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

Qingbo Zhu and Windsor W. Hsu

Presented by: Mehmud Abliz
Background

• Records needs not only be accurate and readily accessible, but also credible and irrefutable.
  
  – *Examples*: email, financial statements, medical images, purchase orders, drug development logs

• Growing proportion of records are subject to regulations that require trustworthy records.
  
  – Detect and deter misconduct, fraud
Motivation

- Write-Once-Read-Many (WORM) storage enables effective preservation of records.
- However
  - WORM does not meet the query response time requirements for large volume of records.
  - Indexing methods for WORM cannot prevent hidden or altered records.
Related Work

**Persistent Search Tree**

1. Insert 12

**Append-Only Trie**

1. Insert 2

**Write-Once B-Tree**
Trustworthy Record Keeping

• Assumptions
  – Record keeper creates **accurate** records for all of the relevant events as they occur.
  – Adversary cannot blatantly destroy the records.

• Key requirement
  – Ensure that in an enquiry, all of the relevant records can be **quickly** located and retrieved in an **untampered** form.

• Approach
  – WORM + fossilized index
Key Properties of Fossilized Index

• Once a record is committed, both the index entry of that record and the path to that entry must be immutable.
• The index must support incremental growth, and scale to extremely large record sets.
• Accessing records via index must be efficient.
• Space overhead of index must be acceptable.
• The index should support selective disposition of index entries to ensure that expired records cannot be recovered from index entries.
Generalized Hash Tree (GHT) Overview

• Generalized Hash Tree
  – is effectively a tree that grows from the root down to the leaves and that is balanced without requiring any dynamic rebalancing.
  – Insert or lookup starts from the root node, and continues to children subtrees until succeeds.
  – When a record cannot be inserted into any of the existing nodes, a new leaf node is created.
  – Possible location of inserting the record are determined by a hash of the record key, hence location is fixed and determined solely by that record.
Generalized Hash Tree (GHT)

• A **record** is *(key, pointer to the actual data)*.

• A GHT is defined by the tuple \(\{M, K, H\}\), where
  
  – \(M = \{m_0, m_1, m_2, \ldots\}\), \(m_i\) *(node size)* is the size of a tree node at level \(i\).

    • **Node** consists of buckets.

  – \(K = \{k_0, k_1, k_2, \ldots\}\), \(k_i\) *(growth factor)* denotes that the tree may have \(k_i\) times as many buckets at level \((i + 1)\) as at level \(i\).

    • **Bucket** is an entry in a tree node to store a record.

  – \(H = \{h_0, h_1, h_2, \ldots\}\), \(h_i\) is the hash function for the level \(i\).
Generalized Hash Tree (cont.)

• Hash function used
  – \( h(x) = ((ax + b) \mod p) \mod r \)
    • \( p \) is a prime larger than all possible keys
    • \( a, b \in \{1, 2, \ldots, p - 1\}; r \) is the hash table size

• Tree functions
  – \textbf{TREE-INSERT}(t, x)\): inserts record \( x \) into tree with root \( t \)
  – \textbf{GetHashTableSize}(i)\): gets size of hash table at level \( i \)
  – \textbf{GetNode}(i, j)\): gets the node which holds the bucket \( j \)
  – \textbf{GetIndex}(i, j)\): gets the index of the bucket \( j \)
TREE-INSERT and TREE-SEARCH

Algorithm 1 TREE-INSERT(t, x)
1: i ← 0; p ← t.root; index ← h₀(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, hᵢ(key))
17: p ← p.child[j]
18: index ← GetIndex(i, hᵢ(key))
19: end loop

Algorithm 2 TREE-SEARCH(t, key)
1: i ← 0; p ← t.root; index ← h₀(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, hᵢ(key))
14: p ← p.child[j]
15: index ← GetIndex(i, hᵢ(key))
16: end loop
TREE-INSERT Example

(a) Before Insertion  
(b) After Insertion

Empty Bucket  Old Record  New Record  Collision

\( m = 4, k = 2 \)
GHT Family

• Depending on how GetHashTableSize(), GetNode(), and GetIndex() are defined, different GHTs can be instantiated.

<table>
<thead>
<tr>
<th></th>
<th>GetHashTableSize(i)</th>
<th>GetNode(i, j)</th>
<th>GetIndex(i, j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin Tree (Hash Trie)</td>
<td>( m ) (if ( i = 0 )); ( m \times k ) (if ( i \neq 0 ))</td>
<td>( j \text{ div } m )</td>
<td>( j \text{ mod } m )</td>
</tr>
<tr>
<td>Fat Tree</td>
<td>( m \times k^i )</td>
<td>( j \text{ div } m )</td>
<td>( j \text{ mod } m )</td>
</tr>
<tr>
<td>Multi-Level Hash Table</td>
<td>( m_0 \times k^i )</td>
<td>0</td>
<td>( j )</td>
</tr>
</tbody>
</table>
GHT Family (cont.)

• Thin Tree
  – Grows in balanced fashion
  – Expected depth: $O(\log(n))$; space cost: $O(m \times n)$

• Fat Tree
  – Pros: less collision; independent hashing at diff levels
  – Cons: hard to ensure uniformity; expensive in space

• Multi-Level Hash Table
  – Pros: simple; access can be parallelized
  – Cons: expensive in space;

• Hash Trie
  – Special case of thin tree, where $m = k$ ($m$, $k$ are power of 2)
  – A cryptographic hash function is used to reduce collision.
Complexity of GHT

• The expected tree depth
  – $O(\log_k n)$

• The expected space cost of a thin tree
  – $\Theta(n)$

• With high probability, the space cost of a fat tree and multi-level hash table
  – $O(n^3)$
Optimizations

• Improve Space Utilization
  – When a collision occurs in a node, linearly search other buckets within the node before probing next level.

• Reduce tree depth
  – Allow the first-level hash table to comprise more nodes than just one node.

• Parallel access multiple levels
  – Suitable for fat tree and multi-level hash table
Disposition of Index Entries

• Records can be reconstructed from the corresponding index entries even after the records are disposed.

• Yet, index entries cannot be disposed together with the corresponding records in some cases.
  – An index entry cannot be disposed of until all the other entries within the disposition unit have expired.
  – Such waiting period can be pretty large
Preventing records reconstructed from index entries

• Scrambled Keys
  – Cryptographically hash record key, and use the hash output as the key to hash tree
  – Cons
    • vulnerable to existence testing
    • Does not work when key space is small

• Logical Disposition
  – Group records by expiration data into disposition groups
  – Each group has one encoding & decoding function, stored separately, and disposed when records are disposed.
  – Encode/decode the record locator (pointer to records)
Logical Disposition Methods

(b) With Indirect Record Locators

(c) With Encrypted Record Locators
Strength

• GHT is can be customized to achieve better trade-off between space efficiency and lookup or insertion efficiency, based on specific application requirements.

• Considers the disposal of index entries, which are not considered in previous indexing methods for WORM; and provides several techniques to dispose index entries.
Weaknesses

• GHT is not very well suited to be used with unstructured data. If multiple documents have the same key, the insertion fails.

• Claims GHT grows in a balanced fashion due to the proposed hash function produces uniformly distributed outputs. **BUT**, no proof to support the hash function is uniform.

• Comparing the time efficiency of GHT against traditional indexing methods **in terms of tree depth** may not provide the most accurate comparison results.