Deleting Index Entries from Compliance Storage

Soumyadeb Mitra and Marianne Winslett
Department of Computer Science
University of Illinois at Urbana-Champaign

{mitra1,winslett}@cs.uiuc.edu

Nikita Borisov
Department of Electrical and Computer
Engineering
University of Illinois at Urbana-Champaign

nikita@uiuc.edu

ABSTRACT

In response to regulatory focus on secure retention of electronic records, businesses are using magnetic disks configured as write-once read-many (WORM) compliance storage devices to store business documents such as electronic mail for their mandated retention periods. A document committed to a compliance storage device cannot be altered or deleted even by a superuser until its retention period is over, and hence is secure from attacks originating from company insiders. Secure retention, however, is only a part of a document's lifecycle: it is often crucial to properly delete documents once their retention period ends. It is relatively simple to delete a document, but much harder to remove its index entries from WORM. Yet if these entries are not obliterated, the contents of the deleted document can often be reconstructed.

In this paper, we formally define *secure deletion* of document entries from an inverted index on compliance storage. We show that previously proposed deletion schemes for compliance storage index entries do not meet the objectives of secure deletion. On the other hand, the naive approach to secure deletion results in very poor query performance. To provide secure deletion of index entries without compromising lookup efficiency, we propose a novel indexing technique that employs noise terms, merged posting lists, and deletion epochs. Experiments with real-life data show that lookups in our scheme are 5 times faster than the naive approach.

1. INTRODUCTION

Documents such as electronic mail, financial statements and meeting memos are valuable assets. Ensuring that these records are readily accessible, accurate, credible, and irrefutable is particularly important given recent legal and regulatory trends. The US alone has over 10,000 regulations that mandate how records should be managed [9]. Many of those focus on ensuring that records are trustworthy during their mandated multi-year retention periods (e.g., SEC Rule 17a 4 [8], HIPAA, and the Sarbanes-Oxley Act [1]).

This has led to a rush to introduce write-once read-many (WORM) compliance storage devices (e.g., [2, 4, 7]) for proper data retention. Until the end of its retention period, a file committed to the WORM device is read-only and cannot be deleted or altered even by a superuser. A WORM device hence secures critical documents from certain threats originating from company insiders or hackers with administrative privileges.

Secure document retention, however, is only a part of the requirement. The ability to *properly* dispose of electronic records after a point in time is often as important, if not more, as the act of maintaining them. Once a record passes its mandatory retention period, the company is no longer required to store it. At that point, the record can be a liability—e.g., it can be subpoenaed in future lawsuits or regulatory enquiries. Corporate policies hence often mandate deletion of expired records. This requirement raises a key question—how do we delete records that have been committed to WORM media?

Fortunately, modern WORM devices are built atop conventional re-writable magnetic disks, with write-once semantics enforced through the firmware/software running inside the device. These devices support *term immutability*—the files on these devices are write-once for a specific period of time. Every file committed to the device has an expiry date (which can be infinity), either assigned explicitly by the committing application or as a system default. Expiry dates can be extended, but not moved forward in time. Once a file expires, the device lets the file be deleted, so that external cleanup applications can erase it. Every record is stored in a separate file, so that it can be deleted or have its expiry date extended independently of all other records.

The actual document itself is not the only source of information about its content. To quickly find records on a specific topic in response to a subpoena, records must be indexed. For example, an inverted index is typically built over semi-structured records like email and memos to support keyword search. These indexes must also be kept on WORM to secure them from tampering [11, 5].

The set of words in a record can be reconstructed from an inverted index. To completely dispose of a record, its index entries must also be erased. Unfortunately, it is hard to support deletion of individual index entries using the file-level expiration-based deletion provided by the WORM devices. The files constituting an index over a group of records should be physically erased only after all the records in that group have expired. This creates a time window between when a record expires and when its index entries can be

deleted. While this time window can be reduced by creating a separate index for each group of documents expiring at approximately the same time, such a scheme leads to very poor index lookup performance due to the many separate indexes that must be scanned during querying. At the other extreme, a single index over all the records will have good query performance but will require us to retain the index entries of all the documents in perpetuity.

In this paper we address the problem of securely deleting entries from an inverted index on compliance storage. We formally define *secure deletion* through an indistinguishability game. We show that previously proposed compliance indexing techniques either do not meet the requirements of secure deletion or have very poor query performance. To address this, we propose a novel technique for deleting index entries that relies on adding random noise terms to the index. We analyze the spectrum of performance and security tradeoffs provided by our scheme. Experimental results on real-world data show that our deletion scheme is 5 times faster than the naive approach, with less than a factor of 2 increase in the index size. This space overhead is reasonable considering the fact that an inverted index usually occupies only 10-15% of the space of the underlying document corpus.

This paper is organized as follows. In Section 2 we discuss background work, specifically the data model, the storage model and the threat model. Section 3 discusses related work. In Section 4, we introduce our strongly secure deletion scheme and compare it against the trivial deletion scheme. We present experimental results in Section 5 and conclude in Section 6.

2. BACKGROUND

2.1 Data Model

We use the term *record* to refer to business documents with a fixed retention period, such as financial notes, email, memos, reports, and instant messages. Ideally, a record should be deleted immediately after it expires. However, deletions require multiple overwriting passes to ensure that the original data is nonrecoverable; this I/O can interfere with the regular I/O activities of the WORM device. In practice, it is usually acceptable to have a slight delay, such as a week, between when a record expires and when it is erased. The length of this *disposition interval* is dictated by company policy. Deletion utilities are run at disposition intervals as a batch operation, during non-peak load. The set of records that are deleted in the same batch is called a *disposition group*.

2.2 Storage Model

In this paper, we consider a WORM device with a file system interface [7], though our techniques are equally applicable to object-based devices [2]. The interface allows users to create new files and to append to existing files. Appends are required for indexing and can be efficiently supported since the underlying media are magnetic [5]. The append feature should be restricted to the specific storage volume holding the index, to prevent appends to committed files that contain ordinary records.

For efficient read access, the WORM devices store files contiguously on disk when possible and periodically defragment the file system to collate non-contiguous file fragments created through multiple append operations. Other than

assuming the existence of these internal optimizations, we treat the WORM device as a black box that offers a file system interface.

2.3 Inverted Indexes

Querying & Indexing. The standard query interface for semi-structured and unstructured business records supports keyword queries, where a user types an arbitrary set of terms and obtains a list of the documents containing some or all of the terms. Queries can be further constrained by a document creation time interval, as in the following:

> *Find all documents containing "Martha" and "Ralph" that were created between 06/2002 and 08/2002.*
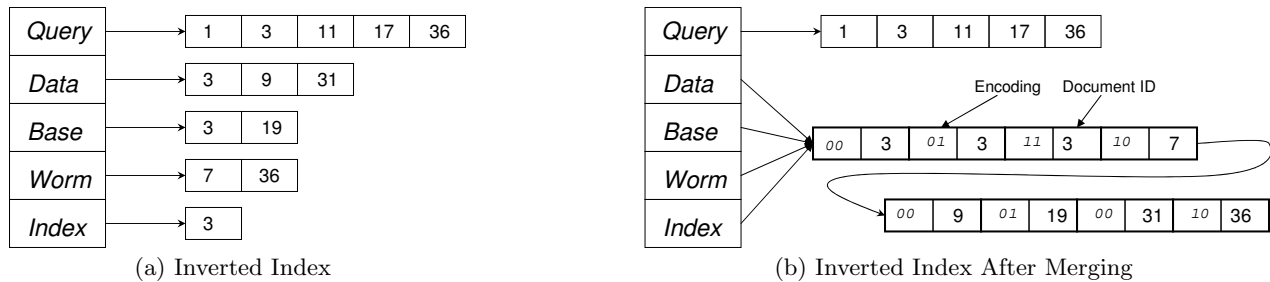
We call this temporal constraint the *time interval* of the query; if none is specified, then all undeleted records must be considered.

The standard implementation of keyword queries uses an inverted index [10]. As shown in Figure 1(a), an inverted index consists of a dictionary of terms and a *posting list* of the identifiers (IDs) of the documents that contain that term. In addition to an ID, each posting list element gives the number of occurrences of that term in the document (its *term frequency*), which is not shown in the figure. The document IDs are usually assigned in order of document arrival, through an increasing counter. Researchers have shown that the individual posting elements can be compressed to about 2 bytes through appropriate encoding schemes [10].

Queries are answered by scanning the posting lists of the terms in the query, thereby obtaining a list of documents having some/all of the keywords. The resulting documents are ranked based on the number of occurrences of the keywords and their relative importance [10]. The I/O cost of scanning the posting lists is the major component of the total query runtime cost. To support time-interval queries efficiently, an auxiliary index on creation times [5] (not shown in the figure) is also maintained on each posting list. The time index can be used to seek to the appropriate position in each posting list, instead of scanning the list from the start.

Compliant Inverted Indexes. Because the volume of compliance data is so large and no one wants to wait hours or days for a query answer, index lookup is the only practical record search method. But this means that an adversary can make a record inaccessible by omitting its entries from the index, or altering the index to point to a different version of the record on WORM. Hence, a record can be *logically deleted* or *logically modified* by suitably altering the index structure. To prevent such tampering, the index must be kept on WORM media [5, 11].

An inverted index can be stored on WORM by keeping each posting list in an append-only WORM file. 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 on average. To address this problem, researchers have proposed merging the posting lists together (into a maximum of as many lists as the number of cache blocks in the

(a) Inverted Index

Query → 1 | 3 | 11 | 17 | 36
Data → 3 | 9 | 31
Base → 3 | 19
Worm → 7 | 36
Index → 3

(b) Inverted Index After Merging

Query → 1 | 3 | 11 | 17 | 36

Encoding   Document ID

Data, Base, Worm, Index → 00 | 3 | 01 | 3 | 11 | 3 | 10 | 7

→ 00 | 9 | 01 | 19 | 00 | 31 | 10 | 36

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.

storage server [5]). Each merged list contains the union of the document IDs from the individual posting lists that are merged together. A keyword encoding is also stored in each posting element, to identify which keyword in the merged set appears in that document. An example of the resulting index structure is shown in Figure 1(b).

Merging increases the length of the posting lists and hence slows down queries. Uniform merging, in which keywords are assigned randomly to posting lists, is the simplest merging scheme to implement and has reasonably good performance, compared to more sophisticated schemes that require prior knowledge of query or keyword probabilities. Experimental results on two different real-world data sets of documents and queries showed that that uniform merging of 100–200 words into each posting list slows down query performance by less than 10% [5] and can support online insertions into the posting lists for typical real-world document arrival rates. We extrapolated these results for use with our own Enron email data set, and merged its 900K words into $\frac{900K}{200} \approx 4096$ posting lists.

2.4 Threat Model for Compliance Records

The three kinds of agents in our system are Alice, an honest company employee; Bob, a regulatory authority who is honest but inquisitive; and Mala, a company insider adversary who may have superuser powers. Alice creates a record, and the application she uses to create it automatically commits it to the storage device. We assume that the commit is trustworthy, that is, Alice properly creates the record and it is committed to the compliance storage device. At some point in the future, Bob queries the record committed by Alice, e.g., as part of an regulatory investigation or litigation.

The record can be attacked by Mala at any point after it is committed. For example, Mala might be the CEO who retroactively wants to hide an illegal email conversation she had with her broker about whether to sell stock in her company. As explained earlier, Mala cannot alter the email itself, because it is on WORM storage. The compliance storage server has built-in protection against reloading the record from a tampered-with backup copy. Mala can physically destroy the storage server, but such an attack will draw attention and regulatory ire will follow any apparent attempt to destroy data. To be successful, Mala's attack must remain undetected. Certain disk replacement attacks are still possible with today's WORM devices, but the next generation of products is moving toward derailing those attacks, so we will not consider them here.

Bob needs to ensure that his query is executed properly and that the records have not been tampered with inside the query engine software. To ascertain this, Bob should obtain direct read access to the WORM device and execute the query using his own trusted search engine. As Bob is law-abiding but inquisitive, after getting read access to the device, he may try to reconstruct or extract information about expired records. Even if the records have been properly erased from the WORM device, he can try to extract information about the documents from the inverted index.

An inverted index contains sufficient information to *reconstruct* the set of indexed terms in a document. For example, from the index in Figure 1(a) Bob can learn that document 3 contains the words Query, Data, Base and Index. From the set of words in a document, an adversary can often infer its meaning (e.g., "fire Harriet tomorrow"). Thus it is critical to clean up the index when documents are deleted. If the deletion scheme prevents Bob from obtaining the set of words in a deleted document, we call it *content secure.* However, content secure deletion is not always sufficient, as illustrated in the examples below.

- *Non-occurrence.* Bob may want to prove the absence of a document with a certain set of keywords in a certain period of time. For example, in a litigation, he might be interested in proving that no email was sent to a particular customer between such and such dates, contrary to the claim of the company.

- *Frequency.* Bob may want to learn about changes in the document frequencies of certain words, i.e., in the number of documents containing that keyword. For example, a sudden increase in the number of occurrences of the name of a dangerous chemical in deleted documents may indicate that the company suspected or knew that its products were contaminated with that chemical, contrary to their claims.

- *Probabilistic occurrence.* Bob may be interested in probabilistic claims. As we show later in the paper in Section 3, one previously proposed deletion scheme [11] allows Bob to learn a set of possible reconstructions of a document. For example, Bob can determine that the document had one of the words "carcinogen," "puppy," or "stewart@msl.com," with each equally probable. In other words, the chance of the document containing "carcinogen" is 33%, which is much higher than its chances of occurring in a randomly selected document. This in itself can be considered as circumstantial evidence in a litigatory environment.

- *Contextual occurrence.* Bob can have additional context information that can help him to reconstruct the document. In the above example, Bob may know that "stewart@msl.com" was not present in the document because Ms. Stewart had already left the company when the email was sent.

- *Dictionary.* In a merged inverted index, the keyword to posting list mapping is stored in a separate dictionary data structure. Bob can learn the set of all keywords used in documents from this dictionary, though he cannot tell during which disposition intervals those words were used. If desired, the rare words can be hidden by including only common words in the shared dictionary, and assigning the rare words to posting lists via a hash function. Thus we do not consider this threat in the rest of the paper.

A compliance index deletion routine should be secure against these and other attacks: Bob should not get *any* information about the deleted documents from the index. This can be formally defined through the following indistinguishability game. Bob chooses a sequence of document set pairs (D_0^1, D_1^1), (D_0^2, D_1^2), ..., (D_0^n, D_1^n) and passes them to Alice. The document sets D_0^1 and D_1^1 have expiry time t_1, D_0^2 and D_1^2 have expiry time t_2, and so on up to t_n. Without loss of generality, assume that $t_1 \leq t_2 \leq \ldots \leq t_n$

Alice then privately chooses a random n-bit string b (where the probability of a bit being 0 or 1 is 0.5 and is independent of the other bits). Hidden from Bob, she then stores and indexes the document sets $D_{b[0]}^1, D_{b[1]}^2, \ldots, D_{b[n-1]}^n$ on an empty WORM storage device S. That is, for each (D_0^i, D_1^i) pair, Alice randomly stores and indexes one document set from the pair. Alice then chooses an integer i such that $1 \leq i \leq n$. She keeps S hidden from Bob until time t_i, invoking the appropriate index deletion routines as document sets $D_{b[0]}^1$ through $D_{b[i-1]}^i$ expire. Alice then gives the storage server S to Bob.

To win the game, Bob must guess any document set Alice stored and indexed on S after they have been deleted. Specifically, Bob wins the game if he can successfully determine the value of any one of $D_{b[k-1]}^k$ $(1 \leq k \leq i)$ for which $D_0^k \neq D_1^k$. We say that an index deletion routine is *strongly secure* if a computationally bound adversary Bob cannot perform any better than random for all choices of the document sets [3]. In other words, his probability of correctly guessing whether S contained document set D_0^k or D_1^k for $1 \leq k \leq i$, $D_0^k \neq D_1^k$ is 50%. We say that an index deletion routine is *weakly secure* if the above holds under the constraint that Bob is only allowed to select document set pairs (D_0^i, D_1^i) where D_0^i and D_1^i $(1 \leq i \leq n)$ have the same number of documents.

The intuition behind this indistinguishability argument is as follows. Consider a sequence of documents $D_1, \ldots, D_n$ committed on a storage server in consecutive disposition groups. Suppose that all documents D_j $(j \leq i)$ have expired. We claim that Bob cannot get any information about the deleted document sets $D_1, \ldots, D_i$ from a strongly secure index. More formally, he would be able to extract exactly the same information if a random sequence of documents $D_1', \ldots D_i'$ was committed instead of the original sequence $D_1, \ldots, D_i$. In other words, any property that Bob is able to derive about some document set D_j by looking at its erased index on S should also be valid for a randomly-chosen doc-

ument set.

We can prove this by setting up an indistinguishability game with the above sequences (D_i, D_i'). If Bob is able to get any information about the sequence D_i which he cannot get for D_i', Bob can win the indistinguishability game.

3. PRIOR WORK

Zhu et al. proposed the concept of *logical deletion* from an inverted index [11], as shown in Figure 2(a). In logical deletion, the posting elements of all the documents in a disposition group (i.e., a set of documents that expire together) are encrypted using a disposition-group-specific key. This key is stored in its own separate file, whose expiry time is set to that of the disposition group. A pointer to the correct key file is stored at the first posting element for each disposition group in each posting list. Once the disposition group expires and the key file is deleted, Bob cannot decrypt the posting list elements and hence cannot reconstruct the contents of a document.

Zhu et al. proposed encrypting the tuple consisting of the document ID and the keyword hash. This is required to prevent the same document ID in two posting lists from being encrypted to the same ciphertext. Specifically, *docID* in keyword w's posting list is encrypted as follows:

$$V = E_K((docID) \oplus H(w)),$$

where E_K is a fast symmetric key algorithm like AES, K is a key, and H is a one-way hash function.

There are several problems with this scheme. Even after the key is deleted, Bob can determine that a set of posting elements belong to the same disposition group, by looking at the pointers to the key files. For example, in Figure 2(a), Bob learns that there were 5, 2, 4, and 3 occurrences of keywords from List1, List2, List3 and List4, respectively. Although Bob cannot determine if these keywords occurred in the same document, with additional context information he might be able to argue about document contents. For example, if posting elements for keywords "Martha", "Ralph", "ImClone", and "sell" occur in the same disposition group, while the remaining keywords have nothing to do with that topic, Bob might be able to claim that those keywords occurred in the same document. Also, as pointed out earlier, the nonoccurrence of a word in a disposition group may in itself be sensitive information.

The deletion scheme of Zhu et al. is not strongly (or weakly) secure. As an input to the indistinguishability game, Bob can pick document set sequence (D_i, D_i') such that the keyword frequencies and hence keyword posting list lengths of D_i and D_i' are different. Even after the index decryption key files are deleted, Bob can see the posting list lengths and hence distinguish between indexes corresponding to D_i and D_i'.

Mitra et al. proposed a deletion scheme that exploits posting list merging [6]. Their approach encrypts the keyword encoding stored with each posting element by XORing it with a key derived from a per-document secret key and the keyword hash. The document ID itself is stored in plain text. The document secret key is stored and deleted with the document. Without this key, the adversary cannot decipher the keyword encoding and hence cannot determine the keyword in the merged set that this posting element corresponds to.
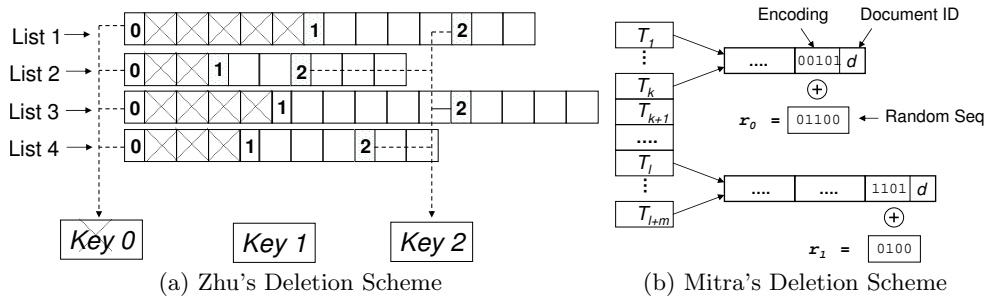
Figure 2: (a) Zhu's index deletion scheme. The posting list elements are encrypted with a per-disposition interval encryption key. The pointer to the encryption key is stored at the beginning of the posting list. The crossed-out elements in the figure show the posting elements that cannot be decrypted (and hence appear as random noise to the adversary) when the key file 0 is deleted. The adversary still sees the document frequencies for each keyword for each disposition group. (b) Mitra's index deletion scheme. The encoding stored with each posting element is encrypted by XORing it with a secret bit sequence (generated from a per document key). Once the key is deleted (along with the document), Bob does not know which keyword in the merged set d had.

This scheme also lets Bob argue about the contents of a document, if he has additional contextual information. For example, if the same document ID appears in the merged posting lists for keywords {Martha, Bush@whitehouse.gov} and {Ralph, service@woodworkers.com} and most pairings of these terms are unlikely to occur together in a single document, then Bob can argue that the document probably contained the words "Martha" and "Ralph". Mitra et. al. proposed heuristic merging strategies to address this problem. However, it is easy to show that these heuristic schemes are not strongly or weakly secure. Furthermore, this scheme is susceptible to all the attacks described in the previous section.

The scheme of Mitra et al. also has high space overhead. Unlike the normal merged inverted index, the keyword encodings stored with each posting element cannot be generated using a variable-length encoding scheme like Huffman coding—otherwise, Bob can guess the keyword from the length of the encoding. A fixed-length encoding, on the other hand, is space inefficient. For example, if 200 keywords are merged together into one list, 8 bits will be required to store the encoding. This increases the size of the index, and hence the index scanning time, by more than a factor of 1.5.

4. STRONGLY SECURE SCHEMES

We use the term "baseline approach" to refer to an ordinary (merged) inverted index on compliance storage. The baseline approach is not weakly secure, but has excellent lookup performance. We also slightly abuse the word *posting list of a keyword k* to actually refer to merged posting lists which can have multiple keywords.

4.1 Split Index

The easiest way to provide strongly secure deletion is to create a separate index for every disposition group, as shown in Figure 3(a). The expiry time of a posting list file is set to that of the corresponding disposition group. Once the disposition group expires, the posting list files are deleted. This deletion scheme is strongly secure: after the posting lists are deleted, the adversary cannot get any information about the deleted documents.

Unfortunately, this *split index* approach has very poor

query performance. A single-keyword query requires scanning as many posting list files as the number of disposition groups spanned by the query interval. Each such access incurs at least one disk seek plus the additional file system overhead of reading in the metadata for the file. For most queries, this will be much slower than the baseline approach, particularly when the disposition interval is small.

We measured the performance of the split index approach on the Enron email corpus (described in detail later). In Figure 3, the y axis shows the time to scan a keyword posting list, averaged across all the keywords. The x axis shows the number of disposition groups that overlap the query's time interval; this determines the number of posting list files that must be scanned. The different curves correspond to different sizes of disposition groups. The baseline curve "baseline" shows the baseline approach, where a single posting list file is created for each keyword, regardless of the number of disposition groups.

The key observation from Figure 3 is that even by choosing a disposition interval of almost 4 months, the split index is slower by a factor of 5 than the baseline approach when the query interval is 2 yrs. Further, the choice of the disposition interval will be dictated by company policies and cannot be chosen arbitrarily. For a 1-week disposition interval, the split index approach is slower than the baseline approach by a factor of 15.

4.2 Overflow Index

Let's revisit the indistinguishability game. In the split index approach, the index is deleted when the documents expire. Without the index (and the documents), Bob can only perform as well as a random adversary in guessing the document set on a storage server. The same level of indistinguishability can be achieved if Alice can somehow make the index structures of every document set appear exactly the same after the documents have expired. For example, if for every document set, the posting lists are the same length, and all the posting list entries are random bits, Bob will not be able to guess which one of document sets D_0^k or D_1^k does the index structure correspond to. This is the main idea behind our next scheme, the *overflow index*, shown in Figure 4.
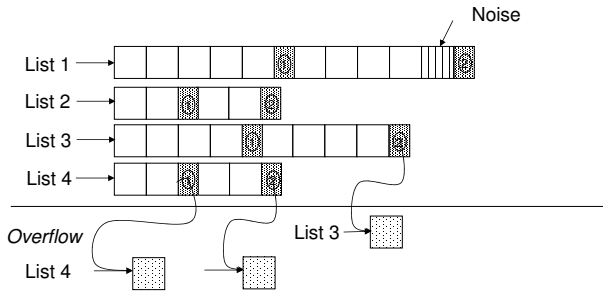
As in Zhu et al.'s scheme, in an overflow index we encrypt

(a) Split Index

(b) Query Performance
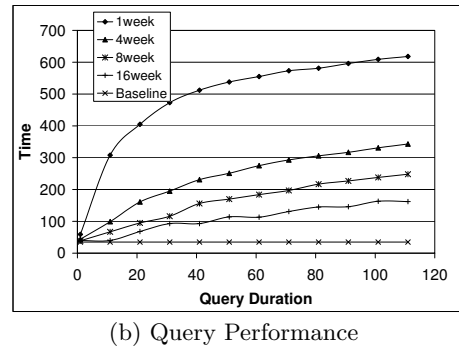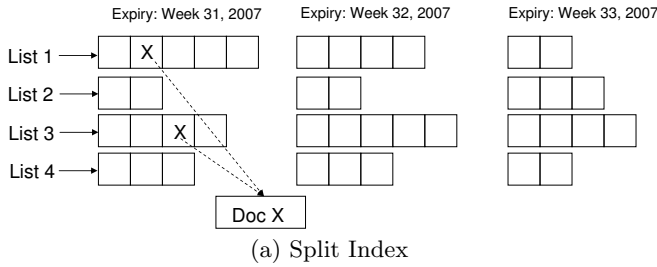
Figure 3: (a) The split index provides strongly secure deletion by creating a separate index for each disposition group. (b) Split index query performance is very poor. Even with a big disposition group, such as 16 weeks, the extra seeks slow down query performance by a factor of almost 6.

Figure 4: Overflow index. The posting list elements are encrypted as before. Additionally, the posting list lengths are made equal to a default value, across all the disposition groups. Noise posting list entries are added (shown as shaded boxes) if the actual posting list length is less than the default value when the disposition interval ends. If the posting list length exceeds the default value, the extra elements are put in overflow lists (shown as dotted boxes). The overflow list for each disposition group and keyword is stored as a separate file and is deleted on expiry.

all the posting list elements using a per-disposition-group key. Additionally, the posting list length for keyword k is set to *cover length* l_k for all disposition groups, as follows:

- If the total length of k's posting list for a disposition interval is less than l_k, then a special start-of-noise-term marker is appended to the posting list. The marker itself is encrypted as for an ordinary posting list element. Then random noise posting elements are added to make the posting list length equal to l_k.

- If the posting list length exceeds l_k, the additional posting elements are stored in an overflow area and a pointer to the area is introduced at the end of the list. A separate overflow area is created for each posting list that overflows in each disposition group. In Figure 4, overflow areas store the overflowing keywords from posting lists 4 and 3. Each overflow area occupies a separate file whose expiry date is set to that of the corresponding disposition group. Once the disposition group expires, its overflow files are deleted. The total length of the non-overflow posting list, including the overflow pointer, is kept at l_k.

During query processing, each posting list is scanned as usual, except that elements beyond a start-of-noise-term marker are ignored. If there is an overflow area for the posting list, then that overflow area file is read, which costs a random I/O.

The overflow index approach is strongly secure because Bob only sees posting list lengths that are the same across all the disposition groups. We formally argue the security of this scheme later.

Choosing l_k

The big challenge in the overflow index approach is to choose appropriate posting list lengths l_k. The choice of l_k offers a tradeoff between index size and query performance. A large l_k reduces the number of overflow accesses and thus has better query performance[1]. On the other hand, a large l_k requires more noise to be added to too-short lists, and hence exhibits poor space efficiency. Choosing the correct l_k values can be framed as the following optimization problem.

---

[1]Although it requires more noise terms to be read it, the overhead is usually negligible as compared to the additional seek time.

Let K be the set of (merged) posting lists. Consider a posting list $k \in K$. Suppose that k's length in the different disposition groups comes from a probability distribution function P_k, where $P_k(i)$ is the probability of this length being i. The expected length B_{l_k} (per disposition group) of the overflow area for the kth posting list can be written as follows:

$$B_{l_k} = \sum_{i>l_k}(i - l_k)P_k(i)$$

The expected total size S_{l_k} (per disposition group) of k's posting list, including the overflow area is (using linearity of expectations):

$$S_{l_k} = l_k + B_{l_k}$$

Hence, the total expected size S of the index is

$$\sum_{k \in K} S_{l_k} = \sum_{k \in K} l_k + \sum_{k \in K}\sum_{i>l_k}(i - l_k)P_k(i)$$

Now consider the query performance. A posting list scan for a keyword k has the following components:

- The initial seek to the start position of the first disposition group (in the query interval) in the posting list. This seek time is incurred only once per posting list scan.
- A possible seek to the start of the overflow area. The expected time O_k for this (per disposition group) is the seek time, multiplied by the probability of overflow:

$$O_{l_k} = C_{seek} \times \sum_{i>l_k}P_k(i)$$

- Disk transfer time to transfer the actual posting elements (including some from the overflow area) and the noise elements (from the main posting list). The expected transfer time per disposition group is the product of the expected posting list length S_k and the disk transfer speed:

$$T_{l_k} = C_{trans} \times S_{l_k}$$

The expected total run time R_k for scanning d disposition groups is obtained by summing the above components:

$$R_{l_k} = C_{seek} + d \times (O_{l_k} + T_{l_k})$$

The total run time for a given query probability distribution can be estimated as follows. Let Q_k be the probability of the keyword k occurring in a query. Also, assume that the time interval (i.e., the restrictions on the document creation time) for a query has expected value d that is chosen independently of the start time of the interval.

$$
\begin{aligned}
QCost(K) &= \sum_{k \in K} R_{l_k} \times Q_k \\
&= \sum_{k \in K} C_{seek} \times Q_k + d \times \sum_{k \in K}(O_{l_k} + T_{l_k}) \times Q_k
\end{aligned}
$$

The optimization problem is to choose l_k for each $k \in K$, in a way that minimizes the total query run time $QCost(K)$, given the space constraint

$$\sum_{k \in K} S_{l_k} \leq C$$
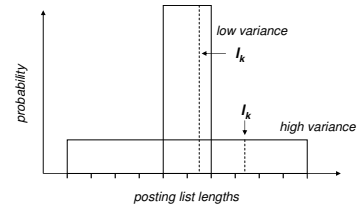
Figure 5: Two example posting list length probability distributions. The graph shows l_k values that have the same probability of overflow for the two distributions. The distribution with a higher variance has more space overhead.

The problem function exhibits an optimal substructure. Let $OptCost(K, C)$ be the optimal total query run time under space constraint C. If l_k is the optimal threshold length for a posting list $k \in K$ and S_{l_k} is the expected space overhead, then the optimal total query run time $OptCost(K - \{k\}, C - S_{l_k})$ for the posting lists $K - \{k\}$ under space constraint $C - S_{l_k}$ is also optimal for the entire set. That is,

$$OptCost(K, C) = R_{l_k} \times Q_k + OptCost(K - \{k\}, C - S_{l_k})$$

This optimization problem can be solved by dynamic programming. One needs to iterate over all different choices of l_k for a keyword k, and choose the one with the minimum total query run time, as given below:

$$\min_{0 \leq l_k, S_{l_k} \leq C}(R_{l_k}Q_k + OptCost(K - \{k\}, C - S_{l_k}))$$

The solutions to the optimal subproblems $OptCost(K, C)$ can be stored in a table to avoid recomputation. We omit the pseudocode due to lack of space.

$P_k(i)$ **Distribution.** The above optimization problem requires the probability distribution P_k as input. There are two ways to estimate P_k: one can start with a probability model (e.g., binomial or Poisson) and estimate the model parameters from a training set of documents using learning techniques like MLE. The other option is to estimate the P_k values directly from a training set. The second option is inaccurate, as posting list lengths that do not occur in the training set are assigned 0 probability. However, it is much simpler than the first option. Coming up with a posting list length probability model is a separate research topic in itself, and we leave it as future work. In our experiments, we have used 10% of the data as a training set and evaluated our scheme in the remaining 90%.

The performance of an overflow index is sensitive to the posting list length distribution $P_k(i)$. For example, the overflow index performs best when the posting lists are the same length across all the disposition groups for each k. To see what happens when we move away from this ideal state, consider two uniform distributions with the same mean but different variance, as illustrated in Figure 5. The query performance of the index is governed by the probability of overflow, which is given the area under the curve to the right of l_k (all posting lists with length more than l_k overflow). To have the same probability of overflow for both distributions, one must pick a larger l_k for the higher variance curve than for the lower variance curve. On the other hand, the space overhead of the index is determined by the number of noise

terms added to too-short posting lists. Thus for the above choices of l_k, the higher variance curve will have higher space overhead. In other words, a posting list length distribution with low variance leads to better space/query performance.

4.3 Normalizing l_k

If P_k has a large variance and the cover lengths l_k are constant across all disposition groups, the overflow index will not perform well during lookups. Unfortunately, this was the case for our test data set of Enron email. Figure 6(a) shows the posting list length distribution for Enron list 0; the other lists had similar distributions. Figure 6(b) plots the number of emails per disposition group and Figure 6(c) plots the posting list lengths normalized by the number of emails. The key observations from this figure are, first, that there is a huge variation in the number of emails per disposition group, ranging from 500 in disposition group 0 to over 10000 in group 52. This results in a huge variance in the posting list length distribution. Second, the normalized posting list lengths distribution has a low variance. In other words, the fraction of emails containing a particular term remains relatively stable across disposition groups. This observation leads us to a possible solution to this problem: since the posting list lengths scale up/down with the number of emails in almost the same proportion, one can scale l_k by the number of emails. Specifically, if N_d is the number of emails in the current disposition group and N_t is the average number of emails in the training set, we use $\frac{l_k * N_d}{N_t}$ as the threshold value for that disposition group.[2] Experiments reported in the next section show that this scheme performs substantially better than the keeping the same l_k across all groups.

The number of records that will be created in a disposition group is not known in advance. However, one can use the number from the last disposition group as an approximation to the expected number of records in the current group. We call this a *normalized overflow index*. The other option is to commit the records to the index at the termination of the disposition group, when the number of records is known. We adopt the former technique in our evaluation.

Security Properties

Let's first consider a non-normalized overflow index. The encryption keys and the overflow areas are deleted after the disposition group expires. The adversary only sees the posting list lengths which are same across all the disposition groups. He hence cannot deduce anything about the document keyword frequencies for a specific deleted disposition group. Formally, this scheme achieves strongly secure deletion. The deleted indexes corresponding to two different document sets look exactly the same. An adversary having access to just the deleted index cannot distinguish between the document sets.

The overflow scheme does leak some information. The adversary can learn about the posting list length distributions P_k from the cover lengths l_k used to create the index. For example, if $l_{k_1} > l_{k_2}$ for two different posting lists k_1 and k_2 and the posting lists are equally likely to be queried,

the adversary will know that $E(P_{k_1}) > E(P_{k_2})$. However, this information is not sensitive. The P_k distribution is an underlying property of the document corpus that remains stable over relatively long periods of time. Thus the adversary can learn those distributions from the set of non-expired documents on the storage server.

Finally, if there is a change in the P_k distribution, a new index must be created. A change in distribution can be detected by comparing the current P_k distribution (learnt over the last few disposition groups) with the training distribution using standard techniques, such as computing the KL-divergence between the distributions. If the KL-divergence exceeds a certain threshold, the cover lengths l_k are recomputed and a new index is created. Once all the documents in the old index have expired, the old posting list files can be discarded. This prevents the adversary from learning the old P_k distribution.

The normalized overflow index, where the l_k cover lengths are scaled up according to the number of documents in the previous disposition group, is weakly secure but not strongly secure. Since the new l_k values depend on the number of emails in the document set, Bob can distinguish between the indexes corresponding to documents sets that have different sizes. After the index is erased, the adversary can learn the number of documents that have been committed since index creation, based on the total length of the posting list. After the disposition group key is deleted, the adversary cannot identify the disposition group start point in the posting list. Hence, he only learn the total posting list length and not the posting list lengths corresponding to each disposition group. In most cases, the number of documents that have been created since disposition group 0 will not be considered sensitive.

5. EXPERIMENTS

Our experiments focus on evaluating the query performance of split, overflow, and normalized overflow indexes. Insertion performance is also important; we do not need to measure it here as insertions into all three kinds of inverted indexes will take the same amount of time as insertions into an ordinary merged inverted index.

5.1 Data

Privacy and confidentiality concerns make it very difficult to get real business documents and queries. As test data, we used a collection of 422,000 emails from the Enron email corpus at `www.cs.cmu.edu/~enron/enron\_mail\_030204.tar.gz`. These emails were exchanged in the 2 year period from January 2000 to December 2001; we omitted the other 50,000 emails in the corpus because they were sprinkled very thinly across the time periods of 1994-2000 and 2002-3. Each email has a metadata tag identifying the sender, receiver, and the time the email was sent. We use this time information to divide the email documents into disposition groups. In most experiments, we set the disposition group interval to 1 week, for a total of 104 disposition groups.

5.2 Query Model

No query log is available for the Enron email corpus, so we use two synthetic query models. Under the *uniform query model*, each posting list is equally likely to be scanned. This assumption is reasonable because we are using *merged* posting lists; the merging process can be used to even out the

---

[2]The alternative is to reformulate the optimization problem with the distribution of the number of emails as a parameter. We leave this as an option for future work, though this distribution is often very hard to estimate, as for Enron email.
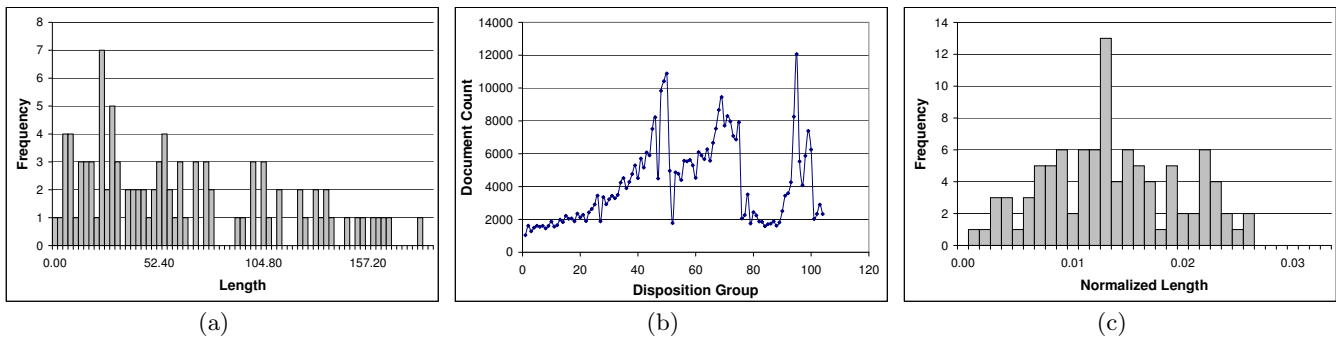
Figure 6: (a) Posting list lengths for list 0, across 104 disposition groups. (b) Number of emails per disposition group. (c) Posting list lengths normalized by the number of emails per group.
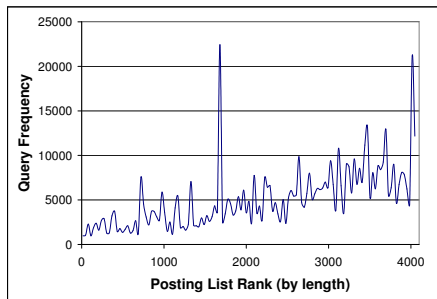

Figure 7: Posting list query frequencies for a corporate intranet data set.

Figure 8: Query performance when the entire data set is used to estimate the P_k values for the overflow index.

access probabilities of the posting lists. In particular, instead of using uniform merging, we can merge keywords so that each posting list was equally likely to be queried in the workload of the recent past.

To validate the uniform query model, we used a set of a million HTML documents obtained from the intranet of a large company, and a set of 300,000 keyword queries submitted by real users on the intranet. (The documents have no associated creation times and hence could not be used in our evaluation of deletion techniques, as there is no justifiable way to split the documents into disposition groups.) We built a merged inverted index, using uniformly random merging of terms. In Figure 7, we plot the posting list query frequencies of this data set. The y axis plots the number of times a given posting list (i.e., any term that has been merged into the list) is queried in the query log. The x axis shows the posting lists ordered by their lengths; the 0th list is the shortest while the 4095th list is the longest. The figure shows that longer lists are queried more often than shorter lists. More importantly, 70% of the posting list query frequencies are within the fraction 0.7 and 1.5 of the mean. Thus a uniform query model is not an unreasonable approximation to the real-world distribution, even without careful merging of terms to even out access probabilities.

Our second query model is called the *query log model*, and it is based on the query probability distribution in Figure 7. We sorted the Enron posting lists by size and assigned each list a query frequency as a function of its rank, as given by Figure 7.
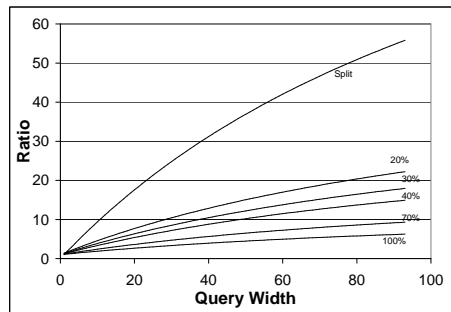
5.3   Estimating Seek & Transfer Times

Most current generation WORM devices run a regular file system internally, with the file write option disabled for committed files. As we do not have access to a WORM storage system, we carried out our experiments on ReiserFS, a high performance Linux file system. We created the posting list files in the same order as they would have been created in the real world, and then scanned them in that same order. For example, for the split index, all the posting list files of the same disposition group were created consecutively on ReiserFS. We computed the average seek time for the split index (needed for our simulations) as the average latency between opening successive files of a posting list, corresponding to successive disposition groups. For the overflow index, we measured the average seek time after creating the overflow area files of the same disposition group successively on the file system. The main posting lists were created as consecutive files on a separate partition. A WORM system that periodically reorganizes the file blocks to make them contiguous on disk is likely to create such a file organization.

5.4   Results

Complete Knowledge. Our first experiment evaluates the hypothetical case where the posting list lengths for each disposition group are known in advance. This experiment gives the best performance improvement that can be achieved by an overflow index, compared to the split index scheme.

We use the keyword document frequencies of all 104 disposition groups to learn the P_k distribution and solve the optimization problem. The index query performance is eval-

uated using the uniform query model, described above. Figure 8 plots the query performance of the resulting indexes. The y axis shows the slowdown of the overflow and split indexes, compared to the baseline (insecure) approach where a single index is created for all the disposition groups. The x axis plots the queried time interval as the number of one-week disposition groups. The different curves correspond to the different space overheads S given to the DP optimization problem; the $p\%$ curve is for an overflow index that uses $p\%$ more space than the baseline index. As evident from the figure, the overflow index is better than the split index by a factor of 3, even with a mere 30% space overhead. With a 100% space overhead the speedup is over a factor of 12.

Learning the keyword query distribution. In our next set of experiments, we evaluate the effectiveness of learning P_k values from a portion of the document corpus. We use 10% of the dataset (10 epochs) as a training set and the remaining as the test set. We solve the optimization problem based on the P_k values learned from the training set, and evaluate the resulting overflow index on the test set. This experiment also uses a uniform query model.

Figure 9(a) compares the query performance of the resulting index with that of a split index. As before, the different curves correspond to the different space overhead parameters passed to the optimization function. With such minimal training, the overflow index performs only marginally better than the split index.

Another interesting observation is the space overhead of the overflow index on the test data, as plotted in Figure 9(b)[3]. The x axis plots the time interval in disposition groups, and the y axis plots the space overhead of the index. The graph shows the space overhead of the cumulative index, which is the index containing entries from the 0th disposition group up to the current x axis disposition group. The overflow index created with a 150% space overhead on the training set has an actual space overhead of less than 10% on the test set.

As explained before, the cause of this phenomenon lies in the number of emails per disposition group, as shown in Figure 6(b). From an average of 500 emails in the first 10 disposition groups, which are our training set, the count jumps to about 10,000 emails per group in the subsequent 70 groups. The posting list lengths in the subsequent groups also scale by the same factor.

The posting list cover lengths l_k are learned from the short posting lists of the training set, and hence are too short to work well for the long-listed test set. As a result, most of the test set posting lists overflow. Scanning the resulting index requires almost one random I/O per epoch to fetch the overflow area from disk. At the same time, because of the large number of overflows, very little space is wasted in storing the noise terms. In effect, this reduces the overflow index to the split index both in terms of query time and space overhead.

Normalized Overflow Index. We address the problem of poor training data by using a normalized overflow index, which scales up the cover lengths l_k based on the number of documents in the previous disposition group. Specifically, if the previous disposition group had N documents, then $l_k \times \frac{N}{N_t}$ is the cover length for the current disposition interval. N_t, the average number of documents per disposition group in the training set, was 1514 in our case. Figure 9(c) plots the query performance of the normalized overflow index on the same data and queries as in Figure 9(a). As before, the different curves correspond to different space overhead parameters. The normalized overflow index outperforms the split index by almost a factor of 6, with less than 100% space overhead. The space overhead of the normalized overflow index on the test set was 12.9%, 19%, 27%, 34%, 50% and 65%, corresponding to 20%, 30%, 40%, 50%, 70% and 100% space overhead allowed on the training set.

Uniform Length Subset. To further investigate the effect of document arrival rates on index lookup performance, we evaluated the unnormalized overflow index on a portion of the data for which the number of documents per disposition group was relatively constant—the period between weeks 40 and 80. Figure 10(a-c) compares the query performance of the overflow index with the split (baseline) index for different sizes of the training and test sets. On this data, the overflow index outperforms the split index by almost a factor of 5, while incurring less than a factor of 2 space overhead.

Varying the Disposition Group Size. By choosing a larger disposition interval, we can reduce the number of separate indexes that have to be maintained in the split index scheme. A larger disposition interval also reduces the number of overflow areas for the overflow index. Hence, the query performance of both these indexing schemes should improve as the disposition interval increases.

Figure 11 plots the index query performance for disposition intervals of 4, 8 and 16 weeks. Although the split index performance improves as the interval grows, the overflow index is still 2-4 times faster than the split index.

Varying the Query Distribution. In the next set of experiments, we evaluate the index performance under the query log model of query patterns. Figure 12(a) shows the query performance of the overflow index under the query log distribution shown in Figure 7. The query distribution is used as an input to the cover length optimization problem and is also used to generate the queries used to test the index's performance on the Enron email. The overflow index still outperforms the split index under this distribution.

We also evaluate the effect of using one query probability distribution for choosing the cover lengths for the overflow index and a different distribution at run time. Such changes are important to consider, because query patterns can shift during the multi-year retention periods of compliance data. Figure 12(b) shows the case where cover lengths are based on a uniform query distribution but the actual workload follows the query log distribution. The key observations from the figure are as follows.

- The overflow index is still a factor of 5-6 faster than the split index. The overflow scheme reduces the random I/Os incurred in a posting list scan, which more than offsets the additional I/O time required to read the noise terms.
- The split index performs better for the query log distri-

---

[3]The curves in Figure 9(a) are the space overhead of the overflow index on the training data for which the optimization problem is solved.

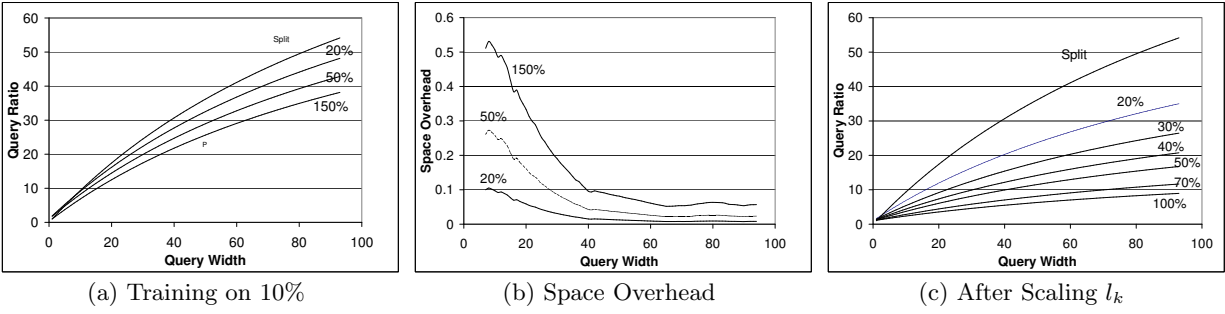(a) Training on 10%   (b) Space Overhead   (c) After Scaling l_k

Figure 9: P_k values are learned from 10% of the data. (a) The performance of the overflow index on the remaining 90% of the data is almost as bad as that of the split index. (b) The space overhead of the overflow index is close to that of the baseline index. (c) The overflow index performance greatly improves after scaling the cover lengths by the number of records in the group.
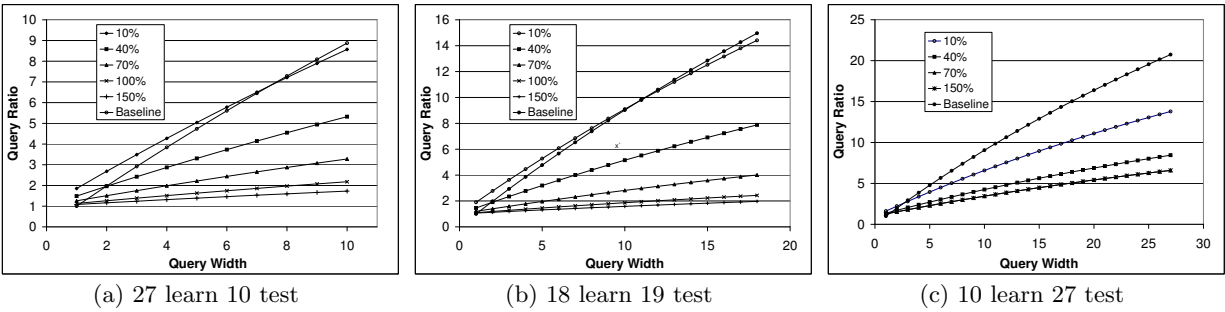

(a) 27 learn 10 test   (b) 18 learn 19 test   (c) 10 learn 27 test

Figure 10: Index query performance for different test and training set sizes.

(a) Disposition Interval = 4 Weeks   (b) Disposition Interval = 8 Weeks   (c) Disposition Interval = 16 Weeks
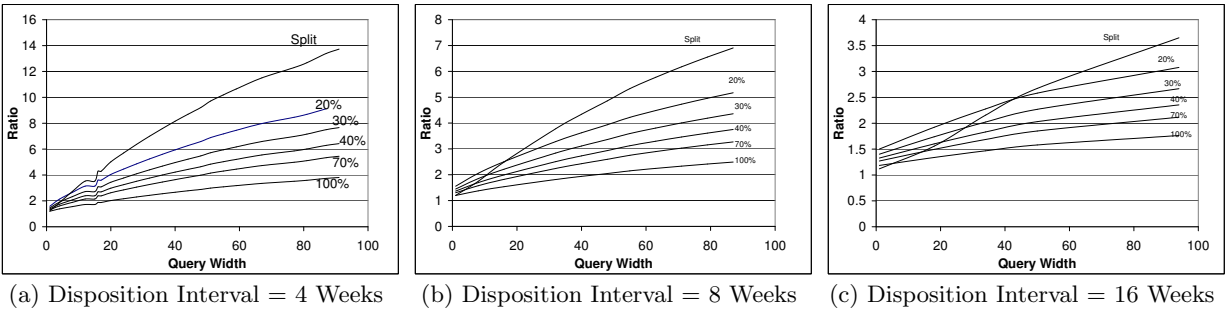
Figure 11: Overflow and split index query performance for different disposition intervals.

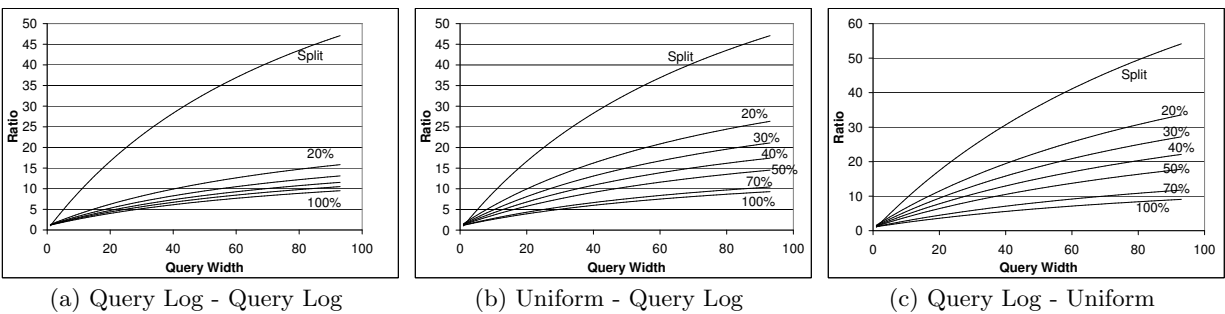(a) Query Log - Query Log   (b) Uniform - Query Log   (c) Query Log - Uniform

Figure 12: Overflow index performance when cover lengths are chosen using distribution 1, but queries actually follow distribution 2. (a) 1 = 2 = query log distribution. (b) 1 = uniform, 2 = query log. (c) 1 = query log, 2 = uniform.

bution than for the uniform query distribution. The longer posting lists have relatively higher query frequencies under the query log model, which is not the case with the uniform query model. At the same time, longer lists incur relatively less performance penalty in the split index, as for such lists the transfer time dominates the seek time, and the transfer time is the same for the split index and the baseline index. This has an overall positive effect on the performance of the baseline index.

6. CONCLUSION

To date, the work on compliance storage has focused primarily on ensuring that documents and their associated inverted index entries are immutable during their retention periods. For many applications of compliance storage, however, it is equally important to eradicate the (potentially incriminating) documents and index entries once their retention periods are over. It is extremely hard to remove entries from an inverted index for compliance storage; in this paper we have shown that previously proposed schemes either leave dangerous traces of the index entries, or else lead to extremely poor lookup performance. As a solution, we proposed *overflow indexes*, which combine good query performance, a moderate space overhead, and good security guarantees. Overflow indexes achieve these goals by having each posting list represent a set of terms rather than a single term, assigning a predetermined *cover length* to each posting list, adding noise entries to posting lists that do not reach their predetermined lengths after a certain period, and using overflow files for posting lists that exceed their predetermined lengths during a certain period (the *disposition interval*). For workloads with fairly steady document arrival rates and a given size limit for the posting lists, optimal cover lengths can be predetermined for all future disposition intervals using a dynamic programming approach; the resulting overflow index is strongly secure, in the sense that an adversary will not be able to infer anything about the content of the set of documents created during a disposition interval, once those documents have been deleted from the index. An overflow index is 5-6 times faster for processing queries than the naive approach of keeping a separate index for each disposition interval, with a factor of 2 storage overhead. This space overhead is very reasonable considering the fact that an inverted index occupies only 10-15% of the space of the underlying corpus.

For workloads with highly variable arrival rates, good query performance requires scaling the cover lengths for the current disposition interval, based on the total number of documents that were inserted during the previous disposition interval. The resulting *normalized overflow index* is weakly secure, in the sense that it leaks a small amount of information about the number of documents inserted during each interval.

7. REFERENCES

[1] Congress of the United States of America. Sarbanes-Oxley Act, 2002. Available at http://thomas.loc.gov.

[2] EMC Corp. EMC Centera Content Addressed Storage System, 2003. Available at http://www.emc.com/products/ systems/centera_ce.jsp.

[3] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[4] IBM Corp. IBM TotalStorage DR550, 2006. http://www-03.ibm.com/systems/storage/index.html.

[5] S. Mitra, W. Hsu, and M. Winslett. Trustworthy Keyword Search for Regulatory Compliance. In *Very Large Data Bases (VLDB)*, Sept 2006.

[6] S. Mitra and M. Winslett. Secure Deletion from Inverted Indexes on Compliance Storage. In *Storage Security and Survivability Workshop*, Oct 2006.

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

[8] Securities and Exchange Commission. Guidance to Broker-Dealers on the Use of Electronic Storage Media under the National Commerce Act of 2000 with Respect to Rule 17a-4(f), 2001. Available at http://www.sec.gov/ rules/interp/34-44238.htm.

[9] The Enterprise Storage Group, Inc. Compliance: The effect on information management and the storage industry, May 2003. Available at www.enterprisestoragegroup.com.

[10] I. H. Wittenm, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufman, San Francisco, CA, 1999.

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