Secure Indexing/Search for Regulatory-Compliant Record Retention

There is a need for trustworthy record keeping

- Digital Information Explosion
- Corporate Misconduct
- Soaring Discovery Costs
- Spending on eDiscovery Growing at 65% CAGR
- Average F500 Company Has 125 Non-Frivolous Lawsuits at Any Given Time


What is trustworthy record keeping?

Establish solid proof of events that have occurred

Bob should get back Alice’s data

This leads to a unique threat model

Adversary could be Alice herself
Traditional schemes do not work

Cannot rely on Alice’s signature

WORM storage helps address the problem

Adversary cannot delete Alice’s record
WORM storage helps address the problem

Build on top of conventional rewritable magnetic disk, with write-once semantics enforced through software, with file modification and premature deletion operations disallowed.

Adversary cannot delete Alice’s record

Index required due to high volume of records
In effect, records can be hidden/altered by modifying the index

The index must also be secured (fossilized)

Btree for increasing sequence can be created on WORM
B+tree index is insecure, even on WORM

Path to an element depends on elements inserted later – Adversary can attack it

Is this a real threat?

- Would someone want to delete a record after a day its created?
- Intrusion detection logging
  - Once adversary gain control, he would like to delete records of his initial attack
- Record regretted moments after creation
- Email best practice - Must be committed before its delivered
Several levels of indexing …

To find documents containing keywords “Query” and “Data” and “Base”
* Retrieve lists for Query, Data and Base, and intersect the document ids in the list

GHT: A Generalized Hash Tree Fossilized Index

- Tree grows from the root down to the leaves without relocating committed entries
- “Balanced” without requiring dynamic adjustments to its structure
- For hash-based scheme, dynamic hashing scheme that do not require rehashing
**GHT**

- Defined by \{M, K, H\}
- \(M = \{m_0, m_1, \ldots\}\), \(m_i\) is size of a tree node (number of buckets) at level \(i\)
- \(K = \{k_0, k_1, \ldots\}\), \(k_i\) is the growth factor for level \(i\)
  - A tree has \(k_i\) times as many nodes at level \((i+1)\) as at level \(i\)
- \(H = \{h_0, h_1, \ldots\}\), \(h_i\) is a hash function for level \(i\)
  - Different \(H\) values lead to different GHT variants

**Standard (Default) GHT – Thin Tree**

- Defined by \{M, K, H\}
- \(M = \{m_0, m_1, \ldots\}\), \(m_i\) is size of a tree node (number of buckets) at level \(i\)
- \(K = \{k_0, k_1, \ldots\}\), \(k_i\) is the growth factor for level \(i\)
  - A tree has \(k_i\) times as many nodes at level \((i+1)\) as at level \(i\)
- \(H = \{h_0, h_1, \ldots\}\), \(h_i\) is a hash function for level \(i\)

\[
m_0 = m_1 = \ldots = 4 \\
k_0 = k_1 = \ldots = 2
\]
Standard (Default) GHT – Thin Tree

- Defined by \( \{M, K, H\} \)
- \( M = \{m_0, m_1, \ldots\} \), \( m_i \) is size of a tree node (number of buckets) at level \( i \)
- \( K = \{k_0, k_1, \ldots\} \), \( k_i \) is the growth factor for level \( i \)
  - A tree has \( k_i \) times as many nodes at level \( (i+1) \) as at level \( i \)
- \( H = \{h_0, h_1, \ldots\} \), \( h_i \) is a hash function for level \( i \)

\[
m_0 = m_1 = \ldots = 4 \\
k_0 = k_1 = \ldots = 2 \\
h_0 = x \mod 4 \\
h_1 = x \mod 8 \\
What about \( h_2 \)? \( x \mod 16? \)
\]
### GHT Variant (Fat Tree)

- **Can tolerate non-ideal hash functions better because there are many more potential target buckets at each level.**
- **Hashing at different levels is independent.**
- **Can allocate different levels to different disks and access them in parallel.**
- **Expensive to maintain children pointers in each node – number of pointers grow exponentially.**

### GHT (Standard) Insertion

**Bucket** = (Level, Child – left or right, Entry within bucket)

<table>
<thead>
<tr>
<th>(0, 0, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1, 2)</td>
</tr>
<tr>
<td>(2, 0, 1)</td>
</tr>
</tbody>
</table>
If hash functions are uniform, tree grows top-down in a balanced fashion
GHT Search

Search for whose hash values at the various levels are shown
- Similar to insertion
- Need to deal with duplicate key values

(0, 0, 1)  

h0(key) = 1

(1, 1, 2)

h1(key) = 6

(2, 0, 1)

h2(key) = 1

(3, 0, 3)

h3(key) = 3

- Only for point queries
- Cannot support range search

Summary

- Trustworthy record keeping is important
- However, need to also ensure efficient retrieval
- Existing indexing structures may be manipulated
- GHT is a “trustworthy” index structure
  - Once record is committed, it cannot be manipulated!
Most business records are unstructured, searched by inverted index

Index must be updated as new documents arrive

- 500 keywords = 500 disk seeks
  - ~1 sec per document
Amortize cost by updating in batch

- 1 seek per keyword in batch
- Large buffer to benefit infrequent terms
  - Over 100,000 documents to achieve 2 docs/sec

Index is not updated immediately

- Prevailing practice – email must be committed before it is delivered
Can storage server cache help?

- Storage servers have huge cache
- Data committed into *cache* is effectively on disk
  - Is battery backed-up
  - Inside the WORM box, so is *trustworthy*

Caching works in blocks (One block per posting list)

- Caching does not benefit infrequent terms (number of posting lists >> number of cache blocks)
Simulation results show caching is not enough

- What if number posting lists ≤ Number of cache blocks?
- Each update will hit the cache
So, merge posting lists so that the tails blocks fit in cache (#posting lists < #cache blocks)

Only 1 random I/O per document, for 4K block size (500 keywords, 8-byte posting)

The tradeoff is longer lists to scan during lookup

- Query answered by scanning posting lists of the terms in the query

Workload lookup cost before merging:

\[
\sum_w t_w q_w
\]
The tradeoff is longer lists to scan during lookup

- Query answered by scanning posting lists of the terms in the query

Workload lookup cost before merging:

\[ \sum w t_w q_w \]

After merging into \( A = \{A_1, \ldots, A_n\} \):

\[ \sum_A \left( \sum w t_w \right) \left( \sum w q_w \right) \]
Which lists to merge?

- Choose $A=\{A_1, A_2 .. A_n\}$
  - $n =$ Cache blocks
  - Minimize $\sum (\sum t_w) \times (\sum q_w)$
- Problem is NP-complete, so need heuristics
- Heuristics (See observation in next slide)
  - Separate lists for high contributor terms
  - Merging heuristics
    - Based on $q_w t_w$
    - Random merging

A few terms contribute most of the query workload cost

![Graph showing workload cost vs term rank]

- $QF$
- $TF$

(workload cost vs term rank graph)
Summary

- To ensure acceptable performance, posting lists have to be properly managed
- We have looked at how buffering/caching can help
  - Merging of posting lists can result in savings
  - However, need to pick the right heuristics

Several levels of indexing ...

To find documents containing keywords “Query” and “Data” and “Base”
* Retrieve lists for Query, Data and Base, and intersect the document ids in the list
Additional index (over the posting lists) support is needed to answer *conjunctive queries* (e.g., \( k_1 \) AND \( k_2 \)) quickly.

**Merge Join**: \( O(m+n) \)

**Index Join**: \( m \log(n) \)

Can use GHT?

An alternative solution is *Jump Indexes*

- Path to an element only depends on elements inserted *before*
- Jump index is *provably* trustworthy
- Leverages the fact that document IDs are increasing
- \( O(\log N) \) lookup: \( N \) - # of documents (typically *weaker* than \( O(\log n) \) in traditional balanced trees like B+-tree where \( n \) is the number of entries)
  - Supports range queries too
- Reasonable performance as compared to B+ trees for conjunctive queries in experiments with real-workload
The Jump Index

- $i^{th}$ pointer points to an element $n_i$
  \[ n + 2^i \leq n_i < n + 2^{(i+1)} \]

Jump index in action

\[ n + 2^i \leq n_i < n + 2^{(i+1)} \]
Jump index in action

\[ n + 2^i \leq n_i < n + 2^{(i+1)} \]

\[ n + 2^2 \leq 7 < 1 + 2^3 \]

\[ 5 + 2^1 \leq 7 < 5 + 2^2 \]

\[ \log(N) \text{ pointers to } N \]
Path to an element does not depend on future elements

![Diagram showing Path to an element does not depend on future elements]

**Lookup (7)**

Start here

Follow Pointer

Got 7

\[ 1 + 2^2 \leq 7 < 1 + 2^3 \]
\[ 5 + 2^1 \leq 7 < 5 + 2^2 \]

Block-based Jump Index

- Storing pointers with every element is inefficient
  - With every document ID, \( \log_2(N) \) pointers are needed
- \( p \) entries are grouped together
- Branch factor \( B \).
  - \((B-1) \log_b(N)\) pointers
  - Pointer \((i,j)\) from block \( b \) points to \( b' \) having smallest \( x \)

\[ l + jB^i \leq x < l + (j+1)B^i \]

<table>
<thead>
<tr>
<th>( p ) entries</th>
<th>Jump Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

47
Jump index elements are stored in blocks

\[ l + j \cdot B^i \leq x < l + (j+1) \cdot B^i \]

What about Data Disposition?

- Regulations may prohibit retention after a certain period
- Company may be free to dispose of records once mandatory retention period has passed
- Term-immutability, rather than immutability
- Software can assign an expiry date on data that cannot be moved forward in time
Documents may be deleted, but indexes contain useful information ...

From the index, one can know that document 3 contain keywords “Query”, “Data”, “Base” and “Index”

Deletion from inverted indexes (on WORM)

- Secure deletion
  - Destroy the media?
  - Another approach
    - Create new copies of the document keyword’s posting lists, minus away those deleted documents’ IDs
    - Original posting list is erased
    - Impractical and costly, setting of expiry time is also difficult since it is not sure when the document will be deleted.

**Physical Deletion**

- What about zeroing-out the document ID+associated metadata from the posting list files?
  - Presence of holes can leak information (since ID are in increasing order)
  - Costly to implement such fine grained deletion in WORM storage

**Logical Deletion**

To reduce overhead, documents with similar expiry date can be grouped into the same disposition group. Encrypt these documents using the same secret key.

![Diagram of encrypted inverted index](image_url)
Logical Deletion

To reduce overhead, documents with similar expiry date can be grouped into the same disposition group. Encrypt these documents using the same secret key.

When all documents associated with keyfile 1 expires, just need to erase keyfile 1

- To prevent "join attack", encrypt (keyword, ID) pair instead.
- Adversary can still determine a set of keywords that were committed in documents in the disposition group, though he cannot determine the exact association of those words with documents.
- Document IDs can still be guessed from those of neighbors.

### Encrypted inverted index

```
Query | 1 | 3 | 11
Data  | 3 | 9 | 31
Base  | 3 | 19|      

keyfile1 | keyfile2
```

Summary

- For trustworthy record keeping, indexes must also be trustworthy
- GHT and Jump Index are examples of trustworthy indexes
- Both can achieve $O(\log(N))$ search time in practice