Trustworthy Keyword Search for Regulatory-Compliant Records Retention

S. Mitra, W. Hsu, M. Winslett

Presented by Thao Pham
Introduction

• Important documents: emails, meeting memos,... must be maintained in a trustworthy fashion

  – *safe* from *destruction* and *modification*
  – while still *readily accessible*
Introduction (tt)

• Write-once-read-many (WORM) storage assures that once data is written, it cannot be deleted or modified

• However...

• The large volume of data requires some index structure for efficient access.
  – Adversary can modify the index so that an access through it no longer returns correct data.

**Fossilized** index is required
Introduction (tt)

- The paper considers inverted index used to index unstructured documents

How to efficiently update the posting lists when a document is added?

How to index the posting lists to support posting list intersection for conjunctive multi-keyword search?
Talk outline

• Storage and threat models

• Efficiently append to the index (posting lists) upon adding a new document
  – Why currently inefficient?
  – Proposed *posting list merging* method
  – Experimental evaluation

• Fossilized index for posting lists
  – Why not B+ tree or General Hashing Tree (GHT)?
  – The proposed *jump index*
Storage model

• Documents and the inverted index are stored in WORM storage
  – *No physical* destruction of modification

• To store index’s posting list, assume a special WORM storage where the WORM rule is enforced by a software built on top of a rewritable media
  – Such WORM storage allows *appending small amount of data* to an existing file and even existing partially written block
Threat model

• The adversary Mala want to hide an existing document D from future keyword searches.
  – Mala can take on the identity of any legitimate users and read or append any data to the WORM storage
  – Mala cannot physically delete or modify data, or modify the behavior of the search engine and operating system.

• Everything is safe through the point the document D is permanently stored in the WORM storage.

• Mala’s only hope is to logically hide D by attacking the index through which D is accessed.
Update inverted index- Problems

• Straightforward approach:
  – For a new document D, append D’s id to the tail of the posting lists corresponding to the keywords in D
  – But, this is prohibitively slow
    • On average, each document contains 500 keywords. So, 500 \textbf{random} I/O per document!
Update inverted index - Problems

• Other solutions: Buffering the index entries in memory and commit them in batches
  – This is effective only when a huge number of index entries is buffered
  – Another problem?

  • Mala can take advantage of the gap between document commit and index update: delete the entries in the buffer, crash the application and delete the recovery logs.

→ Inverted index must be updated in real time
Posting list merging

• Cost of appending document id’s to the posting list can be reduced by caching the lists’ tail blocks in the storage cache
  
  – Data in a WORM storage cache is considered committed to the WORM storage

  – However, even with large cache of 4GB, the cache-miss rate is still high and adding a document still requires 21 random I/O
• Merging the posting lists to reduce the number of lists, so that all the list’s tail blocks can be cached (cache hit rate = 1)
  
  – An I/O is required only when a tail block is full
  – On average, 1 I/O per document!
Posting list merging (cont.)

• Merging, however, has costs!
  
  – Bigger entry – the terms need to be encoded
    • encode using $\log(q)$ bit if q lists are merged together.
    • Huffman encoding.
  
  – Longer posting lists to scan!
    • Merging optimization strategies
    • Experimentally evaluate the cost.
Posting list merging – Merging strategies

• A term’s contribution to the total query workload is
  \[(\text{query frequency}) \times (\text{term frequency})\]
  – query frequency: number of queries asking for the term
  – term frequency: the number of documents containing the term

• **Observation:** a *very small fraction* of the terms has high query frequency and term frequency, thus contributes to *most of* the total workload.

• → **Merging strategies:** *Separated* posting list for *each of the frequent term*, and uniformly merge the remaining ones
Posting list merging – Experimental result

With cache size of 128 or more, the workload cost is almost the same as the case with no merging.

Merging strategies can indeed reduce the cost when the cache is small.

Note: The y axis is the ratio of workload cost (merge) / workload cost (no merge)
Fossilized index for posting lists

• When a query asks for documents that contains all of the specified terms:
  – result is the intersection of the posting lists corresponding to the terms

• A fossilized index for each of the posting lists can speed up the intersection, while still preserving trustworthiness.
Index for posting list – why not B+ Tree?

- For an increasing sequence of document IDs, a B+ Tree can be built bottom up to avoid splitting and merging of nodes
  - However, even by appending nodes and node entries only, the adversary can still fool the search engine!

Now, what happens when the joining algorithm looks for document id of 29 ???
Index for posting list – why not GHT?

• Look at part of the zigzag join of two lists:

```java
if ((*top1) < (*top2)) then
    top1 ← list1.FindGeq(*top2)
    {Find an element greater than or equal to top2}
    continue
end if
if ((*top2) < (*top1)) then
    top2 ← list2.FindGeq(*top1)
    {Find an element greater than equal to top1}
    continue
end if
if ((*top2) = (*top1)) then
    OUTPUT (top1)
```

Hasing based index cannot support range search!

Where can GHT be used in this inverted index scheme?
Jump indexes for posting lists

• Jump index exploits the fact that the sequence of IDs in each posting list is strictly increasing

• Jump index supports the FindGeq(x) – finding an element greater than or equal to x – in $O(\log(N))$
  
  — N is the **greatest number** in the sequence
Jump indexes - Intuition

- Given a number $0 \leq k \leq N$, with binary representation $\langle b_1...b_p \rangle$ where $p = \log(N)$

\[
\begin{align*}
X_0 & \quad X_1 & \quad X_2 & \quad \ldots & \quad k \\
0 & \quad 1 & \quad 2 & \quad \ldots & \quad (X_0 + b_1 \cdot 2^{p-1}) & \quad \ldots & \quad X_1 + b_2 \cdot 2^{p-2} & \quad \ldots & \quad X_{p-1} + b_p \cdot 2^0
\end{align*}
\]

→ from the start of the increasing sequence, jumping $p$ steps gets to $k$
Jump indexes (cont.)

• Jump index tree:

  – Nodes are added in increasing order

  – The root node is the smallest number

  – Each node has a value \( l \), which is the document ID, and a list of jump pointers

(a) Binary Jump Index.
Jump indexes (cont.)

• An \(i^{th}\) pointer points to the node of smallest value \(l'\) satisfying:
  \((1)\) \(l + 2^i \leq l' < l + 2^{i+1}\)

• **All the nodes in the sub-tree of the \(i^{th}\) pointer also satisfy (1)**
  
  – So, at each step, a search for a node of value \(k\) follows the pointer \(i^{th}\) that satisfies: \(l + 2^i \leq k < l + 2^{i+1}\)
Jump index - extension

• Jump pointers point to a **block** consisting of a list of posting list entries rather than a single entry

  – (1) \( l + 2^i \leq l' < l + 2^{i+1} \)

  *l is the greatest* entry in the block, and *l’ is the smallest* on the child block

• Using base B rather than 2.
Jump index (cont.)

• Inserting a new element into the index tree requires appends only, but intermediate blocks need to be fetched.

• To improve performance:
  – For every tail block of a posting list, cache in memory the last pointer and the greatest document ID of all the intermediate blocks on the path from root to it

  – Can this leave a chance for the adversary???
Conclusion

• Merging of posting lists: reduce the numbers of I/O required when adding a new document

• A fossilized index structure for each posting list to support joining of posting lists in answering multi-keyword queries
Conclusion (cont.)

• Strengths:
  – Novel, and interesting approaches to support inverted index.
  – Thoughtful experiments and analysis

– Weaknesses:
  • Based on the assumption that the WORM storage supports append bytes to existing partially written block.