index maintenance performance, from our experiments, we found out that our approaches perform better both in index maintenance performance as well as in query performance. Out of all indexing approaches, *Single-split Immediate merge* offers highest query performance as it maintains a single large index on the disk, but poorest index maintenance performance due to expensive immediate merges.

5. CONCLUSIONS

In this paper, we propose three on-line index maintenance approaches for IR systems based on the idea of *Horizontal Partitioning* of inverted indexes. The splitting criteria used for partitioning of terms into frequent and in-frequent is based on the frequency of terms in the query log. We use a lazy-merge approach for maintaining the infrequent-term index, and an active merge approach for maintaining the frequent-term index. The index maintenance cost for maintaining infrequent-term index, which is a large part of the index, is substantially reduced, because of the use of lazy merge approach. This gain compensates the slight increase in cost for maintaining frequent-term index and thereby reduces the overall index maintenance cost. As the query costs are proportional to number of partitions in the index, and the frequent-term index has a smaller number of partitions compared to other approaches, the query cost for terms in frequent-term index is substantially less. Since the terms in the frequent-term index form a large part of the query load, the overall query costs are reduced significantly, thereby increasing the query performance. Thus, the use of partitioning of indexes based on frequency of terms in the query log gives a significant improvement in query performance with low index maintenance cost.

6. REFERENCES

[1] http://lucene.apache.org.

[2] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 192–202, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[3] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 317–318, New York, NY, USA, 2005. ACM.

[4] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *In SIGIR 2006: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363, 2006.

[5] T. cker Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. Technical report, Experimental Computer Systems Lab, Department of Computer Science, State University of New, 1998.

[6] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411, New York, NY, USA, 1990. ACM.

[7] R. Guo, X. Cheng, H. Xu, and B. Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 751–760, New York, NY, USA, 2007. ACM.

[8] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Jour. of the American Society for Information Science and Technology*, pages 713–729, 2003.

[9] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM.

[10] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[11] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 1, New York, NY, USA, 2006. ACM.

[12] W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, 2005.

[13] K. Shoens, A. Tomasic, and H. García-Molina. Synthetic workload performance analysis of incremental updates. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR*, pages 329–338, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[14] I. M. Strategies, N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge:. In *In Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23. Society, Inc, 2004.

[15] T. Strohman and W. B. Croft. Low latency index maintenance in indri. In *Proceedings of the Open Source Information Retrieval Workshop*, pages 7–11, 2006.

[16] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 1994. ACM.

[17] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.