Trustworthy Keyword Search for Regulatory-Compliant Records Retention

Soumyadeb Mitra[*]
Dept. of Computer Science
University of Illinois at Urbana-Champaign
mitra1@cs.uiuc.edu

Windsor W. Hsu
CS Storage Systems Dept.
IBM Almaden Research Center
windsor@almaden.ibm.com

Marianne Winslett[†]
Dept. of Computer Science
University of Illinois at Urbana-Champaign
winslett@cs.uiuc.edu

ABSTRACT

Recent litigation and intense regulatory focus on secure retention of electronic records have spurred a rush to introduce Write-Once-Read-Many (WORM) storage devices for retaining business records such as electronic mail. However, simply storing records in WORM storage is insufficient to ensure that the records are trustworthy, i.e., able to provide irrefutable proof and accurate details of past events. Specifically, some form of index is needed for timely access to the records, but unless the index is maintained securely, the records can in effect be hidden or altered, even if stored in WORM storage. In this paper, we systematically analyze the requirements for establishing a trustworthy inverted index to enable keyword-based search queries. We propose a novel scheme for efficient creation of such an index and demonstrate, through extensive simulations and experiments with an enterprise keyword search engine, that the scheme can achieve online update speeds while maintaining good query performance. In addition, we present a secure index structure for multi-keyword queries that supports insert, lookup and range queries in time logarithmic in the number of documents.

1. INTRODUCTION

Documents such as electronic mail, financial statements, meeting memos, drug development logs, and quality assurance documents are valuable assets. Key decisions in business operations and other critical activities are based on information in these documents, so they must be maintained in a trustworthy fashion—safe from improper destruction or modification, and readily accessible. Businesses increasingly store these documents electronically, making them relatively easy to delete and modify without leaving much of a trace. Ensuring that records are readily accessible, accurate, credible, and irrefutable is particularly imperative given recent legal and regulatory trends. The US alone has over 10,000 regulations that mandate how records should be managed. Many of those focus on ensuring that records are trustworthy (e.g., Securities and Exchange Commission (SEC) Rule 17a 4 and the Sarbanes-Oxley Act).

This has led to a rush to introduce Write-Once-Read-Many (WORM) storage devices (e.g., [8, 18, 23]) to enable proper records retention. However, storing records in WORM storage is inadequate to ensure that they are trustworthy, i.e., able to provide irrefutable evidence of past events. The volume of records and stringent response time requirements dictate the use of a direct access mechanism such as an index to access the records. Furthermore, the records are likely to be accessed not only during litigation and audits, but also as an integral part of day-to-day business activity—companies prefer to maintain only a single copy of each record if possible, due to the cost of maintaining multiple copies and the need for trustworthy input to business decisions.

If the index through which a record is accessed can be suitably manipulated, the record can, for all practical purposes, be hidden or deleted, even if it is stored in WORM storage. For example, if the index entry pointing to the record is removed, or made to point to a different record, the original record becomes inaccessible. Hence the index itself must be maintained in a trustworthy fashion.

To address this issue, researchers proposed the concept of a *fossilized index*, which is impervious to such manipulations. One such index is the Generalized Hash Tree (GHT) [29] which supports exact-match lookups of records based on attribute values and hence is most suitable for use with structured data. However, most business records, such as email, memos, notes, meeting minutes, etc., are unstructured or semi-structured. The natural query interface for these documents is keyword search, where the user provides a list of keywords and receives a list of documents that contain some or all of the keywords. Keyword based searches are typically handled by an *inverted index* [9].

In this paper, we analyze the requirements for a trustworthy index for keyword-based search. We argue that trustworthy index entries must be durable—the index must be updated when new documents arrive, and not periodically deleted and rebuilt. To this end, we propose a scheme for efficiently updating an inverted index, based on judicious merging of the posting lists of terms. Through extensive

[*]This research was partially supported by an IBM internship.
[†]This research was supported by NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695.

simulations and experiments with an IBM intranet search engine, we demonstrate that the scheme achieves online update speed while maintaining good query performance. We also present and evaluate *jump indexes*, a novel trustworthy and efficient index for join operations on posting lists for multi-keyword queries.

The rest of this paper is organized as follows. In Section 2, we discuss the threat model, analyze related work, and derive the key requirements for a trustworthy inverted index. We also propose enhancements to WORM storage to facilitate such an index. In Section 3, we develop the idea of merging posting lists to enable online update of inverted indexes. We present a trustworthy indexing scheme for posting lists in Section 4. In Section 5, we discuss a rank-based attack and propose countermeasures. Section 6 concludes the paper.

2. ISSUES

2.1 Threat Model

We are concerned with a very specific threat model: a legitimate user Alice creates a document (record R) and commits R to WORM storage, through an application. After R has been committed, a user Mala begins to regret its existence. Mala will do everything she can to prevent a future user Bob (e.g., a regulatory authority) from receiving R as the answer to one of his queries.

Coverups are often directed by high-level company insiders (e.g., CEOs and CFOs). To model these attacks, we assume that Mala can take on the identity of any legitimate user or superuser in the system, and perform any action that person can perform. For example, Mala can write any data to the WORM device as long as the write does not overwrite existing data, and she can read any data on the device. This means that we cannot rely on conventional file/storage system access control mechanisms [12, 22] to ensure that documents and indexes are only modifiable by legitimate applications. However, we assume that physical access to the WORM device is restricted or monitored so that Mala cannot steal or destroy it without raising red flags and triggering suspicion and a presumption of guilt. We also assume that Bob is sufficiently cautious that he will check to make sure he is running a certified version of the search engine and operating system, so Mala cannot alter Bob's search engine or redirect Bob's I/O requests at the file system level. Similarly, we trust the document insertion application Alice uses to commit R (i.e., R does reach WORM storage initially), and assume the WORM device operates properly (i.e., it never overwrites data).

No one regrets the existence of R until R is already permanently in WORM storage. Thus Mala's only hope is to keep R out of the index that Bob uses in his search. She can do this by preventing R from ever getting into the index, or by ensuring that R is not in the index Bob uses. This suggests a strategy for us. First, we can ensure that R is entered in the index *before* Mala regrets R's existence. Second, we can ensure that any data that ever enters the index stays accessible through it forever (or at least for a mandated retention period). In other words, *the index should be trustworthy in the sense of it being in WORM storage and never "losing" any old entries when new entries are added.*

To stop Mala from preventing R from entering the index, one approach is to insert R and construct the index entry for R as a single action, because we trust the document insertion code to get R into WORM storage initially. If Mala alters the document insertion code after R is inserted, R will still be on WORM storage, so we do not need to trust the document insertion code once R has been inserted. If Mala alters the index update code, that alteration will take place after R has been entered into the index. Thus we must ensure that the altered index creation code, altered document insertion code, or any other application of Mala's cannot hide R's index entry from the search engine.

2.2 Storage Model

Magnetic recording currently offers better cost and performance than optical recording. Moreover, while immutability is often specified as a requirement for records, what is required in practice is that the records be "term-immutable", i.e., immutable for a specified retention period. Thus almost all recently-introduced WORM storage devices are built atop conventional rewritable magnetic disks, with write-once semantics enforced through software [8, 18, 23]. The software running on these WORM boxes provides a file-system-like (or object) interface to the external world, but with file modification and premature deletion operations disallowed.

Such devices allow files to be written and to subsequently have their contents committed. However, this interface is too restrictive for WORM indexes. For efficiency, we need the ability to commit small amounts of data without creating a new file. For example, to add a new document to the inverted index, we need the ability to append the document ID to individual posting lists (maintained as separate files). Furthermore, for creating index structures like jump indexes (introduced later), we also need the ability to append new bytes to partially-written file blocks.

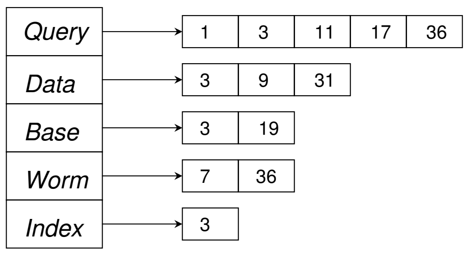
Since rewritable media are the underlying medium, the WORM device's interface can be extended to support append operations on otherwise immutable files and blocks [17]. Based on discussions with several storage vendors, we believe such an extension to the current interface can be accomplished relatively easily. In the remainder of this paper, we assume that such a WORM device is used for the inverted indexing, while a conventional WORM is used for the documents themselves.

2.3 Requirements and Related Work

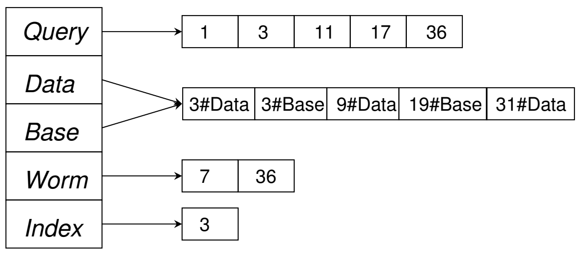
Search engines typically use inverted indexes to support keyword search [9]. As shown in Figure 1(a), an inverted index comprises a dictionary of keywords and associated *posting lists* of document identifiers (with additional metadata such as keyword frequency, type, position) for each keyword.

In a trustworthy index, the posting list entries for a document must be durable, and the path to each entry must also be durable. This can be achieved by keeping each posting list in an append-only file in WORM storage. The index can be updated when a new document is added, by appending its document ID to the posting lists of all the keywords it contains. Unfortunately, this operation can be prohibitively slow, as each file append will require a random I/O. For example, in the data set used in our experiments, each document contains almost 500 keywords on average. If each append incurs a 2 msec random I/O, it would take 1 second to index a document.

A number of solutions have been proposed for supporting efficient inverted index updates [3, 6, 10, 11, 16, 27]. The



(a) Inverted Index



(b) Inverted Index After Merging

Figure 1: Posting Lists. With each keyword, a *posting list* of documents containing that keyword is stored. After merging, the keyword (or its hash) must also be stored in the posting list.

underlying principle behind these approaches is to amortize the cost of random I/O by buffering the index entries of new documents in memory or disk and committing them to the index in batches. Specifically, the keywords of newly arriving documents are appended to an in-memory or on-disk log containing ⟨*keyword, document ID*⟩ pairs. This log is periodically sorted on *keyword* to create an inverted index for the new documents, which is then merged with the original inverted index. However, this strategy is effective only when a huge number of index entries are buffered. For example, researchers have found that one must buffer over 100,000 documents to achieve a document commit rate of 2 documents per second on a 20 GB collection [21]. Buffering creates a time lag (100,000/2 seconds, or half a day, in the previous example) between when a document is created and when the index on WORM is updated. For trustworthy indexing, we cannot leave such a gap between document commit and index update—Mala can get rid of an index entry while it is still in the buffer, or crash the application and delete the recovery logs of uncommitted posting entries. Keeping the recovery logs on WORM is not a solution because scanning the entire log on every restart would be very inefficient, while relying on an end-of-committed-log marker would be insecure. Mala can append such markers to fool the application into believing that no recovery is required.

The time lag between when a document is committed and when someone may regret its existence is domain-specific and has no a priori lower bound. Furthermore, any delay in committing index entries introduces unnecessary risk and complexity into the compliance process. For example, the prevailing interpretation of email retention regulations is that a regulated email must be committed as a record *before* it is delivered to a mailbox. Thus generic compliance indexing should not assume any safe time window for committing index entries after returning to the invoking application—a trustworthy index should be updated in real time.

Search engines answer multi-keyword conjunctive queries (queries in which all the keywords must be contained in the document) by calculating the intersection of the posting lists of the query keywords. To make this fast, additional index structures such as B+ trees are typically maintained on the posting lists. Mala can effectively conceal a document if she can omit it from these posting list indexes. In Section 4, we show that Mala can conceal entries from B+ trees even if they are maintained in WORM storage, by appending malicious entries.

The problem of secure indexing has been explored before, but under a different threat model. Merkle hash trees let one verify the authenticity of any tree node entry by trusting the signed hash value stored at the root node. Authenticated dictionaries [2, 13] support secure lookup operations for dictionary data structures. These and many other proposals are designed for the outsourcing threat model, where the data store is untrusted [15]. The correctness of outsourced query answers is guaranteed by the data owner by attaching appropriate signatures that can be verified by the querier. These approaches rely on the data owner to sign data and index entries appropriately. In the compliance storage threat model, Mala can take on the data owner's identity, so she could modify outsourced data/indexes and re-sign them. Another key difference from the outsourcing threat model is that we trust the data store (WORM device) not to delete committed records.

Indexing methods for WORM (e.g., [1, 7, 20, 24]) were motivated by the desire to store data on optical disks, which once had an advantage over magnetic disks in terms of cost and storage capacity. These methods were designed for minimizing storage overhead and maximizing performance, and do not provide trustworthy recordkeeping.

3. ONLINE UPDATE OF POSTING LISTS

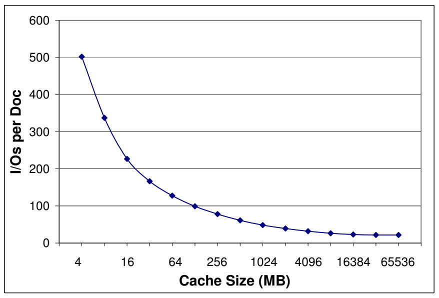


Figure 2: Random I/Os Per Inserted Document.

As pointed out in the previous section, an inverted index must be updated in real time to ensure compliance. We can reduce the cost of incremental posting list updates by caching the list tails in the storage cache. Modern storage devices (storage servers) have on-board non-volatile caches, with sizes ranging from a few hundred megabytes to a few gigabytes. Data in the non-volatile cache of a WORM device are effectively committed to WORM storage, from the application's point of view. To determine the effectiveness of caching for improving the performance of index insertions, we simulated the incremental insertion of one million documents obtained from IBM's intranet into an initially empty index. In the simulation, the tail blocks of as many posting lists as possible are cached in the storage server's (initially dirty) cache. If there is a cache hit when writing an index entry, then no I/O occurs (unless the block becomes full, in which case it is written out). If there is a cache miss, then the least recently used cache block is written out, and the needed block is read.

Figure 2 shows the average number of random I/Os per added document, as a function of the cache size. The curve levels off slowly due to the Zipfian distribution [30] of the keywords in typical document databases—most words occur in very few documents, and posting list updates for these rare words benefit little from caching. Even for very large caches beyond 4 GB, the number of random I/Os remains very high, at about 21 per document on average. Because caching is performed at the granularity of disk blocks, a random I/O is incurred for writing out an evicted posting list block even if the block is not yet full.

We can markedly improve the situation by merging multiple posting lists together as shown in Figure 1(b). If we merge posting lists until the total number of lists is no more than the number of cache blocks in the storage device, then all posting list updates will hit the cache. Also, by merging posting lists smaller than one block into larger lists, we reduce the number of random I/Os by decreasing the number of partial blocks written out. With this scheme, an index update requires I/O only when a cache block fills up and must be written to disk. Assuming 4 KB blocks and 500 8-byte postings per document, we incur, on an average, $\frac{500*8}{4096}=1$ random I/O per document insertion. This is a factor of 20 speedup over simply using a cache of 4 or more gigabytes, and 500 speedup over the uncached case.

Merging, however, has associated costs:

• Query response time will increase because longer posting lists must be scanned. For example, a query for the keyword 'Data' in Figure 1(b) now requires scanning five posting list entries, as compared to only three in the unmerged case.

• To remove false positives, we must store (an encoding of) the keyword with each entry in a merged list. The encoding can be stored in $\log(q)$ bits, where q is the number of posting lists that are merged together. This overhead can be reduced further if an encoding scheme like Huffman encoding is used, since keyword occurrences within merged posting lists are unlikely to be uniformly distributed. This encoding can be added to the metadata in each list entry. We do not include this overhead in our analysis, but it would appear as a constant factor overhead in the query cost results in Section 3.3.

The challenge, therefore, is to devise merging strategies that have minimal impact on query performance.

3.1 Query Cost and Optimization Problem

Keyword queries are answered by scanning the posting lists of the terms in the query[1]. The documents in the posting lists are assigned scores based on similarity measures like cosine [28] or Okapi BM-25 [25]. The scores are used to rank the documents.

The total workload cost for answering a set of queries can hence be modeled as follows: Let q_i be the number of queries that contain the ith term (its *query frequency*), and let t_i be the length of the unmerged posting list for the ith term (its *term frequency*). The total workload cost without any merging is proportional to

$$\sum_{1\leq i\leq n} t_i * q_i.$$

We wish to merge the n posting lists into M lists, where M is the number of cache blocks available in the storage device, in a manner that will minimize query response time. If we merge the posting lists into M lists, $A_1, \ldots, A_M$, then the workload cost is proportional to

$$Q = \sum_{1\leq i\leq M} (\sum_{k\in A_i} t_k)(\sum_{k\in A_i} q_k). \qquad (1)$$

The original scan of the ith posting list has been replaced by a scan of all the posting lists merged with the ith list. The optimization problem is to choose the sets $A_1, \ldots, A_M$ that minimize the total workload cost Q, given M, q_i and t_i. Unfortunately, this problem can be shown to be NP-complete by reduction from the minimum sum of squares problem [5]. The challenge, therefore, is to design heuristic solutions that perform well in practice. We propose and analyze two such schemes in Section 3.3.

3.2 Experimental Data

Confidentiality concerns make it difficult to obtain query workloads for corporate email and documents, which are the primary targets of this work. We are aware of only one publicly available business email archive [19], and it has no query log. Since business intranet queries typically are searching for specific information, such as specific business or policy documents, or for specific web pages, we believe that they are a reasonable approximation to a record retention workload. Therefore, our experiments use a collection of one million documents crawled by an IBM intranet search engine. For the query workload, we use 300,000 actual user queries logged by the search engine.

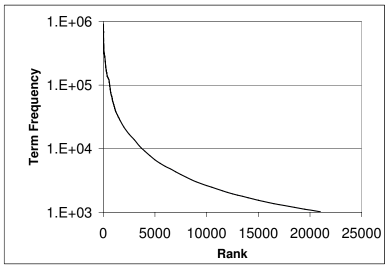
3.3 Merging Strategies

The distributions of unmerged posting list lengths (t_i) and query frequencies (q_i) give useful insights into possible merging strategies. Figure 3(a) shows that the distribution of t_i for our documents is Zipfian, as one might expect based on published web studies. Figure 3(b) shows the distribution of q_i for the 300,000 queries from our query log. We found that the most common terms in the queries (high q_i) are also very common in the documents (high t_i). This makes sense, as people generally query on terms that they know about [4]. However, some terms (like 'following') are common in documents but rarely queried.

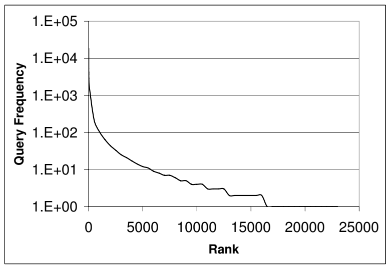
The ith term's contribution to the total workload cost is $q_i t_i$. Figure 3(c) shows the cumulative workload cost distribution, with the x axis representing the frequency order of

[1]Researchers have proposed heuristic techniques that do not require scanning entire posting lists. These techniques often do not perform well in practice [28] and can omit relevant documents. Thus these cannot be used in a regulatory environment, where it is imperative that all relevant documents be produced on demand.

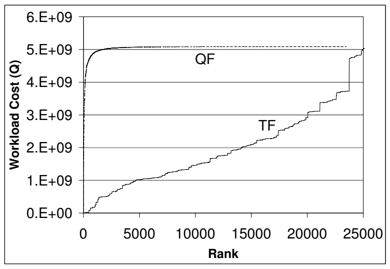
Document Statistics



(a) Distribution of Term Frequencies

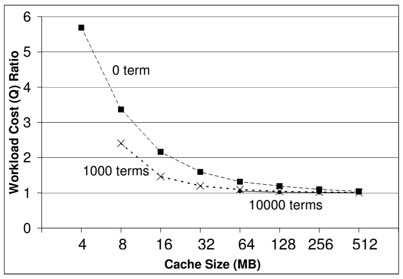


(b) Distribution of Query Frequencies

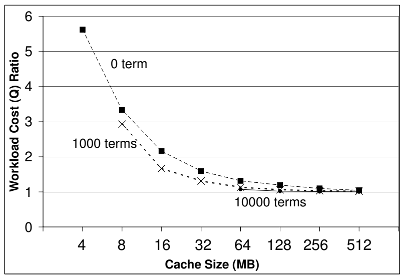


(c) Cumulative Workload Cost

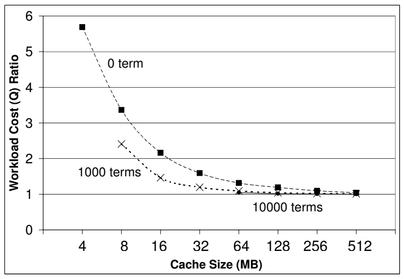
Ratios of Workload Cost ((d)-(g))



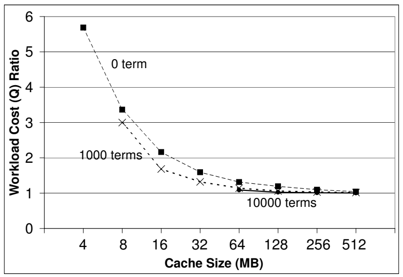
(d) Popular Query Terms Not Merged



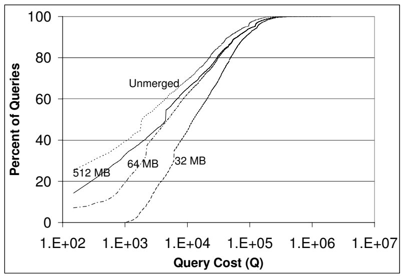
(e) Popular Document Terms Not Merged



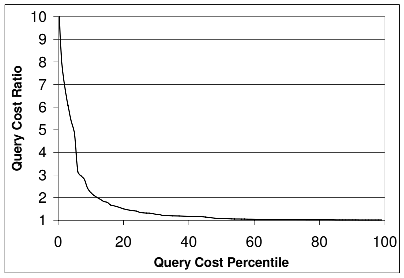
(f) Popular Query Terms Not Merged - With Learning



(g) Popular Document Terms Not Merged - With Learning



(h) Cumulative Query Cost Distribution



(i) Query Slowdown Distribution

Figure	Description
3(d),3(e)	These figures show the ratios of total workload costs (merged versus unmerged) as a function of cache size, when the terms with unmerged lists are those with high q_i and t_i respectively. The different curves correspond to different numbers of terms with unmerged posting lists (0, 1,000, or 10,000). The remaining posting lists are merged uniformly, with hashing used to assign terms to posting lists.
3(f),3(g)	These figures show the effectiveness of learning q_i and t_i statistics. To study the stability of these statistics, we computed the most popular terms for the first 10% of the documents crawled and the first 10% of the queries submitted, and used those statistics to make merging decisions for the entire index. Figures 3(f) and 3(g) show that the resulting workload query cost ratio is almost unchanged from that of Figures 3(d) and 3(e), respectively.
3(h),3(i)	These figures show the effect on individual queries of merging posting lists. Figure 3(h) shows the cumulative distribution of query costs for different cache sizes with uniform merging and with no merging. This figure shows that merging slows down the shortest queries the most (the x axis is log scale), while the long running queries are comparatively unaffected. This is also illustrated in Figure 3(i), which plots the query slowdown against the query cost, for a cache size of 512 MB. The figure shows that the longest-running half of the queries in the workload have no visible slowdown on average, and the next longest-running 30% of the queries are 25% slower on average.

Figure 3: Workload and Query Cost Under Different Merging Strategies.

the terms. The figure shows the contribution of the 25,000 most popular terms, out of more than 1,000,000 terms in the documents. The two curves in the figure show terms ranked by query frequency q_i (QF) and by term frequency t_i (TF). The key observation is that a very small fraction of the terms account for almost the entire workload cost. Thus one possible merging heuristic is to have separate posting lists for the frequent terms and merged posting lists for the remainder. The curve in Figure 3(c) corresponding to document popularity peaks slowly, compared to the query-popularity curve, due to terms that occur in many documents but few queries. However, the fraction of terms that contribute a lot to query cost is still small, compared to the total number of terms.

These merging heuristics require learning the frequencies t_i or q_i. Figures 3(f) and 3(g) show that the frequencies are quite stable over time and space. In an environment where the frequencies are less stable, the system can learn the frequencies online, and the merging strategy can be adapted accordingly. One possible approach is to divide time into epochs and maintain a separate index for the documents inserted in each epoch. The choice of posting lists to merge in any particular epoch can be determined by the statistics collected during the previous epoch. Queries must be answered by scanning the indexes of all epochs. By keeping the epoch interval large enough, one can keep the number of separate indexes reasonable. Furthermore, document database query interfaces typically support query conditions on document creation time (e.g., to retrieve only documents created in 2004). For such queries, one only needs to consider those indexes whose epochs overlap with the time interval specified in the query. The exact time range restrictions can be checked in a filtering phase after the keyword search, or can be integrated into the keyword search phase using jump indexes, as discussed in a later section.

3.4 Simulations

We conducted extensive simulations to evaluate the effects on query performance of the posting list merging heuristics proposed in the previous section. We varied the cache size and the number of popular terms (based on both t_i and q_i) that are not merged. The cache size determines the final number M of posting lists. Assuming a block size of 8 KB, $M = \frac{\text{Cache Size}}{\text{8KB}}$.

To estimate the effect of merging posting list on query throughput, we computed the ratio of the workload cost Q after merging the posting lists to the workload cost with no merging. The results are shown in Figures 3(d)-3(g), along with a detailed description. Figures 3(d) and 3(e) show the workload cost ratio as a function of cache size, when the terms with unmerged lists have high q_i and t_i respectively. The key observation from these figures is that even for modest cache sizes (128-256 MB), the workload cost with merging is almost as good as without merging. Figures 3(f) and 3(g) summarize the effectiveness of learning term statistics for the epoch-based scheme outlined in Section 3.3. In this experiment, we used terms in the first 10% of the documents and queries to make merging decisions for the entire index. The resulting workload cost ratio is almost unchanged from that of Figures 3(d) and 3(e). This suggests that the term statistics are likely to be relatively stable in practice, and can be learned for the purpose of merging posting lists.

Figures 3(h) and 3(i) show the effect on individual queries

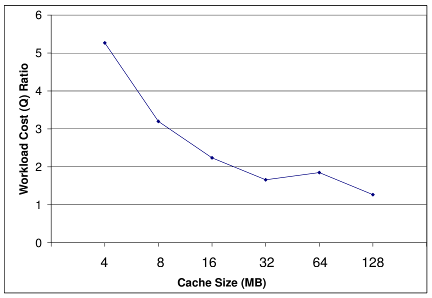


Figure 4: Actual Workload Run Time Ratios.

of merging posting lists. Overall, merging slows down the shortest running queries the most, while the longest-running queries are relatively unaffected. As shown in Figure 3(i), the longest-running half of the queries in the workload have no visible slowdown and the next longest-running 30% of the queries are 25% slower on average. From separate experiments, we know the actual running time for the average-length query is 500 msec on the platform described in the next section. The shortest-running 20% of the queries are slowed down by a factor of 4 on average, but this slowdown is unlikely to be visible to the user since the user-visible response time for such fast queries (far less than 500 msec) is likely to be dominated by factors such as network latency and human response time, rather than index scan time.

One important observation from all these figures is that if all the posting lists are uniformly merged (the curves corresponding to "0 term" in Figures 3(d) and 3(e)), the workload slowdown is close to that of the other schemes, particularly for larger cache sizes (128-512 MB). Due to the Zipfian distributions of q_i and t_i, very few terms have high t_i or q_i. As long as the posting lists of the popular terms do not get merged with each other, merging does not slow down query response significantly. If the number of posting lists is substantially larger than the number of popular terms (as with the larger cache sizes), these unlucky merges are unlikely to occur. In other words, uniform merging, being straightforward to implement, is likely to be the method of choice in practice.

3.5 Experimental Validation

The simulation results in the previous section showed that uniform merging of all posting lists offers very reasonable query performance. Further, it is straightforward to implement, as it does not require learning q_i or t_i statistics; nor does it require the creation or use of multiple indexes for different epochs. We implemented this scheme in IBM's Trevi [11] search engine to validate our simulation results. Our experiments used a dual-processor 3.2 GHz Intel Xeon Server with 10K RPM SCSI disks configured as RAID-0. We emulated the storage cache by caching in software and disabling file system caching (forcing all writes to disk).

Running all 300,000 queries on the server would have taken very long, so we instead used a 1% random sample from the query log. We used 32K separate posting lists (corresponding to a 128 MB cache size), which gives very reasonable query performance. Figure 4 shows the workload performance with uniform merging as a ratio to the

```
ZIGZAG(list1,list2)
 1: top1 ← list1.Start()
 2: top2 ← list2.Start()
    {Initialize the iterators to point to list heads}
 3: loop
 4:   if ((top1=list1.End()) OR (top2=list2.End())) then
 5:     return
 6:   end if
 7:   if ((*top1)<(*top2)) then
 8:     top1 ← list1.FindGeq(*top2)
        {Find an element greater than or equal to top2}
 9:     continue
10:   end if
11:   if ((*top2)<(*top1)) then
12:     top2 ← list2.FindGeq(*top1)
        {Find an element greater than equal to top1}
13:     continue
14:   end if
15:   if ((*top2)=(*top1)) then
16:     OUTPUT (top1)
        {Next element in join}
17:     top1 ← list1.FindGeq(*top1+1)
18:     top2 ← list2.FindGeq(*top2+1)
19:     continue
20:   end if
21: end loop
```

Figure 5: Zigzag Join.

case with no merging, for different cache sizes. The plot is quantitatively similar to that obtained from the simulation results (Figure 3(e), plot labeled "0 term"), except for a slight perturbation at a cache size of 64 MB, due to a change in system behavior as the number of open files passes 16,000.

4. MULTI-KEYWORD QUERIES

Our analysis so far has focused on queries in which any document containing a subset of the query terms is a match. Conjunctive queries in which *all* the query keywords must be present in a document are also very common in business environments. For example 'all emails from X to Y' is a conjunctive query on the two email addresses.

Conjunctive queries can be answered by taking the intersection of the keyword posting lists. This intersection can be computed without scanning the entire lists if an additional index structure, such as a B+ tree, is maintained on the individual lists. For example, one can exploit the fact that the posting lists are sorted on document ID and use the zigzag join [14] algorithm (Figure 5), together with an auxiliary index on posting lists, to skip over portions of lists that cannot be in the query result. In particular, the FindGeq() operation in steps 8 and 12 of Algorithm 5 can be done efficiently using such an auxiliary index.

One can create a B+ tree for an increasing sequence of document IDs without any node splits or merges, by building the tree from the bottom up, as shown in Figure 6(a) for the special case of a 2-3 tree. New elements are added at the leaf (posting list) level. When a leaf node fills up, a new leaf is created and an entry is added to the parent that points to the new leaf. If there is no room in the parent, the process is repeated until the root is reached. When the root fills up, a new level can be introduced, with a new root. These steps only require append and create operations on nodes and can be implemented in WORM storage.

Unfortunately, B+ trees are not secure, even when stored in WORM storage. For instance, Figure 6(b) shows that Mala can hide entry 31 by creating a separate subtree that does not contain 31, and adding an entry 25 at the root to lead to the new subtree. A subsequent lookup on 31 will be directed to Mala's subtree. FindGeq will also return incorrect results: the call FindGeq(28) will return 30 instead of 29. Mala's attack works because in a B+ tree, the path taken to look up entry 31 depends on entries that were added to the index after entry 31 was added. Other techniques like binary search can also be compromised by the adversary, by appending smaller numbers at the tail. For example, binary search on the leaves of the tree in Figure 6(b) would miss 31 because of the malicious entry 30 at the end.

An alternative strategy for supporting fast joins of posting lists is to build a GHT [29] for each posting list. For every entry in the smaller posting list, we consult the GHT to find matching entries in the longer posting list. However, GHTs only support exact-match lookups and have poor locality due to the use of hashing. A GHT-based join would be much slower than a zigzag join on sorted posting lists, especially for roughly equal sized lists.

4.1 Jump Indexes for Posting Lists

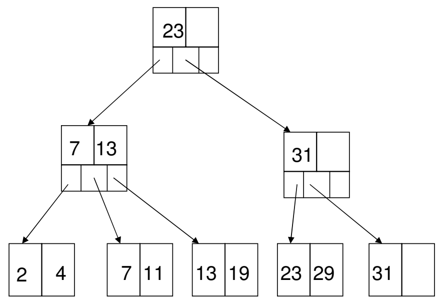
To address the issues outlined above, we designed a trustworthy index called a *jump index*. This index exploits the fact that the document IDs in posting lists are strictly monotonically increasing sequences. Jump indexes support FindGeq() in $O(\log(N))$ pointer follows in the worst case, where N is the largest number in the sequence. This bound is generally weaker than the $O(\log(n))$ bound for B+ tree lookups, where n is the number of entries in the tree. However, in this particular application, N is equal to the number of documents in the database, because document IDs are assigned through an increasing counter. Hence our bound is logarithmic in the number of documents.

The intuition behind jump indexes is that one can get to any number $k \leq N$ in $O(\log_2(N))$ steps, by taking jumps in powers of 2. Consider a sequence of numbers $0, \ldots, N-1$, where $N = 2^p$. Let $b_1 \cdots b_p$ be the binary representation of an integer $0 \leq k < N$. One can reach k in p steps by starting at the beginning of the sequence, then successively jumping forward $b_1 * 2^{p-1}$ places, then $b_2 * 2^{p-2}$ places, and so on until a $b_p * 2^0$ jump forward brings us to k.

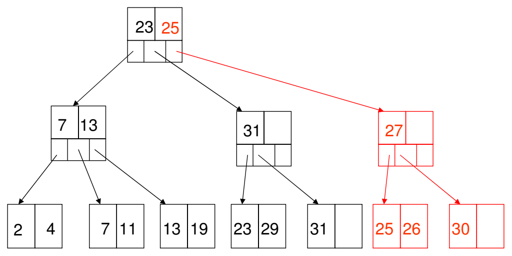
If we apply this approach to a posting list, the list will not contain every document ID, so we must store explicit jump pointers that tell us how far ahead to jump. The ith jump pointer stored with a list entry l points to the smallest list entry l' such that $l + 2^i \leq l' < l + 2^{i+1}$. Figure 7(a) shows an example jump index where the 0th pointer from 1 points to 2, as $1 + 2^0 \leq 2 < 1 + 2^1$, the 2nd pointer points to 5 since $1 + 2^2 \leq 5 < 1 + 2^3$, and so on.

More generally, let the posting list entries be $n_1, \cdots, n_N$. Any entry can be looked up by following jump pointers from the smallest number in the sequence. To look up an entry, say n, one needs to first find i_1 such that $n_1 + 2^{i_1} \leq n < n_1 + 2^{i_1+1}$ and follow the i_1st jump pointer from n_1 to a number, say n_{i_1}. From n_{i_1}, one needs to find i_2 such that $n_{i_1} + 2^{i_2} \leq n < n_{i_1} + 2^{i_2+1}$, and follow the i_2nd pointer to n_{i_2}, and so on until one reaches n. To look up 7 in Figure 7(a), one follows the 2nd pointer ($i_1 = 2$) from 1 to 5 and the 1st pointer ($i_2 = 1$) from 5 to 7.

The algorithms for Insert(k), Lookup(k) and FindGeq(k) are presented in Figure 4.2. FindGeq(k) simply makes a call to FindGeqRec(). The notation $ptr_s[i]$ refers to the ith jump pointer for index entry s. The pseudocode includes



(a) B+ Tree in WORM Storage



(b) B+ Tree Manipulated to Hide Entries

Figure 6: Untrustworthy B+ Trees. *The diagram on the left shows a B+ Tree in WORM storage. By adding entry 25 to the root and pointing it to a spurious subtree, one can effectively hide entry 31 from subsequent queries.*

assert checks, violations of which should trigger a report of attempted malicious activity.

4.2 Complexity

Proposition 1: Let $i_1, \ldots, i_j$ be the values of i selected in step 9 in successive iterations of the loop in Lookup(k). Then $i_1 > \cdots > i_j$.

Proof: Let the document IDs whose jump index nodes are visited by Lookup be $s_1, \ldots, s_j$, where s_1 is the smallest number in the posting list. From step 9, we have $s_1 + 2^{i_1} \leq k < s_1 + 2^{i_1+1}$ (a). Also, since the i_1st pointer points to s_2, and s_2 must have been inserted previously, we also have $s_1 + 2^{i_1} \leq s_2 < s_1 + 2^{i_1+1}$, i.e., $s_2 \geq s_1 + 2^{i_1}$ (b). Further, we have $s_2 + 2^{i_2} \leq k < s_2 + 2^{i_2+1}$ (c). From (a) and (c), we have $s_2 + 2^{i_2} < s_1 + 2^{i_1+1}$ (d). From (b) and (d) we have $s_1 + 2^{i_1} + 2^{i_2} < s_1 + 2^{i_1+1}$. Hence we have $i_1 > i_2$. By repeating the same argument, we get $i_1 > \cdots > i_k$. QED

From step 9 of Lookup, we have $i_1 \leq \lfloor \log_2(k) \rfloor + 1$. Thus it takes at most $\lfloor \log_2(k) \rfloor + 1$ jumps to find k. It can similarly be argued that Insert() and FindGeq() also require $O(\log_2(k))$ pointer follows. If there are N documents in the index, the complexity of the operation is, therefore $O(\log_2(N))$.

4.3 Trustworthy Jump Indexes

A straightforward approach to storing jump pointers in a WORM device is to maintain each node of the jump index in a separate disk block. Because of the monotonicity property of document IDs, the pointers are also set in increasing order—$ptr_s[i]$ is always set after $ptr_s[i']$ if $i' < i$. Hence the pointer assignment operation can also be implemented as an append operation. Under this approach, jump indexes and their associated posting lists have the following properties:

Proposition 2: Once an ID has been inserted into a jump index and the associated posting list, it can always be looked up successfully.

Proof Outline: The pointers set up during Insert (step 11) are written to WORM, so they and the entries themselves cannot be altered afterwards. The values of i ($i_1, \ldots, i_j$) selected by Insert() are identical to those selected by Lookup(). Hence an entry that has been inserted is always visible to Lookup(). QED

Proposition 3: Let v be an ID in the posting list. If $k \leq v$, then FindGeq(k) will never return a value greater than v.

Proof Outline: Suppose the path to v in the index is through jump pointers $j_1, \ldots j_{o_j}$ (i.e., Lookup(v) selects $j_1, \ldots j_{o_j}$ in successive iterations of the loop step 9). Also suppose FindGeq(k) returns l and $i_1, \ldots i_{o_i}$ is the path to l in the index, that is, $i_1, \ldots i_{o_i}$ are selected in step 4 (or step 15 if the checks in lines 5 or 9 fail) of successive calls to FindGeqRec(k).

We first show that $i_1 \leq j_1$. Consider the first call to FindGeqRec. In step 4, we choose $i \leq j_1$, as $k \leq v$. If the checks in step 5 and 9 succeed for that i, we have $i_1 = i$ and hence $i_1 \leq j_1$. If either of the checks fails, FindGeqRec selects an i in step 15. However, since v is in the index, $ptr_0[j_1]$ is not NULL. So we never go beyond j_1, hence $i_1 \leq j_1$. Now two cases arise: (i) $i_1 < j_1$. In this case, we have $l < v$ ($l < s_0 + 2^{i_1+1} \leq s_0 + 2_1^j \leq v$). (ii) $i_1 = j_1$. In this case, we can inductively apply the argument to the next call to FindGeqRec(). QED

Proposition 3 guarantees that no document ID can be hidden when joining two posting lists. Consider a document ID d present in both posting lists being joined. The zigzag join starts from the smallest numbers in the lists and makes successive calls to FindGeq(). Proposition 3 guarantees that no number greater than d can be returned in a FindGeq() call, before d is returned. In other words, d will eventually be returned in a FindGeq() call on both the lists and hence will always appear in the final result.

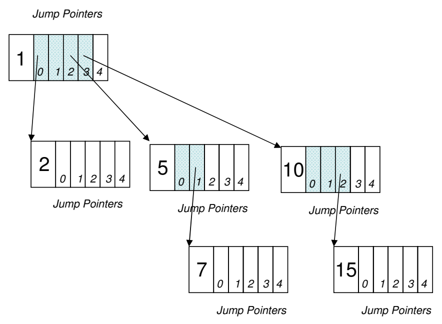
4.4 Block Jump Indexes

To reduce the overhead of storing the jump pointers and the depth of the index, which impacts performance, we can store p posting entries together in blocks of size L and associate pointers with blocks, rather than with every entry. We can also define jump pointers using powers of B rather than powers of two, where $p \geq B$. Specifically, we maintain $(B-1)\log_B(N)$ pointers with every block, with each pointer uniquely identified by a pair (i, j), where $0 \leq i < \log_B(N)$ and $1 \leq j < B$. The pointers are set up as follows:

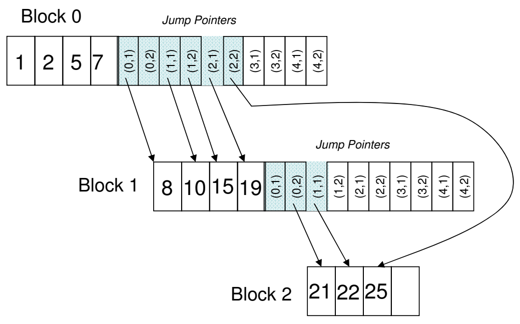
Let l_1 be the largest document ID stored in block b. The (i, j) pointer in b points to the block containing the smallest document ID s such that

$$l_1 + jB^i \leq s < l_1 + (j+1)B^i.$$

Figure 7(b) illustrates this for the case where $p = 4$ and $B = 3$. The figure shows only pointers for $0 \leq i \leq 4$. The



(a) Binary Jump Index.



(b) Block-structured Jump Index.

(a) Shaded pointers that are not filled are NULL. Only the first four pointers are shown. (b) For clarity the pointers are shown pointing to numbers, though pointers actually only store the address of the block. Only pointers with $i \leq 4$ are shown.

```
Insert(k)    —Insert ID k into the jump index

1: if jump index is empty then
2:    Create a new jump index with a node containing $k$
3:    return
4: end if
5: $s \leftarrow$ the smallest document ID in the jump index
6: assert $s < k$
   {The index entries must be monotonically increasing}
7: loop
8:    Find $i \geq 0$, such that $s + 2^i \leq k < s + 2^{i+1}$
9:    if ($ptr_s[i]$==NULL) then
10:      Create a new jump index node containing $k$
11:      $ptr_s[i] \leftarrow k$'s location    // Set the $i$th pointer of $s$
12:      return DONE
13:   else
14:      $s' \leftarrow$ the document ID at $ptr_s[i]$
         {Follow the pointer to a new $s$}
15:      assert $s' < k$
16:      $s \leftarrow s'$
17:      continue
18:   end if
19: end loop
```

```
Lookup(k)    —Find k in the jump index

1: $s \leftarrow$ the smallest document ID in the jump index
2: loop
3:    if $s > k$ then
4:       return NOT_FOUND
5:    end if
6:    if $s == k$ then
7:       return FOUND
8:    end if
9:    Find $i \geq 0$ such that $s + 2^i \leq k < s + 2^{i+1}$
10:   if $ptr_s[i]$==NULL then
11:      return NOT_FOUND
12:   else
13:      $s' \leftarrow$ the document ID at $ptr_s[i]$
         {Follow the pointer to a new $s$}
14:      assert $s + 2^i \leq s' < s + 2^{i+1}$
15:      $s \leftarrow s'$
16:      continue
17:   end if
18: end loop
```

```
Lookup_block(k)    —Find ID k in the jump index

1: $b \leftarrow$ the first block of the index
2: loop
3:    $n_b \leftarrow$ the largest ID in block $b$
4:    if ($k \leq n_b$) then
5:       Search for $k$ in $b$, and return FOUND or NOT_FOUND
6:    end if
7:    Find $(i, j)$ such that $0 \leq i < log_B(N)$, $1 \leq j < B$, and
      $n_b + j * B^i \leq k < n_b + (j+1) * B^i$
8:    if $ptr_b(i, j) \neq$ NULL then
9:       $b \leftarrow ptr_b(i, j)$
         {$ptr_b(i, j)$ is the $(i, j)$th pointer in block $b$}
10:      continue
11:   else
12:      return NOT_FOUND
13:   end if
14: end loop
```

```
FindGeqRec(k,s)    —Function implementing FindGeq(k)

1: if ($s \geq k$) then
2:    return $s$
3: end if
4: Find $i \geq 0$ such that $s + 2^i \leq k < s + 2^{i+1}$
5: if ($ptr_s[i] \neq$ NULL) then
6:    $t \leftarrow$ document ID at $ptr_s[i]$
7:    assert $s + 2^i \leq t < s + 2^{i+1}$
8:    $res \leftarrow$ FindGeqRec($k$, $t$)
      {Recursively call FindGeqRec() by following the ptr}
9:    if ($res \neq$ NOT_FOUND) then
10:      assert $s + 2^i \leq res < s + 2^{i+1}$
11:      return $res$
12:   end if
13: end if
   {No number $\geq k$ could be found by following $i$th ptr. Now
   we return the first non-null ptr}
14: $i \leftarrow i + 1$
15: while ($i < \log_2(N)$) do
16:   if ($ptr_s[i] \neq$ NULL) then
17:      $t \leftarrow$ document ID at $ptr_s[i]$
18:      assert $s + 2^i \leq t < s + 2^{i+1}$
19:      return $t$
20:   end if
21:   $i \leftarrow i + 1$
22: end while
23: return NOT_FOUND
```

```
FindGeq(k)    —Find number ≥ k

1: return FindGeqRec($k$, the smallest number in the sequence)
```

```
Insert_block(k)    — Insert document id k in the jump index

1: last_block $\leftarrow$ last block in the index
2: If last block is full (has $p$ entries), allocate a new block and
   set last_block to new block.
3: Append ($k$) to the last_block.
4: $b \leftarrow$ first_block
5: loop
6:    if (b==last_block) then
7:       return DONE
8:    end if
9:    $n_b \leftarrow$ the largest ID in block $b$
10:   assert $n_b < k$
11:   Find $(i, j)$ such that $0 \leq i < log_B(N)$, $1 \leq j < B$, and
      $n_b + j * B^i \leq k < n_b + (j+1) * B^i$
12:   if $ptr_b(i, j) \neq$ NULL then
13:      $b \leftarrow ptr_b(i, j)$
         {$ptr_b(i, j)$ is the $(i, j)$th pointer in block $b$}
14:      continue
15:   else
16:      $ptr_b(i, j) \leftarrow$ last_block
17:      return
18:   end if
19: end loop
```

Figure 7: Jump Index Essentials.

largest number in block 0 is 7. The $(0, 1)$ pointer points to block 1 because the latter contains 8 and $7 + 1 * 3^0 \leq 8 < 7+1*3^1$. 8 is the smallest number satisfying that constraint. Similarly, the $(2, 2)$ pointer of block 0 points to block 2, because block 2 contains 25 and $7 + 2 * 3^2 \leq 25 < 7 + 2 * 3^3$, and so on.

The algorithms for Insert_block() and Lookup_block() are presented in Figure 7. As when $B = 2$, the pointer set operation in step 15 of Insert_block() can be implemented by an append operation. As with $B = 2$, one can show that if the lookup proceeds by following pointers $i_1, \ldots, i_k$, then $i_1 < \cdots < i_k$. This gives a bound of $\log_B(N)$ jumps for Lookup().

4.5 Evaluation

The key jump index parameters are L (jump index block size), p (number of posting list elements in a block), and B (branching factor). At the end of each posting list block, we leave space to store jump pointers. Assuming 4-byte pointers and 8-byte posting elements, we have

$$8 * p + 4 * (B - 1) \log_B(N) \leq L.$$

The number of pointers per index block ($(B-1)\log_B(N)$) depends on N, the largest document ID expected to be indexed. For our evaluation, we set $N = 2^{32}$, roughly 4 billion, which should be adequate for typical business usage. To store more than N documents, we can chain additional blocks of jump pointers off the end of the posting list block. Figure 8(a) shows the space overhead of a jump index, computed as the ratio of the space allocated for pointers to the space occupied by actual posting elements. For $B = 32$ and $L = 8\ KB$, a jump adds 11% space overhead.

We carried out extensive simulations to evaluate the performance of jump indexes for different values of block size L and branching factor B. Due to space constraints, we present and analyze the results for $L = 8$ KB, and briefly discuss how the results differ for other values of L.

Our first set of experiments evaluates the index update performance as a function of the cache size. When a new document is added, apart from appending the document ID to the tail block of the posting lists, the jump index pointers must also be followed and set in some of the intermediate posting list blocks. This increases the I/Os per document unless we increase the cache size. To reduce the I/O for following jump index pointers, our index code tracks in its own memory (not the storage server memory) the largest document ID and the last pointer for all the blocks on the path from root to the tail block, for every posting list. The pointer to be followed while appending a new document ID is determined from these largest document ID values (step 11 in Insert_Block), without fetching the corresponding block from disk. With this optimization, a block fetch is required only when setting a new pointer, which is infrequent compared to following a pointer. The memory requirement for storing the largest document ID and pointers for all the posting lists is $8k \log(N)$ bytes, where k is the number of posting lists and N is the number of documents. For example, the memory requirement for k = 32,768 and $N = 2^{32}$ is 8 MB, which is quite reasonable for the indexing code.

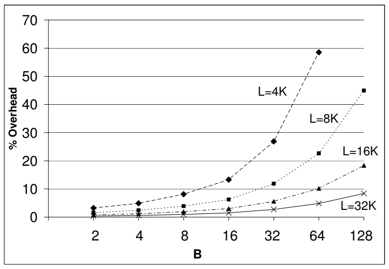
To evaluate update performance, we incrementally inserted our 1 million documents into an initially empty index. We used a cache simulator to measure the I/Os incurred while updating the index. As in Section 3.5, we uniformly merged the posting lists into 32,768 lists. Figure 8(b) plots the I/Os incurred per document inserted as a function of cache size, for $B \in \{2, 32, 64\}$. A higher value of B requires more jump index pointers to be set and hence requires more I/O per document. The effect is particularly prominent for smaller cache sizes like 128 MB, which is barely enough to hold the posting list tail blocks. However the curves level off eventually as the cache size increases to hold all the pointer blocks (depth of the tree) in memory. For example at 288 MB, the curves almost converge at 1.1 I/Os per document. This is close to the 1 I/O per document required to just append the document IDs to the posting lists. Increasing the block size L beyond 8 Kbytes (not shown in the figure) reduces the I/Os per document, by reducing the storage overhead for jump pointers.

Our next set of simulations evaluates jump index query performance. We use the same workload of 300,000 queries as before, treating each query as a conjunctive query. Multi-keyword queries are answered with zigzag joins of the posting lists, starting with the shortest two lists. We plotted query speedup, defined as the ratio of the number of blocks read when no jump index is kept (using a sequential scan-merge join) to the number of blocks read in a zigzag join using the jump index. Figure 8(c) shows the speedup as a function of the number of query keywords, for different values of B. The figure shows that the benefit of jump indexes increases with the number of keywords in the query, rising smoothly from almost no benefit with 3-keyword queries to a factor of 3 speedup for 7-keyword queries. Two-keyword queries are approximately 10% slower when a jump index is used, which can be explained as follows.

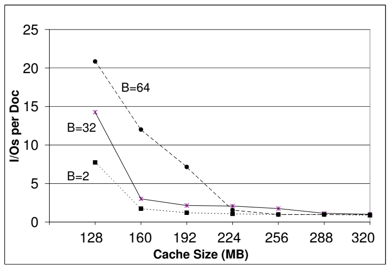
Consider two sorted lists of length l_1 and l_2 containing numbers uniformly distributed in $[1, N]$ (that is, each number is picked with probability $\frac{l_i}{N}$). If l_1 is roughly the same size as l_2, then a zigzag join on the two lists is $O(l_1 + l_2)$, even with an index. This is because any two consecutive elements of one list are likely to bracket an element from the other list, so that a zigzag join visits every element of each list, approximating a scan-merge join. However, if $l_1 \ll l_2$, then a zigzag join is $O(l_1 \log(l_2))$, approximating a scan of the smaller list with index lookups on the longer list.

Because we uniformly merge posting lists, most of the resulting lists are of roughly equal size. A 2-keyword query on these lists approximates a scan. The additional overhead of jump pointers makes a sequential scan using the jump index slower than a scan over posting lists without a jump index. For queries with more than two keywords, the partial result obtained by intersecting the first two lists is zigzag joined with the next shortest list, and so on. On each iteration, the partial result becomes smaller and the next list to be joined becomes longer. Thus subsequent zigzag joins move toward a $O(l_1 \log(l_2))$ computation time. This is why the benefit of a jump index increases with the number of conjunctive terms in the query.

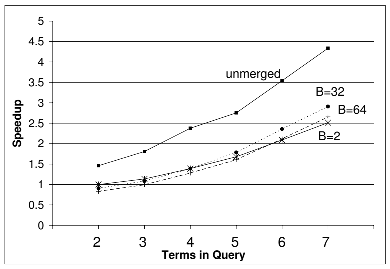
A second observation from Figure 8(c) is that queries with few keywords perform slightly better with small B, while many-keyword queries benefit slightly from larger B. This is because the scan-merge-like zigzag join on smaller queries runs fastest on the index with lowest space overhead (smaller B). In contrast, the zigzag joins of large queries benefit from the decreased depth of the jump index (higher B). In our experiments with $B = 2^k$, $1 \leq k \leq 6$ (for clarity, not all are shown in the figures), $B = 32$ provides the best tradeoff,



(a) Space Overhead of Jump Index



(b) I/Os Per Document Inserted with Jump Index



(c) Query Performance for Jump Index

Figure 8: Jump Index Performance.

performing best for queries with more than five keywords and within 5% of the best for smaller queries.

As a reference, Figure 8(c) also shows the ideal speedup achievable with no merging of posting lists and with a separate B+ tree (block size 8 KB) for each posting list. This ideal case benefits from unmerged posting lists to answer queries, and from the larger fanout of B+ trees compared to jump indexes, for a fixed block size. A jump index stores $(B-1)\log_B(N)$ pointers in each block, while B+ trees only use B pointers. Nonetheless, Figure 8(c) shows that the query performance of jump indexes is within a factor of 1.4 of the theoretical maximum—while simultaneously supporting immediate index updates on document insertions and providing trustworthiness guarantees that are unachievable with the unmerged, B+ tree-based approach.

Finally, jump indexes slow down disjunctive query workloads (Section 3.1) by the same factor as the space overhead of the jump index. For example, the slowdown is 1.5% and 11% for $B = 2$ and $B = 32$, respectively, for 8 KB blocks.

The choice of whether to include a jump index depends on the expected query workload. If most queries are disjunctive or involve only two or three keywords, one should use merged posting lists with no jump index. If most queries conjoin many keywords, it is best to use merged posting lists and a jump index with $B = 32$. One can use the epoch scheme outlined in Section 3.3 to learn the query pattern in one epoch and use it to decide whether to include a jump index for the next epoch.

5. RANKING ATTACKS

The effectiveness of a search query depends on the ranking of the documents it returns. Mala can try to hide a document D by adding spurious documents to the posting lists of all D's keywords or by directly altering the statistics maintained for ranking D, so that D will be ranked low when Bob issues his query. In an investigation of potential illegal activity, Bob is likely to examine all returned documents, so Mala's attack will make Bob's job harder but will not prevent him from uncovering evidence of misdeeds. Further, Bob is extremely likely to notice her cover-up attempt, rousing his suspicions still further, as explained below.

If Mala stuffs the posting lists with IDs of documents that do not exist or do not contain the query keywords, the search engine can detect this and alert Bob to malicious activity. Bob will then carefully examine the felonious cover-up attempt and unearth D. If Mala stuffs the posting lists with documents containing the same keywords as D, it will be very hard for her to create a large number of meaningful distinct documents that contain D's keywords, without raising suspicion. For example, if Bob queries a corporate email archive for [Stewart Waksal ImClone], Mala cannot invent many believable emails that discuss Stewart, Waksal, and ImClone. Though the search engine may be fooled by them, Bob's suspicions are likely to be aroused when he examines the top-ranked email.

In most investigative situations, Bob will also be able to supply a target time range for illegal activity (Nov.-Dec. 2001 in this case). To support such queries, we need a trustworthy index on document commit times—Mala must not be able to retroactively insert email supposedly committed during an earlier period, or eliminate any entry from the result of a query on commit time. A jump index on commit times can provide these guarantees and foil Mala's cover-up attempt.

6. CONCLUSION & FUTURE WORK

We have presented a threat model for trustworthy record-keeping for legislative compliance, and identified key requirements for trustworthy indexes for this environment. One such requirement is the need for immediate indexing of newly inserted documents, which renders inapplicable the traditional approach of logging new posting list entries and periodically rebuilding the posting lists from scratch. To meet these requirements, we proposed a scheme based on judicious merging of posting lists and the optional use of *jump indexes* for faster conjunctive query processing.

Through extensive simulations and experiments with an IBM intranet search engine and a workload of IBM intranet queries and documents, we demonstrated that merged posting lists and jump indexes offer excellent performance. Compared to a baseline approach that uses a multi-GB storage server cache for posting lists, does not merge posting lists, and keeps a separate B+ tree for each posting list to speed up conjunctive queries, our scheme is 20 times faster for document insertions, while using only a modest 128 MB storage server cache (256 MB with jump indexes)—and our scheme offers trustworthiness guarantees. For a disjunctive keyword query workload, merged posting lists without jump indexes are only 14% slower than the baseline approach, and merged posting lists with jump indexes (32-way branching) are only 26% slower than the baseline approach (due to the

11% space overhead of including a jump index), while still offering trustworthiness guarantees. On a conjunctive query workload, merged posting lists with jump indexes are 47% faster than merged posting lists without jump indexes, and are only 30% slower than the baseline approach. We conclude that the query overhead of our scheme is a very reasonable tradeoff for its impermeability to insider attempts to hide evidence of misdeeds by circumventing records retention policies.

Jump indexes and merged posting lists share the desirable property that attempted malicious activity is easy to detect, in the form of a violation of a monotonicity property. One topic for future work is an elegant course of action once malicious attempts have been detected (malicious index entries and documents cannot simply be removed, as they reside on WORM). Another interesting topic is the use of merged posting lists for online update in non-compliance scenarios.

7. REFERENCES

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-tree. *VLDB Journal*, 5:264–275, 1996.

[2] K. Blibech and A. Gabillon. Chronos: an authenticated dictionary based on skip lists for timestamping systems. In *Workshop on Secure Web Services*, 2005.

[3] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB*, 1994.

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

[5] P. Crescenzi and V. Kann. *A compendium of NP optimization problems.* Available at http://www.nada.kth.se/.

[6] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR*, 1990.

[7] M. C. Easton. Key-Sequence Data Sets on Indelible Storage. *IBM J. Research & Development*, May 1986.

[8] EMC Corp. EMC Centera Content Addressed Storage System, 2003. www.emc.com/products/systems/centera_ce.jsp.

[9] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, vol. 17, pp. 49-74, 1985.

[10] C. Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *VLDB*, 1992.

[11] M. F. Fontoura, A. Neumann, S. Rajagopalan, E. Shekita, and J. Zien. High performance index build algorithms for intranet search engines. In *VLDB*, 2004.

[12] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, 2003.

[13] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX II*, 2001.

[14] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2000.

[15] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.

[16] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. for Info. Sci. & Tech.*, 54:8, Jun. 2003.

[17] L. Huang, W. Hsu, and F. Zheng. CIS: Content Immutable Storage for Trustworthy Record Keeping. In *NASA MSST*, 2006.

[18] IBM Corp. IBM TotalStorage DR550, 2004. Available at http://www-1.ibm.com/servers/storage/disk/dr.

[19] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *CEAS*, 2004.

[20] T. Krijnen and L. G. L. T. Meertens. Making B-Trees Work for B.IW 219/83. The Mathematical Centre, Amsterdam, 1983.

[21] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Conf. on Australasian Computer Science*, 2004.

[22] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In *FAST*, 2002.

[23] Network Appliance, Inc. SnapLock^{TM} Compliance and SnapLock Enterprise Software, 2003. Available at http://www.netapp.com/products/filer/snaplock.html.

[24] P. Rathmann. Dynamic Data Structures on Optical Disks. In *ICDE*, 1984.

[25] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau. Okapi at TREC. *TREC*, 1992.

[26] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[27] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *VLDB*, 1994.

[28] I. H. Witten, A. Moffatt, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 1999.

[29] Q. Zhu and W. Hsu. Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records. In *ACM SIGMOD Conference*, June 2005.

[30] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Cambridge, 1949.