Efficient Update of Indexes for Dynamically Changing Web Documents

Contact Author
Lipyeow Lim
IBM T. J. Watson Research Ctr.
19 Skyline Dr.
Hawthorne, NY 10532, USA.
liplim@us.ibm.com
tel: 914-784-6156
fax: 919-784-7455

Min Wang
IBM T. J. Watson Research Ctr.
19 Skyline Dr.
Hawthorne, NY 10532, USA.
min@us.ibm.com

Sriram Padmanabhan
IBM Silicon Valley Lab.
555 Bailey Av.
San Jose, CA 95141, USA.
srp@us.ibm.com

Jeffrey Scott Vitter
Purdue University
150 N. University St.
West Lafayette, IN 47907, USA.
jsv@purdue.edu

Ramesh Agarwal
IBM Almaden Research Ctr.
650 Harry Rd.
San Jose, CA 95120–6099, USA.
rargarwal@us.ibm.com

Abstract

Recent work on incremental crawling has enabled the indexed document collection of a search engine to be more synchronized with the changing World Wide Web. However, this synchronized collection is not immediately searchable, because the keyword index is rebuilt from scratch less frequently than the collection can be refreshed. An inverted index is usually used to index documents crawled from the web. Complete index rebuild at high frequency is expensive. Previous work on incremental inverted index updates have been restricted to adding and removing documents. Updating the inverted index for previously indexed documents that have changed has not been addressed.

In this paper, we propose an efficient method to update the inverted index for previously indexed documents whose contents have changed. Our method uses the idea of landmarks together with the `diff` algorithm to significantly reduce the number of postings in the inverted index that need to be updated. Our experiments verify that our landmark-diff method results in significant savings in the number of update operations on the inverted index.

keywords: Web Search, Indexing Methods, Document Management

1 Introduction

The inverted index is the indexing technique of choice for web documents. Search engines use an inverted index for HTML documents [25], and DBMSs use it to support containment queries in XML documents [21, 32]. An *inverted index* is a collection of inverted lists, where each list is associated with a particular word. An *inverted list* for a given word is a collection of document IDs of those documents that contain the word. If the position of a word occurrence in a document is needed, each document ID entry in the inverted list also contains a list of location IDs. Positional information of words is needed for proximity queries and query result ranking [25]. Omitting positional information in the inverted index is therefore a serious limitation. Positional information is usually stored in the form of location IDs. The location ID of a word is the position in the document where the word occurs. An entry in an inverted file is also called a *posting*; it encodes the information $\langle word\_id, doc\_id, loc\_id \rangle$.

Since web documents change frequently, keeping inverted indexes up-to-date is crucial in making the most recently crawled web documents searchable. A *crawler* is a program that collects web documents to be indexed. Cho and Garcia-Molina [9] have shown that an in-place, incremental crawler can improve the freshness of the inverted index. However, the index rebuild method commonly used for updating the inverted index cannot take advantage of an incremental crawler, because the updated documents crawled in between rebuilds are not searchable until the next index rebuild.

In this paper we study the problem of keeping inverted indexes up-to-date. We first describe two general approaches and then show how our proposed landmark-diff method addresses the deficiencies in these two approaches.

The first approach is to rebuild the index more frequently. As the interval between rebuilds gets smaller, the magnitude of change between the two snapshots of the indexed collection also becomes smaller [9]. A large portion of the inverted index will remain unchanged, and a large portion of the work done by the rebuild is redundant.

The second approach is to store the updates in between rebuilds in a searchable update log. This is similar to the 'stop-press' technique [31] used to store the postings of documents that need to be inserted into the indexed collection. Each entry in this update log is a delete or insert posting operation. Query processing will need to search both the inverted index and the update log, and merge the results of both. If positional information is stored in each posting, which is often the case, the size of the update log will be prohibitively large.

Frequent rebuild is inefficient because it rebuilds portions of the index that did not change. The update log is unsatisfactory because it is too large and affects query response time. A better way of updating the inverted index is needed. An update method has to handle three types of changes: (1) documents that no longer exist (henceforth "deleted documents"); (2) new documents (henceforth "inserted documents"); (3) common documents that have changed. In order not to re-index common documents that did not change, an incremental update method is needed. Previous work on incremental inverted index updates has addressed changes due to inserted documents [8, 17, 27] and deleted documents [11, 12, 27]. Changes due to common documents that have changed have not been addressed.

In this paper, we propose the landmark-diff update method that addresses the problem of updating the inverted index in response to changes in the common documents. We show that solving this particular problem results in very efficient solutions to the more general index update problem.

Our landmark-diff method is based on the `diff` of the old and updated documents, and the encoding of positional information using landmarks in a document. Positional information is stored as a landmark-offset pair and the position of a landmark in a document is stored in a separate landmark directory for each document. The landmark encoding scheme reduces the number of inverted index update operations obtained from the `diff` output, because postings using landmarks are more “shift-invariant”.

1.1 Our Contributions

First, our method addresses how to incrementally update the inverted index for previously indexed documents whose contents have changed. This problem has not been addressed before.

Second, our method uses a landmark-offset pair to represent positional information. This representation has three advantages: it renders postings more “shift-invariant”, it does not increase the index size, and it does not affect query processing (Section 3.5). The mapping of a landmark to its position in a document is maintained in a landmark directory. A landmark directory is very small compared with the size of the document and hence does not significantly affect query response time. We show that this landmark-offset representation significantly reduces the number of update operations on the inverted index when used with the `diff` approach.

Third, our landmark-diff method is a general method that can be applied in a variety of ways. It can also be used to optimize many existing methods. Some possible applications are discussed in Section 3.4.

Fourth, we show analytically that the number of inverted index update operations generated by our landmark-diff method is independent of the size of the document and is dependent only on the size of the document change and the maximum distance between two landmarks. In contrast, the number of inverted index update operations generated by previous methods are all dependent on the size of the document. We also derive bounds on the update and query running time performance.

Fifth, in our experiments, we show that our landmark-diff method is three times faster than the forward index method (Section 3.1) in updating the inverted index for common documents that have changed. We also show how our landmark-diff method can be used in the partial rebuild method (Section 3.4) to solve the more general inverted index update problem where all three types of changes (deleted, inserted, and common documents) are addressed. The partial rebuild method is twice as fast as a complete rebuild. The partial rebuild method uses an array-based implementation of the inverted index, because the set of inserted documents is of the same magnitude as the set of common documents. As the update frequency increases, the number of inserted documents and the number of update operations for the common documents will be very small. Using our landmark-diff update method on a B-tree implementation of the inverted index will result in an even more dramatic speedup.

In the rest of this section we discuss related work. In the next section, we present an empirical analysis of web document change that motivate our work. In Section 3 we describe current indexing techniques and our landmark-diff method. In Section 4 we evaluate our landmark-diff method analytically, and in Section 5 we provide empirical validation.

1.2 Relation with Prior Work

Information retrieval using inverted indexes is a well studied field [3,31]. Although most of the update techniques generalize to keyword searches in web documents (HTML and XML files), several assumptions made by those techniques need to be re-examined. First, web documents are more dynamic than the text document collections assumed by those techniques. Past work on incremental updates of inverted indexes [3,8,13,27,31] deals mostly with additions of new documents to a collection of static documents. Static documents are existing documents in the collection whose contents do not change over time. The problem addressed in this paper deals with existing documents whose contents do change over time.

Second, previous work assumes that the inverted index resides on disk or in some persistent storage and try to optimize the propagation of in-memory update posting operations to the inverted file in persistent storage [8, 11, 12, 27]. Current search engines keep a large portion of the inverted index in memory[1], especially the most frequently queried portions of the inverted index. Even though the indexed collection is growing, we have three reasons to believe that a large portion of the inverted index will still be kept in memory: (1) query volume is increasing and there is a demand for increasing query processing speed; (2) the cost of memory chips is relatively low and still decreasing; (3) the ease of scaling with parallel architecture. Not only does the inverted index in persistent storage need to be updated, but the portion of the inverted index in memory needs to be updated as well. The update method we propose in this paper addresses the problem of updating the inverted index as a data structure independent of whether it resides in memory or on disk. In many cases, our method could be used in conjunction with existing techniques to speedup the propagation of updates to the off-memory inverted index. Several of these existing techniques will be described in greater detail next.

For append-only inverted index updates, Tomasic et al. [27] proposed a dual inverted list data structure: the in-memory short list and the disk-based long list. New postings are appended to their corresponding short lists and the longest short list is migrated into a long list when the storage area for the short lists is full. Brown et al. [8] proposed another incremental append strategy that uses overflow 'buckets' to handle the new postings in the relevant inverted lists. These overflow buckets are chained together and have sizes that are powers of 2. Both [27] and [8] deal with append-only incremental updates and assume that a document never changes once indexed. Our method tackles the complementary problem of updating previously indexed documents that have changed.

Clarke et al. [11, 12] addressed the deficiency of these append-only inverted index techniques and proposed a block-based processing of the inverted index that supports deletion of documents as well. The entire document collection is concatenated into a long sequence of words. Positional information of a posting is reckoned from the beginning of the document collection (as opposed to the beginning of each document). Hence, even though Clarke's inverted index supports updates at the postings level, it does not solve the problem of small changes in documents causing a shift in the positional information of many postings unrelated to the change. In fact, the positional shift problem is exacerbated by the use of absolute positional information reckoned from the beginning of the collection.

Our landmark encoding scheme is based on the idea of *blocking*. The idea of partitioning a document or document collection into blocks has been used for various purposes in [4, 18, 23, 26, 33]. The Glimpse [23] system uses a block addressing inverted index in order to keep the index size small. A block addressing inverted index stores, for each keyword, pointers to the blocks of text containing the keyword. The actual occurrence of the keyword in each block is not stored and is found by scanning the actual text itself using a fast sequential scan algorithm, such as Knuth-Morris-Pratt [19] or Boyer-Moore [5]. In [18, 26, 33], the idea of block addressing is used to improve the quality or the relevance of the information retrieved for a given query. In our work, landmarks are used to provide relative addressing for efficient index update using `diff`. The inverted indexes we consider are full inverted indexes that differ from block addressing inverted indexes in that the position of the occurrences of each word in the text is stored.

[1]For example, Google has thousands of machines with at least 8 GB of memory each.

No. of docs at time n	6042
No. of docs at time $n+1$	6248
No. of deleted docs	2788
No. of inserted docs	2994
No. of common docs	3254
No. of common docs unchanged	1940

Table 1: Summary of the representative data set used for data analysis.

2 An Analysis of Web Document Change

We analyzed a small set of web documents in order to understand the update characteristics of web documents. The results are summarized in this section as motivation for our landmark-diff index update method. Detailed results can be found in [22]. Other than scale, our analysis differs from [15] in that our distance measure for two documents is based on exact edit distance whereas a shingle-based approximate similarity measure [7] is used in [15]. Because our landmark-diff index update method is based on edit transcripts, the exact edit distance is a more appropriate distance measure in the context of this work. We have chosen to restrict our study to a small sample of the web (relative to that used in [15]) due to resource limitations. Despite the difference in distance measure and scale, our results agree with that in [15].

We call the set of web documents associated with a snapshot of the web at a particular time a *sample*. The time between two consecutive samples is the *sampling interval*. Consider two consecutive samples S_n and S_{n+1}. Any document can only belong to one of the three partitions: (1) common documents $S_n \cap S_{n+1}$, (2) deleted documents $S_n - S_{n+1}$, and (3) inserted documents $S_{n+1} - S_n$. Correspondingly, three operations are needed to update the index of S_n so that it reflects S_{n+1}. First, postings corresponding to the deleted documents need to be removed from the current index. Second, postings corresponding to the inserted documents need to be inserted into the index. Third, postings corresponding to the common documents that have changed need to be updated in the index.

We analyze the changes occurring in the set of common documents in terms of the magnitude of change and the clusteredness of change. For our analysis, we used several samples of the web recursively crawled from several seed URLs at 12 hour intervals. The list of seed URLs consists of www.cnn.com, www.ebay.com, www.yahoo.com, espn.go.com, and www.duke.edu. We present our analysis using 2 samples that are representative of the general update behavior. Other characteristics of our data are summarized in Table 1.

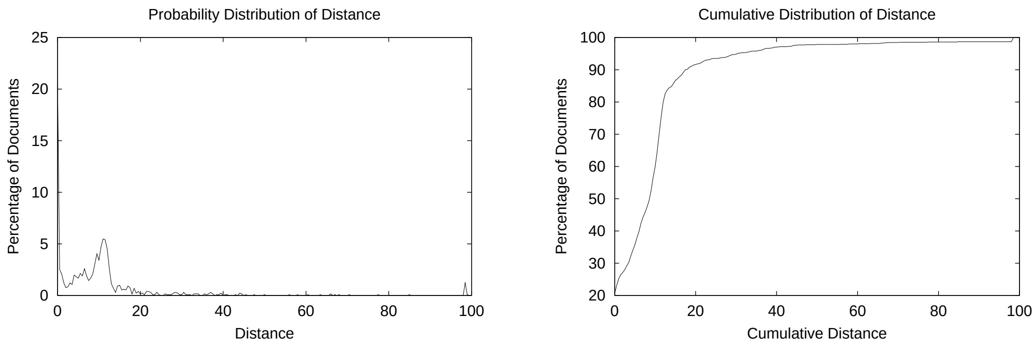
2.1 Magnitude of Document Change

For common documents whose contents have changed, we are interested in how large the changes are. The magnitude of change between two versions of a document can be quantified by the *edit distance*. The edit distance of two documents is the minimum number of *edit operations* (word insertions or deletions) required to transform one document to the other. Let δ be the minimal number of words deleted or inserted to transform one document to another. We define the distance between two documents A and B as

$$d(A,B) = \frac{\delta}{m+n}, \qquad (1)$$

where m and n are the size (in words) of document A and document B respectively. Two identical documents will have zero distance and two completely different documents will have a distance of one.

For our empirical analysis, we measure the distance of each pair of old and updated documents (same document ID) in the set of common documents and put them into bins according to the distance. Each bin has a width of 0.5 %. The number of documents in each bin is normalized to a percentage and plotted in Figure 1. We observe that most documents fall into the bins between



(a) Probability Distribution (b) Cumulative Distribution

Figure 1: Distribution of documents with respect to our distance measure.

distance 0 % and 20 % (Figure 1). In fact, the corresponding cumulative distribution plot shows that more than 90 % of the documents have changes smaller than a distance of 20 %. Moreover this behavior is consistent across the consecutive samples we crawled and we conjecture that the set of common word occurrences is very big in general. The implication is that a large portion of the information maintained by a web index can remain unchanged at an update.

2.2 Clusteredness of Change

Another important characteristic of change is how clustered the changes are. For example, from an update processing point of view, the insertion of a paragraph of 10 words to the beginning of a document is quite different from inserting the same 10 words at 10 different random and non-contiguous locations in the document. The former has locality that could be exploited in update processing.

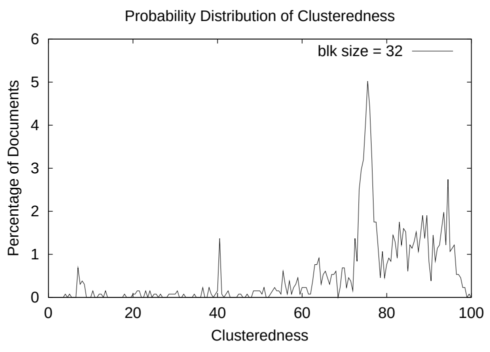
We measure how clustered the document changes are as follows. Partition the document into blocks according to some partitioning scheme and count the number of blocks affected by the changes. The fraction of affected blocks over the total number of blocks is then used as a measure of how clustered the changes are under the partitioning scheme used. A simple partitioning scheme is to used fixed size blocks of b words each. Other partitioning scheme that exploits the embedded structure of the document can also be used. One example is to use HTML tags such as the paragraph tag `<p>`. Choosing a partitioning scheme is discussed in greater detail in Section 3.3.

For a given partitioning scheme with parameter b, we define the *clusteredness* of the changes required to transform document A to document B as

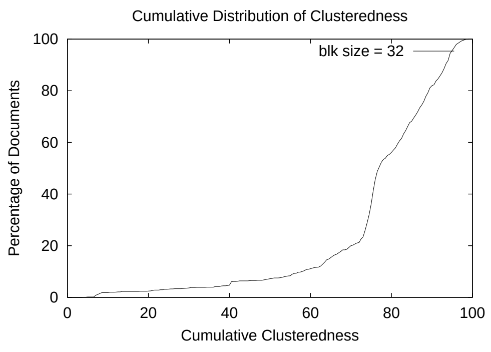
$$c(A, B, b) = 1 - \frac{\Delta}{\lceil m/b \rceil}, \tag{2}$$

where Δ is the number of blocks affected by the change, m is the size of the old documents in words and b is the parameter of the partitioning scheme used. For fixed size partitioning, b is the block size in words. If there are no changes, Δ will be zero and the clusteredness will measure one. If all the changes are clustered into one block and assuming that the block size b is sufficiently small, the clusteredness will be close to one. If the changes are distributed over all the blocks, Δ will be equal to $\lceil m/b \rceil$ and clusteredness will be zero. The block size b must be chosen such that it is much smaller than the document size m for this measure to be meaningful.

We study the empirical distribution of the common documents that have changed using this clusteredness measure. Our results are obtained by computing the clusteredness measure for each document in the set of common documents. The clusteredness values are normalized to a percentage and classified into bins of 0.5 %. The number of values in each bin is counted and normalized to a percentage of the total number of common documents. From the probability distribution plot with



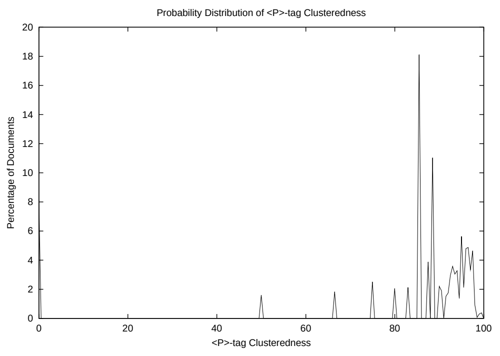
(a) Probability Distribution



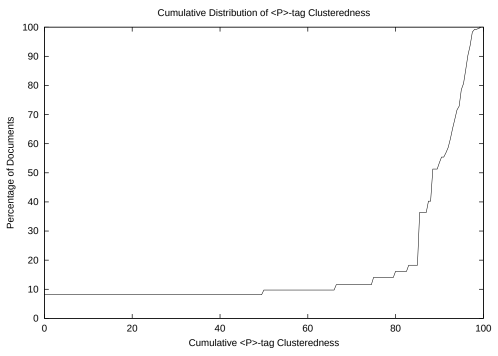
(b) Cumulative Distribution

Figure 2: Distribution of documents with respect to clusteredness $c(A, B, 32)$.

respect to our clusteredness measure $c(A, B, 32)$ (Figure 2), we observe that most documents have changes that are more than 50 % clustered, that is, the changes affect less than half of the blocks. The corresponding cumulative distribution plot shows that only about 20 % of the documents have changes that are less than 70 % clustered. Using HTML paragraph tags, we observe in the



(a) Probability Distribution



(b) Cumulative Distribution

Figure 3: Distribution of documents with respect to clusteredness $c(A, B, \texttt{<p>}\text{-tag})$.

probability distribution plot with respect to the clusteredness measure $c(A, B, \texttt{<p>}\text{-tag})$ (Figure 3) that many document have changes that are more than 50 % clustered; however significant spikes occur consistently at the 0-0.5 % clusteredness bin. This is because some documents do not use the `<p>`-tag at all. For such documents there is only one block in total and any changes must occur in that block and hence $c(A, B, \texttt{<p>}\text{-tag})$ is zero. This is consistent with the previous $c(A, B, 32)$-distribution plot (Figure 2) where no documents have changes distributed to every block. Other observable artifacts are the spikes at the 50% and 66% clusteredness marks. These are mostly due to the documents with only two to three paragraphs in total.

We draw three conclusions from our empirical analysis that are true for the web samples we collected and we conjecture that they are true in general if the sampling interval is sufficiently small: (1) common documents represents at least 50 % of the currently indexed collection, (2) most of the changes in the common documents, are small, and (3) most of the changes are spatially localized within the document. The landmark-diff update method that we propose in the next section is a scheme that exploits these properties.

3 Updating Inverted Indexes

3.1 Current Methods

Before we describe our method in detail, we briefly describe three naive update methods: the index rebuild, document update, and forward index methods. The forward index method is used as a benchmark in our experiments because it is the most efficient of these naive methods.

Index Rebuild. In this method, the old inverted index is discarded, and a new inverted index is constructed by scanning and sorting the entire updated document collection (see [31] for details). The disadvantages are: (1) the entire document collection has to be crawled at periodic intervals, and (2) every word in the collection has to be scanned to construct the inverted file. Distributed rebuilding techniques, such as [24], parallelizes (and pipelines) the process, but does not eliminate the need of scanning every word in every document. If the magnitude of change is small, scanning and re-indexing the words in documents that did not change is wasteful and unnecessary. Rebuilding is recommended only if (1) there are very few common documents ($|S_n \cap S_{n+1}|$ is small), or (2) the set of common word occurrences is small, i.e., a large portion of each updated document has changed.

Document Delete and Insert. One improvement over index rebuild is to process only documents that have changed. For each document that has changed, we delete all the postings for the old version of that document in the inverted index and insert the postings of the new document. The worst case number of postings deleted and inserted is $O(m + n)$, where m and n are the number of words in the old and new version of the document, respectively. The advantage of this method is that web documents can be crawled at different sampling intervals depending on their rate of change[2]. The document delete and insert method is recommended if (1) the rate of change of documents has a large variance, and (2) the magnitude of change within an updated document is large.

Forward Index Update. A forward index is a list of postings associated with a single document (see Figure 4) and differs from the inverted index in that the list of postings of the form $\langle \textit{word\_id}, \textit{doc\_id}, \textit{loc\_id} \rangle$ is sorted first by $\textit{doc\_id}$, then by $\textit{word\_id}$ and $\textit{loc\_id}$. The forward index

[2] Incremental crawling has been addressed in [6, 9, 10, 20] and optimal crawling frequency is discussed in [6, 10, 20].

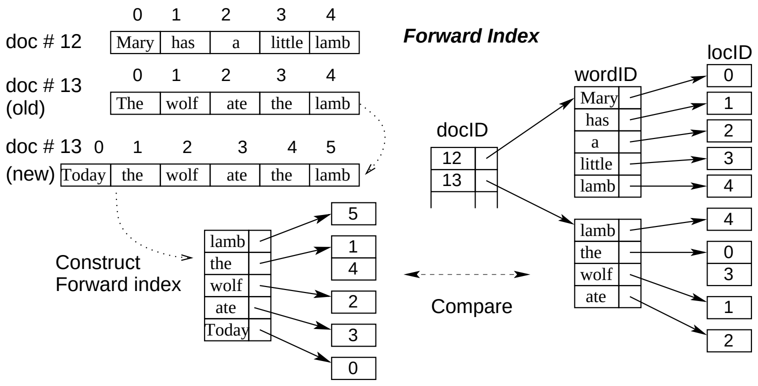


Figure 4: An example of a forward index. For the new version of the document #13 (with "Today" inserted to the beginning), a forward index representation is constructed. This forward index is then compared with the forward index of the old version stored by the system. Note how the location IDs of the words changes.

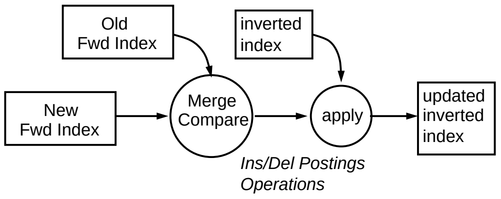


Figure 5: Updating an inverted index using forward indexes.

data-structure was introduced in Page and Brin's Google paper [25] as an auxiliary data structure to speedup inverted index construction. We deduce that it can be used for index updates as described next. Given the forward indexes of the old and new document, we generate a list of postings operations (insert posting and delete posting) that will transform the forward index of the old document to that of the new document. Since the postings stored in a forward index are the same postings that are stored in the inverted index, we can use these posting operations to update the inverted index. The data flow diagram in Figure 5 illustrates the update procedure for a single document. To update the inverted index, we apply the procedure for every common document that has changed.

The advantages of using the forward index method are: (1) as with the document delete and insert method, incremental crawling can be used, (2) when the content change occurs near the end of the document, the postings corresponding to the words before the position of that change need not be deleted and re-inserted. However, the disadvantages are: (1) an additional forward index needs to be stored for each document, (2) each forward index requires as much space as the document itself, and (3) the number of postings deleted and inserted for a document that has changed is still $\Theta(m + n)$ in the worst case, even if the difference between the two documents is small. For a pathological example, consider an insertion of one word to the beginning of the old document. All subsequent *loc_id*s will shift by one and hence all corresponding postings in the index will need to be updated.

3.2 Our Landmark-diff Update Method

In this section, we present our landmark-diff index update method. Our method has the following desirable properties: (1) it is document based and incremental crawling can be used; (2) the

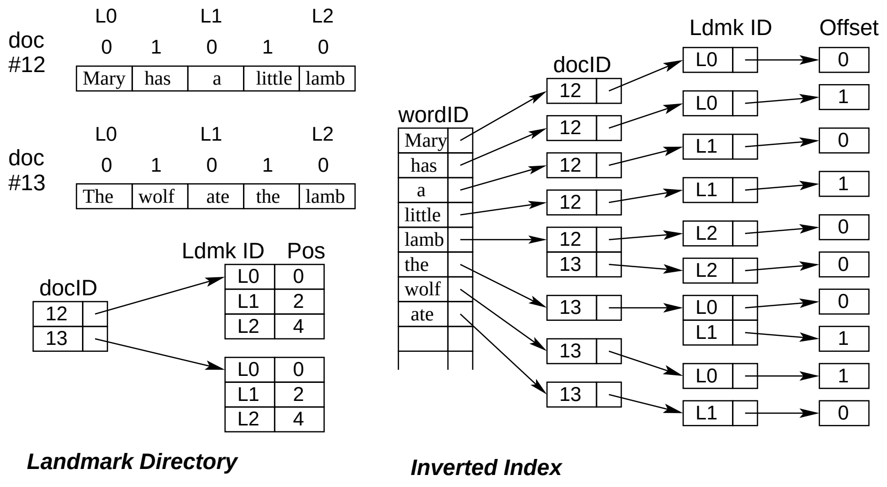


Figure 6: An inverted index with fixed sized landmarks. Note that this pointer-based representation is purely conceptual.

number of update operations on the inverted index (the number of postings deleted and inserted) is independent of the document size, and depends on the size of the content change only; (3) the landmark-diff method is especially efficient when the set of common documents is large, and changes are small and clustered.

Our landmark-diff method combines two important ideas: encoding the position of words using landmarks in a document, and using the *edit transcript* or `diff` output of the old and new document to obtain the index update operations. An edit transcript is a list of operations that transform an old version of a document to the new version of the document. The landmark encoding scheme allows us to translate the edit transcript for the document into a very compact list of update operations on the inverted index.

We first introduce the landmark encoding scheme and then describe the landmark-diff update procedure. For the rest of this paper, we reserve the term *edit transcript* for the `diff` output of a document and its updated version, and we use the term *update operation list* to refer to the list of inverted index update operations (delete or insert posting) that will update an inverted index.

Landmarks. The purpose of relative addressing using landmarks is to minimize the changes to the location IDs stored in the inverted index when documents change. Given a document, a number of positions in the document are chosen as landmarks. The position of each landmark in the document is recorded in a landmark directory structure for the document. The location ID for each word in the document is then encoded as a $\langle \text{landmark ID}, \text{offset} \rangle$ pair. Given a $\langle \text{landmark ID}, \text{offset} \rangle$ pair, the original location ID of the word occurrence is recovered by retrieving the position of the landmark from the associated landmark directory and adding the offset to that position. Each landmark acts as a reference point for the words between itself and the next landmark.

Putting landmarks in a document can also be thought of as partitioning a document into blocks. The starting position of a block corresponds to a landmark and the location of every word in a block is encoded as an offset from the start of that block. We use the term *block* to denote the words between consecutive landmarks. Each landmark corresponds to a block and vice versa. For the following discussion, we assume that a document is partitioned into fixed sized blocks. Other ways of choosing landmarks are discussed in Section 3.3. Figure 6 shows an inverted index with landmarks.

The landmark encoding does not increase the space requirement of the inverted index itself; however, auxiliary landmark directories need to be stored for each document. Suppose k bits are allocated for each *loc_id* in an inverted index without landmark encoding. In an equivalent index using landmark encoding, the same k bits in each posting can be used to encode a $\langle\text{landmark ID}, \text{offset}\rangle$ pair with the most significant $b < k$ bits used for storing the landmark ID and the least significant $k - b$ bits for the offset.

The landmark directories for each document are usually small. For a fixed block size of l words, the number of entries in a landmark directory for a document of m words is $\lceil m/l \rceil$. For other ways of choosing landmarks, such as using HTML paragraph tags, the number of entries in a landmark directory is less dependent on the document size. Note also that a landmark directory lookup is not always required during query processing. Further analytical evaluation and implementation issues for the landmark directories are presented in Section 4.

Update Procedure Using Landmarks. We show how to update an inverted index using our landmark-diff method in response to a content change in a document. The goal is to obtain an update operation list (an "edit transcript" for the inverted index) that can be applied to the inverted index to bring it up-to-date. The key idea is to obtain this update operation list from the edit transcript for the old and updated document. The landmark encoding scheme allows us to construct this update operation list using the edit transcript of the updated document, without increasing the number of delete or insert posting operations per document to $\Omega(m + n)$, where m and n are the number of words in the old and new version of a document. The details of the procedure for transforming a document edit transcript to an update operation list is lengthy but trivial, and we illustrate it by way of the example in Figure 7. The entire landmark-diff update procedure is also outlined in Figure 7. For each document that has changed, we obtain the edit transcript for updating that document using a `diff` procedure. The `diff` output is then transformed into corresponding entries in the update operation list for the inverted index using landmark information. The update operation list is then used to update the inverted index. All the procedures before the apply step require as input the old document, its landmark directory, and the new version of the document only; therefore these procedures lend themselves to parallel processing.

In addition to updating the index, the landmark directory for each changed document needs to be updated as well, because insertion and/or deletion of words from the blocks within a document change the absolute position of landmarks within that document. Updating a landmark directory can be done very efficiently in a single sequential scan through the landmark directory data structure. This process is linear in the size of the landmark directory and the number of landmarks deleted and inserted. The overhead incurred for storing and maintaining landmark directories is not significant compared to the savings gained in the number of inverted update operations and in the update time as shown by our analytical and empirical evaluations in Section 4 and Section 5. Further analysis of landmark directories is given in Section 4.

3.3 Choosing Landmarks

Our landmark-diff update method exploits the clusteredness of the document updates and the partitioning of the document into blocks to localize the clusters of updates. Choosing a partitioning scheme, or, analogously, choosing landmarks, is therefore an important aspect of the method.

A *landmarking policy* describes how landmarks are chosen in a document and also how these landmarks are to be processed during updates. To understand the crux of the problem, consider the hypothetical question: how would an oracle, i.e., a person who knows everything about fu-

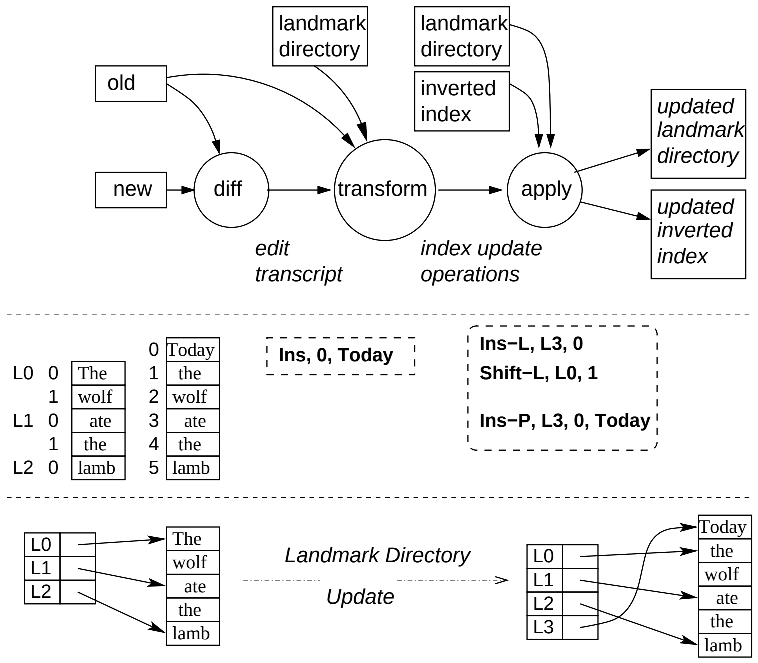


Figure 7: The inverted index update method using landmarks. The top diagram shows the data flow; the middle diagram shows what the edit transcripts look like with an example; the bottom diagram show how the landmark directory is updated. The landmark directory is represented here conceptually as a table.

ture updates (within a certain time window), choose landmarks? An oracle would partition the document such that

1. each longest contiguous sequences of words that do not change is put into a block and

2. each longest contiguous sequences of words that changes is localized into a block.

The oracle would not only optimize these localization constraints for each update, but across all updates over the time window. Figure 8 shows an example of such a partitioning across all updates within a time window. The resultant policy ensures that the minimal number of blocks are used and that changes are localized into as few blocks as possible. Hence, the size of the landmark directory is minimized and the number of blocks touched by update processing is also minimized.

Since we do not have foreknowledge of future updates, we can only choose landmarking policies based on the document itself. If no assumptions can be made on the structure of the document contents, fixed size partitioning can be used. If we know something about the structure of the document contents, for example from HTML/XML tags, metadata, or semantic structure, we can use this information to choose landmarks.

Fixed size partitioning. The fixed size partitioning policy partitions each document into blocks of a fixed size during index construction. During update processing, two ways of dealing with how an edit operation affect a block are possible and the landmarking policy needs to specify which one to use. Consider an edit operation that inserts a piece of text at some position within a block, the landmarking policy has to specify whether to make the block bigger or to split the block (Figure 7 shows the case when an insert always splits a block). Similarly, when a piece of text in the middle of a block is deleted, the landmark policy has to specify whether the remaining text forms one block

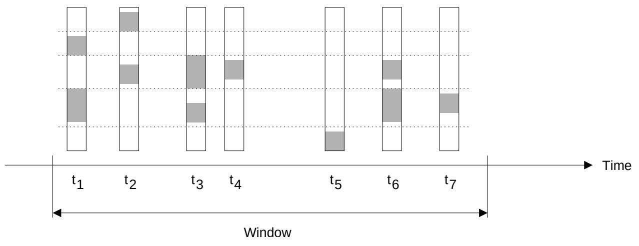


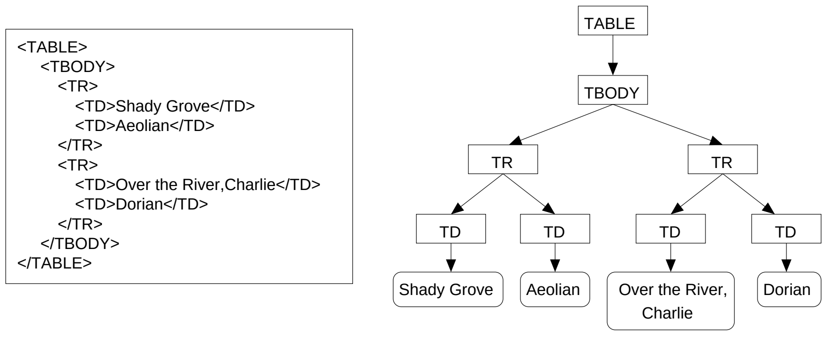
Figure 8: The evolution of a document over time. A tall rectangular represents a document at time t_i. The shaded regions in a document denotes the presence of updates in those regions at that time. The horizontal dotted lines denote the partition boundaries that would localized all updates occurring within the time window.

or two blocks. A *split-block* policy is a policy where an edit operation always splits a block. A *grow-block* policy is a policy where an edit operation always makes a block larger. The split-block policy has the property that each block never exceeds the chosen size, while the grow-block policy has the property that the landmark directory never exceeds a certain size. In both cases, the variance in block size increases with the number of updates – a condition we term loosely as *fragmentation*. For the split-block policy fragmentation results in large landmark directories and hence increases the overhead in query and update processing that require accesses to the landmark directories. For the grow-block policy, fragmentation results in large blocks of text and increases the number of index update operations required. Defragmentation or index rebuild should be performed when update performance degrades. Since content changes in the updated documents are likely to be small, fragmentation would not occur frequently.

In fixed size landmarking, the landmarks are not inherent in the structure of the document. A brute-force `diff` on the old and updated documents is required to generate the edit transcript. Using the landmark information of the old document and the edit transcript, inverted index and landmark directory update operations are generated (see Figure 7 for an example). Each edit operation in the edit transcript is mapped to landmark directory update operations and index update operations in terms of landmark-offset pairs.

HTML/XML tags. HTML tags such as the paragraph tag (`<p>`) can also be used as landmarks. In contrast to the fixed size policy, the landmarks in this case are inherent in the document and the problem of whether an edit operation should split or grow a block is irrelevant, because block boundaries are decided by the HTML tags, say the `<p>` tag. A more efficient, block-based, approximate `diff` procedure could be used instead of the brute force `diff`. For example, we could hash each block of the original and new versions and do the `diff` on the substantially smaller sequence of hashed values.

A natural question arises: which tags or sequence of tags should be used as landmarks? Linear-time heuristics can be used to check which tags (or sequences of tags) are suitable as landmarks. For example, the *Document Object Model (DOM)* tree [1] of a HTML or XML file can be generated and the top few levels used as the partitioning policy. Note that we really only need to generate the top few levels of the tree. Figure 9 shows an example of a HTML document and its DOM tree. In the example, the first three layers of the tree can be used as the landmarking policy and we could use the `<TR>` tags as the landmarks. Using such techniques to determine landmarks for each document means that a brief description of the landmarking tags will need to be stored together with the landmark directory for each document.



```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River,Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

Figure 9: A HTML document and its DOM tree. The first three layers can be chosen to be the landmarking policy, i.e., the `<TR>` tag can be used to partition the document into three blocks.

Characteristic	Fixed Size Partitioning	Tag-based Partitioning
Block boundaries	Imposed by fixed size constraint	Inherent in document structure
`diff` requirements	Brute force	Block-based
Variance of block sizes	Determined by split/grow-block policy and de-fragmentation	Determined by document structure
Assumption on updates	Localized by small number of blocks	Localized by document structure
Storage overhead per document	A landmarking block size will need to be stored, if a different block size is used for each document.	A landmarking tag description needs to be stored, if different tags are used for each document.

Table 2: A summary of the differences between fixed size partitioning and tag-based partitioning.

3.4 Some Application Scenarios

We have described the landmark-diff method for updating the index for common documents that have changed. We now give three example scenarios to illustrate how our landmark-diff method can be used to solve the general index update problem.

Update Log. If the inverted index is maintained using an update log (similar to "stop-press") in between complete index rebuilds, our landmark-diff method can be used to reduce the size of the update log. The naive update log corresponds to the list of inverted index update operations generated by the forward index update method (Section 3.1). Our landmark-diff update method can be used to significantly reduce the size of this update log and thus the query processing time.

Partial Rebuild. In contrast to a complete rebuild, a partial rebuild avoids re-indexing the common documents that did not change. Suppose that the inverted index is stored as a sorted array A_{index} of postings, the *doc_id*s of the deleted documents are stored in a bitmap $\mathcal{B}_{deleted}$, and the postings of the inserted documents are stored in a sorted array $A_{inserted}$ (stop-press). The landmark-diff method can be used to maintain a reduced-size sorted update log A_{update} for the common documents. The inverted index can then be updated in a single merging pass of the three sorted arrays A_{index}, A_{update} and $A_{inserted}$ (checking $\mathcal{B}_{deleted}$ for deletes). Our experiments described in Section 5.3 show that partial rebuild can be twice as fast as complete rebuild.

Distributed Index Update. Suppose the document collection is partitioned among M machines and indexed independently. At each update, each machine updates its index using a bitmap for the deleted documents and the landmark-diff method for the common documents. Inserted documents are always processed at a free machine that builds an index for them. When no free machines are available, the two machines with the smallest indexes are found and the two indexes are merged to free one machine.

3.5 Query Processing with Landmarks

Query processing using an inverted index with landmarks does not differ significantly from using a traditional inverted index. The main difference is that when positional information is required (either to check positional query constraints or to compute relevance rank), the landmark directories of all documents involved need to be retrieved to compute the location ID's of the postings. To minimize the number of landmark directories retrieved, the retrieval of landmark directories should be done after all document ID based filtering.

For phrase queries, the landmark directory retrieval and lookup can be avoided completely if positional information is not required. An additional field is associated with the last word occurrence of every block to store the ID of the next landmark. Given a phrase query (key_1 key_2), the postings for each keyword is retrieved from the index independently. Postings with the same document ID are grouped together and two postings, $\langle key_1, doc\_id, landmark\_id_1, offset_1\rangle$ and $\langle key_2, doc\_id, landmark\_id_2, offset_2\rangle$ form a phrase only if (1) $landmark\_id_1$ is equal to $landmark\_id_2$ and the difference between $offset_1$ and $offset_2$ is 1, or (2) $landmark\_id_1$ is not equal to $landmark\_id_2$, but $offset_2$ is zero and the posting for key_1 has the additional next landmark field that is equal to $landmark\_id_2$.

The cost of retrieving a landmark directory depends on the size of the landmark directory which depends on the landmarking policy. The cost of looking up the location ID of a landmark, depends on the implementation of the directory and we discuss the details in Section 4.3 and Section 4.7.

4 Analytical Evaluation

In this section we evaluate the performance of our landmark-diff update method analytically using the number of update operations as well as the more traditional running time complexity. We also show that the complexity of the `diff` operation is not a bottleneck, and that landmarking has minimal impact on query processing.

4.1 Update Performance in Number of Operations

Consider the number of updates (insert and delete postings) to an inverted index generated by our update method for a single edit operation on a single document. The following theorem states a key property of the landmark-diff method that all existing update methods lack:the number of updates generated by our method is independent of document size and dependent only on the size of the edit operation in the document and the maximum block size. The number of updates generated by existing update methods, on the other hand, is dependent on the size of both the original and the updated document.

Theorem 1 *The number U of updates to an inverted index caused by a single edit operation that deletes or inserts Δ contiguous words is at most*

$$U \leq \Delta + \epsilon,$$

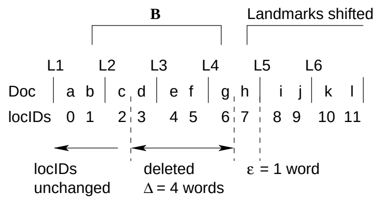


Figure 10: Consider a document with 12 words ('a' to 'l'). The block size is set at two words. Deletion of $\Delta = 4$ words generates $\Delta + \epsilon$ update posting operations on the inverted index. The set $\mathbf{B}$ contains all the segments affected by the change. Landmark ID $L3$ has to be deleted, the position of landmark $L4$ has to be changed to 3, the offset of the word 'h' has to be changed to 0, and the position of landmarks $L5$ and $L6$ has to be shifted by -4.

where ϵ is the number of words that are not deleted in the last modified block.

Proof: Let $\mathbf{B}$ be the set of landmarks or blocks affected by this deletion or insertion of $\boldsymbol{\Delta}$ words in the document. For the example in Figure 10, $\mathbf{B} = \{L2, L3, L4\}$. All postings in the inverted index corresponding to the words in $\mathbf{B}$ will have to be updated except postings for the words occurring prior to the start of the deletion or insertion point. The number of updates is thus bounded by $\Delta+\epsilon$ words, where ϵ is the number of words following the change in the last block in $\mathbf{B}$. In Figure 10, the posting for the word 'c' does not change, but the posting for the word 'h' changes, because its offset becomes zero after the deletion of four words. □

The following lemma follows from the analysis of the gap distribution given in [29]. The positions of the L landmarks of a document are modeled as random, because the initially uniform landmark positions are no longer uniform after several random edit operations.

Lemma 1 *If the L landmarks of a particular document are randomly located, the expected number U of updates to an inverted index caused by a single edit operation of size Δ on a document of size m words satisfies*

$$U \leq \Delta + (m - L)/(L + 1).$$

This lemma provides an upper bound in terms of m and L for the expected value of the ϵ term in Theorem 1. If the number of landmarks L is a fixed constant, then the expected value of ϵ is linear in the document size m. If the number of landmarks is a factor of m (for example in fixed size partitioning, $L = m \div \textit{blocksize}$), the expected value of ϵ is upper bounded by the block size.

4.2 The Complexity of the `diff` Operation

The use of edit transcripts (`diff` output) is a key idea in our method. The problem of computing an edit transcript given two documents is known formally as the longest common subsequence (LCS) problem. The worst case running time for a dynamic programming solution to the LCS problem is $\Theta(mn)$, where m and n are the number of words in the old and the new documents, respectively. A heuristic-based algorithm due to Ukkonen [28] achieves a running time of $O(D \min\{n, m\})$ using $O(D \min\{n, m\})$ space, where D is the minimum edit distance. The UNIX `diff` program uses an output-sensitive heuristic algorithm similar to that of [28], so that the running time is near-linear when D is small and quadratic if D is linear.

Since `diff` is near-linear for small updates and the updates are usually small between two consecutive samples [22], `diff` is not a bottleneck in the processing in most cases. If the `diff` operation did form a bottleneck, we could represent each block in a document by its hashed value and perform the `diff` operation on the sequence of hash values rather than on the raw blocks directly.

Theorem 2 *Let L and L' be the number of landmarks in the old and new documents, respectively. The block-based* `diff` *variant takes $O(D' \min\{L, L'\} + m + n)$ time to generate the edit transcript, where $L \leq m$, $L' \leq n$, and D' is the block-wise minimum edit distance.*

False negatives (i.e., two identical blocks that are reported to be different) do not affect the correctness of the update method, but increases the size of the edit transcript and hence the number of update operations. Note that hash functions in general do not produce false negatives. False positives (i.e., two different blocks that are reported to match) could affect the correctness of our update method and need to be eliminated by doing a linear scan of the blocks that are reported to be identical to verify identity. All the blocks that are found different (both from the `diff` step and the linear scan step) will be the ones that need to be deleted and inserted into the system. The edit transcript would then consist of a list of delete landmark operations for blocks deleted from the old document and a list of insert landmark and insert postings operations for the blocks of text inserted into the new document.

4.3 Implementing the Landmark Directory

The running time complexity of the landmark-diff method is dependent on how the landmark directory is implemented. We briefly describe the operations that an implementation of the landmark directory must support.

Insert(pos**)** inserts a new landmark at position pos.

Delete($landmark\_id$**)** deletes the landmark $landmark\_id$.

DeleteRange($landmark\_id_1$**,** $landmark\_id_2$**)** deletes all landmarks occurring between $landmark\_id_1$ and $landmark\_id_2$.

Shift($landmark\_id$**,** $value$**)** adds $value$ to the position of all the landmarks that occur after $landmark\_id$.

Find_Pos($landmark\_id$**)** returns the position of the landmark $landmark\_id$.

Find_Ldmk(pos**)** returns the landmark corresponding to the position pos.

Find_All_Ldmk(pos_range**)** returns all the landmarks corresponding to the positions in the range pos_range.

An offset tree [30] is the theoretically efficient data structure for a landmark directory. In practice, an array is used because of its compactness and good locality of memory reference. We analyze the offset tree data structure for landmark directories in Section 4.4, and the array data structure for landmark directories in Section 4.5. A summary of the analysis is given in Table 3. For the rest of the analysis we will assume that the landmark directories are implemented as arrays.

4.4 Offset Trees for Landmark Directories

Offset trees are binary trees that store an offset value in each node. They resemble the document representation data structures proposed in [14,16,30]. In this paper, we are not concerned with document representation, but with index representation together with novel index update techniques. Offset trees are frequently used in similar problems (such as in [30]) and they are theoretically very

Operation	Offset Tree	Simple Array
Insert	$O(\log L)$	$O(1)$
Delete	$O(\log L)$	$O(1)$
DeleteRange	$O(\log L)$	$O(B)$
Shift	$O(\log L)$	$O(L)$
Find_Pos	$O(\log L)$	$O(1)$
Find_Ldmk	$O(\log L)$	$O(L)$
Find_All_Ldmk	$O(\log L + B)$	$O(L)$
Update Ops Generation	$O(\log L + B + \Delta)$	$O(L + \Delta)$
Query Overhead	$O((\sum_i s_{key_i}) \times \log L_{\max})$	$O(\sum_i s_{key_i})$
Space Overhead	$40L$	$16L$

Table 3: Summary of the performance of the offset tree data structure versus that of the array data structure for landmark directories. The term L is the number of landmarks in the document, B is the number of landmarks affected by change or within a range, and Δ is the number of words deleted or inserted.

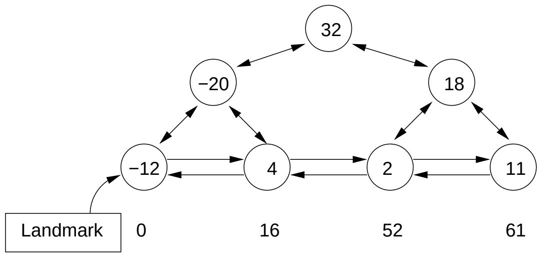


Figure 11: The offset tree data structure.

efficient. However, they require too much space as compared with arrays and exhibit poor locality of memory reference in practice.

In our context, each node in the offset tree stores an additional parent pointer and each leaf node corresponds to a landmark of the document. The leaf nodes also store the following extra fields: the **size** field, the **byte position** field, the **left** and **right** pointers. The **size** field stores the extent (in number of words) of that landmark. The **byte position** field stores the byte position of the landmark in the document. The **left** and **right** pointers point to the adjacent leaf nodes and make the leaf nodes a doubly linked list.

The actual word position of a landmark is the sum of the offsets of the nodes in the path from the root to the leaf node corresponding to that landmark. Conceptually we can think of a landmark as a pointer to a leaf node without affecting the bounds we describe.

Offset trees can be height-balanced by using splay trees (amortized) or red-black trees (worst-case) [30]. Using offset trees, all the landmark directory operations (described in Section 4.3) take $O(\log L)$ time except for the operation Find_All_Ldmk(*pos_range*) which runs in $O(\log L + B)$ time, where L is the number of landmarks and B is the number of landmarks in the given range. We omit the proof for reasons of brevity.

Space Requirement. For L landmarks, how much space does this offset tree structure require? Each tree has L leaf nodes and $L - 1$ internal nodes. Each node stores an offset value and three pointers (parent, left and right). Each leaf node stores an additional two integers for the size and byte position fields. Assuming 4-byte pointers and integers, this structure requires $(L-1) \times 4 \times 4 + L \times 6 \times 4 \approx 40L$ bytes.

Update Performance. Suppose Δ contiguous words are deleted or inserted in an existing document. Using an offset tree, $O(\log L + B + \Delta)$ time is required to generate the update operations that need to be done on the inverted index and to update the offset tree given the position(s) and the word IDs of the Δ words that are deleted or inserted.

Query Performance. The additional time required to process a conjunctive query of k keywords $key_0 \wedge key_1 \wedge \ldots \wedge key_{k-1}$ is $O\left(\left(\sum_i s_{key_i}\right) \times \log L_{\max}\right)$, where s_{key_i} is the selectivity of keyword key_i and $L_{\max}$ is the maximum number of landmarks per document among all the inverted index entries involved in this query. The selectivity of a keyword is the number of inverted index entries with *word_id* equal to the given keyword.

4.5 Arrays for Landmark Directories

Although offset trees are theoretically efficient, they are not space-efficient for our application in practice. Arrays offer a more practical solution. Several array implementations of a landmark directory are possible depending on the space-time trade-off. We describe one simple implementation here as an illustration and analyze its update and query performance.

For a particular document, suppose there are at most L landmarks and we allocate landmark IDs as integers in $[0, L-1]$. The landmark directory can be implemented as an array of size L indexed by the landmark IDs. Each array entry stores the actual word position of that landmark, a previous landmark pointer, a next landmark pointer, and the actual byte position of the landmark in the document.

Space Requirement. Again assuming 4-byte integer/pointer for each of the four fields (word position, previous landmark, next landmark, and byte position), the space required by the array is $4 \times 4 \times L = 16L$ bytes. Some additional space could be used to accommodate future insertions, or else new space for the array can be allocated when L changes. Thus, arrays provide significant saving in space over the use of offset trees.

Update Performance. The time complexity of the various update operations appears in Table 3. All lookups using a landmark ID takes constant time; hence Find_Pos(*landmark_id*) takes constant time. However, the Find_Ldmk(*pos*) operation requires $O(L)$ time, because the array is not indexed by word position and we have to scan through the array in order to find the corresponding landmark given its position. The Find_All_Ldmk(*pos_range*) operation also takes $O(L)$ time because we need one scan to locate the landmark ID corresponding to the starting position and then we can follow the next landmark pointers and check if the ending position has been reached. Since at most two passes through the array is required, it requires $O(L)$ time.

The insert, delete, shift, DeleteRange operations are not needed if a new updated landmark directory is generated together with the inverted index update operations. This is the update method of choice since only one linear scan is required per document to generate the update landmark directory. Nevertheless, for the sake of completeness, we give brief analysis of these operations.

If we maintain a free list of unused landmark IDs, the insert operation requires only $O(1)$ time. The delete operation requires a constant time lookup, a constant time nullify operation, and a constant time update previous landmark operation.

The shift operation requires changing the word position of all subsequent landmarks. The next landmark pointer is used to traverse through the subsequent landmarks and there are at most $L-1$ subsequent landmarks.

For the DeleteRange operation, we lookup the starting landmark ID and traverse the range of landmark IDs using the next landmark field and nullify each entry; hence if B is the number of landmarks in the given range, DeleteRange takes $O(B)$ time.

Note that the Delete and Shift operations are not required if the landmarking policy relies on the structure of the document (e.g., HTML tags), since we can rebuild the entire landmark directory from the new version of the document.

Compared with the logarithmic time lookup using offset trees, this simple array implementation offers constant time lookup of the word position given the landmark ID, an operation required during query processing. Query performance of arrays is therefore superior to offset trees; update performance requires the linear time reverse lookup operation (given position find the landmark ID) and hence is more expensive compared with offset trees.

4.6 Update Performance Using Arrays

Theorem 3 *Let Δ be the number of contiguous words that were deleted or inserted and L be the number of landmarks in the existing document. Using an array, $O(L + \Delta)$ time is required to generate the update operations on the inverted index and update the landmark directory given the position(s) and the word IDs of the Δ words that are deleted or inserted.*

Proof: In the case where UNIX `diff` is used, the edit transcripts consist of the positions of the words deleted or inserted and the words themselves. For delete, a position range of the words deleted is usually given. To generate the update entries for the deleted words, the operation Find_All_Ldmk(*pos_range*) is called. The operation Find_All_Ldmk(*pos_range*) outputs the landmark ID and offset of the starting word, the landmark ID and offset of the ending word and a list landmark IDs with their corresponding extents. Using this list and the information inside the edit transcript, we can generate the delete entries for the inverted file. The operation Find_All_Ldmk takes $O(L)$ time, where L is the total number of landmarks for that document. Generating the delete landmark entries and applying them on the landmark directory take another $O(L)$, because at most $O(L)$ landmarks can lie in the given range and each delete landmark operation requires constant time. Generating the delete and insert posting entries for the words in the first and last block requires constant time assuming a constant block size. Hence to delete Δ contiguous words, $O(L + \Delta)$ time is required to generate the update entries for the inverted file and update the landmark directory given the edit transcript. A similar argument can be made for inserts. From our data analysis, we can deduce that Δ is typically very small.

In the case where a block-based variant of `diff` such as that described in Section 4.2 is used, an extra access to the old file or new file is required to obtain the words that are deleted or inserted. We show that this additional file access does not degrade the bound obtained in the previous paragraph.

Suppose that the variant `diff` returns an edit transcript in the following format. For delete, it returns the landmark ID of the block to be deleted. For insert it returns to the word position of the insertion point and the byte positions in the new document where the words to be inserted are located.

For deletion of a block of Δ words, we can obtain the word position, the byte position and the extent of the landmark in constant time. Using the byte position, we can access the starting point of the block of text to be deleted in the old document in $O(1)$ time. Reading the Δ words require another $O(\Delta)$ time assuming a constant word length. We can now generate the delete entries for the inverted file and update the landmark directory as before. Since the file access required only $O(\Delta)$ time the overall bound is still $O(L + \Delta)$ time. The argument for insert is analogous. $\square$

4.7 Query Performance Using Arrays

Theorem 4 *The additional time required to process a conjunctive query of k keywords $key_0 \wedge key_1 \wedge \ldots \wedge key_{k-1}$ is $O(\sum_{i=0}^{k-1} s_{key_i})$, where s_{key_i} is the selectivity of keyword key_i.*

Proof: Recall how conjunctive queries are processed. For each keyword key_i, we obtain from the inverted index a list l_i of inverted index entries of the form $\langle word\_id, doc\_id, loc\_id\rangle$. The the selectivity of keyword key_i is $s_{key_i} = |l_i|$, the number of inverted index entries with *word_id* equal to key_i. Entries from documents that do not contain all the keywords are filtered out of each l_i.

If the query post processing does not require the position of each occurrence of the word in a document, then we do not need to access the landmark directories and no additional cost is incurred. If positional information is needed (for example, to compute some relevant metric), then there is an additional $s_{key_i} \times O(1)$ time overhead to look up the actual position of each of the s_{key_i} landmark IDs for a keyword key_i, assuming in the worst case that the the landmark IDs are all distinct. Hence, for a query consisting of a list of k keywords $\{key_i \mid 0 \leq i < k\}$ joined by the boolean operator AND, the additional overhead is $O(\sum_i s_{key_i})$ time. Note that an offset tree implementation of the landmark directory is an order of magnitude slower, because each lookup requires $O(\log L)$ time (see Table 3). □

5 Experimental Evaluation

In this section we describe the experiments used to evaluate our landmark-diff method. We compare the performance of the landmark-diff method with the forward index method, because both methods are designed for updating common documents whose contents have changed. For the performance on general index update, we compare the the partial rebuild method using landmark-diff with the complete rebuild method, because the complete rebuild method is the industry workhorse.

Through our experiments, we answer five important questions about the performance of our landmark-diff method:

1. How does landmark or block size affect performance?

2. Does the landmark-diff method significantly reduce the number of edit operations on the inverted index compared to other methods (e.g., forward index method)?

3. Does the reduction in the number of inverted index update operations actually translate to savings in real execution time when applying these operations?

4. Does generating update operations using the
landmark-diff method require more time than other methods?

5. Does the landmark-diff method provide a more efficient solution for the general inverted index maintenance problem than complete rebuild method, especially when the change between two consecutive samples is large?

Our Implementation. Our text indexing system is implemented in C. Two implementations of the inverted index have been used. The binary tree (of postings) implementation is used for measuring the time required to apply update operations on the index, because the binary tree represents the worst case tree-based data structure. We discuss a B-tree representation in Section 5.4 which will take better advantage of our landmark-diff method when updates are very frequent and therefore very small in magnitude. A b-way search structure implicitly represented by a linear array of postings is used for the comparison between the partial rebuild method and the complete rebuild method, because both methods rely on an external k-way merge sort and linear arrays give the complete rebuild method the most advantage. Landmark directories and forward indexes are implemented using linear arrays, because these are small data structures.

Landmarking Policy and Block Size. Fixed size partitioning is used to choose landmarks. Intuitively, the square root of the average document size is a good block size[3]. For the experiments we present, a default block size of 32 words is chosen since the average document size is roughly 1000 words. Insertion of words always split a block.

Performance Measures. We evaluate the performance of our landmark method using four measures: (1) the number of inverted index update operations (Table 4); (2) the time needed to perform those operations (Table 5); (3) the time needed to generate those operations (Table 6); (4) the time needed to bring the inverted index up-to-date using the partial rebuild method (Table 7).

An edit operation is defined to be a delete posting or insert posting operation. The number of edit operations on the inverted index is a natural performance measure, since the goal of a good update method is to reduce the number of edit operations on the inverted index. Moreover, unlike the execution time, the number of edit operations depends neither on the implementation of the system nor on the hardware architecture. The number of edit operations is therefore a good measure of update performance across different search engine architectures and implementations.

Data Set. Two sets of data are crawled from the web. Data Set I consists of two samples of the web crawled from 100 seed web sites. The time between the start of the two web crawls is 71 hours and the recursion depth is limited to 5 levels. The first sample has 63,336 documents and the second has 64,639 documents with 37,641 documents common to both. Each sample contains about 1.5 GB worth of HTML files. Data Set II consists of 6 samples of the web crawled from 5 seed web sites (www.cnn.com, www.ebay.com, www.yahoo.com, espn.go.com, and www.duke.edu). The sampling interval is 12 hours and the recursion depth is 5 levels. Note that 4 out of the 5 listed web sites have fast changing contents.

Document Preprocessing. Every HTML file is preprocessed into a canonical form by stripping off HTML tags, scripting code, and extra white space. Every letter is capitalized so that different capitalizations do not result in different word IDs. If the canonical form of a file has less than 10 words, it is discarded.

5.1 Block Size and Performance

In this section, we address two questions: how does block size affect the number of inverted index update operations, and how does block size affect the time to generate these operations.

We measured the minimum number of index update operations required for different block sizes (using data set I described previously) and verified that as the block size gets smaller the number of index update operations decreases at the expense of increasing landmark directory size. Figure 12 shows the distribution of the documents with respect to the number of index update operations normalized by the sum of the sizes of the old and new documents. While small block sizes result in less update operations on the index, it also results in larger landmark directories, which translates to an increase in the cost of manipulating the landmark directory.

[3]The reasoning is similar to a 2-level B-tree where the number of blocks and the block size is 'balanced'.

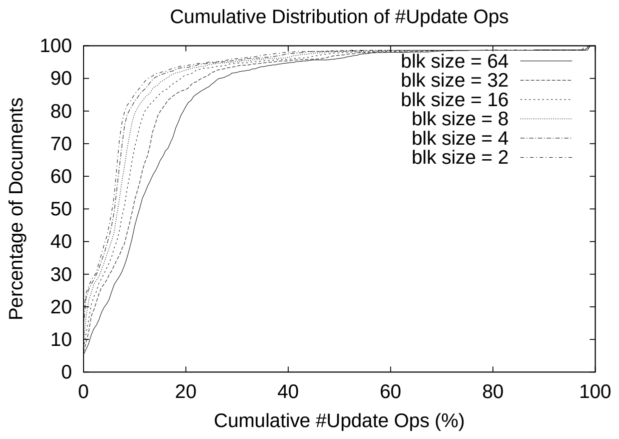


Figure 12: Cumulative distribution of documents with respect to the number of inverted index update operations for different block sizes. Note that the lines from right to left are in the same order as the labels in the legend (top-down), so the right most line is for block size 64. If a vertical line is drawn at 20 % cumulative update operations, the intersection with each of the plotted lines gives the percentage of documents that each require less than 20 % number of update operations.

The increasing cost of manipulating the landmark directories as block sizes decreases can be seen in our experiment for investigating how the landmark size affects the time required to generate the update operations for the common documents (data set I) that have changed. Figure 13 shows that smaller landmark size results in less processing time to generate update operations for block sizes greater than eight words. For block sizes less than eight words, the cost of manipulating large landmark directories dominate and the processing time increases. Note that the block size that minimizes the time to generate update operations is dependent on the magnitude and clusteredness of change of the data as well as the implementation of the system.

In principle it is possible to determine the block size that minimizes the time for the entire update process for common documents given a data set and a particular system implementation. However, that is computationally intensive, and the computed optimal block size is not very useful, because it is dependent on the data, which change over time. One approach to reduce the computational overhead is to compute the optimal block size using a small subset (a statistical sample) of the data. A simple strategy to deal with changes in data characteristics is to recompute the block size during periodic index re-organization (de-fragmentation).

5.2 Performance on Common Documents

We compare the update performance of the landmark-diff method on common documents with the forward index method.

Number of Update Operations. We count the number of inverted index update operations generated by the forward index method and our landmark-diff method on two data sets. Fixed size landmarking policy is used with a block size of 32 words. Table 4 shows that the landmark-diff method generates significantly less update operations on the inverted index than the forward index method (which represents the best of the naive methods). The performance of the landmark-diff update method is consistent over a broad range of web sites including web sites with fast changing content. We indicate in Section 5.4 that this measure will mirror real-world performance when updates are frequent.

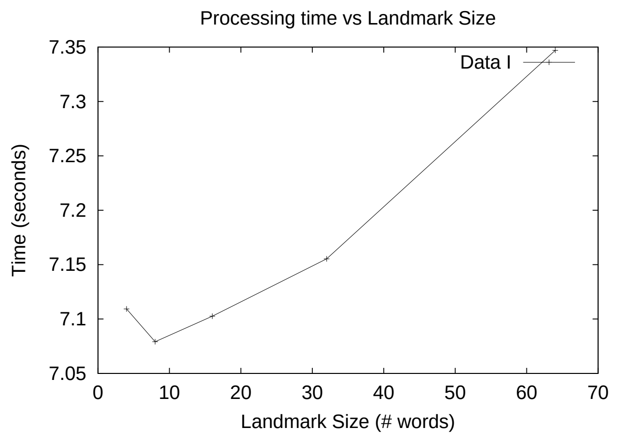


Figure 13: Landmark size versus time to generate update operations for common documents that have changed.

Data	$\mathcal{C}$(docs)	$\Delta\mathcal{C}$(docs)	Fwd. Index	Landmark
I	37,641	10,743	10,501,047	3,360,292
IIa	2,026	1,209	1,109,745	330,695
IIb	5,456	1,226	1,566,239	350,802
IIc	5,335	3,096	1,855,618	534,088
IId	5,394	1,278	1,426,163	378,661
IIe	5,783	1,605	1,762,018	539,594

Table 4: The number of update operations generated by the update methods in our experiments. The symbol $\mathcal{C}$ denotes the number of common documents ($|S_n \cap S_{n+1}|$) and the symbol $\Delta\mathcal{C}$ denotes the portion of $\mathcal{C}$ that has changed.

Method	$\lvert S_n \rvert$	No. Update Ops.	Time (s)
Fwd. Index	63,336	10,501,047	28.3
Landmark	63,336	3,360,292	9.2

Table 5: Time required to apply the update operations on a binary tree implementation of the inverted index. The symbol $|S_n|$ denote the number of documents in the index and corresponds to the index size.

		Generation Time (s)	
Method	No. Update Ops	$\mathcal{C}$	$\Delta\mathcal{C}$
Fwd. Index	10,501,047	148.9	18.9
Landmark	3,360,292	17.5	7.5

Table 6: The time (in seconds) required to generate the inverted index update operations for the landmark-diff method and the forward index method. The $\mathcal{C}$ column gives the required time if all the common files (37,641) have to be checked and processed. The $\Delta\mathcal{C}$ column gives the required time, if incremental crawling is used and the system knows *a priori* whether a common file has been modified since the last crawl.

Update Time. Do the reductions in the number of inverted index update operations actually translate to savings in the time to apply these operations? We measure the execution time for applying the update operations generated by the forward index method and the landmark-diff method to a binary tree implementation of the inverted index for data set I. The experiment is performed on a Sun Blade-1000 computer running SunOS 5.8 with 4 GB of RAM. Our results as summarized in Table 5 show that a reduction in the number of update operations does translate to a proportional reduction in the time required to update the inverted index. Note that the measured time does not include the time for updating the landmark directories, because we are interested in the time required to update the inverted index data-structure. The landmark directory is a per document data-structure and therefore is small and can be updated in parallel for each document. The inverted index, on the other hand, is a data-structure for a collection of documents that is used for query processing. The update processing time on the inverted index is therefore critical to the performance of the retrieval system. Updating each landmark directory requires only one linear pass through the associated document and can be done at the same time as generating the update operations (see Section 5.2).

Time for Generating Update Operations. Does generating update operations using the landmark-diff method require prohibitively more time than the forward index method? We measured the time used to generate the update operations for data set I using the same configuration as in Section 5.2. For the forward index method, the measured time includes the time to read each pair of old and new document from disk, the time to create forward indexes for them, and the time to compare the forward indexes and generate update operations. For the landmark-diff method, the measured time includes the time to read each pair of old and new document from disk, the time to performing a `diff` on them, the time to translate the `diff` output to update operations, and the time to generate the updated landmark directory. Table 6 shows that for common documents that have changed the landmark-diff method is roughly three times as fast as the forward index method in generating update operations. This speedup is roughly the ratio of the output size (the number of update operations). Observe that `diff` is a more efficient method for detecting identical, i.e. unchanged, documents than the forward index method. In our experiments, the landmark directories are stored in memory, but even if the landmark directories are stored on disk, the additional time required to read a landmark directory into memory is still small compared with reading a document, since it is much smaller in size.

For common documents, our experiments have shown that the landmark-diff method does indeed

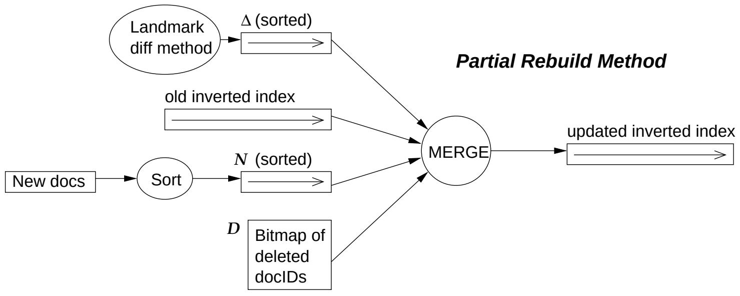


Figure 14: The schematic diagram for the partial rebuild method. Our landmark-diff method produces a list of update operations Δ which is then sorted. The new documents are processed into a sorted list of postings $\mathcal{D}$ to be inserted. The *doc_id*s of the deleted documents are stored in a bitmap. These three data structures are merged with the old inverted index in a linear pass.

	No. of docs	No. of postings
$\mathcal{D}$	25,695	-
$\mathcal{C}$	37,641	58,575,596
$\Delta\mathcal{C}$	10,743	-
Δ	-	3,360,292
$\mathcal{N}$	26,998	43,366,308
Partial Rebuild	513.8 s	
Complete Rebuild	1113.7 s	

Table 7: Running time performance of the partial rebuild method using landmark-diff and the complete rebuild method. The symbols $\mathcal{D}$, $\mathcal{C}$, $\Delta\mathcal{C}$, Δ and $\mathcal{N}$ denote the deleted documents, the common documents, the common documents that have changed, the update operations for the common documents and the new documents that have been inserted. We give the sizes of these sets in units of documents as well as postings.

reduce the number of update operations as compared to the forward index method. Moreover, the reduced number of update operations does translate to speedups of roughly the same ratio in update generation and application time. We next show how the landmark-diff method can be used for efficient general index update (not restricted to common documents).

5.3 Performance on General Index Update

We investigate the efficiency of using the landmark-diff method for solving the general index update problem. We compare the performance of the partial rebuild method using landmark-diff (see Section 3.4) with that of the complete rebuild method.

In both methods, the inverted index is implemented as a sorted array of postings residing on disk (searching can be accomplished using an implicit complete b-way tree similar to a binary heap implemented as an array). For the complete rebuild method, the entire updated document collection is scanned into a list of postings and sorted using external k-way merge sort and written out to disk. For the partial rebuild method, the original and updated document collection is scanned to obtained (1) a list of document IDs of the deleted documents stored as a bit map in memory, (2) a sorted list of postings of the inserted documents stored as an array in memory, and (3) a sorted list of index update operations for the common documents, generated using our landmark-diff method, and stored as an array on disk. The old inverted index (on disk), the bit map, the array of new postings, and the sorted update operations are merged (in a fashion similar to mergesort) into an updated inverted index and written to disk (see Figure 14).

The two update methods are applied on data set I and the elapsed time measured. Table 7 shows that the partial rebuild using landmark-diff method is twice as fast as the complete rebuild method even for two web samples that are crawled 71 hours apart and thus have a large number of changed common documents and inserted documents. The partial rebuild method illustrates that the efficient update processing of the common documents can improve overall efficiency by avoiding the expensive re-indexing of the portion of the common documents that did not change. Hence, the partial rebuild method is more efficient because it only has to perform the expensive indexing operation on the new documents, whereas the complete rebuild method has to index both the updated common documents and the new documents.

Observe that the time required to process common documents (Table 5 and 6) is quite small compared to the overall update time of the partial rebuild method using landmark-diff (Table 7). A large portion of the elapsed time for the partial rebuild method is spent on the expensive indexing operation done on the new documents. One could argue that optimizing updates of common documents is meaningless because the bottleneck lies in indexing the new documents; however, the goal of this work is incremental updates. In the incremental update scenario, the time between updates is small and the new documents that arrive between updates are also fewer. Update processing of the common documents would account for the bulk of the processing time in incremental updates. If the set of new documents is significantly large, other ways of dealing with the new documents can be used. For example, a separate index can be built for the new documents as suggested in Section 3.4.

5.4 Discussion

Updating inverted indexes given a new sample of the web involves four sets of items:

1. the postings of deleted documents $\mathcal{D}$,
2. the postings for inserted/new documents $\mathcal{N}$,
3. the postings for original common documents $\mathcal{C}$, and
4. the update operations Δ for $\mathcal{C}$.

The existing inverted index (consisting of $\mathcal{C}$ and $\mathcal{D}$) has to be 'merged' with the sets $\mathcal{D}$, $\mathcal{N}$ and Δ, so that $\mathcal{D}$ is deleted, $\mathcal{N}$ is inserted, and Δ is applied to the inverted index (the alternative is to rebuild from scratch). The set $\mathcal{D}$ can be processed very efficiently by storing the *doc_id*s using a bit map. Updating the index with Δ can be done very efficiently using our landmark-diff method (three times faster than the forward index method). The efficiency of incorporating $\mathcal{N}$ to the inverted index will depend on the size of $\mathcal{N}$.

The partial rebuild method using landmark-diff exploits the following three facts to achieve the factor of two speedup compared to the complete rebuild method: (1) $\mathcal{N}$ can be processed in a sequential manner very efficiently; (2) our landmark-diff method produces a very small Δ relative to $\mathcal{C}$ very efficiently; (3) the landmark-diff method avoids re-indexing the portion of $\mathcal{C}$ that did not change. The goal of this paper is to investigate fast incremental update of inverted index. As the sampling interval decreases, the sizes of $\mathcal{N}$ and Δ also decrease relative to $\mathcal{C}$ [9]. In the limit, $\mathcal{N}$ will be very small and random access updates to an inverted index implemented as a B-tree will be faster than an array implementation. A similar trade-off between random access data structures and stream-based processing has been observed in the processing of spatial joins [2]. Therefore, the speedup in incremental update time using a B-tree implementation approaches

$$\frac{|\mathcal{C}| + |\mathcal{N}| + |\Delta|}{|\mathcal{N}| + |\Delta| + |\mathcal{D}|} \tag{3}$$

compared with a complete rebuild. We have showed in Table 7 that the array-based partial rebuild using landmark-diff is twice as fast as complete rebuild for a relatively large update interval of 71 hours. As the update interval decreases, the speedup will be more significant and even more dramatic if we use a B-tree implementation of the inverted index.

6 Conclusion

Keeping web indexes up-to-date is an important problem for research and in practice. Naive update methods such as index rebuild is inadequate in keeping the inverted index up-to-date especially in the context of in-place, incremental crawling. That web document changes are generally small and clustered, motivates the use of incremental update algorithms based on `diff`. However, storing positional information in the inverted index presents a problem in using the `diff` approach. The landmark representation that we proposed allows the `diff` approach to be used to efficiently update inverted indexes that store positional information. We show that the performance of our method is theoretically sound and our experiments show that our method results in significant savings in the number of edit operations performed on the inverted index and hence in the update time. We further showed how the landmark-diff method can be used with partial rebuild to update inverted index in half the time it takes to rebuild the index from scratch. One important insight is that not re-indexing the portion of the document collection that did not change can result in more efficient update processing. The implication for incremental updates is significant, because smaller time intervals between updates mean that more of the document collection remains unchanged. An orthogonal issue not addressed in this paper is update processing for documents whose content did not change, but whose URL or document ID changes. In the spirit of not re-indexing content that did not change, such updates in URL should be detected and re-indexing avoided.

References

[1] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. Document Object Model Level 1 (W3C Recommendation), 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. *Proceedings of the 7th Intl. Conf. on Extending Database Technology (EDBT '00)*, 1777:413–429, 2000.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[4] Ricardo A. Baeza-Yates and Gonzalo Navarro. Block addressing indices for approximate text retrieval. *Journal of the American Society on Information Systems*, 51(1):69–82, 2000.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1976.

[6] Brian Brewington and George Cybenko. Keeping up with the changing web. *IEEE Computer*, 33(5):52–58, May 2000.

[7] Andrei Z. Broder. On the resemblence and containment of documents. *Proceedings of Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[8] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *20th Intl. Conf. on Very Large Data Bases*, pages 192–202, 1994.

[9] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. *26th Intl. Conf. on Very Large Data Bases*, 2000.

[10] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change. *ACM Transactions on Internet Technology*, 3(3):256–290, 2003.

[11] C. Clarke and G. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system. *Tech. Report CS-95-01, Univ. of Waterloo CS Dept.*, 1995.

[12] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with on-line update. *Tech. Report CS-94-40, Univ. of Waterloo CS Dept.*, 1994.

[13] Doug Cutting and Jan Perdersen. Optimizations for dynamic inverted index maintenance. *Proceedings of SIGIR*, pages 405–411, 1990.

[14] Malcom C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, pages 230–241, May 1986.

[15] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *Proc. of the 12th Intl. conf. on World Wide Web*, pages 669–678. ACM Press, 2003.

[16] Michael J. Fischer and Richard E. Ladner. Data structures for efficient implementation of sticky pointers in text editors. *Dept. of Computer Science, Univ. of Washington, Tech. Report 79-06-08*, June 1979.

[17] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[18] Marcin Kaszkiel and Justin Zobel. Passage retrieval revisited. In *Proceedings of the 20th Annual Intl. ACM SIGIR Conf.*, pages 178–185. ACM, 1997.

[19] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[20] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[21] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *27th Intl. Conf. on Very Large Data Bases*, pages 361–370, 2001.

[22] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh C. Agarwal. Characterizing web document change. In *Advances in Web-Age Information Management, 2nd Intl. Conf., WAIM 2001*, pages 133–144, 2001.

[23] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the Winter 1994 USENIX Conf.*, pages 23–32. USENIX, 1994.

[24] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *Proceedings of the 10th Intl. WWW Conf.*, 2001.

[25] Lawrence Page and Sergey Brin. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the 7th Intl. WWW Conf.*, pages 107–117, 1998.

[26] Gerard Salton, James Allan, and Chris Buckley. Approaches to passage retrieval in full text information systems. In Robert Korfhage, Edie M. Rasmussen, and Peter Willett, editors, *Proceedings of the 16th Annual Intl. ACM-SIGIR Conf*, pages 49–58. ACM, 1993.

[27] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. *Proceedings of 1994 ACM SIGMOD Intl. Conf. of Management of Data*, pages 289–300, May 1994.

[28] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[29] Jeffrey S. Vitter. Faster methods for random sampling. *Communications of the ACM*, 27, July 1984.

[30] Jeffrey S. Vitter. An efficient I/O interface for optical disks. *ACM Trans. on Database Systems*, pages 129–162, June 1985.

[31] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[32] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of 2001 ACM SIGMOD Intl. Conf. of Management of Data*, pages 361–370, 2001.

[33] Justin Zobel, Alistair Moffat, Ross Wilkinson, and Ron Sacks-Davis. Efficient retrieval of partial documents. *Information Processing and Management*, 31(3):361–377, 1995.