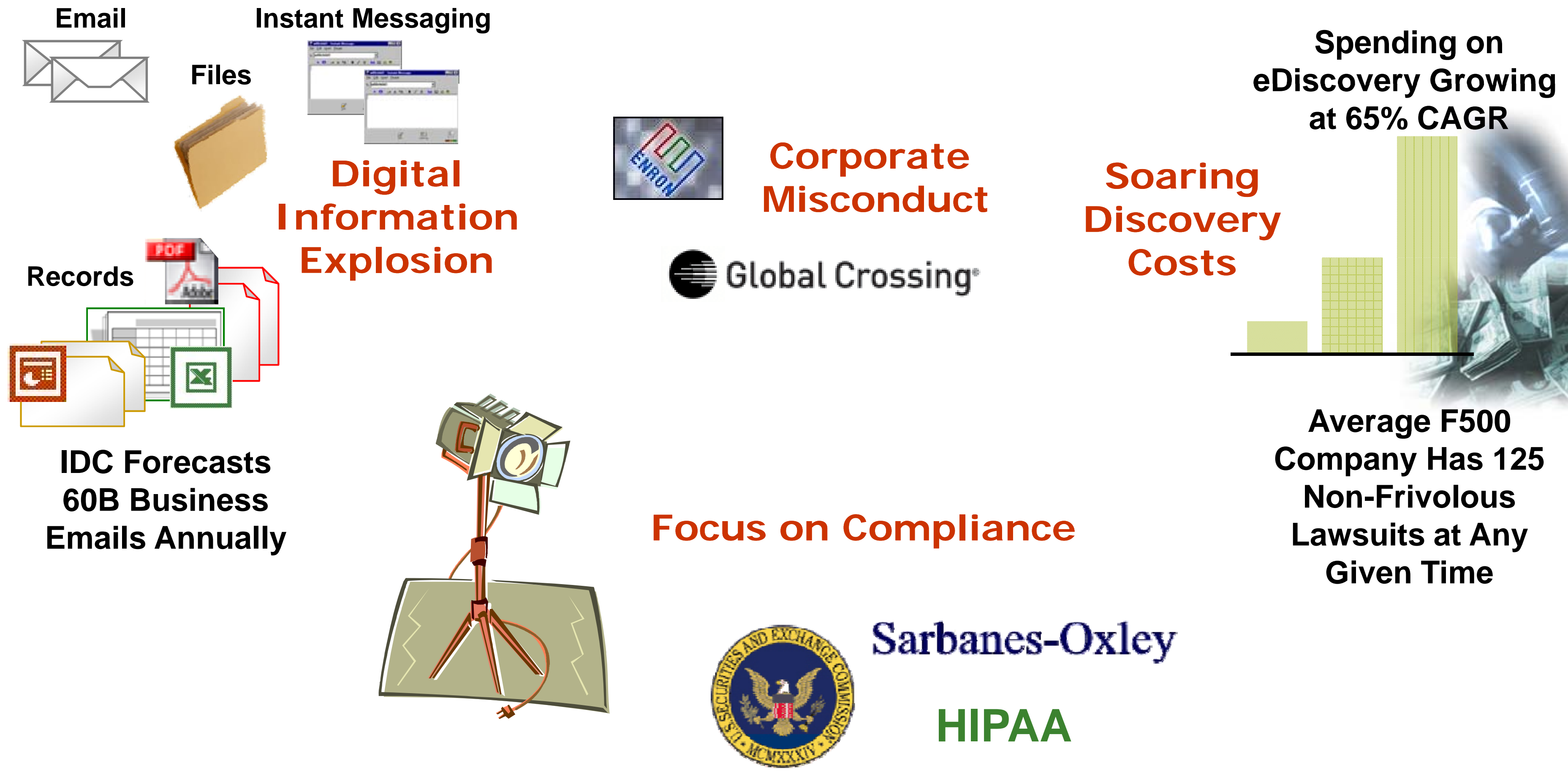
Secure Indexing/Search for Regulatory-Compliant Record Retention

There is a need for trustworthy record keeping

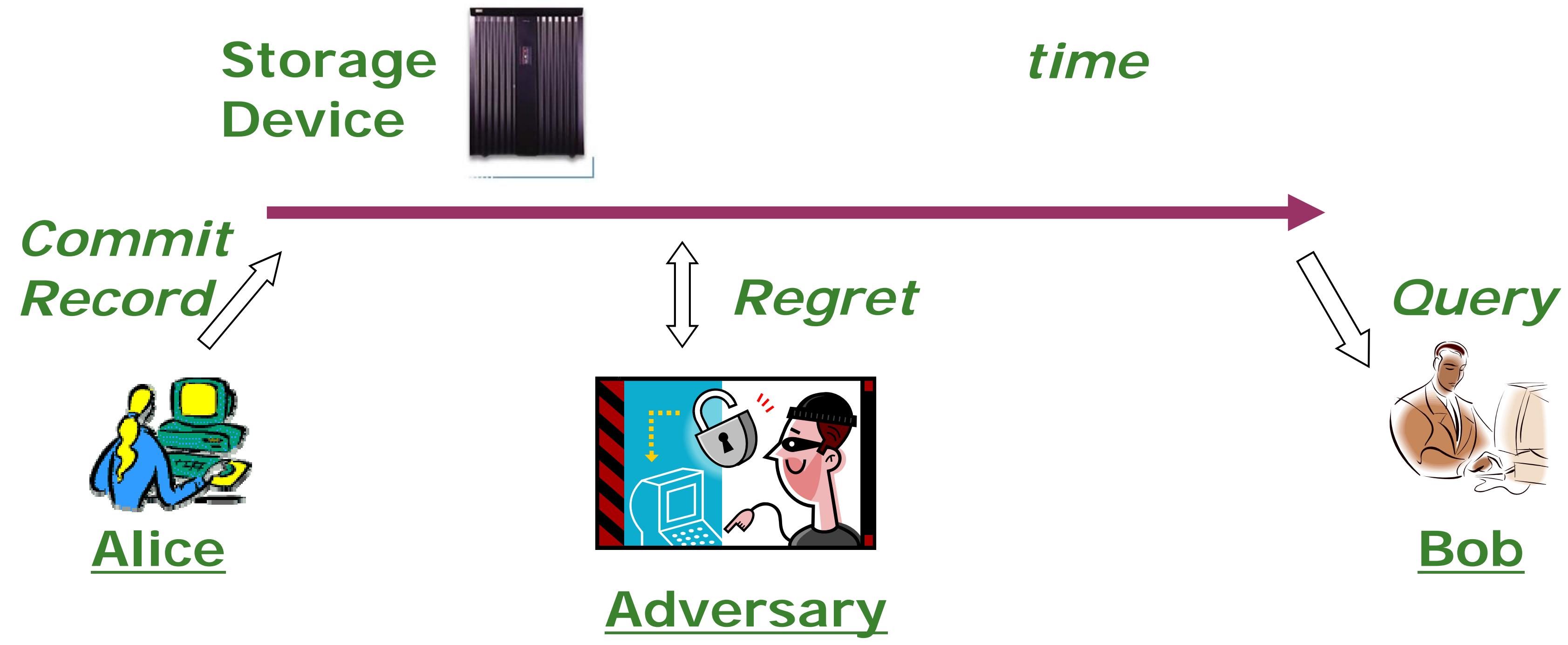


Sources: IDC, Network World (2003), Socha / Gelbmann (2004)

Q. Zhu, W. W. Hsu: Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records. SIGMOD'2006, 395-406, 2006

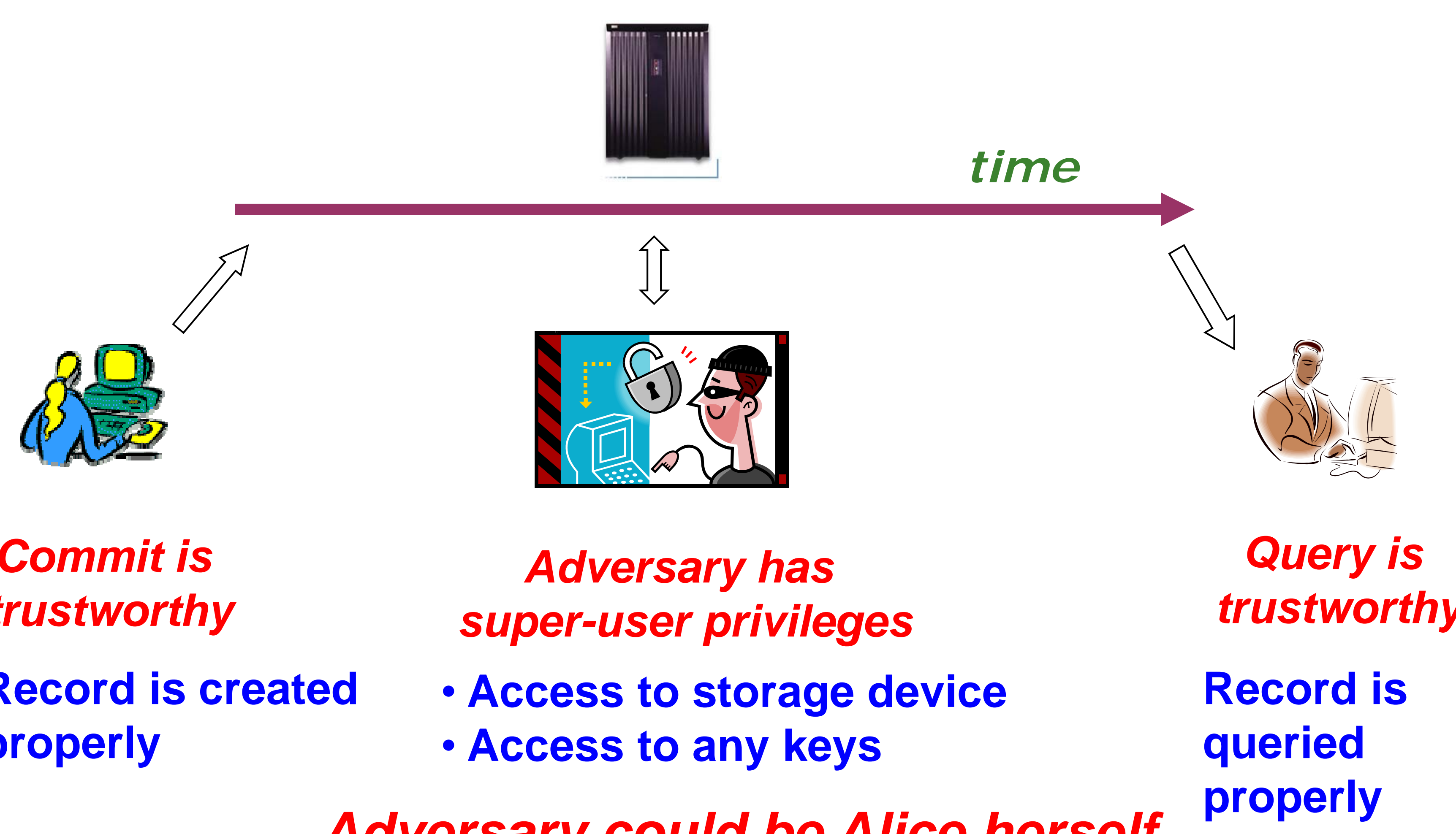
What is trustworthy record keeping?

Establish solid proof of events that have occurred

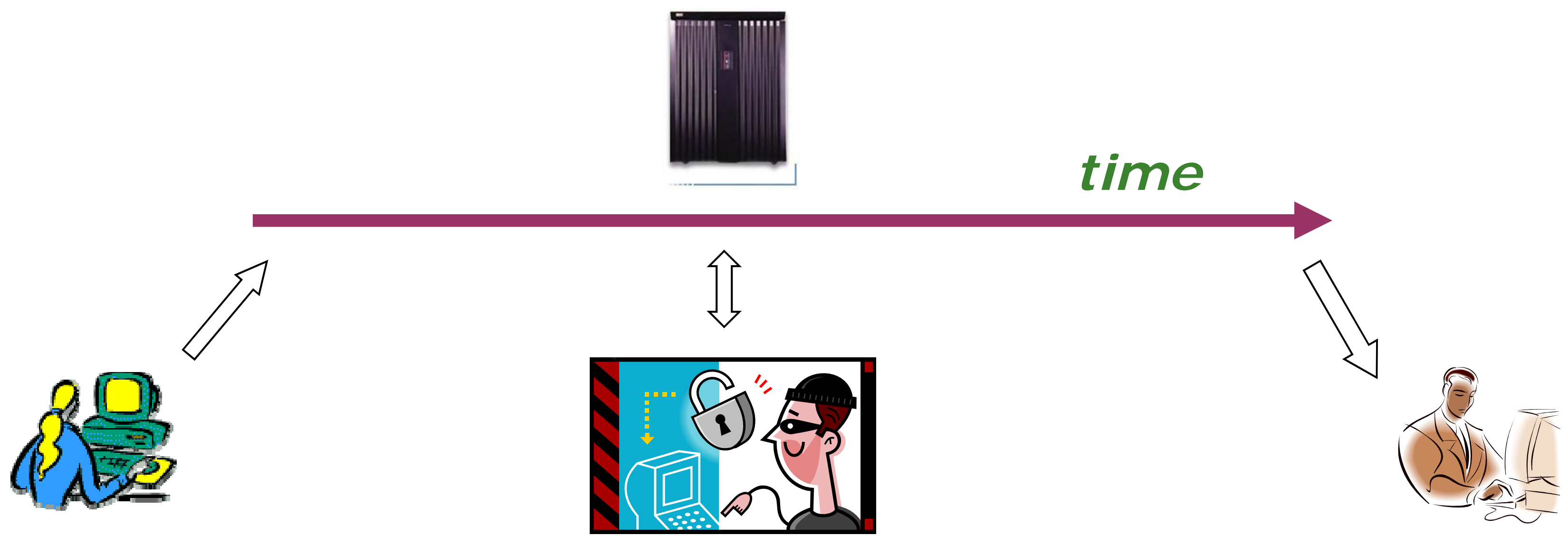


Bob should get back Alice's data

This leads to a unique threat model

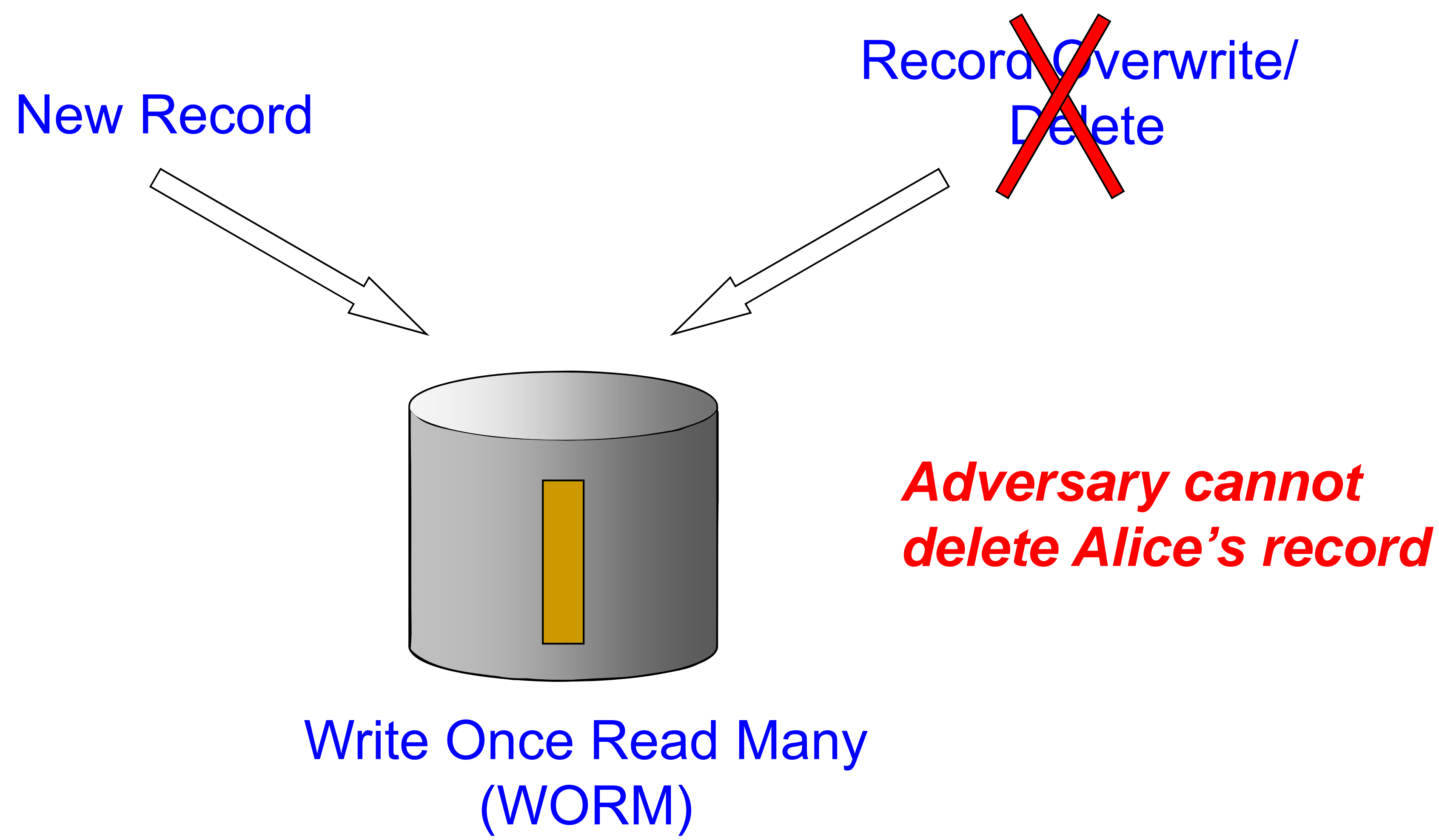


Traditional schemes do not work

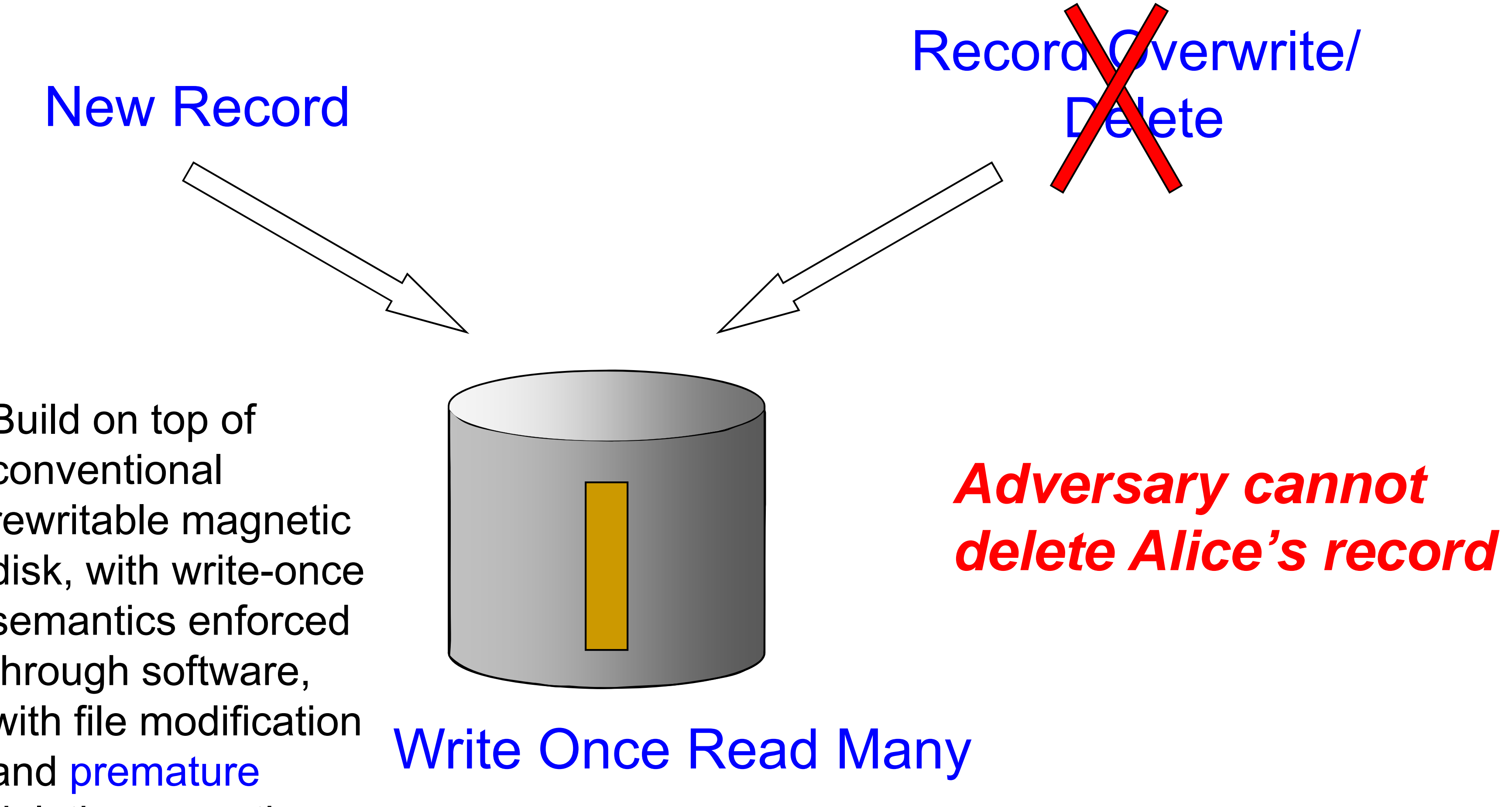


Cannot rely on Alice's signature

WORM storage helps address the problem

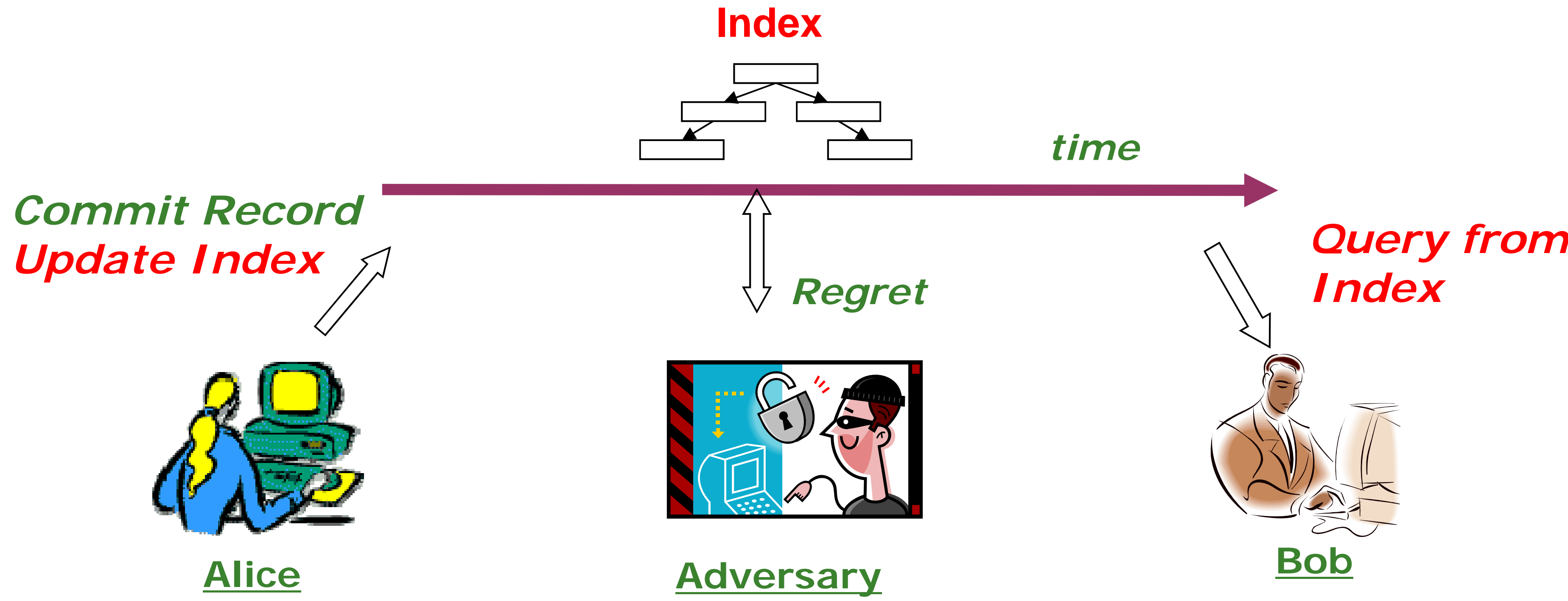


WORM storage helps address the problem

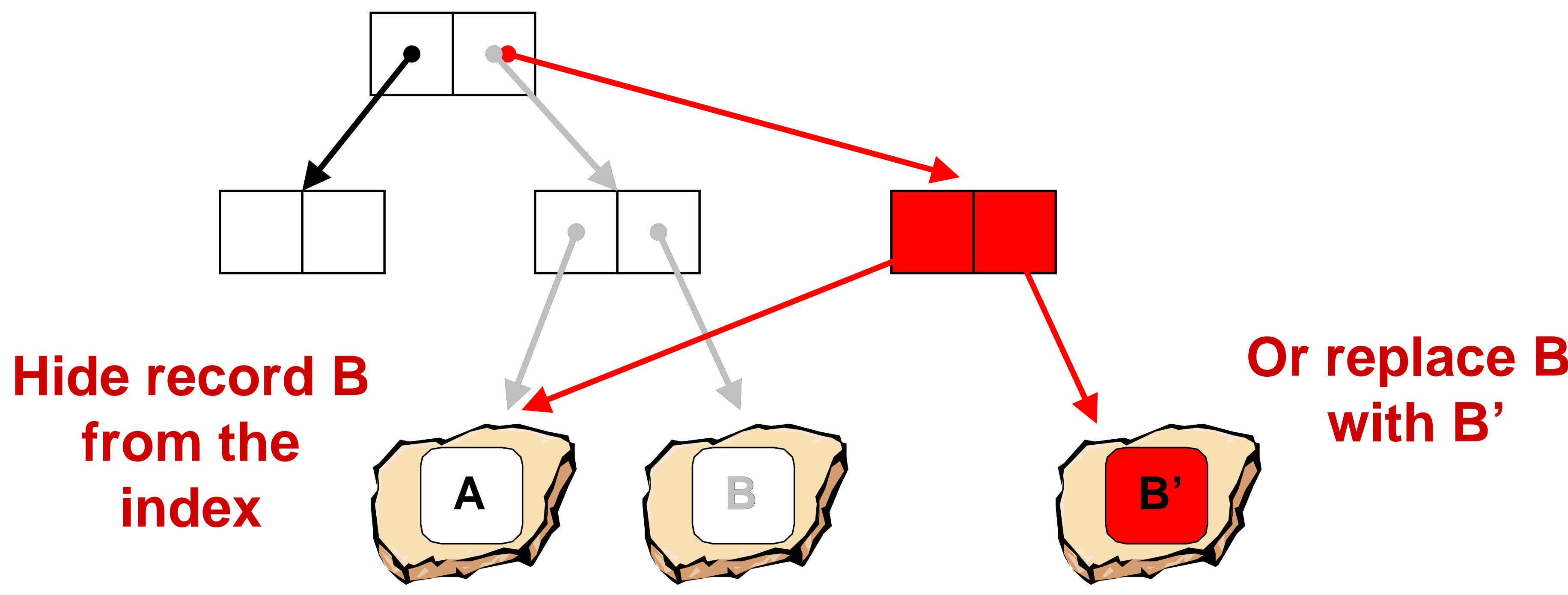


Build on top of conventional rewritable magnetic disk, with write-once semantics enforced through software, with file modification and premature deletion operations disallowed.

Index required due to high volume of records

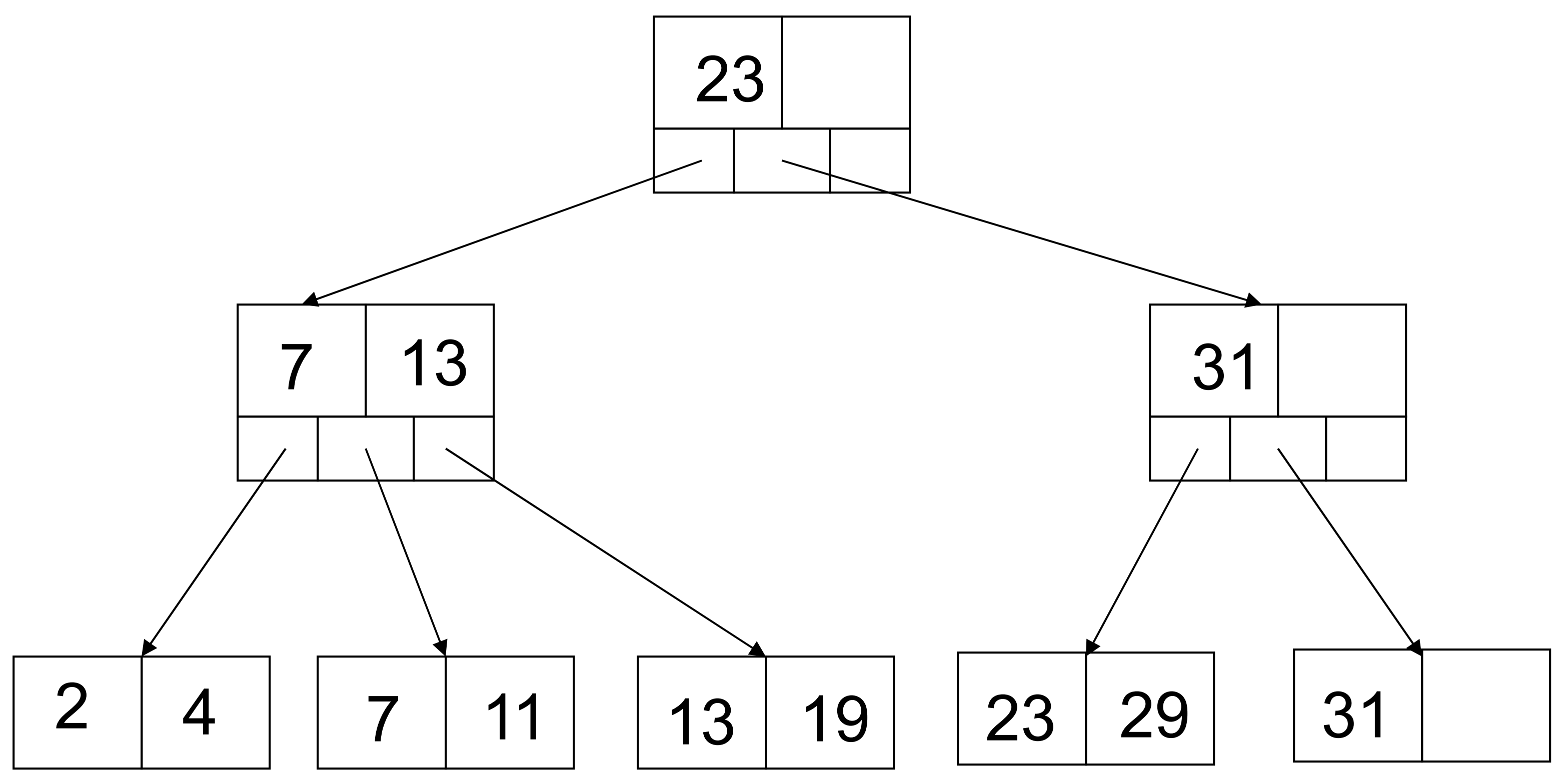


In effect, records can be hidden/altered by modifying the index

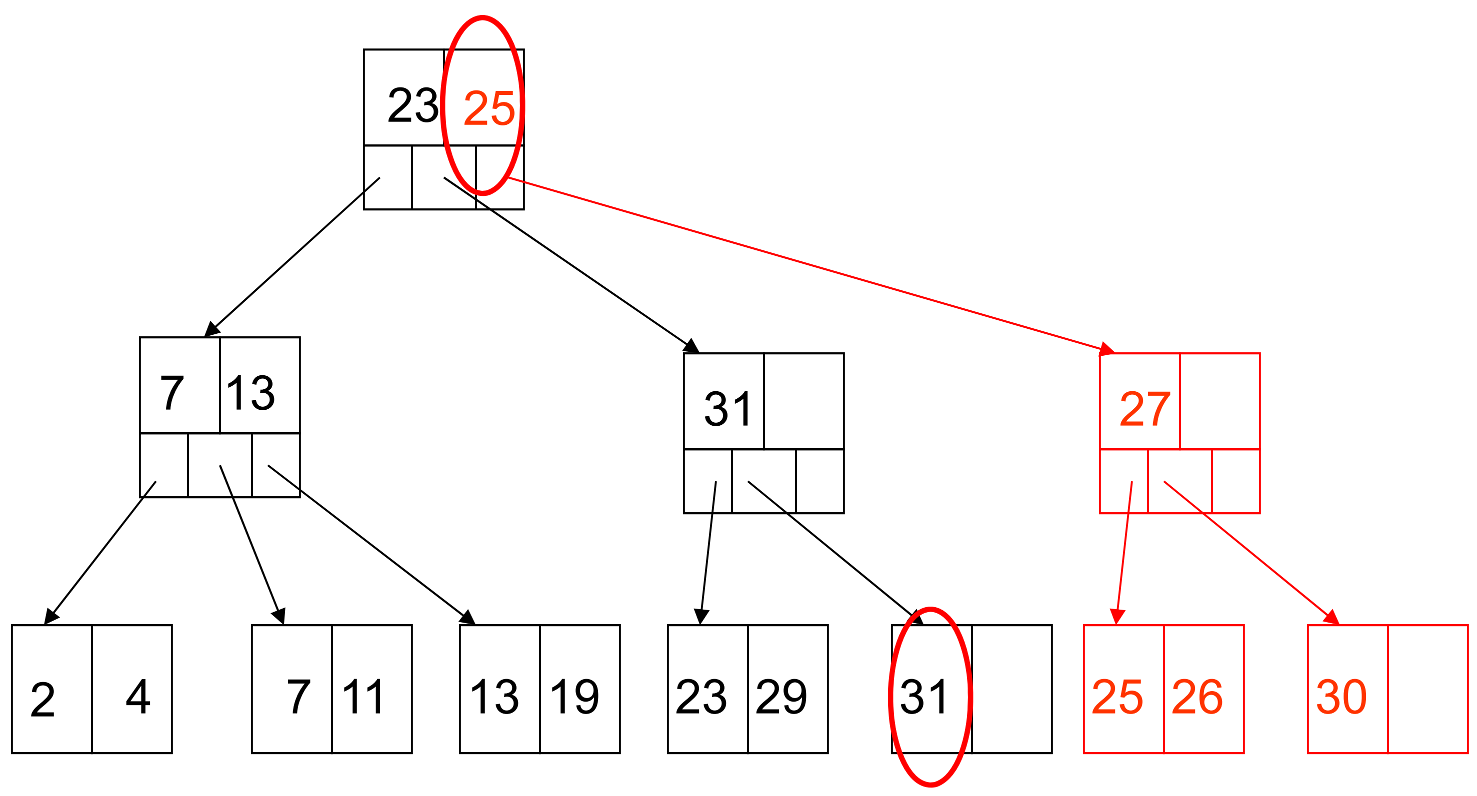


The index must also be secured (fossilized)

Btree for increasing sequence can be created on WORM



B+tree index is insecure, even on WORM

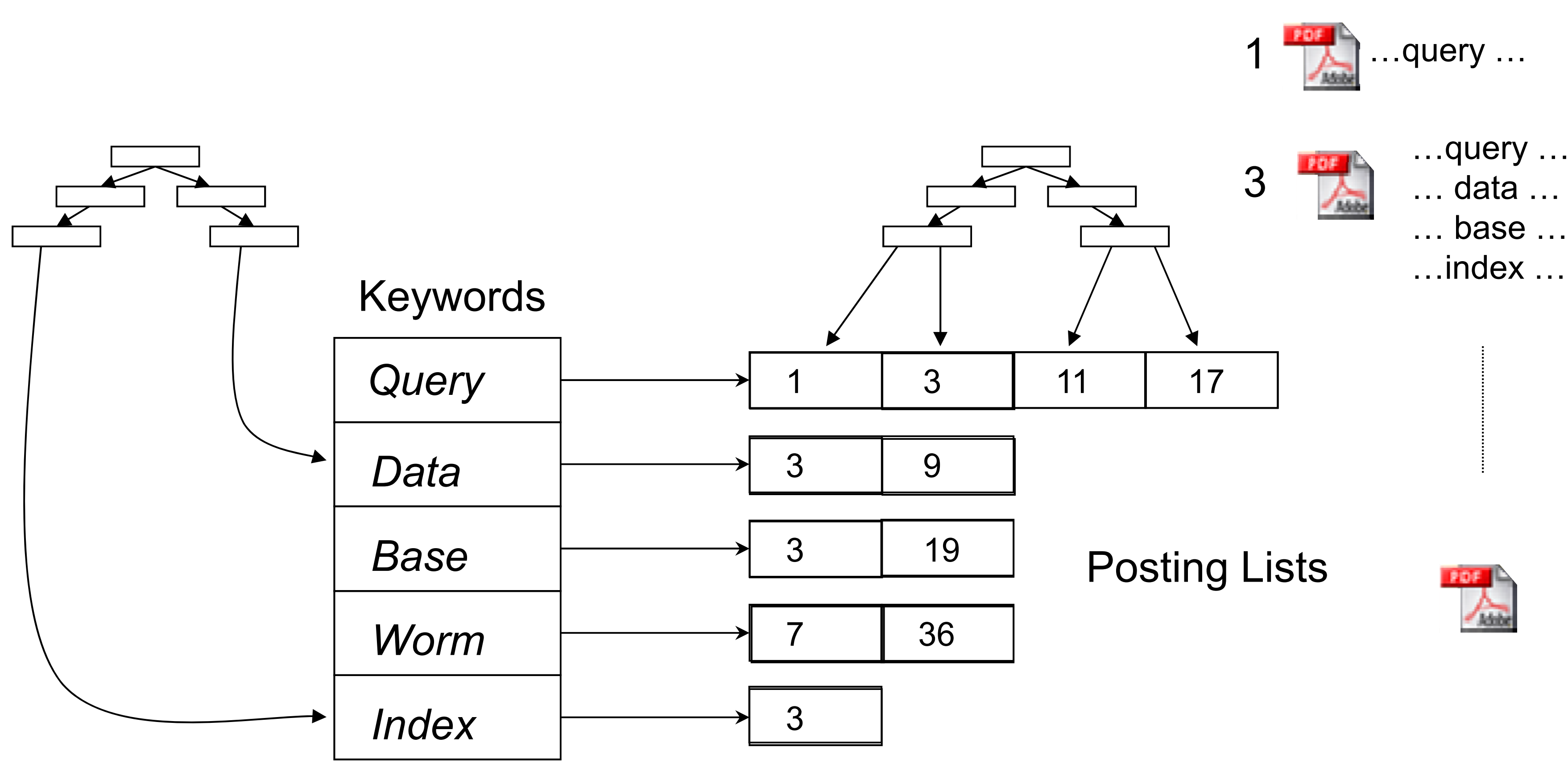


- Path to an element depends on elements inserted later – Adversary can attack it

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”

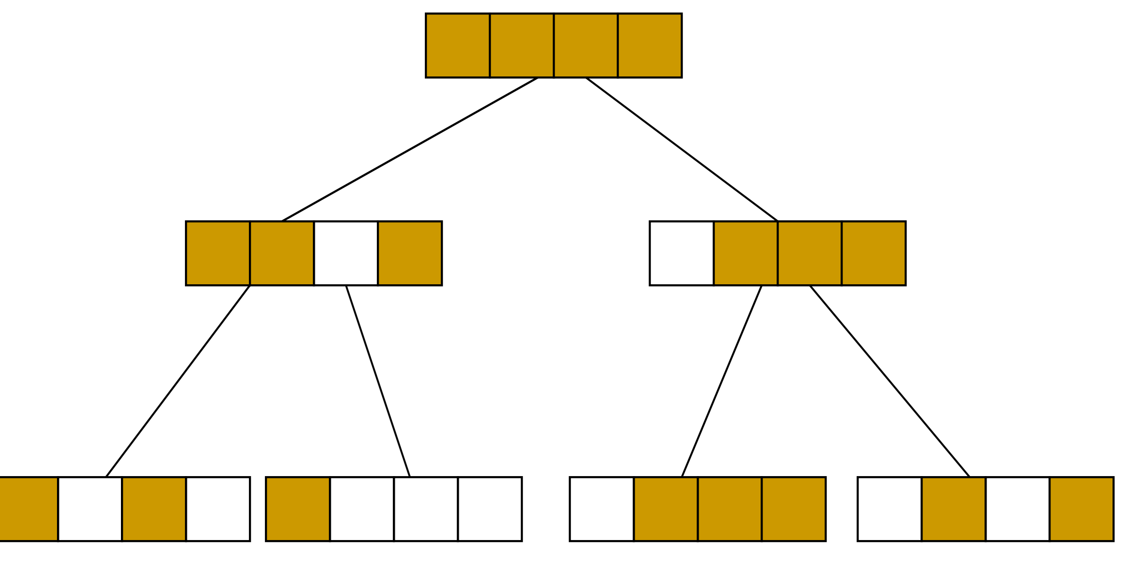
* Retrieve lists for Query, Data and Base, and intersect the document ids in the list

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

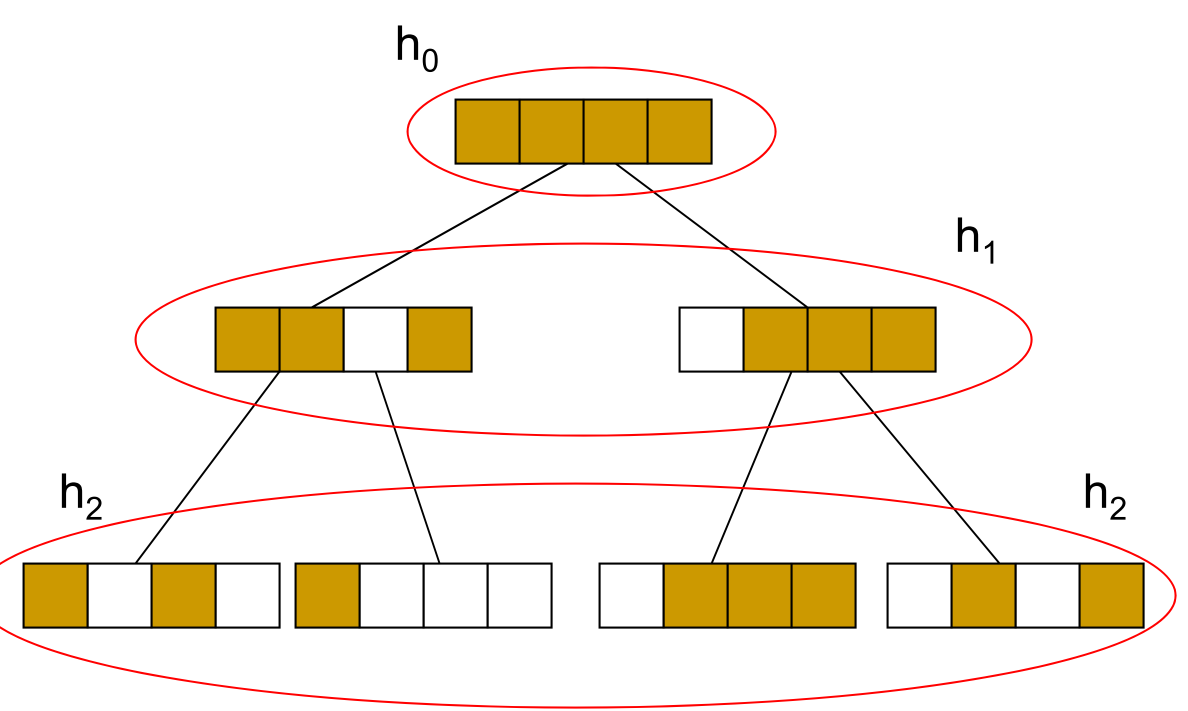
- Defined by {M,K, H}
- $M = \{m_0, m_1, \ldots\}$, m_i is size of a tree node (number of buckets) at level i
- $K = \{k_0, k_1, \ldots\}$, k_i is the growth factor for level i
    - A tree has k_i times as many nodes at level (i+1) as at level i
- $H = \{h_0, h_1, \ldots\}$, h_i is a hash function for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



$m_0 = m_1 \ldots = 4$
$k_0 = k_1 \ldots = 2$

Standard (Default) GHT – Thin Tree

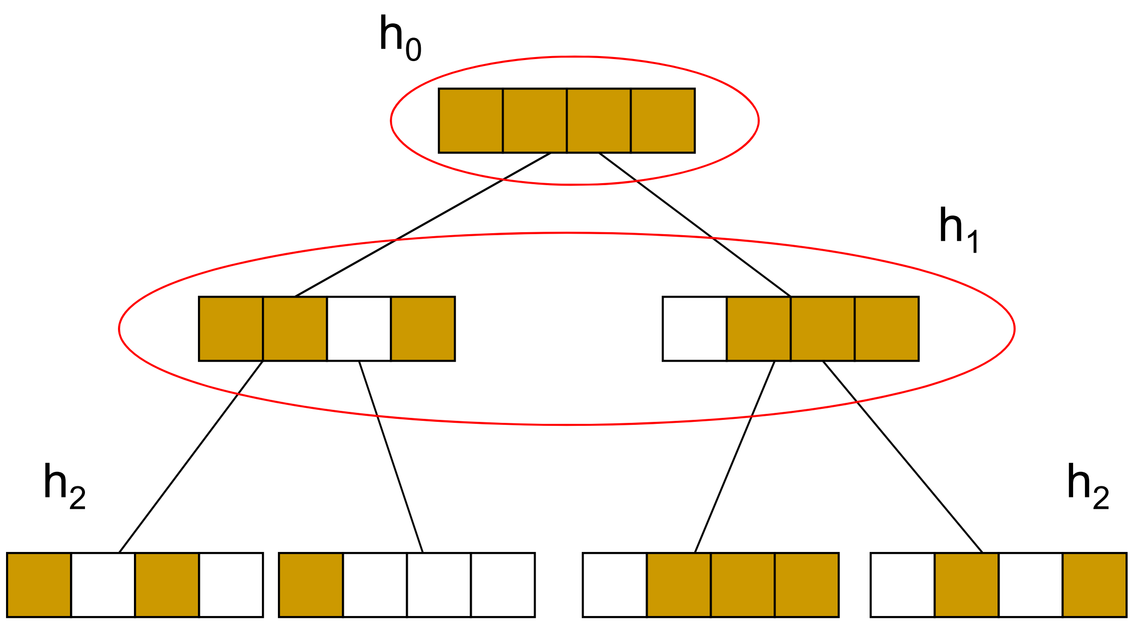
- Defined by {M,K, H}
- $M = \{m_0, m_1, \ldots\}$, m_i is size of a tree node (number of buckets) at level i
- $K = \{k_0, k_1, \ldots\}$, k_i is the growth factor for level i
  - A tree has k_i times as many nodes at level (i+1) as at level i
- $H = \{h_0, h_1, \ldots\}$, h_i is a hash function for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



$m_0 = m_1 \ldots = 4$
$k_0 = k_1 \ldots = 2$

Standard (Default) GHT – Thin Tree

- Defined by {M,K, H}
- M = {m_0, m_1, …}, m_i is size of a tree node (number of buckets) at level i
- K = {k_0, k_1,…}, 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,…}, h_i is a hash function for level i

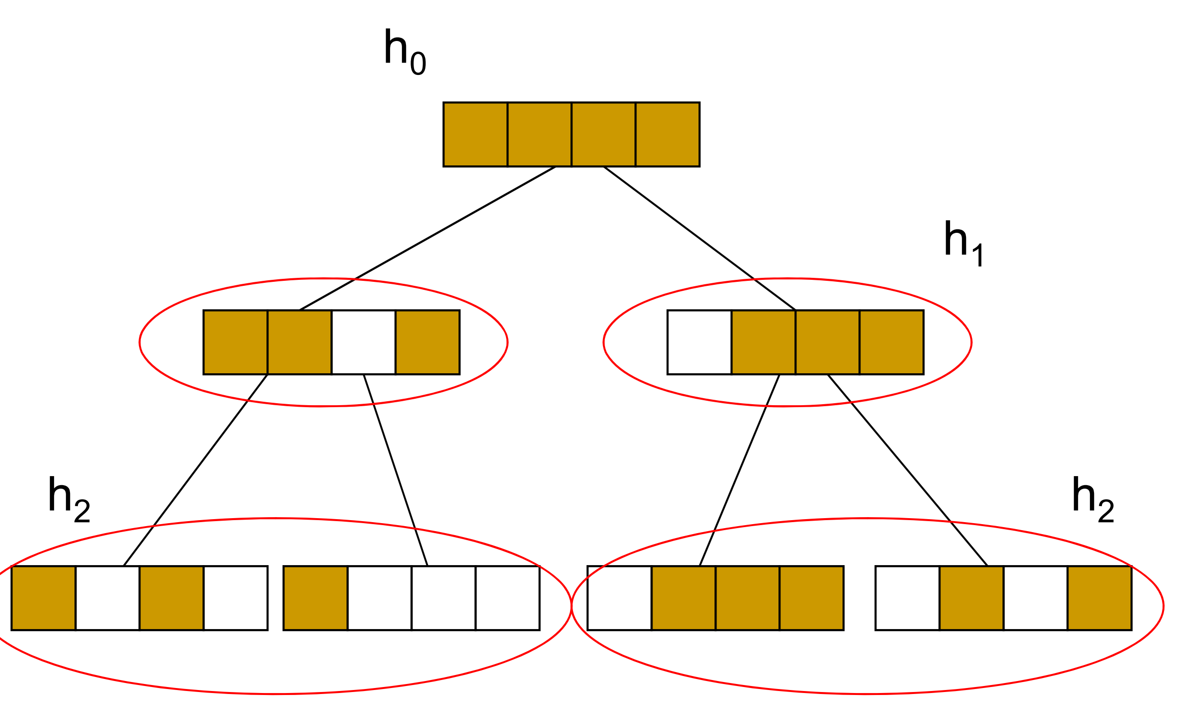


$m_0 = m_1 \ldots = 4$
$k_0 = k_1 \ldots = 2$
h_0 = x mod 4
h_1 = x mod 8

What about h_2? x mod 16?

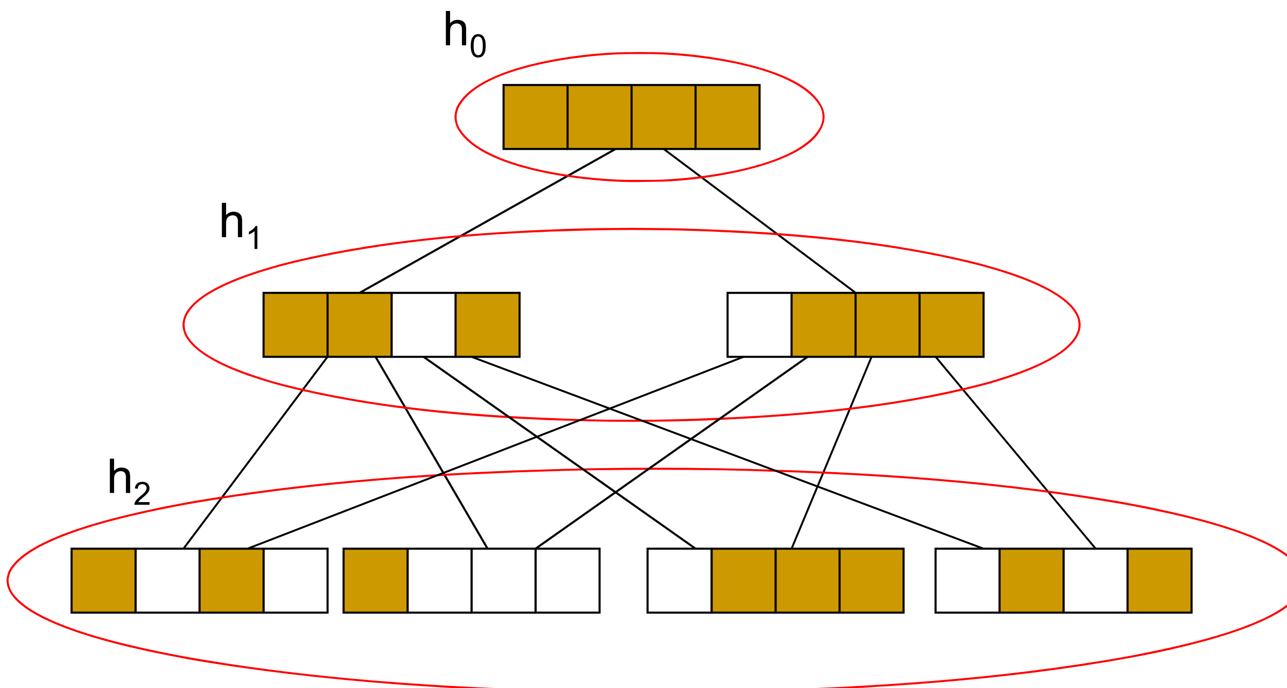
Standard (Default) GHT – Thin Tree

- 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



$m_0 = m_1 \ldots = 4$
$k_0 = k_1 \ldots = 2$
$h_0 = x \bmod 4$
$h_1 = x \bmod 8$
$h_2 = h_3 = \ldots = x \bmod 8$

GHT Variant (Fat Tree)



$m_0 = m_1 \ldots = 4$
$k_0 = k_1 \ldots = 2$

$h_0 = x \bmod 4$
$h_1 = x \bmod 8$
$h_2 = x \bmod 16$
$h_i = x \bmod 4*2^i$

Can tolerate non-ideal hash functions better because there are many more potential target buckets at each level

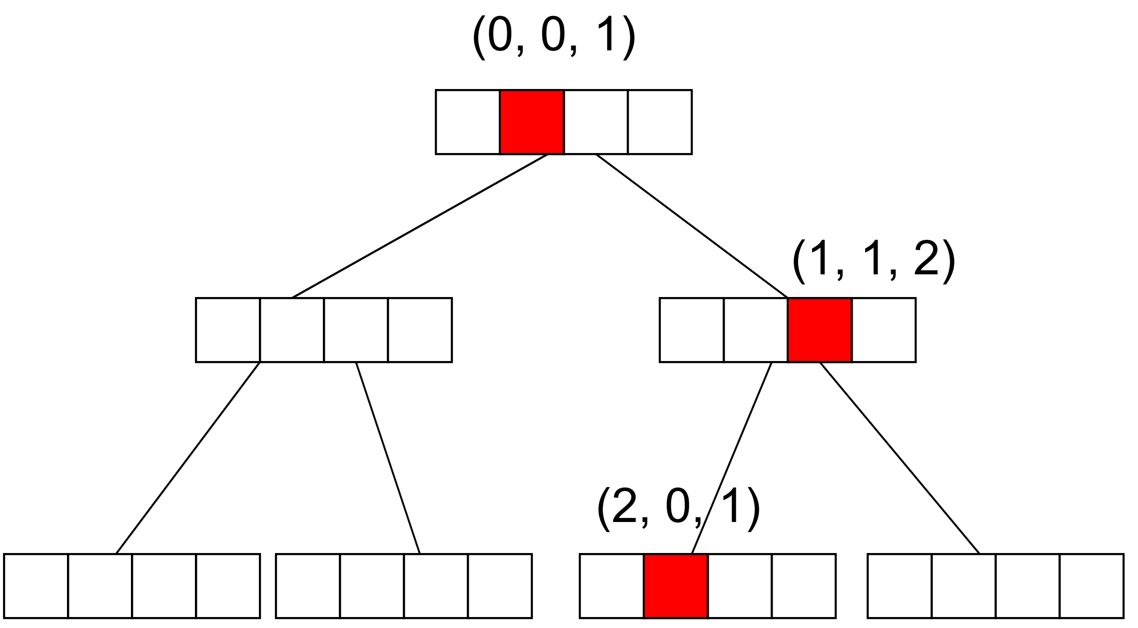
Hashing at different levels is independent

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

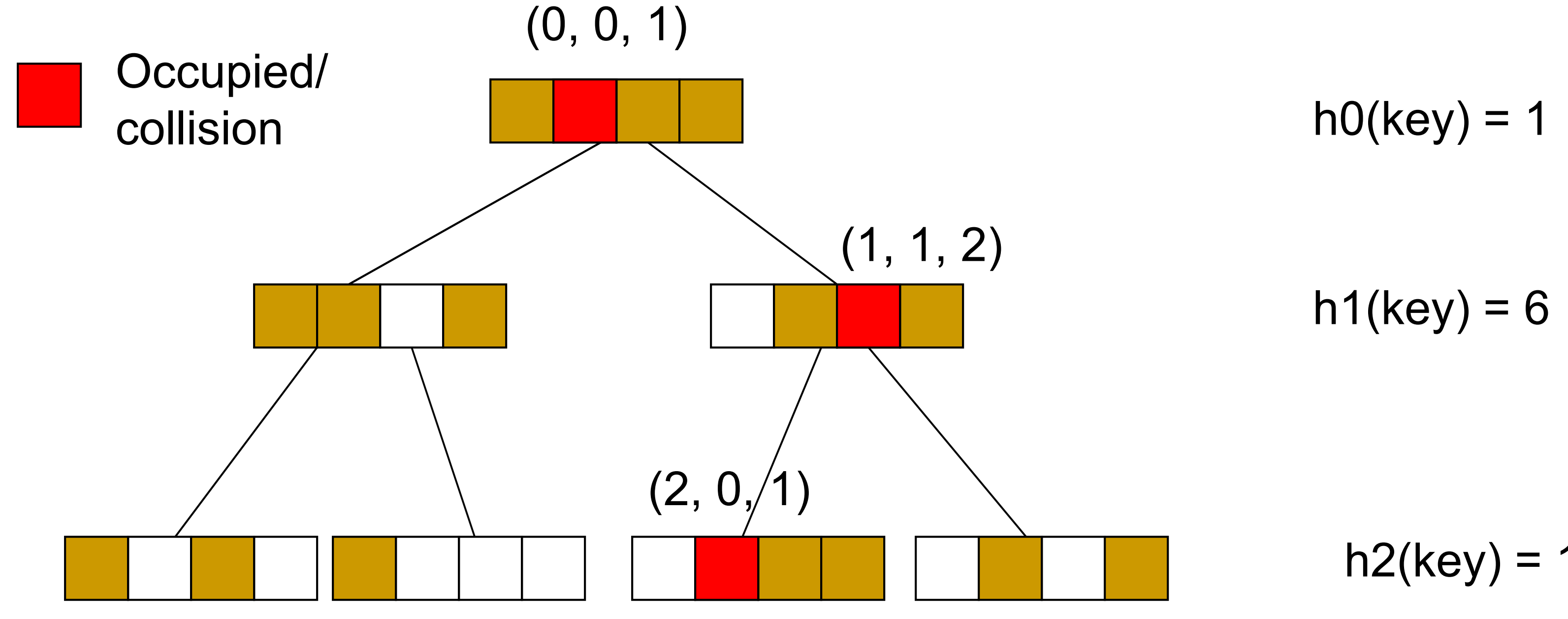
GHT (Standard) Insertion

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



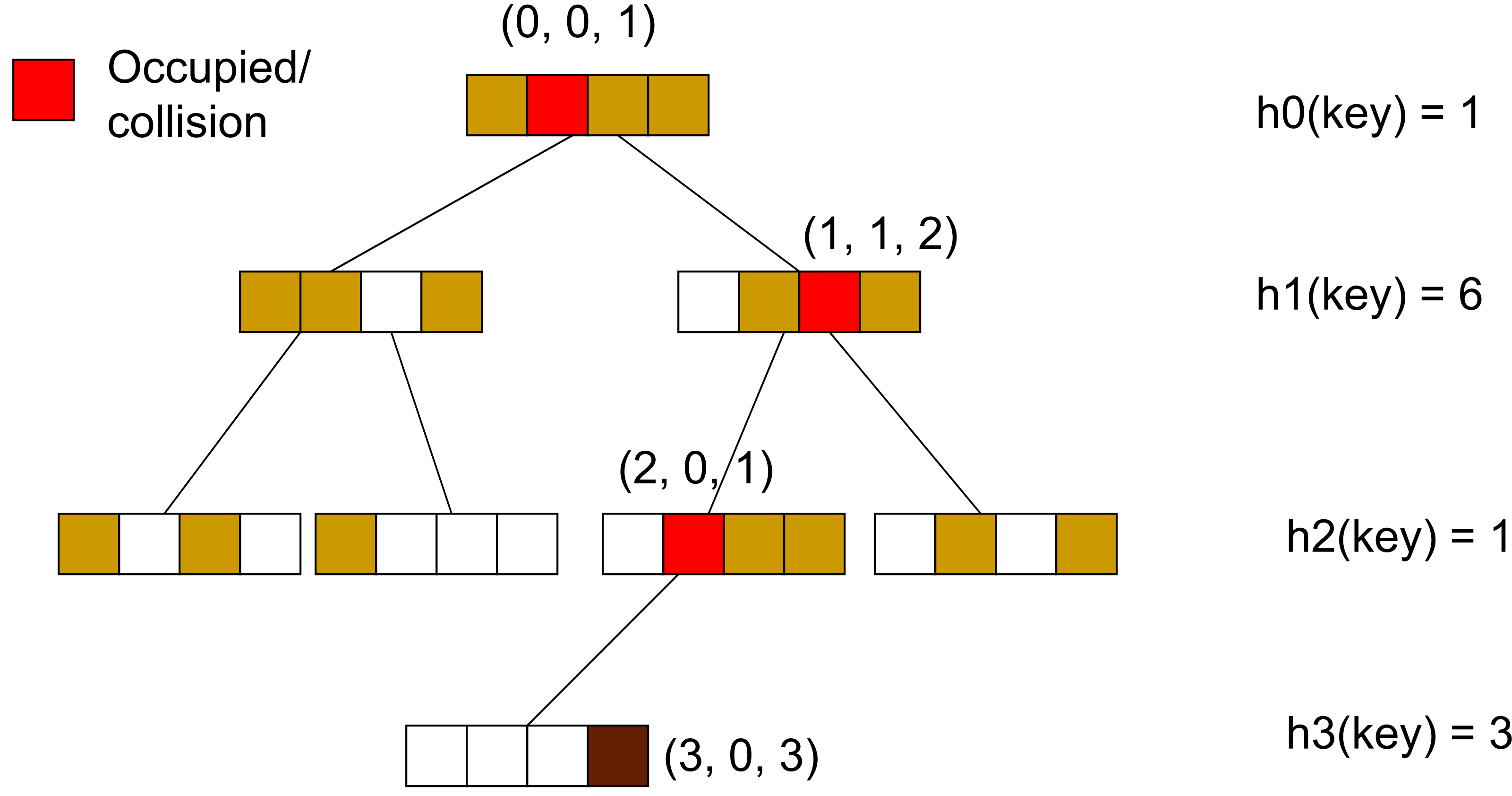
GHT Insertion

Insert ■ whose hash values at the various levels are shown.



h3(key) = 3

GHT Insertion

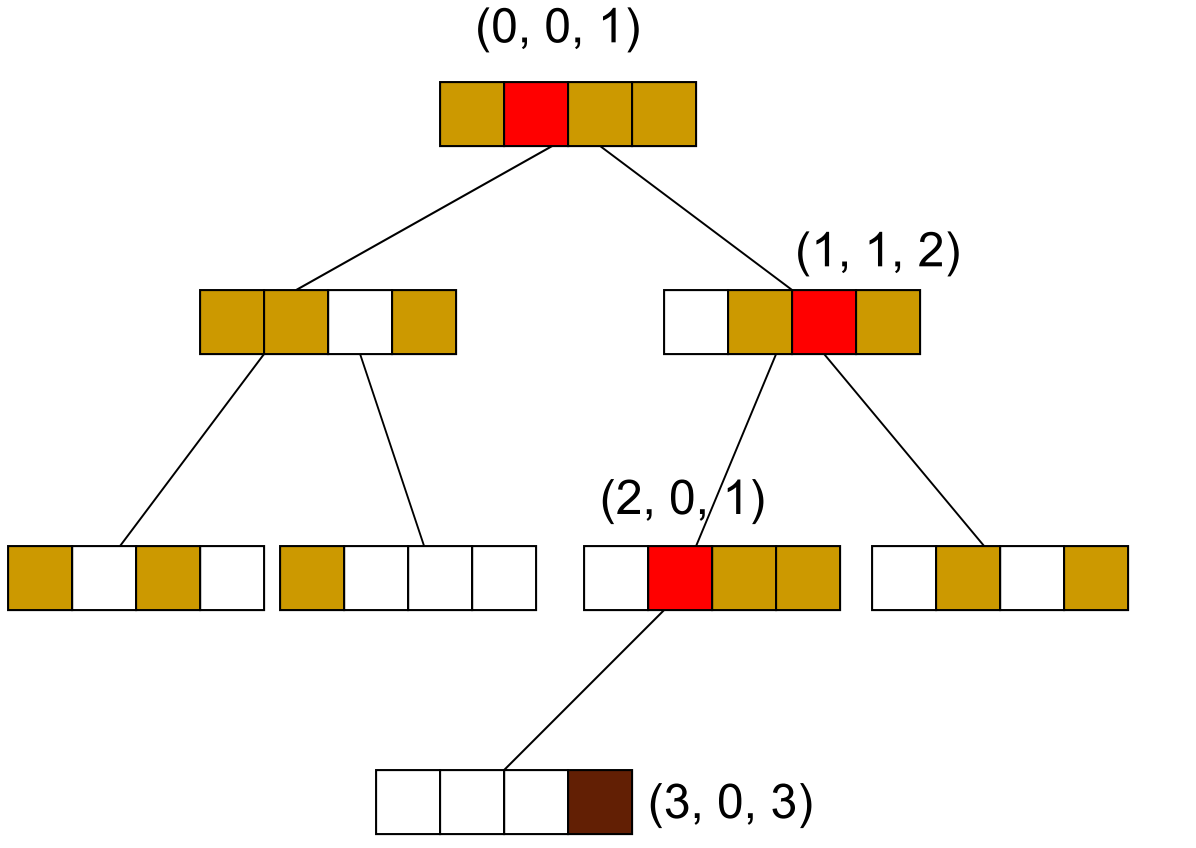


If hash functions are uniform, tree grows top-down in a balanced fashion

GHT Search

Search for ■ whose hash values at the various levels are shown

- Similar to insertion
- Need to deal with duplicate key values

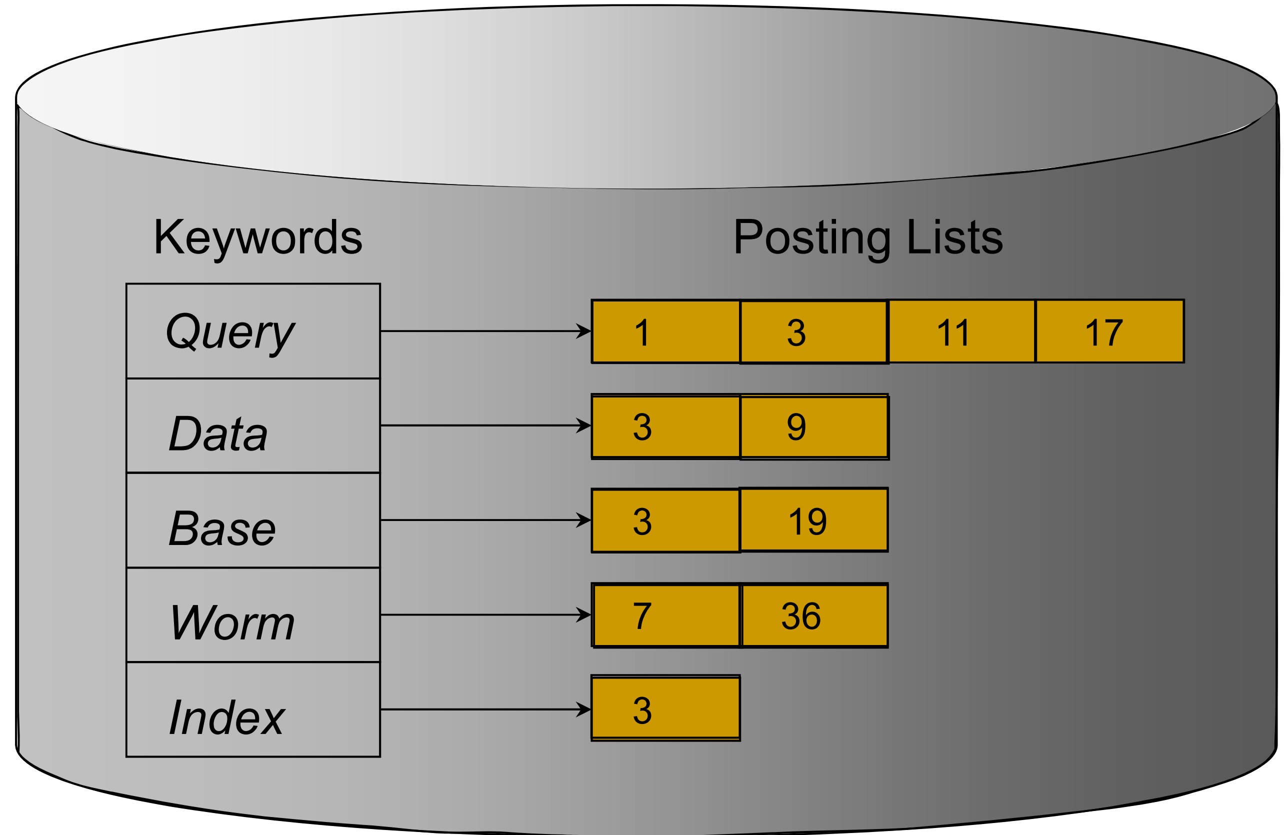


- Only for point queries
  - Cannot support range search

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!

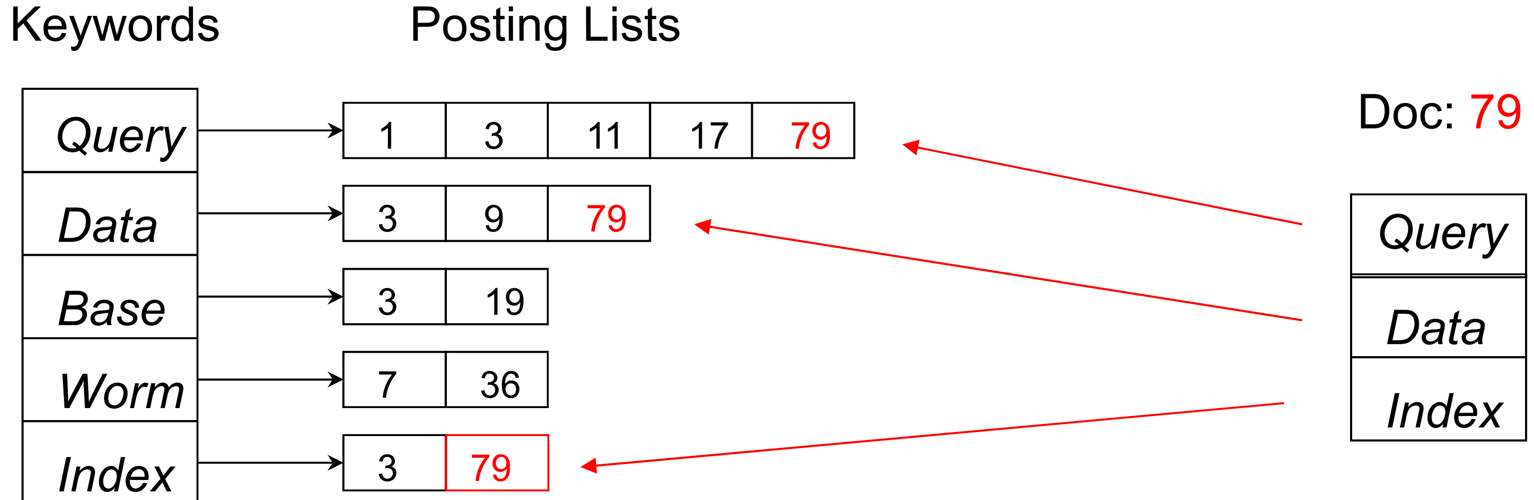
Most business records are unstructured, searched by inverted index



One WORM file for each posting list

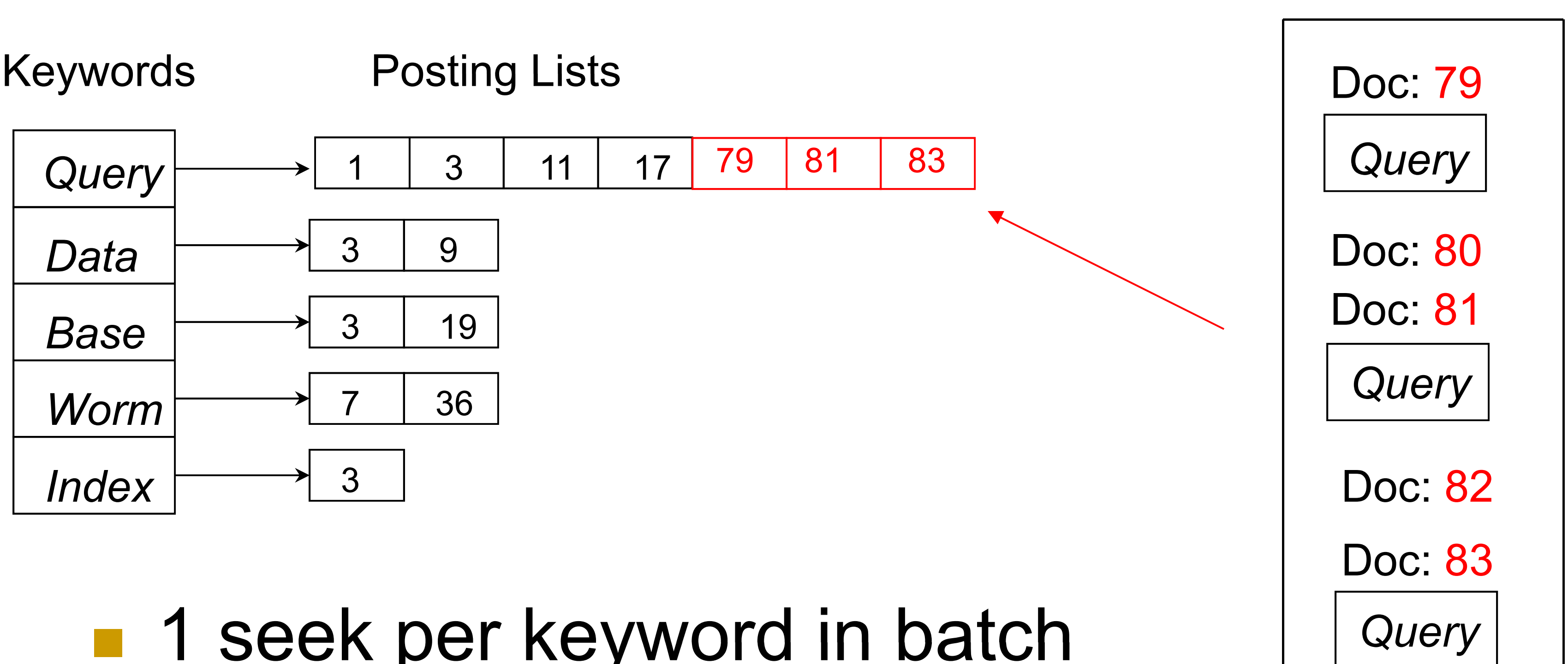
S. Mitra, W. W. Hsu, M. Winslett: Trustworthy Keyword Search for Regulatory-Compliant Record Retention. VLDB'2006, 1001-1012, 2006

Index must be updated as new documents arrive



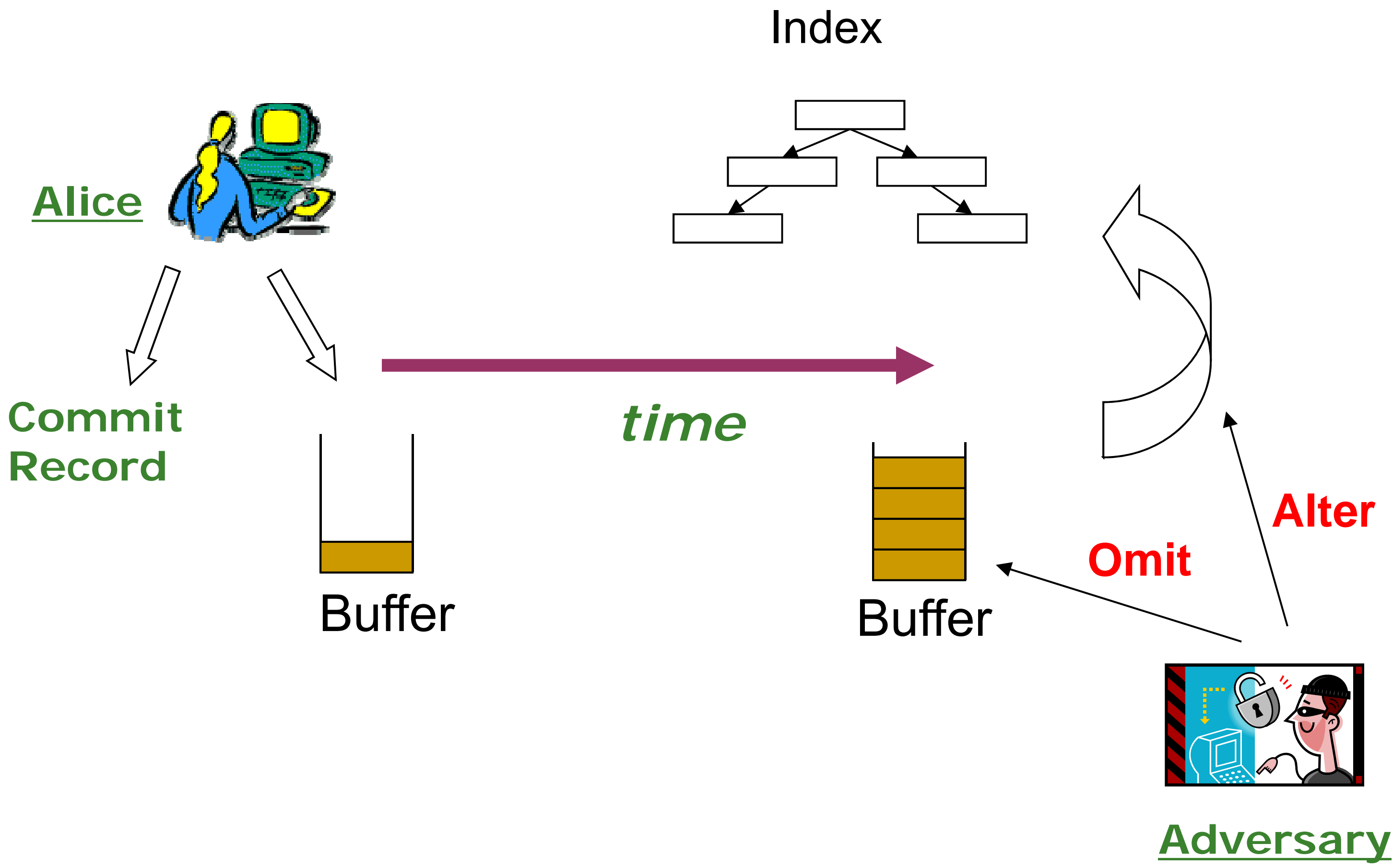
- 500 keywords = 500 disk seeks
  - ~1 sec per document

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

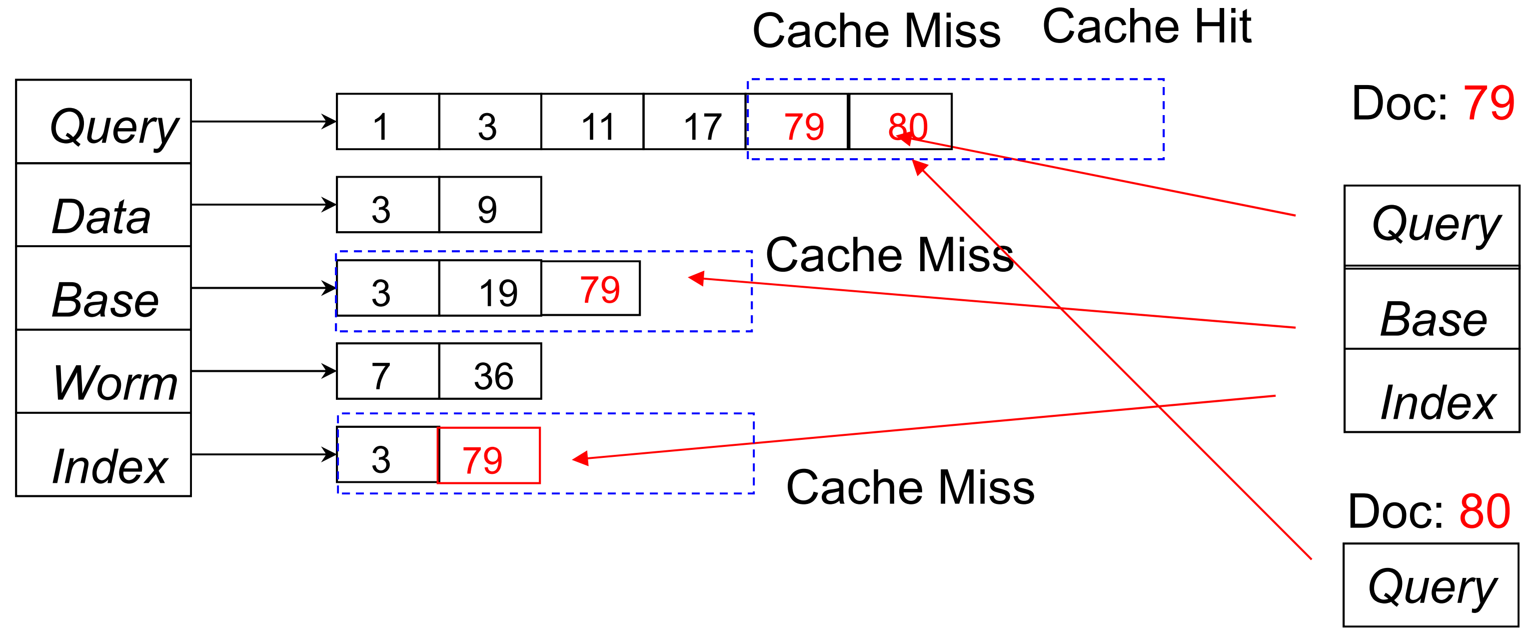


- **Prevailing practice – email must be committed before it is delivered**

Can storage server cache help?

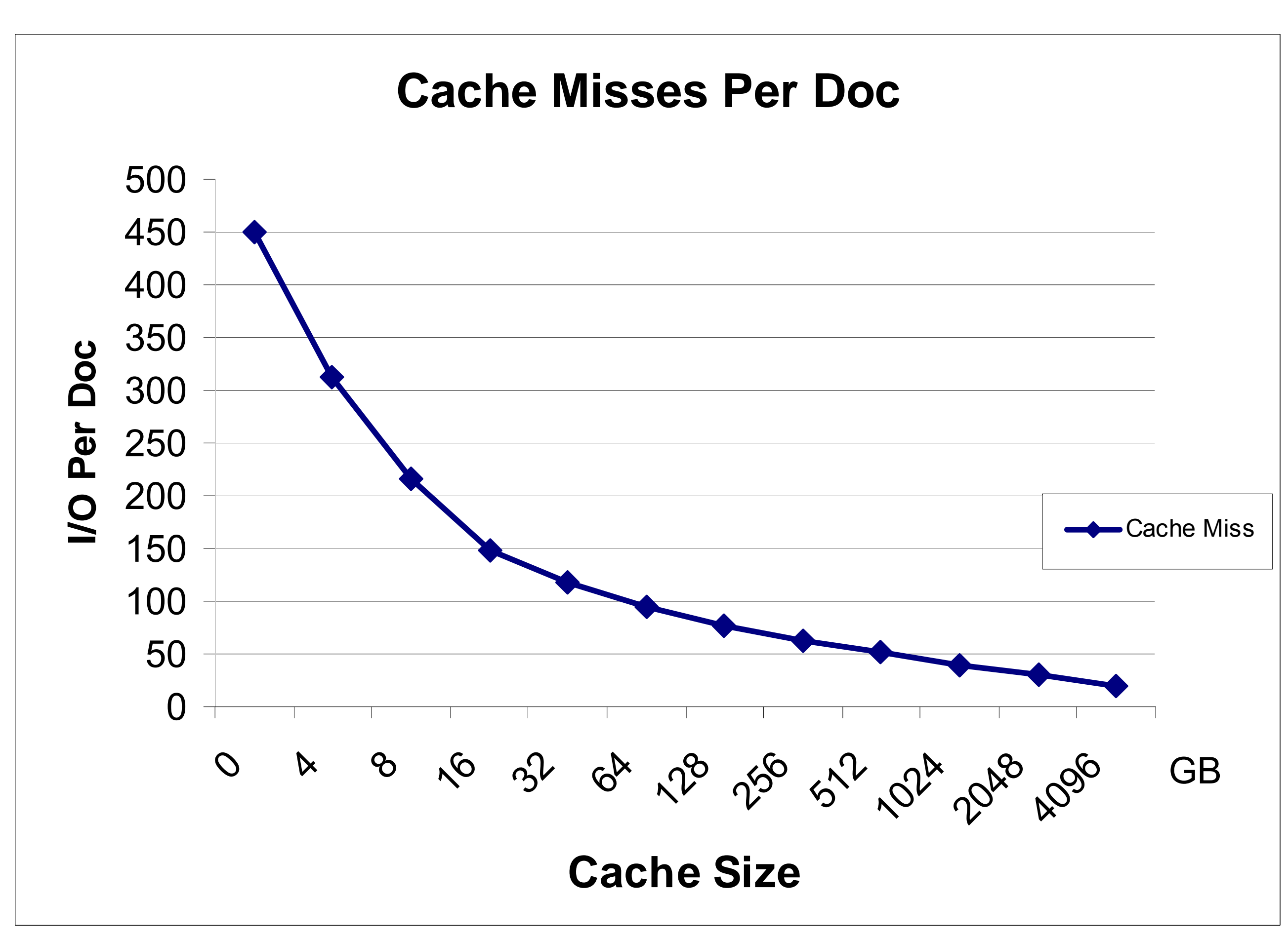
- Storage servers have huge cache
- Data committed into *cache* is effectively on disk
  - Is battery backed-up
  - Inside the WORM box, so is *trustworthy*

Caching works in blocks (One block per posting list)



- Caching does not benefit infrequent terms (number of posting lists >> number of cache blocks)

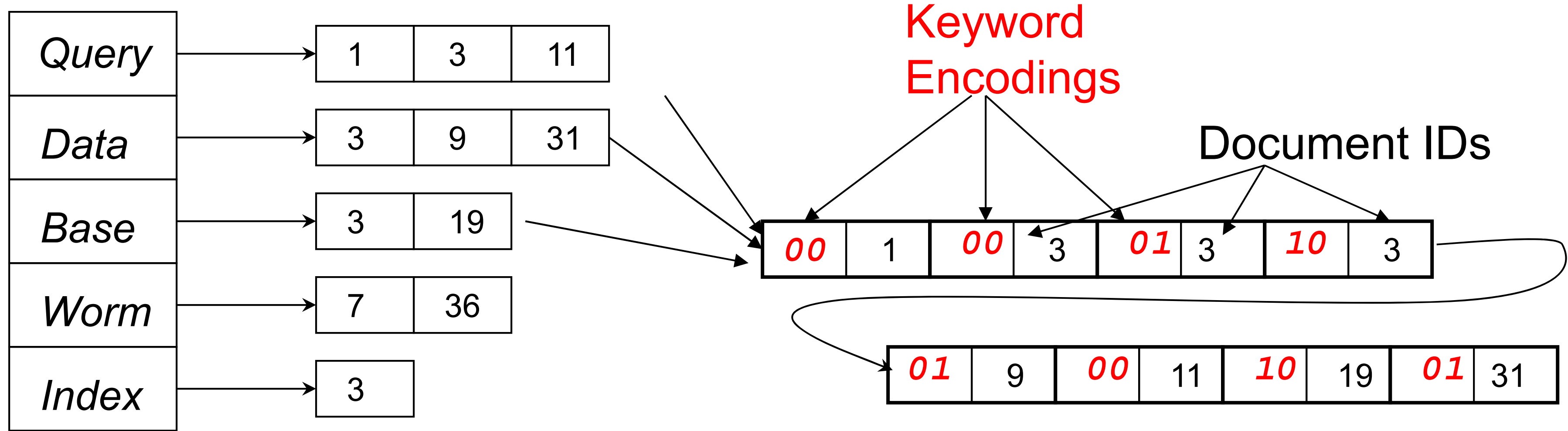
Simulation results show caching is not enough



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

- Que… …nning posting lists o… …ry

length of posting list for keyword *w*

# of times w is queried in workload

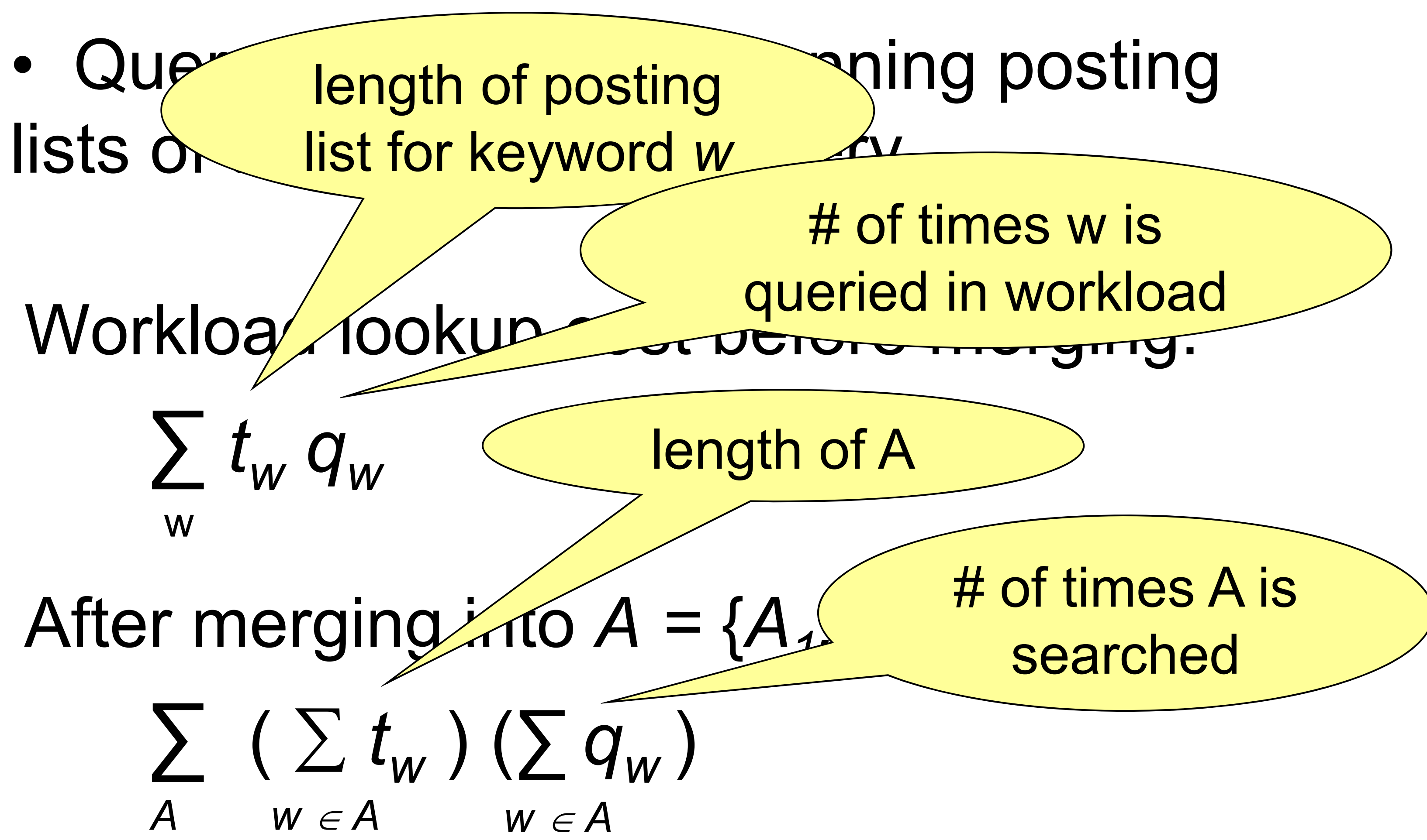
Workloa… lookup … before merging:

$$\sum_{w} t_w \, q_w$$

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

$$\sum_{A} \Big(\sum_{w \in A} t_w\Big)\Big(\sum_{w \in A} q_w\Big)$$

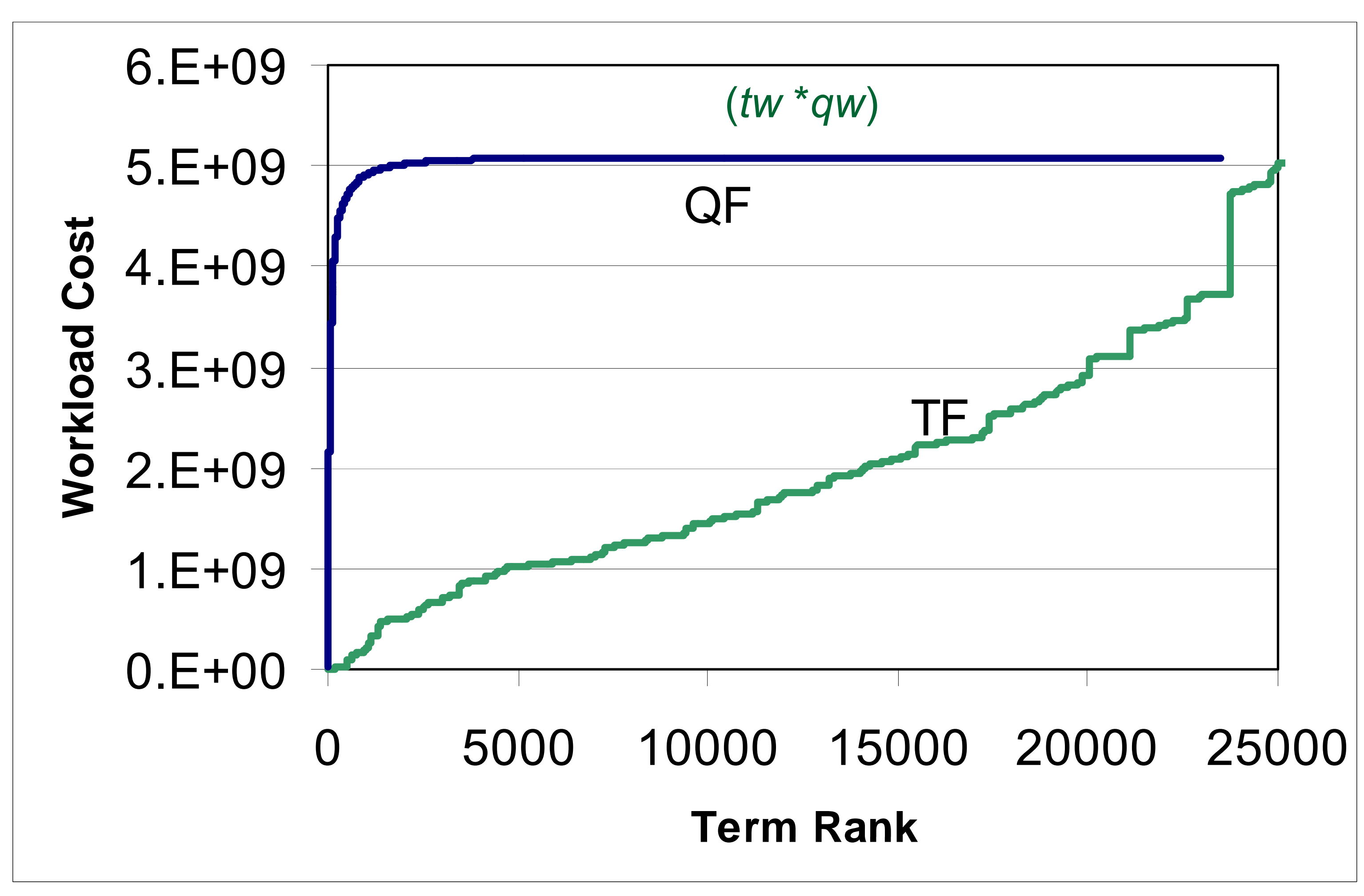
The tradeoff is longer lists to scan during lookup



Which lists to merge?

- Choose $A=\{A_1, A_2 .. A_n\}$
  - n = Cache blocks
  - Minimize $\sum \ (\sum t_w) * (\sum q_w)$
- 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

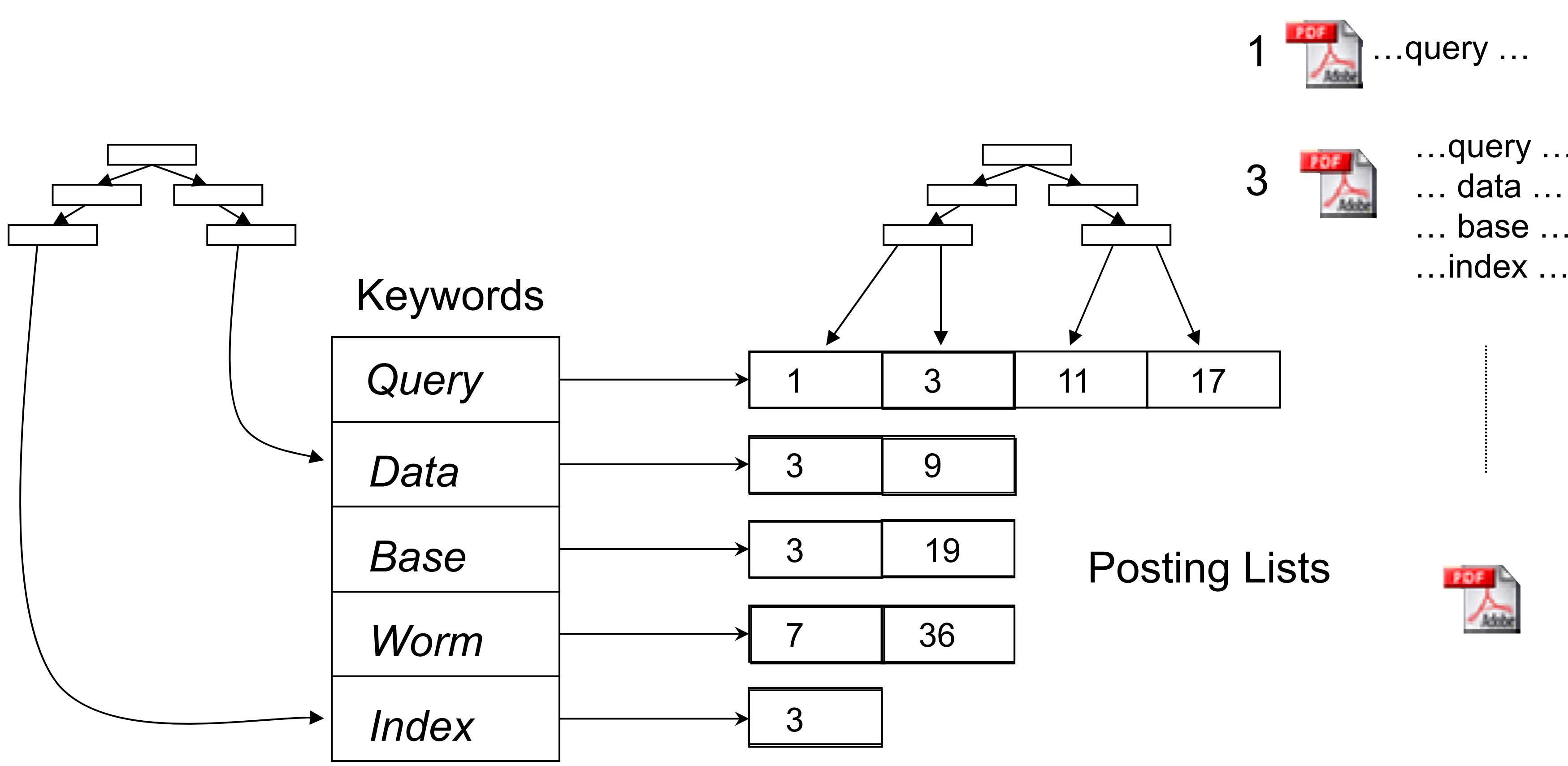
A few terms contribute most of the query workload cost



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

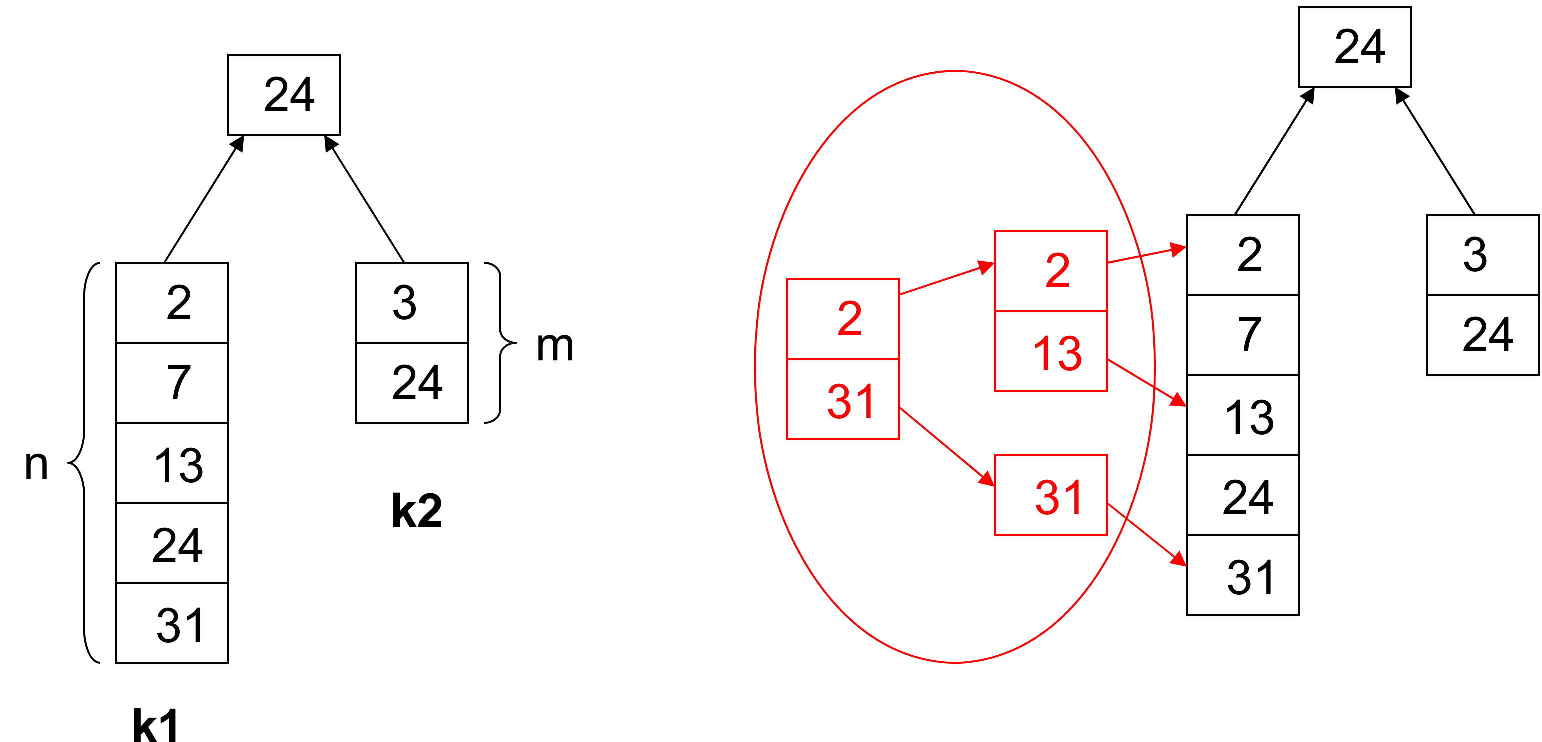
Several levels of indexing …



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., k1 AND k2) quickly



Merge Join : $O(m+n)$

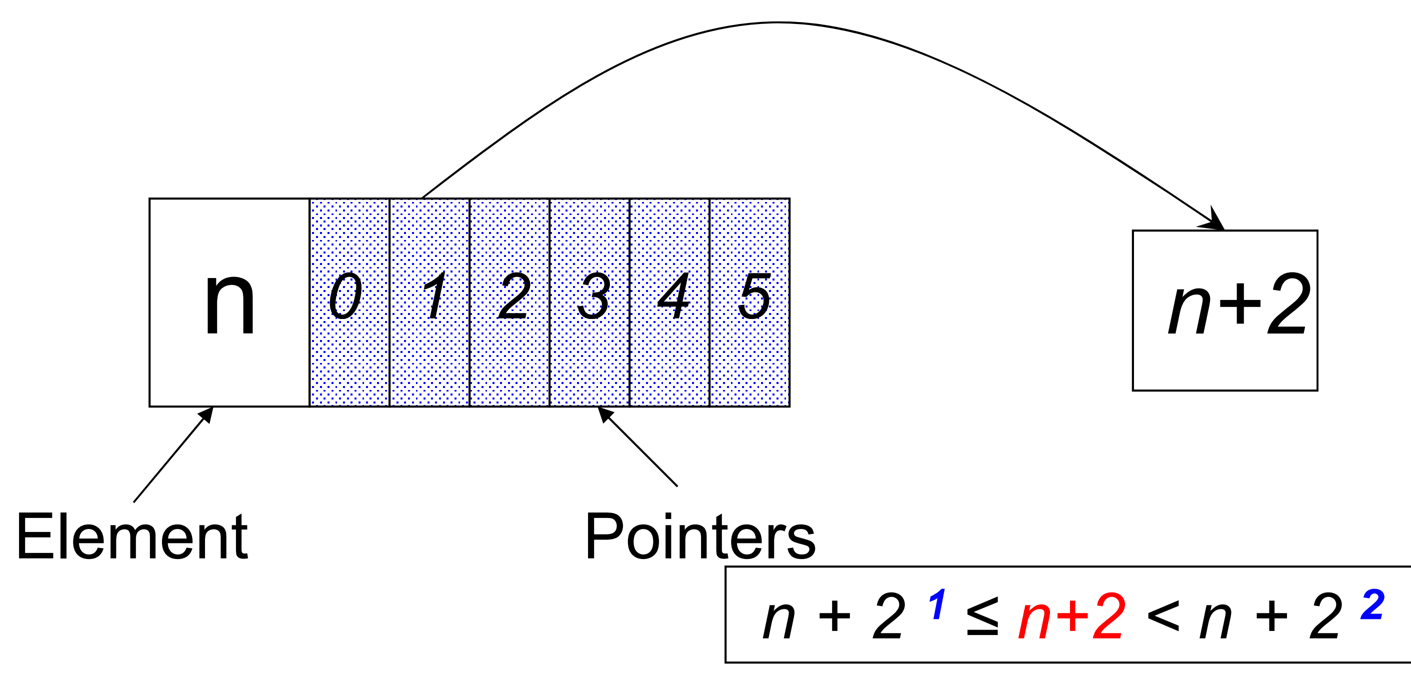
Index Join : $m \log(n)$

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

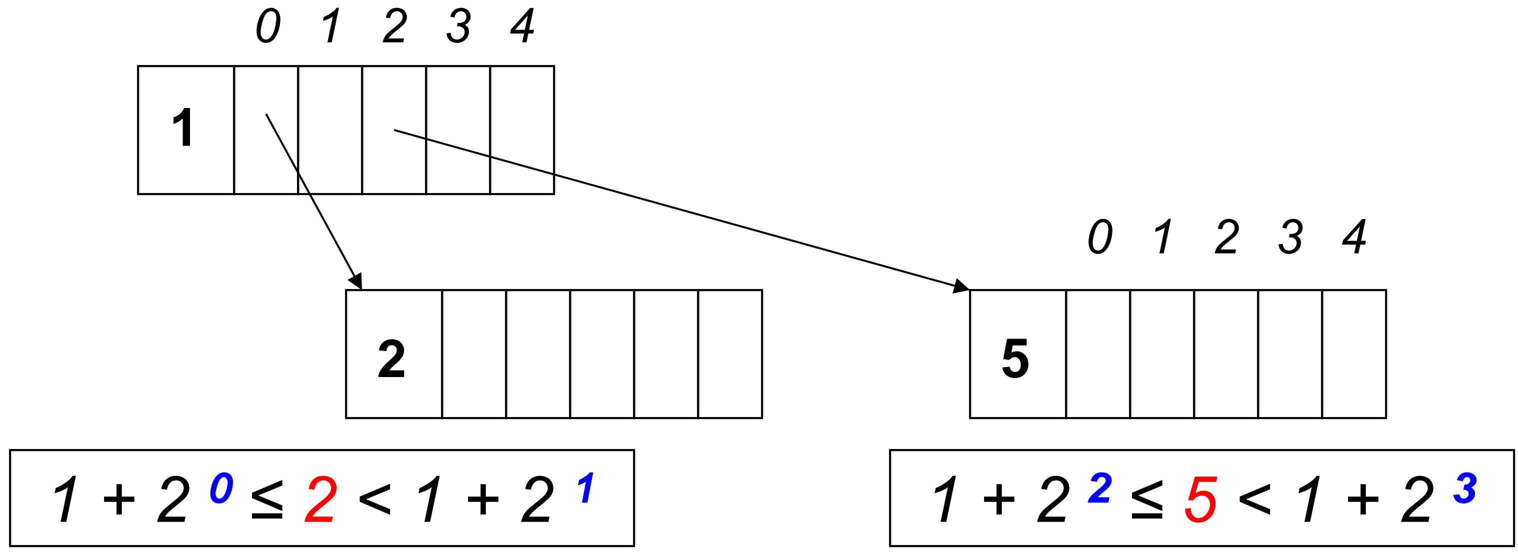


- i^{th} pointer points to an element n_i

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

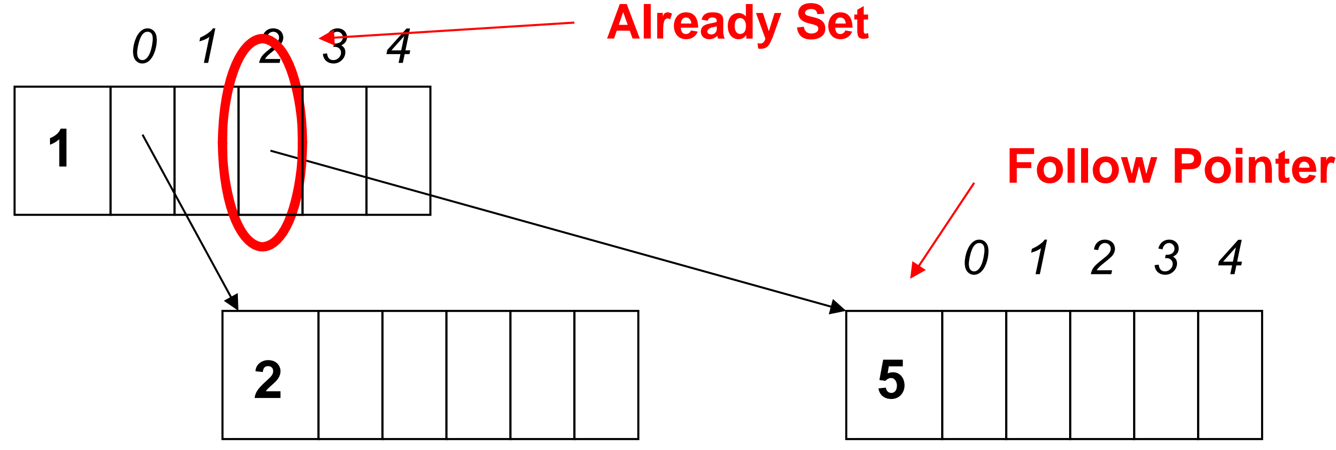
Jump index in action

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



Jump index in action

$n + 2^{i} \le n_{i} < n + 2^{(i+1)}$

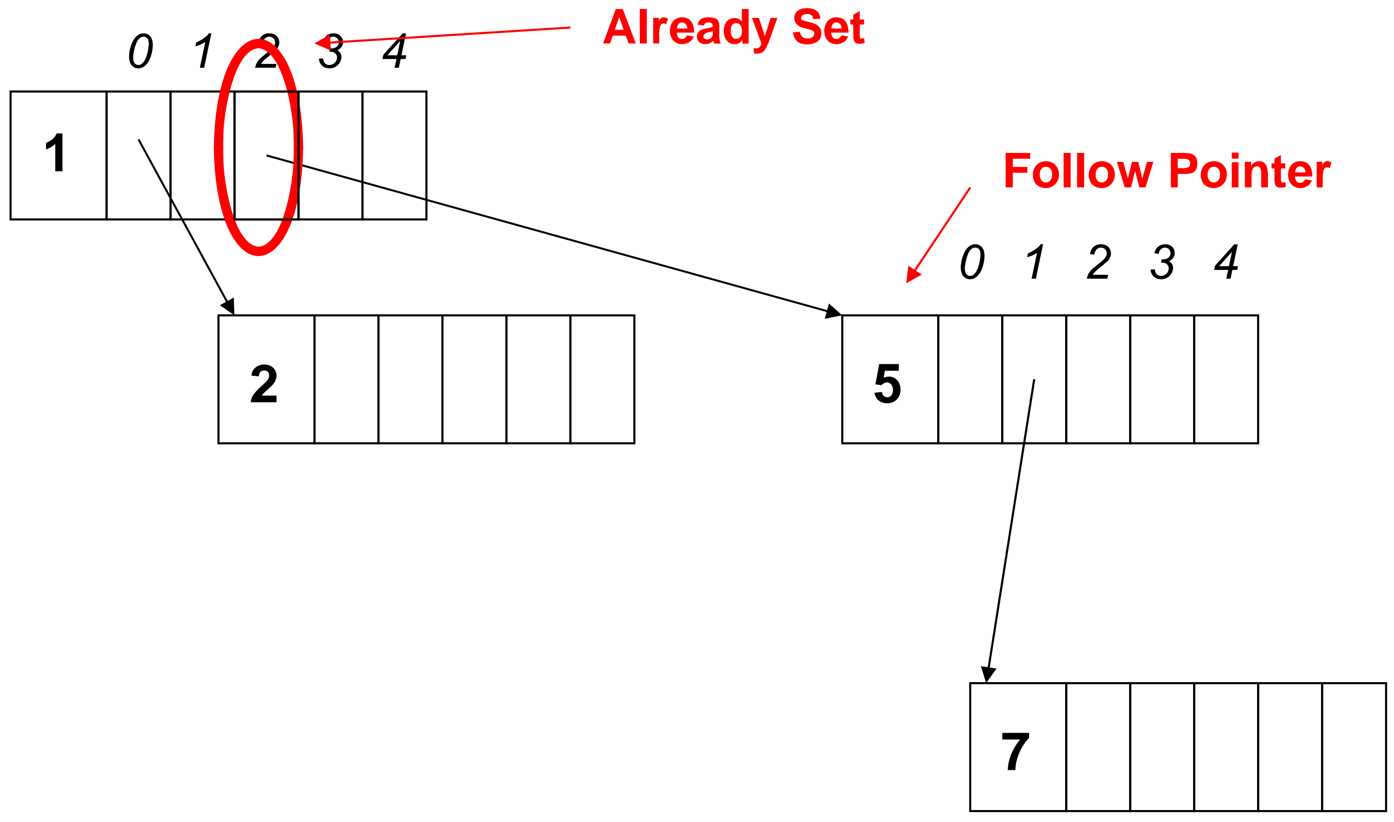


$1 + 2^{2} \le 7 < 1 + 2^{3}$

7

Jump index in action

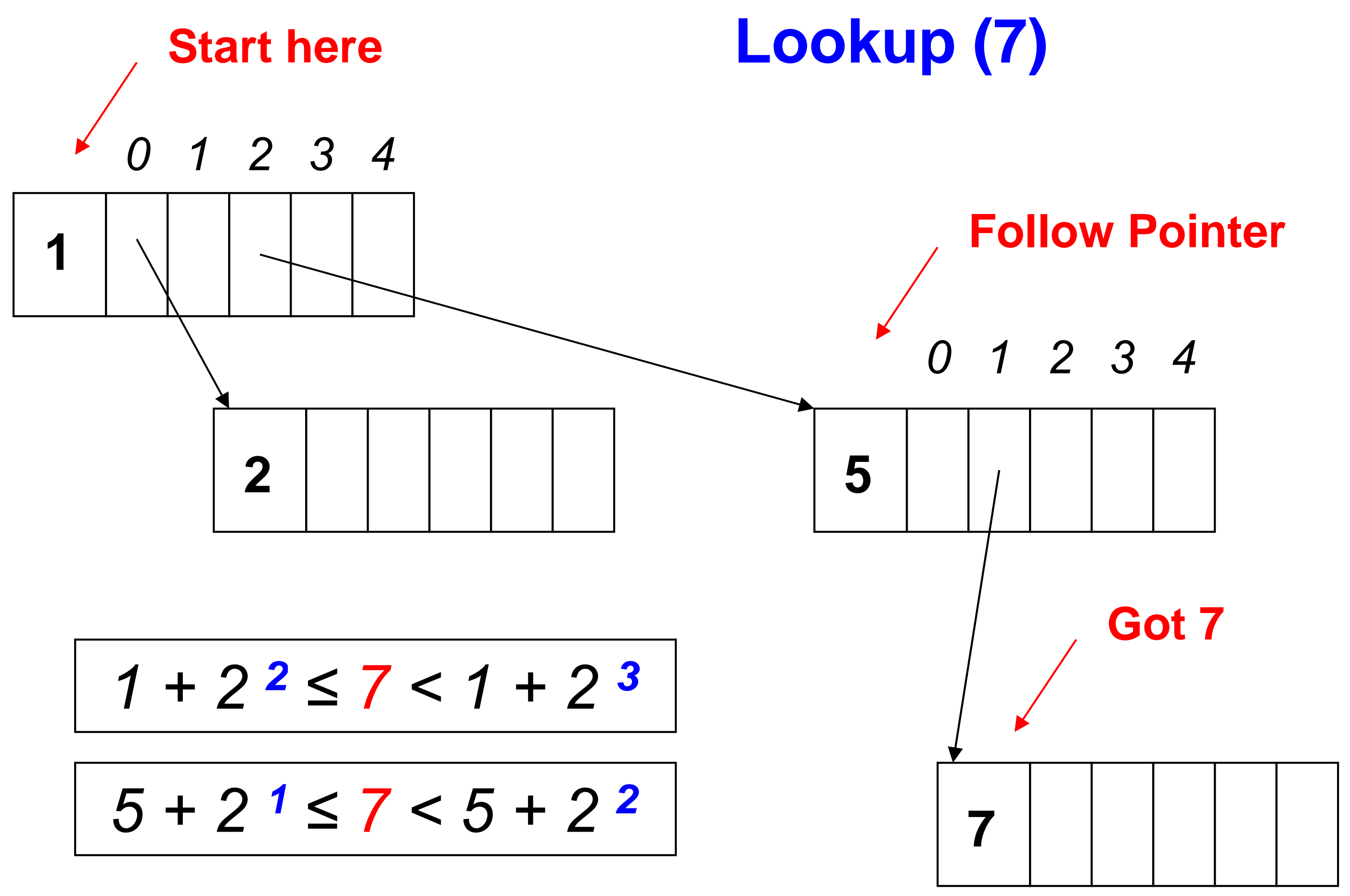
$n + 2^i \leq n_i < n + 2^{(i+1)}$



$5 + 2^{1} \leq 7 < 5 + 2^{2}$

log(N) pointers to N

Path to an element does not depend on future elements

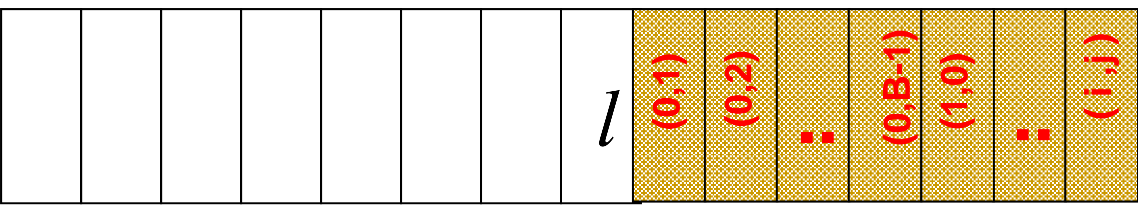


Block-based Jump Index

- 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*

$$l + j*B^i \leq x < l + (j+1)*B^i$$

p entries *Jump Pointers*



Jump index elements are stored in blocks

$$l + j*B^i \leq x < l + (j+1)*B^i$$

$P=4$ entries, B = 3

1	2	5	7	(0,1)	(0,2)	(1,1)	(1,2)	(2,1)	(2,2)

Block 0

$7+1*3^0 \leq 8 < 7+2*3^0$

$7+2*3^2 \leq 25 < 7+3*3^2$

Block 1

8	10	15	19	(0,1)	(0,2)	(1,1)	(1,2)	..	(i,j)

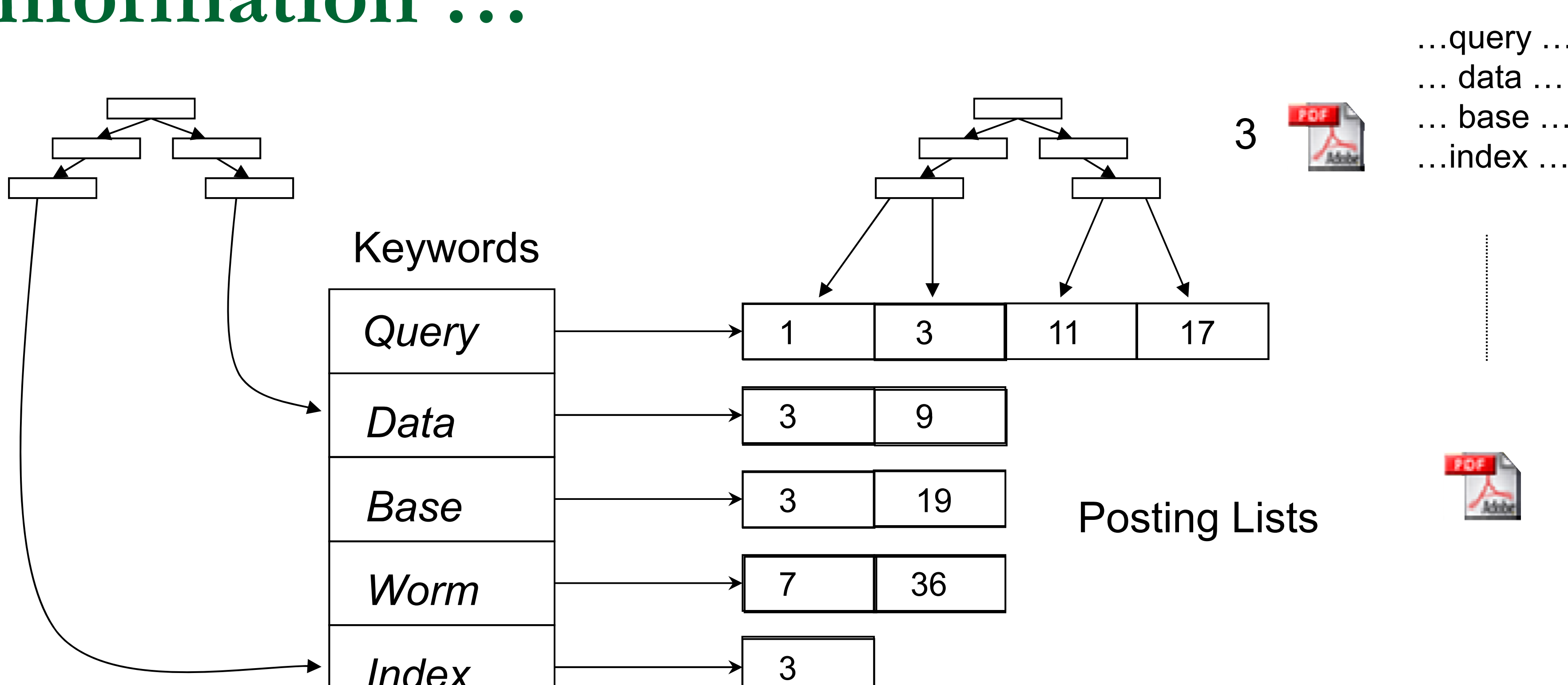
Block 2

21	22	25		(0,1)	(0,2)	(1,1)	(1,2)	..	(i,j)

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.

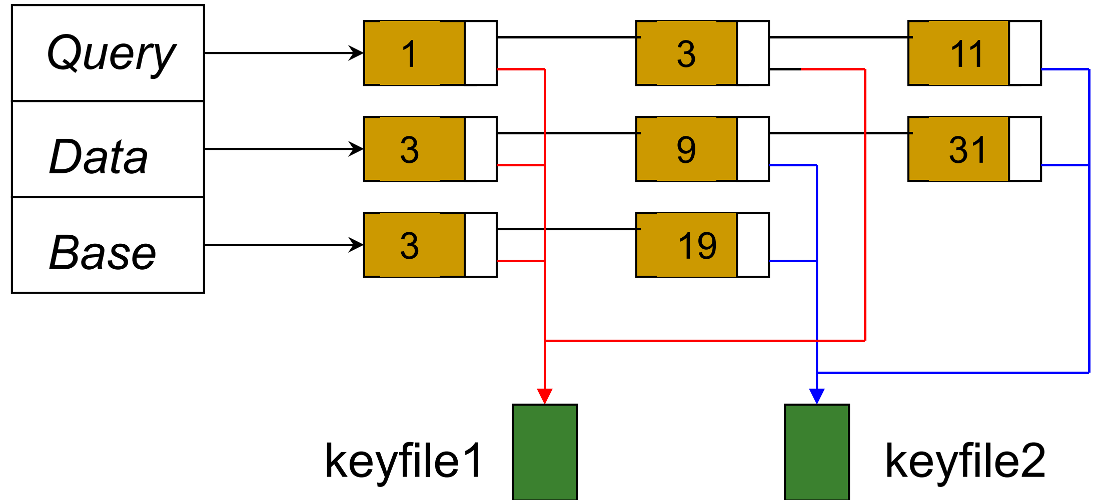
S.Mitra, M. Winslett: Secure Deletion from Inverted Indexes on Compliance Storage. Proceedings of the 2006 ACM Workshop On Storage Security And Survivability, StorageSS 2006, Alexandria, VA, USA, October 30, 2006, pp. 67-72.

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.

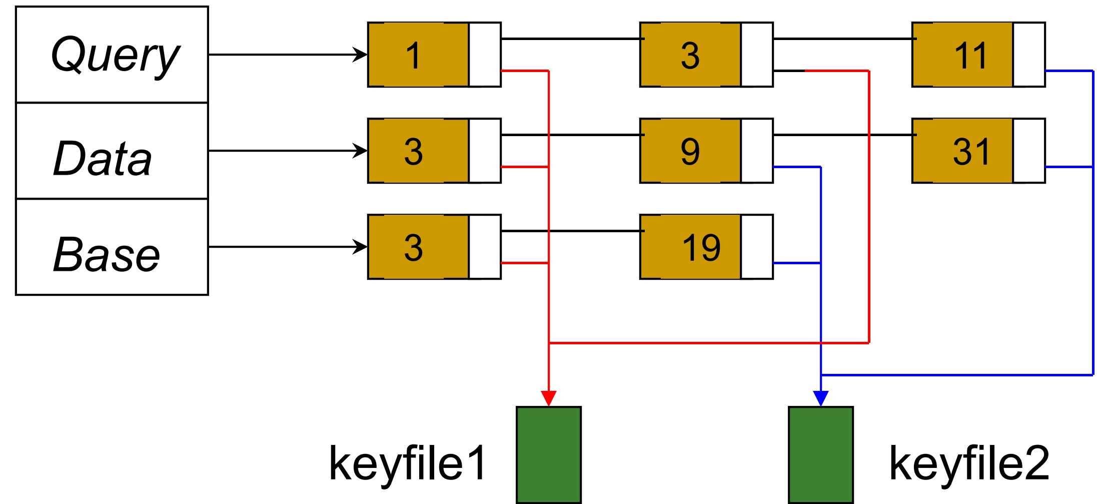


Encrypted inverted index

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 keyfile1 expires, just need to erase keyfile 1



Encrypted inverted index

- 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

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