Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records*

Qingbo Zhu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
qzhu1@cs.uiuc.edu

Windsor W. Hsu
Computer Science Storage Systems Department
IBM Almaden Research Center
San Jose, CA 95120
windsor@almaden.ibm.com

ABSTRACT

As critical records are increasingly stored in electronic form, which tends to make for easy destruction and clandestine modification, it is imperative that they be properly managed to preserve their trustworthiness, *i.e.*, their ability to provide irrefutable proof and accurate details of events that have occurred. The need for proper record keeping is further underscored by the recent corporate misconduct and ensuing attempts to destroy incriminating records. Currently, the industry practice and regulatory requirements (*e.g.*, SEC Rule 17a-4) rely on storing records in WORM storage to immutably preserve the records. In this paper, we contend that simply storing records in WORM storage is increasingly inadequate to ensure that they are trustworthy. Specifically, with the large volume of records that are typical today, meeting the ever more stringent query response time requires the use of direct access mechanisms such as indexes. Relying on indexes for accessing records could, however, provide a means for effectively altering or deleting records, even those stored in WORM storage.

In this paper, we establish the key requirements for a fossilized index that protects the records from such logical modification. We also analyze current indexing methods to determine how they fall short of these requirements. Based on our insights, we propose the Generalized Hash Tree (GHT). Using both theoretical analysis and simulations with real system data, we demonstrate that the GHT can satisfy the requirements of a fossilized index with performance and cost that are comparable to regular indexing techniques such as the B-tree. We further note that as records are indexed on multiple fields to facilitate search and retrieval, the records can be reconstructed from the corresponding index entries even after the records expire and are disposed of. Therefore, we also present a novel method to eliminate this disclosure risk by allowing an index entry to be effectively disposed of when its record expires.

1. INTRODUCTION

Records such as electronic mail, financial statements, medical images, drug development logs, quality assurance documents and purchase orders are valuable assets. They represent much of the data on which key decisions in business operations and other critical activities are based. Having records that are accurate and readily accessible is therefore vital. Records also serve as evidence of activity, but to be effective the records must be credible and accessible. Given the high stakes involved, tampering with the records could yield huge gains and must be specifically guarded against, especially as the records are increasingly in electronic form, which makes them relatively easy to delete and modify without leaving so much as a trace. Ensuring that the records are trustworthy, i.e., not only readily accessible and accurate, but also credible and irrefutable, is particularly imperative in the litigious US. On average, a Fortune 500 company is the target of 125 non-frivolous lawsuits at any given time, and the damages awarded are increasingly rapidly, as is the cost of electronic data discovery, which is projected to rise at 65% per year to reach $2 billion in 2006 [22].

Furthermore, a growing proportion of the records are subject to regulations that specify how they should be managed. In the US alone, there are currently more than 10,000 such regulations [26]. The key focus of many of these regulations (*e.g.*, Sarbanes-Oxley Act [5], SEC Rule 17a-4 [21]) is to ensure that records are trustworthy. Non-compliance with these regulations could result in stiff penalties. For example, unprecedented fines have recently been levied by several regulatory bodies, including the Securities Exchange Commission (SEC) and the Food and Drug Administration (FDA), for non-compliance. The bad publicity of non-compliance and the ensuing investor flight alone could cost an organization dearly. As information becomes more valuable to organizations and with recent headlines of corporate misdeeds, accounting scandals and securities fraud, the number and scope of such regulations are likely to grow. Worldwide, the volume of compliant records is projected to increase by 64% per year to almost 2 PB in 2006 [26].

Thus, there has been a recent rush to introduce Write-Once-Read-Many (WORM) storage devices (*e.g.*, [8, 12, 16, 23]) to enable the effective preservation of records. However, simply storing records in WORM storage, as is the current focus, is far from adequate to ensure that the records are trustworthy. We contend that while WORM storage is essential for establishing trustworthy electronic records, it alone effectively offers no value in ensuring that records are trustworthy. The key issue is that with today's large volume of records and increasingly stringent query response time [4], some form of direct access mechanism such as an index must be available as needed to access the records. But unless the index is properly realized, the records stored in WORM storage can in effect be hidden or altered. For example, Figure 1 illustrates that a record stored in WORM storage can, for all intents and purposes, be modified or deleted if the index through which it is accessed can be suitably manipulated.

*Work conducted at IBM Almaden Research Center, San Jose, CA in the summer of 2004.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'05, June 14-16, 2005, Baltimore,Maryland, USA
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

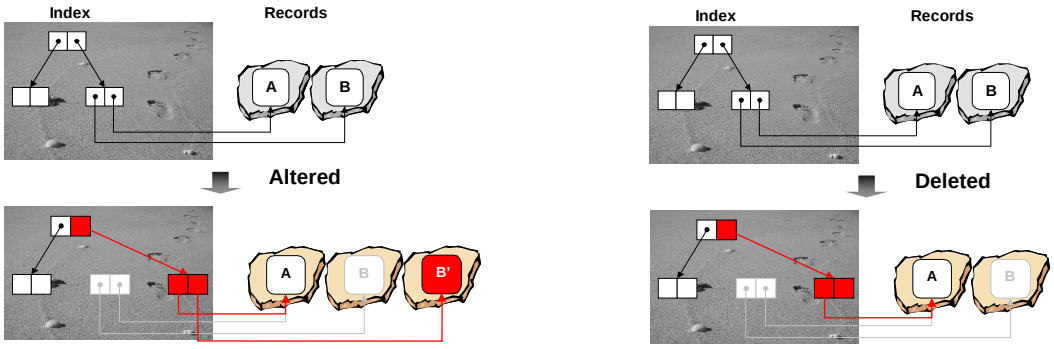


Figure 1: Non-Fossilized Index Allows Logical Modification. *This diagram shows that even if record B is stored in WORM storage, it can effectively be altered or deleted if the index through which it is accessed can be suitably manipulated. The records are depicted as cast in slabs of stone to indicate that they are not modifiable. The index blocks are illustrated as written in sand to show that they can be easily changed.*

In this paper, we establish the basic requirements for an indexing method that is impervious to such logical modification of records. We call such an index a *fossilized index* because its structure is effectively cast in stone. A fossilized index can be used to ensure that once a record is preserved in WORM storage, the record will be quickly accessible in an unaltered form through the index. We analyze current indexing methods to determine how these methods, even those designed for WORM storage, fall short of the fossilized index requirements. Based on the insights we obtained, we propose a novel indexing method, the *Generalized Hash Tree* (*GHT*), that satisfies all of the requirements. The GHT is, in effect, a tree that grows from the root down to the leaves in a balanced fashion without requiring dynamic adjustments to its structure. Using theoretical analysis and simulations with real system data, we demonstrate that the GHT can provide performance and cost that are comparable to regular indexing techniques such as the B-Tree. In other words, it is practical to protect records from logical modification.

Note that 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. For example, SEC Rule 17a-4 [21] specifies a retention period of three years for email, attachments, memos, instant messaging, *etc.* Even after records have passed any mandated retention periods, if they are available, they are subject to discovery, and typically at great expense to the owning organization [22]. Thus, it is important for an organization to properly dispose of records that are no longer useful to the organization and have passed any mandated retention periods. However, as records are indexed on multiple fields to facilitate search and retrieval, the records can be reconstructed (*e.g.*, by performing a "join" on the record ID) from the corresponding index entries even after the records expire and are disposed of. Therefore, in this paper, we further present a novel method to eliminate this disclosure risk by allowing an index entry to be effectively disposed of when the corresponding record expires.

The rest of the paper is organized as follows. We establish the key requirements for a fossilized index in Section 2. We analyze how previous indexing methods fall short of these requirements in Section 3. In Section 4, we present the GHT and in Section 5, we discuss selective disposition of index entries. Simulation results are reported in Section 6. Section 7 concludes the paper.

2. TRUSTWORTHY RECORD KEEPING

The fundamental purpose of record keeping is to establish solid proof and accurate details of events that have occurred. Trustworthy records are, therefore, those that can be relied upon to achieve this purpose. In [11], we make the case that trustworthiness has to be established from an end-to-end perspective, *i.e.*, from when the records are created to when they are actually used by an agent conducting an audit, a legal or regulatory discovery, an internal investigation, *etc.* We further develop a process called fossilization for achieving end-to-end trust in record keeping. Here, we briefly outline some of the key ideas in fossilization.

The process of creating accurate records for all of the relevant events as they occur is generally trusted. This is the case especially if the records are used in the normal course of business, and are hence required for the proper functioning of the organization. Furthermore, record creation is an ongoing process for which periodic audits are effective in ensuring proper execution. More importantly, it is extremely unlikely that one can anticipate at record creation time how a record could be modified for personal gain. In any case, if one is privy to such knowledge, one would arguably be better off trying to influence the events themselves. The basic objective of record keeping is not to prevent the writing of history, but to prevent the changing of history; in other words, changing the records after the fact.

Therefore, the key requirement for trustworthy record keeping is to ensure that in an enquiry, all of the relevant records can be quickly located and retrieved in an untampered form. This means that the records have to be protected from any physical modification during their storage. Modification of records could result from user errors, such as issuing the wrong commands and replacing the wrong disks, and from software bugs. Given our increasing reliance on computer records, the potential gain from manipulating and modifying the records is huge. Thus, more importantly, the records have to be protected from intentional attacks, even inside attacks launched by disgruntled employees, company insiders, or conspiring technology experts. An adversary is likely to have the highest (*e.g.*, executive) level of support and insider access, privilege and knowledge. He can be thought of as the super system administrator. Although the adversary has physical access to the records, he cannot destroy them in a blatant fashion because it would result in severe penalties and even the presumption of guilt. His mission is to clandestinely hide or modify specific records.

Typically, the records are stored in some form of WORM storage to protect them from modification. With the growing volume of records and the ever more stringent response time to enquiries, direct access mechanisms such as indexes increasingly have to be maintained to ensure that all of the records relevant to an enquiry can be discovered and retrieved in a timely fashion, typically within days and sometimes even within hours [4]. As illustrated earlier in

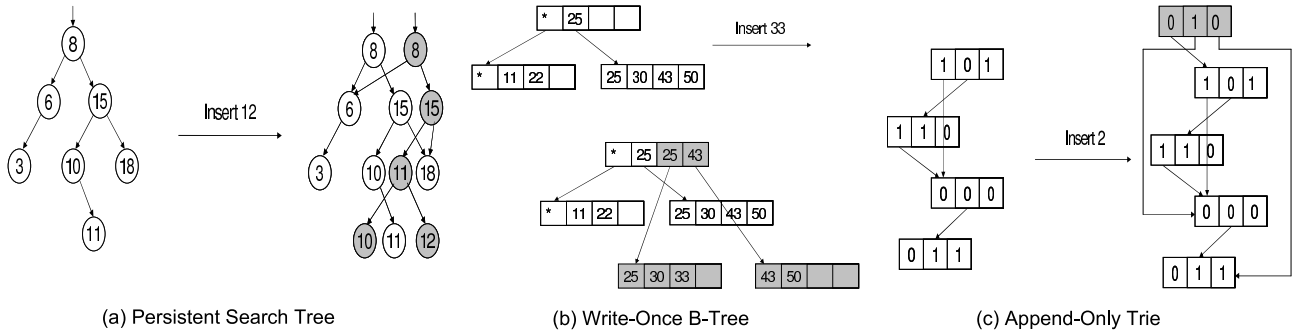


Figure 2: Examples of Previous Indexing Methods

Figure 1, however, if records are accessed through an index, even if they are stored in WORM storage, they will still be vulnerable to logical modification if the index can be suitably manipulated. In particular, an adversarial system administrator could write carefully crafted data to the WORM storage to subvert the index and logically hide or modify selected records. He could, for example, fake an adjustment to the index structure (e.g. tree rebalance), which would enable him to omit crucial records from the new structure. He could also prevent timely access to crucial records by causing the index to degenerate (e.g. by inserting carefully chosen spurious records).

The linchpin of trustworthy records is, therefore, a *fossilized index* that ensures that once a record is preserved in WORM storage, it is accessible in an unaltered form and in a timely fashion. The key properties for such an index are as follows:

- Once a record is committed, both the index entry for that record and the path to that entry must be immutable. Specifically, the insertion of a new record into the system must not in any way affect how previously inserted records are accessed through that index. This means that once the insertion of a record into the index has been committed to WORM storage, the record is guaranteed to be accessible through that index unless the WORM storage is compromised. In other words, the accessibility of the record is dependent only on the non-rewritability of the WORM storage.

- The index must support incremental growth and be able to scale to extremely large collections of records to support the rapidly growing volume of records.

- Records must be quickly accessible through the index when events such as investigation or litigation occurs. The index must not, for example, degenerate into requiring a linear scan.

- The space overhead of the index must be acceptable. Although storage has become relatively inexpensive with the rapid improvement in disk areal density, with the large volume of records typical today, storage efficiency is still a major consideration. This is the case especially because there is a tendency in the short term to view some of the record keeping largely as an overhead needed just for satisfying the current intense regulatory scrutiny.

- The index should support selective disposition of index entries to ensure that expired records cannot be recovered or reconstituted from the index entries, even with the use of data forensics.

Note that a fossilized index must be used for any trusted means of finding and accessing a record. Examples of such include the file system directory which allows records to be located by the file name, the database index which enables records to be retrieved based on the value of some specified field or combination of fields, and the full-text index which allows records containing some particular words or phrases to be found.

3. INSIGHTS FROM PREVIOUS WORK

There is a lot of previous work on indexing techniques, including several focused on effective indexing methods for WORM storage. For the most part, these indexing techniques for WORM storage were motivated by the desire to store data on optical disks, which at that time had an advantage over magnetic disks in terms of cost and storage capacity. Thus, they were designed mainly for storage and operational efficiency, and not for trustworthy record keeping.

Index in Rewritable Storage: In this approach, the records are stored in WORM storage while the index structures are stored in rewritable storage (*e.g.*, [24]. Clearly, regardless of the type of index used, an adversary can easily modify the index to cause records to be effectively hidden or altered.

Copy on Rebalance: A balanced tree can be maintained in WORM storage simply by making a new copy of the tree each time an insertion occurs. The time and space overhead of this approach is, however, prohibitive. A straightforward optimization used to realize *persistent search trees* is to copy only the nodes that are involved in rebalancing the tree [13, 18, 19, 25]. Because any node that contains a pointer to a node that is copied must itself be copied, copying a node requires copying the entire path to the node from the root of the tree. Thus this method is called "path copying". As shown in Figure 2(a), the effect of this method is to create a set of search trees having different roots but sharing common subtrees. If an index entry is omitted or counterfeited during path copy, the corresponding record could be effectively deleted or modified. More importantly, an adversary could modify records at will by exploiting this provision for creating new paths to any node. Note that even if all of the previous versions of the tree are preserved in WORM storage, probing each in an effort to guard against tampering is impractical.

Grow from Leaves Up: One way to avoid having to rebalance the tree is to grow the tree from the leaves up to the root. For example, in *write-once B-tree* [7] (Figure 2(b)), a node is split into two new nodes when it overflows. Two pointers are added to the end of its parent node, superseding the earlier pointer to the old node. If the parent node overflows, it is split as well and only the most recent

pointers are copied. When the root node splits, a new root is created. Such a design allows an adversary to effectively modify any record he wishes to by creating a new version of the appropriate nodes. Again, even if every alteration of the nodes is preserved in WORM storage, it is infeasible to look up each version of the tree to defend against tampering. The same issue applies to the *multi-version B-tree* [2] and the *append-only trie* [17](Figure 2(c)).

Relocate to Grow: In *dynamic hashing* [6, 9] and its extensions (*e.g.,* [3]), when the number of records in a hash table exceeds a high watermark, a new hash table with a larger size is allocated and all the records are rehashed and moved into the new table. In this way, dynamic data growth can be supported with good performance. The ability to relocate records, however, clearly provides an opportunity for an adversary to alter the records. The trie proposed in [1], which stores records in the leaf nodes and moves records to new leaf nodes as the trie grows, suffers similar exposure. Dynamic hashing can, in principle, be adapted to defeat logical modification of records by, for example, storing all versions of the hash table in WORM storage and probing each of the versions on a lookup. Later in Section 4.2, we will consider an optimized adaptation that prevents any logical modification of records and that does not require duplicating entries and probing all the hash tables.

4. GHT: GENERALIZED HASH TREE

Although the previous indexing methods for WORM storage fall short of what is required to achieve a fossilized index, they provide valuable insights into how to design one. First, any approach that requires the rebalancing of a tree is vulnerable to compromise because it inevitably allows an adversary to create new paths to records. Second, trees that grow from the leaves up to the root are similarly exposed because an adversary could modify records at will by exploiting the provision for creating new versions of the tree nodes. Third, any method that permits index entries to be relocated is inherently not trustworthy because it opens the door for an adversary to create new versions of any entry.

Therefore, a fossilized index essentially requires that we design:

- a tree that grows from the root down to the leaves without relocating committed entries and that is balanced without requiring dynamic adjustments to its structure.
- efficient dynamic hashing schemes that do not require rehashing.

In this paper, we describe a fossilized index method based on a novel data structure called the *Generalized Hash Tree* (*GHT*). The GHT is effectively a tree that grows from the root down to the leaves and that is balanced without requiring any dynamic rebalancing. Attempts to insert or lookup a record are made beginning at the root node of the tree and if unsuccessful, the process is repeated at one or more of its children subtrees. When a record cannot be inserted into any of the existing nodes, a new node is created and added to the tree as a leaf. At each level, the possible locations for inserting the record are determined by a hash of the record key. Consequently, the possible locations of a record in the tree are fixed and determined solely by that record. Moreover, inserted records are never rehashed or relocated. We consider the tree "generalized" because it represents a family of hash trees. By using different parameters and hash functions, we achieve hash trees with different characteristics.

We first discuss the design of the GHT in Section 4.1 and describe four instances – *thin tree, fat tree, multi-level hash table* and *hash trie* – in Section 4.2. In Section 4.3, we present theoretical analysis of the GHT's performance and cost, and in Section 4.4, we consider several optimizations. We assume that the underlying storage supports reads from and writes to a random location, and ensures that any data that has been written cannot be overwritten, at least for a specified period. This assumption is in line with the current industry trend of providing WORM storage using magnetic disks (e.g. [8, 12, 16]). For improved storage efficiency, it is preferred that the storage has a small write granularity, which can be achieved, for example, by allowing data to be appended to a sector.

4.1 General Design

To make our presentation precise, we first define some terms.

1. We consider only single key retrieval in which a *record* is represented by a key and a pointer to the actual data.
2. A *bucket* is an entry in a tree node to store a record.
3. A *tree node* consists of buckets and is the basic allocation unit of a tree. The size of a tree node may vary with its level in the tree. Let $M = \{m_0, m_1, m_2, ...\}$ where m_i is the size of a tree node at level i. To reduce clutter, if the size of the tree node is constant across the levels, we denote it by m.
4. A *growth factor*, k_i, denotes that the tree may have k_i times as many buckets at level $(i+1)$ as at level i. The growth factor may vary for each level. Let $K = \{k_0, k_1, k_2, ...\}$ where k_i is the growth factor for level i. We denote the growth factor by k if it is constant across the levels.
5. Let $H = \{h_0, h_1, h_2, ...\}$ denote a set of hash functions where h_i is the hash function for level i.

A GHT is defined by the tuple $\{M, K, H\}$. When traversing the GHT, the candidate buckets, *i.e.,* the target hash table, at a given level is composed of the children of the node at which the collision occurred at the previous level. The root node is itself a hash table.

There are three criteria to consider when choosing the hash functions, H. First, they should be independent to reduce the chances that records colliding at one level of the tree will also collide at the next. Second, they should be efficient to calculate. Third, they should be insensitive to the size of the target range. In this paper, we use universal hash functions defined as follows:

$$h(x) = ((ax + b) \bmod p) \bmod r,$$

where p is a prime chosen such that all the possible keys are less than p, r is the size of the target range, $a \in \{1, 2, ..., p-1\}$, and $b \in \{0, 1, 2, ..., p-1\}$. The hash function for each level is selected randomly at the time the level is created by picking random values for a and b. Using randomly selected hash functions prevents the GHT from degenerating largely into a list on specific inputs and allows it to achieve provably good performance on average.

To insert a record, we use the procedure TREE-INSERT. Given a pointer to the root of a tree, t, and a record, x, TREE-INSERT returns SUCCESS if the insertion succeeds and FAILURE if a record with the same key already exists in the tree. Given a key, we calculate the target bucket at a given level by using the corresponding hash function. The target range of the hash function or, in other words, the size of the hash table at a given level, is determined by GetHashTableSize(). GetNode() gives the tree node which holds the bucket and GetIndex() returns the index of the bucket in that tree node. Depending on how these functions are defined, we can realize a family of GHTs. Table 1 lists some sample instantiations of these functions for the various hash trees we will describe in section 4.2.

	GetHashTableSize(i)	GetNode(i, j)	GetIndex(i, j)
Thin Tree (Hash Trie)	m (if $i = 0$); $m \times k$ (if $i \neq 0$)	j div m	j mod m
Fat Tree	$m \times k^i$	j div m	j mod m
Multi-Level Hash Table	$m_0 \times k^i$	0	j

Table 1: Tree-Specific Functions

Algorithm 1 TREE-INSERT(t,x)

```
 1: i ← 0; p ← t.root; index ← h_0(key)
 2: loop
 3:    if node p does not exist then
 4:       p ← allocate a tree node
 5:       p[index] ← x
 6:       return SUCCESS
 7:    end if
 8:    if p[index] is empty then
 9:       p[index] ← x
10:       return SUCCESS
11:    end if
12:    if p[index].key = x.key then
13:       return FAILURE
14:    end if
15:    i ← i+1 {Go to the next tree level}
16:    j ← GetNode(i, h_i(key))
17:    p ← p.child[j]
18:    index ← GetIndex(i, h_i(key))
19: end loop
```

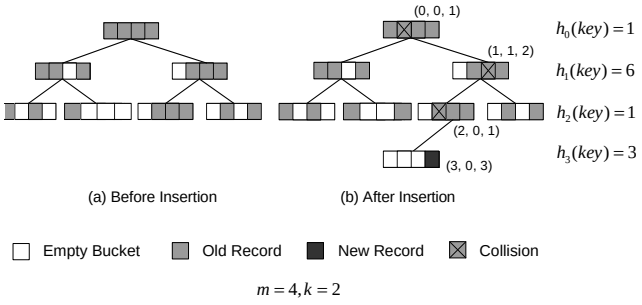


Figure 3: Example of Inserting Records into a GHT

Figure 3 illustrates the insertion of a record into a tree where $m = 4$, $k = 2$ and H is as defined earlier. Let us assume that $(h_0, h_1, h_2, h_3)(key) = (1, 6, 1, 3)$ and use (level, node, index) to denote a bucket. We first try to insert the record into the root node. However, the target bucket (0,0,1) is not empty. Next, we try again in the hash table at the next level, which is formed by the two children nodes of the root. $h_1(key) = 6$ indicates that the target bucket is (1, 1, 2) which is not empty either. Since the collision happens in node 1, its two children nodes form the next-level hash table. Unfortunately, the next attempt collides in the bucket (2,0,1). The fourth attempt succeeds since the tree node containing the target bucket does not exist. We allocate a new tree node and insert the record into the bucket (3,0,3). Intuitively, if the hash functions are uniform, the tree grows from the root down in a balanced fashion. We will prove this assertion of balanced growth later in Section 4.3.

The procedure TREE-SEARCH lists the steps to retrieve a record given its key. The procedure returns the record if it exists in the tree and NULL otherwise. When a target bucket is full, we test if its key matches the search key. If they match, then the record is found; otherwise we follow the same process as in insertion to probe the next level of the tree. When a target bucket is empty or the target tree node has not been allocated, the search fails.

Algorithm 2 TREE-SEARCH(t, key)

```
 1: i ← 0; p ← t.root; index ← h_0(key)
 2: loop
 3:    if node p does not exist then
 4:       return NULL
 5:    end if
 6:    if p[index] is empty then
 7:       return NULL
 8:    end if
 9:    if p[index].key = key then
10:       return p[index]
11:    end if
12:    i ← i+1 {Go to the next tree level}
13:    j ← GetNode(i, h_i(key))
14:    p ← p.child[j]
15:    index ← GetIndex(i, h_i(key))
16: end loop
```

We do not consider deletion here but will discuss disposition of index entries in Section 5.

4.2 Case Studies

The GHT is actually a family of hash trees. The members differ in the way the tree is organized. Each of them has its own strengths and weaknesses. In this section, we discuss four representative cases: thin tree, fat tree, multi-level hash table and hash trie. Their tree-specific functions are shown in Table 1.

Thin Tree: Thin trees are standard trees in which each node has a fixed size, a fixed number of children nodes, and at most one parent node. Figure 4(a) depicts a thin tree with $m = 4$ and $k = 2$. Earlier in Section 4.1, we used such a thin tree to illustrate the record insertion operation. Thin trees can be implemented with each node containing m buckets and k pointers to its children nodes.

If the hash functions are uniform and independent, a new record is equally likely to follow any path from the root to a leaf node. Thus, thin trees grow from the root down in a balanced fashion. The expected depth of the tree is $O(log(n))$ where n is the number of records. The most attractive property of thin trees is its linearly bounded space cost. Since a node is allocated only when a record is about to be inserted into it, each node contains at least one record. The space cost is thus $O(m \times n)$ in the worst case.

Fat Tree: The fat tree is a hash tree in which each node has multiple parents. The fatness indicates how many parents each node can have. Figure 4(b) shows a fully fat tree where all the nodes at the level above a node are its parents. For simplicity, we discuss only fully fat trees and refer to them simply as fat trees. In Figure 4(b), $m = 4$ and $k = 2$, as is the case in the thin tree example. The

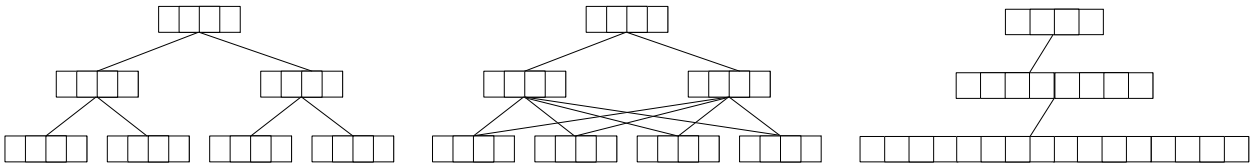


Figure 4: A Family of GHTs

hash functions are, however, different because the hash table size is $m \times k^i$ for each level i. In other words, when a collision occurs, the record can be inserted into any node at the next level.

If the hash functions are uniform and independent, fat trees behave similar to thin trees and, thus, the expected tree depth is also $O(log(n))$. However, in reality, it is difficult to ensure that the hash functions are exactly uniform. One advantage of fat trees is that they can tolerate non-ideal hash functions better than thin trees because they have many more potential target buckets at each level.

Another advantage of fat trees is that the hashing at different levels is actually independent. In thin trees, the target hash table at a given level is dependent on the node at which the collision occurred at the previous level. In contrast, all the nodes at the same level in a fat tree form the target hash table. Therefore, if we locate each level of a fat tree in a different disk, we can access these levels in parallel using their corresponding hash functions.

In implementing fat trees, since the number of children node increases exponentially with the level in the tree, it is expensive in space to maintain the children pointers for each node. As the children of a node are the same as those of its peers, we maintain an extra array for each level to track whether a tree node is allocated and if so, its location.

Multi-Level Hash Table: If we double the size of the tree nodes at each level, and still set the growth factor, k, to be 2, we end up with the multi-level hash table shown in Figure 4(c). In this case, $m_i = m_0 \times k^i$ where m_0 is the size of the root node. For simplicity, however, we still use $m = 4$ to denote the structure.

The hash functions are the same as those of the corresponding fat trees. Therefore, the multi-level hash table has the same tree depth as the corresponding fat tree for a given insertion sequence. Access to the multi-level hash table can also be parallelized.

The multi-level hash table has the advantage of simplicity. However, a collision in the leaf may lead to the allocation of a large chunk of space – k times as large as the current leaf. As a result, the space usage of the multi-level hash table grows much faster than that of the fat tree, as we shall see in Section 6.

Hash Trie: If we hash the key of a record and store the record in a trie based on this hash value, we obtain the hash trie. For example, suppose that the size of a trie node is 256 buckets, which means that the fan-out is 256. We use 8 bits of the hash value as an index to the target bucket at each level. If the target bucket is empty, the record is inserted; otherwise, we proceed to the subtrie pointed to by the index and use the next 8 bits of the hash value as a new index. To reduce the chances of collisions, a cryptographic hash function such as SHA-1 [14] can be used.

In essence, a hash trie is a special case of the thin tree where m and k are equal and a power of 2. The sample trie above is a thin tree with $m = k = 256$. The corresponding hash functions are as follows: at the first level, use the first 8 bits of the hashed record key as the hash value; at the second level, use bits 0-15 as the hash value; at the third level, use bits 8-23; and so on.

4.3 Theoretical Analysis

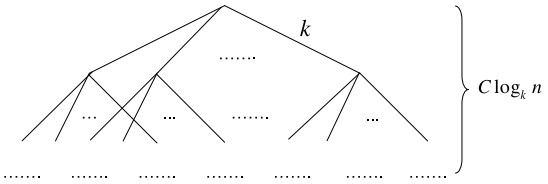


Figure 5: A Full Thin Tree with a Depth of $Clog_k n$

Theorem I: The expected tree depth of a GHT is $O(log_k n)$.

Proof: First, we consider a thin tree where $m = 1$. We construct a full thin tree with a depth of $Clog_k n$ where $C \geq 3$ and n is the number of records. Figure 5 shows such a tree. We use this tree to establish a bound on the probability that an initially empty thin tree will become deeper than $Clog_k n$ after n insertions. We then utilize this bound to derive the expected depth of a thin tree after n insertions.

Let $P(i, j)$ be the probability that the insertion of the i^{th} record ($i \in [1, n]$) results in a tree deeper than $Clog_k n$ via a particular leaf node j. Assuming uniform and independent hashing,

$$P(i,j) \leq (\frac{1}{k})^{Clog_k n} = \frac{1}{n^C} \quad (1)$$

Since a tree grows deeper only when a collision occurs in the leaf nodes and there are at most n leaf nodes that are filled with records, $P(i)$, the probability that the insertion of the i^{th} record results in a tree deeper than $Clog_k n$, satisfies the following inequality:

$$P(i) \leq n \times \frac{1}{n^C} = \frac{1}{n^{C-1}} \quad (2)$$

Therefore, the probability that the tree is not deeper than $Clog_k n$ after n insertions is

$$P(d \leq Clog_k n) \geq (1 - \frac{1}{n^{C-1}})^n \quad (3)$$

By the Bernoulli Inequality, $(1-x)^n \geq 1 - nx$ for $x < 1$,

$$P(d \leq Clog_k n) \geq 1 - n \times \frac{1}{n^{C-1}} = 1 - \frac{1}{n^{C-2}} \quad (4)$$

It follows that the probability that the tree becomes deeper than $Clog_k n$ after n insertions is

$$P(d > Clog_k n) \leq \frac{1}{n^{C-2}} \quad (5)$$

Using this bound, we prove that the expected tree depth is

$$E(d) = \sum_{d=1}^{n} d \times P(d) \leq Clog_k n \times 1 + n \times \frac{1}{n^{C-2}} = \theta(log_k n) \quad (6)$$

For $m > 1$, inequalities 1-4 still hold by considering buckets in leaf nodes rather than leaf nodes. Therefore, the expected tree depth of a thin tree is $O(log_k n)$.

For a fat tree, the following relationship holds for $P(i, j)$, the probability that the insertion of the i^{th} record leads to a tree deeper than $Clog_k n$ via a particular leaf node j:

$$P(i, j) \leq \frac{1}{k^{Clog_k n}} = \frac{1}{n^C} \quad (7)$$

This is because there are n^C leaf nodes into which a record is equally likely to be hashed. Other inequalities remain as above. Therefore, the expected tree depth of a fat tree is $O(log_k n)$.

Since a multi-level hash table has the same depth as its corresponding fat tree, its expected tree depth is also $O(log_k n)$. □

Theorem II: The expected space cost of a thin tree is $\theta(n)$.

Proof: A thin tree only allocates a tree node when a record is about to be inserted into it. In other words, an allocated tree node contains at least one record. Therefore, s, the space cost of a thin tree, has the following property:

$$(m + k)\lceil\frac{n}{m}\rceil \leq s \leq (m + k) \times n, \quad (8)$$

where for simplicity, we assume that the size of both a bucket and a pointer is 1 unit.

It follows that the expected space cost of a thin tree is $\theta(n)$. □

Theorem III: With high probability, the space cost of a fat tree and a multi-level hash table is $O(n^3)$.

Proof: When a tree grows to a depth of $Clog_k n$, a fat tree allocates a pointer array of size $k^{Clog_k n} = n^C$ while a multi-level hash table allocates a tree node of size $m \times n^C$. Inequality 4 indicates that

$$P(d \leq 3log_k n) \geq 1 - \frac{1}{n} \quad (9)$$

Therefore, with high probability, the space cost of a fat tree and a multi-level hash table is $O(n^3)$. □

In the worst case, each record insertion results in a hash collision, causing the tree to grow one level deeper. Thus, all the trees have the same worst-case tree depth, namely, $\text{WST}(d) = n$. The worst-case space costs are, however, different. The thin tree, in particular, has a space cost that is linearly bounded.

Thin Tree:	$\text{WST}(s) = (m + k) \times n = O(mn)$
Fat Tree:	$\text{WST}(s) = m \times n + \frac{k^n - 1}{k - 1} = O(k^n)$
Multi-Level HT:	$\text{WST}(s) = m \times \frac{k^n - 1}{k - 1} = O(mk^n)$

4.4 Optimizations

In this section, we discuss several optimization techniques to further reduce the time and space costs.

Linear Probing: A potential shortcoming of hash trees is that, in the worst case, each tree node might have only a few buckets that are occupied, resulting in low space utilization. To improve the situation for thin trees and fat trees, we apply linear probing within a tree node. When a collision occurs in a tree node, we linearly search the other buckets within the node before probing the next level in the tree. Specifically, at each level i, we use the following series of hash functions: $h_i(j, key) = (h_i(key) + j) \ mod \ m$, where $j = 1, 2, ..., m - 1$. For the multi-level hash table, we introduce the concept of "virtual node". The single tree node at each level is divided into fixed-size virtual nodes and we probe linearly within the virtual nodes.

As we shall see later in Section 6, with linear probing, empty buckets appear only in the leaf nodes and space utilization is, therefore, improved significantly. Meanwhile, average access time remains logarithmic in the number of records. In practice, linear probing incurs little overhead because it uses large block I/O, which effectively leverages the high sequential transfer rate of modern storage devices. Other hash table optimizations such as double hashing can also be applied to the hash tree.

Large First-Level Hash Table: In hash trees, the first level hash table is, by default, the root node. Unless the tree node is huge, this makes the tree unnecessarily deep, which reduces performance. One optimization is to decouple the first-level hash table from the root node to allow the first-level hash table to comprise a number of tree nodes. In other words, we effectively remove the first few levels of the basic hash tree.

In section 6, we evaluate the effect of increasing the first-level hash table size. Ideally, the first-level hash table should be large enough to allow efficient insertion and retrieval but small enough to avoid over-provisioning.

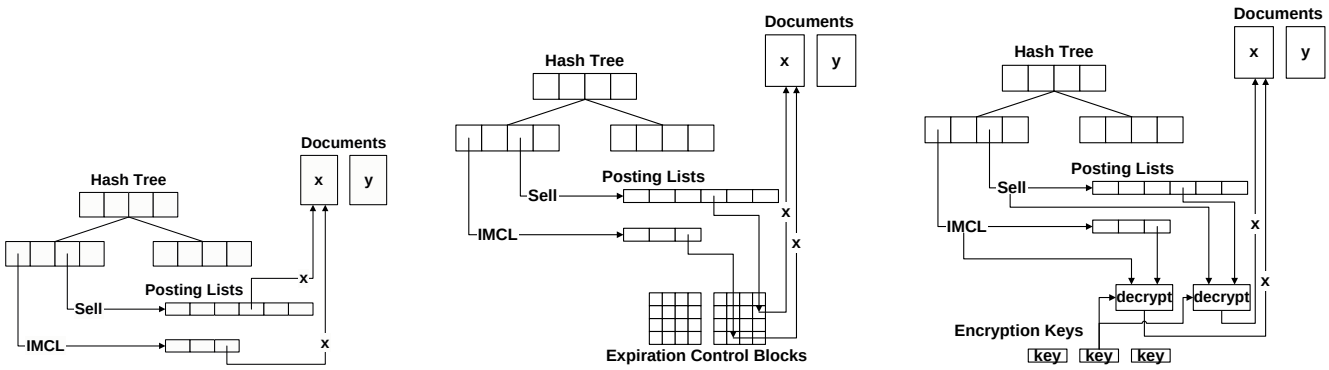
Parallel Access: As discussed in Section 4.1, for thin trees, including hash tries, the target hash table at a given level depends on the node at which the collision occurred at the previous level. On the other hand, for fat trees and multi-level hash tables, the hashing at each level is independent. Therefore, if we have multiple disks and we locate each level of a fat tree or multi-level hash table on a different disk, we can access all the levels in parallel.

5. DISPOSITION OF INDEX ENTRIES

5.1 Problem Statement

As discussed earlier, it is crucial for an organization to properly dispose of records that are no longer useful to the organization and have passed any mandated retention periods. Otherwise, the records are subject to discovery, and typically at great expense to the organization [22]. Proper disposition of records include deleting the records. In some cases, the records have to be shredded such that they cannot be recovered or discovered even with the use of data forensics. Such disposition can be achieved by physical destruction of the storage. For disk-based WORM storage (*e.g.,* [8, 12, 16]), an alternative is to overwrite the data multiple times with specific patterns so as to completely erase remnant magnetic effects which could otherwise enable the data to be recovered [10].

However, as records are indexed on multiple fields to facilitate search and retrieval, the records can be reconstructed from the corresponding index entries even after the records expire and are disposed of. For example, Figure 6(a) shows an inverted index [20] consisting of a *dictionary* (*e.g.,* a GHT) of words occurring in a set of documents and *posting lists*, which contain information about the occurrence of the words such as the record locator (*e.g.,* ID of documents containing the words, pointer to the documents) and, in some cases, positional information (*e.g.,* offset within the documents). By performing a join on the record locator in the posting lists, we can discover that the document X contains both the words "Sell" and "IMCL". If this is a full-text index, the whole document can be reconstructed from the index even after the document has been disposed of. Thus, it is imperative to dispose of all the index entries associated with a record when the record is disposed of.



(a) Inverted Index (b) With Indirect Record Locators (c) With Encrypted Record Locators

Figure 6: Illustration of Logical Disposition

An arbitrary index entry, however, cannot typically be disposed of together with its record because the unit of disposition tends to be much larger than the unit of writing. Data stored on a WORM optical disc, for example, is regularly disposed of by physically destroying the entire disc. In disk-based WORM storage, some metadata such as the expiration date must be kept for each unit of disposition. To reduce the amount of metadata required, the disposition unit is, therefore, large. In general, an index entry is likely to be much smaller than a disposition unit, meaning that an index entry cannot be disposed of until all the other entries within the disposition unit have expired. Because index entries are organized based on the index key and are not clustered on the expiration date, there tends to be a long period during which information about a record that has already been disposed of can potentially be discovered.

5.2 Scrambled Keys

A straightforward way to prevent a record from being reconstructed from its index entries is to scramble the keys before inserting them into the index. Specifically, to insert a record into the index, we hash the index key of the record and use the hash value as the key for inserting the record into the index. With a cryptographic hash function such as SHA-1 [14], it is computationally infeasible to recreate a key from its hash value. Thus, we can simply dispose of an expired record and leave behind its index entries.

This approach is, however, vulnerable to existence testing. For instance, if one is looking for a document containing a few specific terms, one can hash the terms and search the index entries for those specific hash values. Another weakness is that this method does not work when the key space is small. In this case, one can effectively create a reverse map for the hash values of all the keys without having to break the one-way hash. For example, it is not difficult to find the SHA-1 hash value for all the words in the English language, which total only about 616,500 words [27].

5.3 Logical Disposition

Therefore, we introduce a more general technique called *logical disposition* to allow the index entries associated with expired records to be effectively disposed of. In this method, records are classified into disposition groups by, for example, their expiration date. For each disposition group, we generate encoding and decoding functions. These functions are stored by themselves in one or more disposition units so that they can be individually disposed of when the records in the corresponding disposition group expire. The encoding function is used to transform the record locators, which are then stored in the index. The decoding function decodes the stored record locators so that they can be used to locate the record. When a group of records is disposed of, the functions of that group are also disposed of.

In the *indirect pointer* method, the encoding and decoding functions are realized through *expiration control blocks* (ECBs) which introduce a level of indirection in the record locators. Instead of storing the record locator directly in the index, we store an indirect pointer to an entry in the ECB where the record locator is actually stored. Each ECB serves as an indirect table for records that are in the same disposition group. In the above example, the record locators corresponding to "Sell" and "IMCL" in document X would each point to a different entry in the same ECB and both of these entries would point to document X (Figure 6(b)). When the group of records which include document X is disposed of, the associated ECB is also disposed of. After the ECB is disposed of, the only information that could be uncovered is that the words "Sell" and "IMCL" belong to documents that were disposed of together. As added security, we can introduce spurious records so that each disposition group has a minimum number of records. Note that we only need to store the metadata for the spurious records; the records themselves do not have to be instantiated. The drawback of this method is the space needed for the ECBs, which essentially duplicate the record locators.

In the second method, we use encryption and decryption (*e.g.*, AES [15]) to realize the encoding and decoding functions. For each disposition group, we generate and maintain a secret key. Basically, each record locator is first encrypted using the secret key corresponding to its disposition group before it is stored in the index. To avoid having the same cyphertext for each occurrence of a particular record locator, which would enable an adversary to reconstruct the record by performing a join on the cyphertext, we encrypt the combination of the index key and the record locator. When a group of records is disposed of, the associated secret key is also disposed of. Figure 6(c) shows such an example. Note that after the secret key is disposed of, an adversary can still determine that the words "Sell" and "IMCL" belong to documents in the same disposition group since the index entries for these words identify the same secret key. As in the indirect pointer method, spurious records can be introduced so that each disposition group has a minimum number of records. Furthermore, for the encryption method, it is feasible to conceal membership in a disposition group by duplicating the

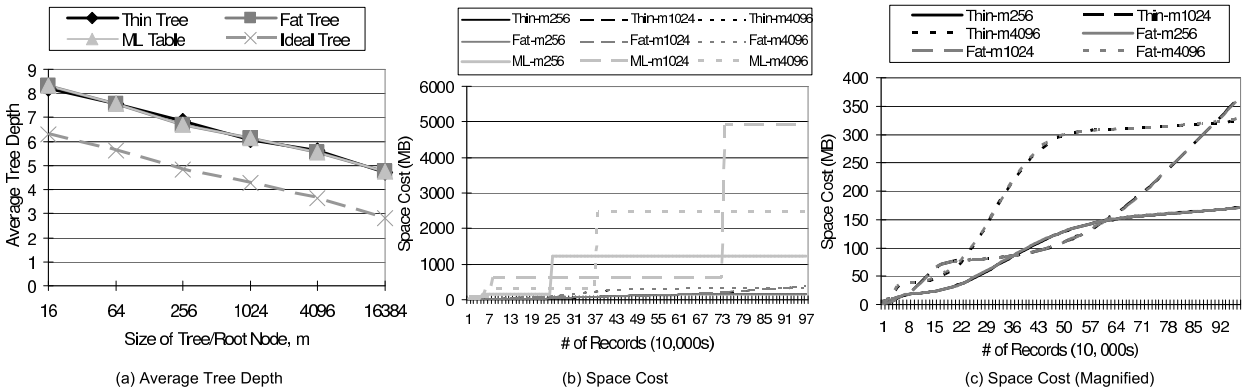


Figure 7: Sensitivity to m, the Size of a Tree/Root Node (k=8)

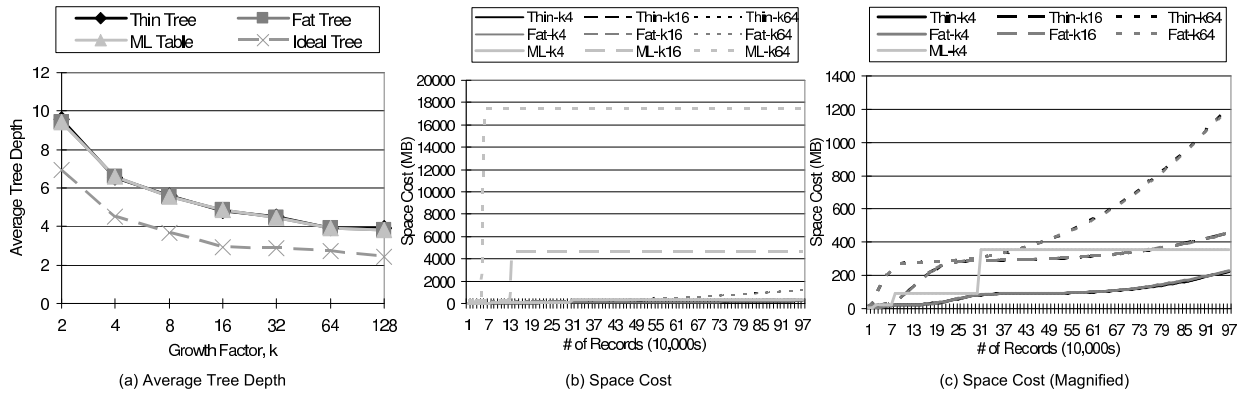


Figure 8: Sensitivity to k, the Growth Factor (m=4096)

secret keys and randomly selecting which copy of a key to identify in an index entry.

6. SIMULATION RESULTS

6.1 Methodology

In order to understand the actual cost of a fossilized index, we use simulation with real system data to compare the GHT with several traditional indexing methods including the *k-way tree*, *rewritable B-tree* and *ideal tree*. For the GHT, we present results for the thin tree, fat tree and multi-level hash table. Since the hash trie is a special case of the thin tree, we do not explicitly show its results. In most cases, it behaves exactly like the generic thin tree.

The k-way tree is a tree in which each node contains $k-1$ keys and k pointers to its children. The pointers and keys are arranged in an alternate fashion such that the subtrees to the left of a key contain only keys that are smaller while the subtrees to the right of a key contain only keys that are larger. The B-tree is a balanced variant of the k-way tree. All the leaves in a B-tree are on the same level and all the nodes except for the root and the leaves have at least $k/2$ children. The rewritable B-tree splits a node that overflows in place and is, hence, much more space-efficient than its write-once counterpart. The ideal tree is a k-way tree that is constructed offline with knowledge of all the keys so that it has perfect balance and space utilization. Its results represent the lower bound.

The two metrics that we focus on are the average tree depth and space cost. Since storage has become relatively inexpensive, the space cost has become less of an issue. Nevertheless, with the large volume of records typical today, it is still important to ensure that the space overhead is not excessively high. The average tree depth is a critical determinant of insertion and access performance. It is defined as $1/n \sum_{i=1}^{n} depth(i)$, where n is the number of records and $depth(x)$ denotes the depth of the x-th key in the tree.

We use two sets of real system data. Zeus2 is a snapshot of a file system used by a group of researchers to do compilation, text editing and software development. Mainstore is a data set created by web crawling and includes the URLs of all of the internal websites of a large corporation. In total, Zeus2 contains 975,861 files while Mainstore includes 5,130,881 URLs. Between them, the data sets are likely to be illustrative of a variety of applications, including those targeted by the GHT. For each file and URL, we create a unique 64-bit key by using SHA-1. In this paper, we present only the results for Zeus2 since the results for Mainstore are virtually identical. The simulations were performed on an IBM xSeries model 325 server with a 64-bit AMD Opteron processor. We use eight bytes for the length of both a key and a pointer.

6.2 Sensitivity to Size of Tree Node

In Figure 7, we present how the average tree depth and space cost of the GHTs are affected as m, the size of the tree node (root node for the multi-level hash table), is increased. Since a larger m means that there are more buckets at each level, the space cost rises with m while the average tree depth decreases. Observe from Figure 7(a) that, largely independent of changes in m, the average tree depth of the GHTs is consistently within two of that of the ideal tree, indicating that the GHTs are very well-balanced.

From Figure 7(b), as m increases, the space cost for the multi-level hash table increases faster as records are added, but remains

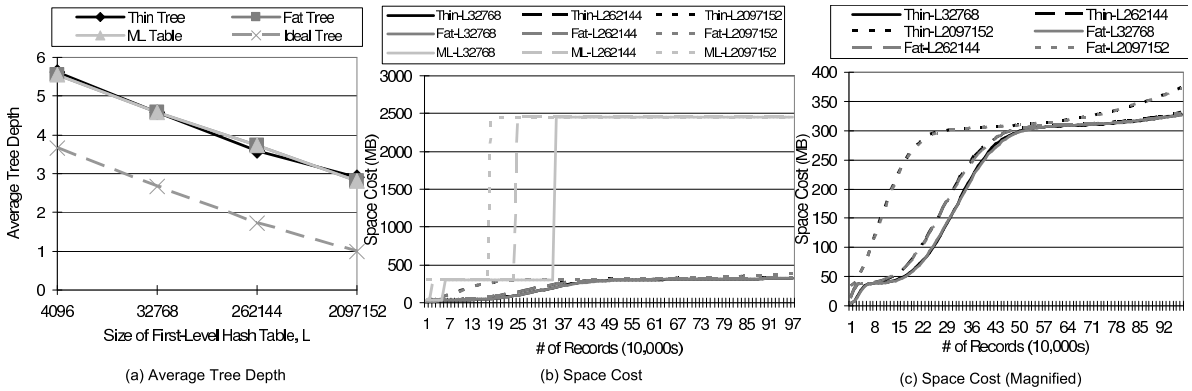


Figure 9: Sensitivity to First-Level Hash Table Size, L (k=8, m=4096)

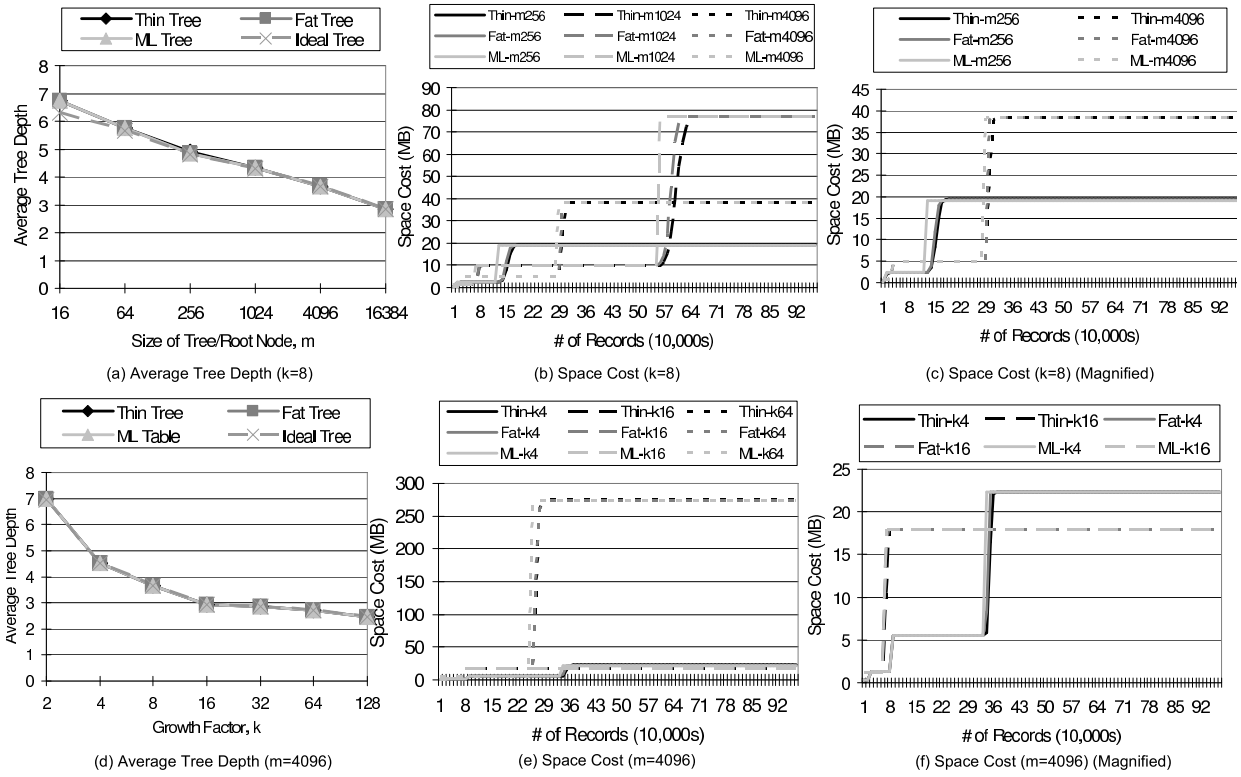


Figure 10: Performance with Linear Probing

in a stable state for a longer period of time. The thin and fat trees exhibit similar behavior, but their increase in space cost happens more smoothly (Figure 7(c)). The plots for the fat and thin trees are almost overlapped because, in practice, the pointer array of the fat tree incurs little overhead due to the small tree depth. In general, a large m is preferred since it decreases the average tree depth without increasing the space cost by very much.

6.3 Sensitivity to Growth Factor

Figure 8 presents the sensitivity of the GHTs to the growth factor, k. As expected, the space cost increases with the growth factor while the average tree depth decreases. Notice that compared to the ideal tree, the GHTs have, on average, only about two more levels. If space is not a very big constraint, as is likely to be the case, a large k is preferred. Otherwise, a small k would lead to reasonable access performance with small space cost. In the next section, we present the results of linear probing within a tree node (virtual node for multi-level hash table) and show that it can significantly improve both space efficiency and access performance.

6.4 Sensitivity to First-Level Hash Table Size

A large first-level hash table effectively means that the first few upper tree levels are removed. As shown in Figure 9, we can use a large first-level hash table to significantly reduce the average tree depth and access time without adversely affecting the space cost.

6.5 Effect of Linear Probing

Figures 10(a) and 10(d) show that for GHTs with linear probing within a node, the average tree depth is almost the same as that of the ideal tree, implying that the GHTs are perfectly balanced. Fur-

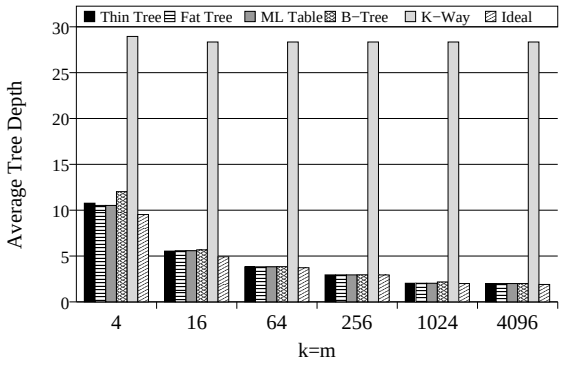


Figure 11: Comparison of Average Tree Depth with Linear Probing for the GHTs

thermore, the space cost is significantly reduced with linear probing, making hash trees very practical. For example, comparing Figures 7(b) and 10(b), we find that for $k = 8$, the space used is lessened by 60 times with linear probing. Moreover, linear probing is well suited to exploit the high sequential transfer rate of modern storage devices by bringing large chucks of data into memory at a time.

Figures 10(b) and 10(e) also indicate that, with linear probing, the thin tree, fat tree and multi-level hash table grow the next level at almost the same time. This is because, with linear probing, a child node is generated only when its parent node becomes full. Therefore, the tree grows in a level-by-level fashion. If the hash functions are uniform, the records tend to be well distributed so that the thin and fat trees end up allocating all the nodes in a level within a short period of time. Notice further that the effect of the parameters are qualitatively similar with or without linear probing, but the scales are different: with larger m, the space cost increases faster as records are inserted, but remains in a stable state for longer; trees grow deeper faster when k is large.

6.6 Comparison Results

In this section, we compare the GHTs with space optimization (linear probing within a node) to the rewritable B-tree, k-way tree and ideal tree. In order to establish a common basis for comparison, we set the size of the tree node, m, to be equal to the growth factor, k, which is a non-optimal configuration for the GHTs. The results for the average tree depth are summarized in Figure 11. Note that the k-way tree is able to prevent logical modification of records, but it has by far the worst tree depth. Since the k-way tree does not rebalance itself, its depth is determined solely by the insertion sequence of records. The B-tree, on the other hand, performs well but is vulnerable to logical tampering of records. The GHTs are able to satisfy the requirements of a fossilized index and yet perform equal or up to 12.5% better than the B-tree. In all cases, the GHTs are very close to the ideal tree in terms of the average tree depth. This indicates that hashing works very well in practice. As a result, the GHTs are well-balanced, implying good search and access performance.

Figure 12 presents the comparison of space cost for $k = m = 4, 16$ and 64. We zoom in on some of the plots in order to provide a more detailed picture of the behavior. For $k = m = 4$, the space cost of the multi-level hash table grows much faster than that of the other trees. This is because it takes only one collision in the leaf node to trigger the allocation of a large chunk of space - four times larger than the current leaf node. In contrast, the space cost of the thin and fat trees with linear probing is comparable to that of the B-tree. In fact, the thin tree has an advantage in space efficiency over the B-tree that exceeds 10%. The small jumps in the curve of the fat tree are caused by the allocation of space for the pointer array whenever the tree grows by a level. As k and m are increased, the thin and fat trees become less space-efficient than the B-tree. This is because with uniform hashing, the records at the bottom level tend to be well distributed among all the leaf nodes so that many leaf nodes are allocated with each containing only a few records. A small k with a large first-level hash table is generally preferred for the GHTs. When thus configured, the GHTs incur reasonable space cost and offers very good performance, thereby showing that a fossilized index can be achieved in a practical manner.

The GHT has been prototyped and implemented to handle the file directory of a large SAN (storage area network) file system. The performance achieved is in line with our expectations.

7. CONCLUSION

Having trustworthy records, *i.e.*, records capable of providing irrefutable proof and accurate details of events that have occurred, is critical to an organization. In this paper, we contend that the current limited focus of storing electronic records in WORM storage is far from adequate to ensure that they are trustworthy. In particular, we demonstrate that the records stored in WORM storage can be logically modified if the index through which the records are accessed can be suitably manipulated. With the increasingly large volume of records and the ever more stringent response time to enquiries, the index poses a large and looming risk. The rising threat of adverse discovery further makes it imperative to dispose of records that are no longer needed. However, unless the corresponding index entries are also properly disposed of, the disposed of records can in effect be reconstructed from the index entries.

Therefore, we identify the key requirements for a *fossilized index* that prevents the records from being logically modified. We also analyze how current indexing methods, including those designed for WORM storage, fall short of these requirements. Based on the insights we gained, we propose the *Generalized Hash Tree (GHT)* as an effective approach to achieving a fossilized index. The GHT represents a family of hash trees, including the thin tree, fat tree, multi-level hash table and hash trie. We further present a novel technique called *logical disposition* that enables the effective disposition of index entries corresponding to records that have been disposed of so as to prevent the records from being reconstructed.

Using both theoretical analysis and simulations with real system data, we show that the GHT offers high performance at a low storage cost. Specifically, the expected tree depth of the GHT is logarithmic in the number of records indexed and, in practice, the GHT is very well-balanced. Furthermore, with optimizations such as linear probing, its space cost is comparable with that of the B-tree. In short, the GHT enables fossilized index with performance and cost that are comparable to traditional indexing techniques, indicating that trustworthy electronic records can be effectively achieved.

8. REFERENCES

[1] P. Bagwell. Ideal Hash Trees. Technical report, Programming Methods Laboratory, Institute of Core Computing Science, School of Computer and Communication Sciences, Swiss Institute of Technology Lausanne, 2001.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal: The International Journal on Very Large Data Bases*, 5:264–275, 1996.

[3] A. Z. Broder and A. R. Karlin. Multilevel Adaptive Hashing. In *1st ACM-SIAM Symposium on Discrete Algorithms*, 1990.

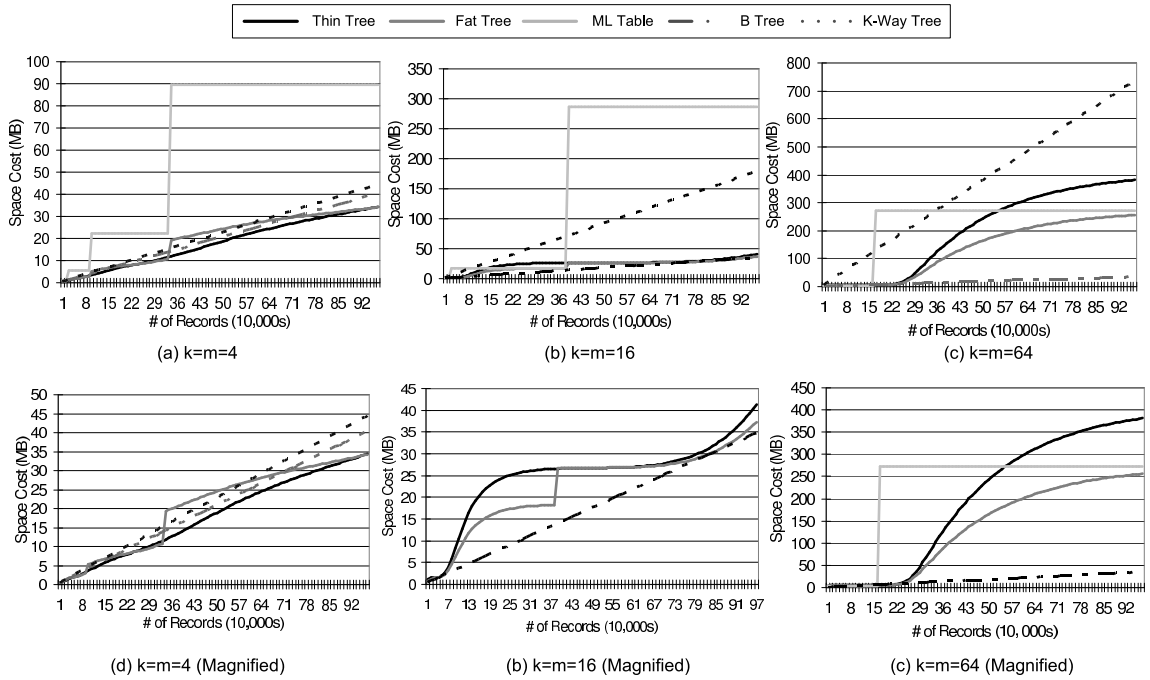


Figure 12: Comparison of Space Cost with Linear Probing for the GHTs

[4] Cohasset Associates, Inc. The role of optical storage technology. White Paper, Apr. 2003.

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

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. A. D. Heide, H. Rohnert, and R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

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

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

[9] R. J. Enbody and H. C. Du. Dynamic Hashing Schemes. *ACM Computing Survey*, 20(2), June 1988.

[10] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *6th USENIX Security Symposium*, July 1996.

[11] W. W. Hsu and S. Ong. Fossilization: A process for establishing truly trustworthy records. Research Report RJ 10331, IBM Almaden Research Center, San Jose, CA, 2004. Available at http://www.research.ibm.com/resources/paper_search.shtml.

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

[13] E. W. Myers. Efficient Applicative Data Types. In *11th ACM Symposium on Principles of Programming Languages*, 1984.

[14] National Institute of Standards and Technology. FIPS 180-1. Secure Hash Standard. US Department of Commerce, 1995.

[15] National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES), 2001.

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

[17] P. Rathmann. Dynamic Data Structures on Optical Disks. In *1st International Conference on Data Engineering*, 1984.

[18] T. Reps, T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language-based Editors. *ACM Transactions on Programming Language Systems*, 5:449–477, 1983.

[19] N. Sarnak and R. E. Tarjan. Planar Point Location Using Persistent Search Tree. *Communications of the ACM*, 29(7), July 1986.

[20] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *25th ACM Conference on Research and Development in Information Retrieval*, 2002.

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

[22] Socha Consulting LLC. The 2004 Socha-Gelbmann Electronic Discovery Survey, 2004.

[23] Sony Corp. AIT-2/AIT-3 WORM Drives & Libraries, 2003. Available at http://www.storagebysony.com/products/prod_hilite4.asp.

[24] M. Stonebraker. The Design of the POSTGRES Storage System. In *13th VLDB Conference*, 1987.

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

[26] The Enterprise Storage Group, Inc. Compliance: The effect on information management and the storage industry, May 2003.

[27] D. Wilton. How Many Words Are There In The English Language? Wilton's Word and Phrase Origins, 2001.