

# UpBit: Scalable In-Memory Updatable Bitmap Indexing

Anwasha, Asianna, Can, Ross, Toby, Tommy








# Presenters





# Brief Overview

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	1
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

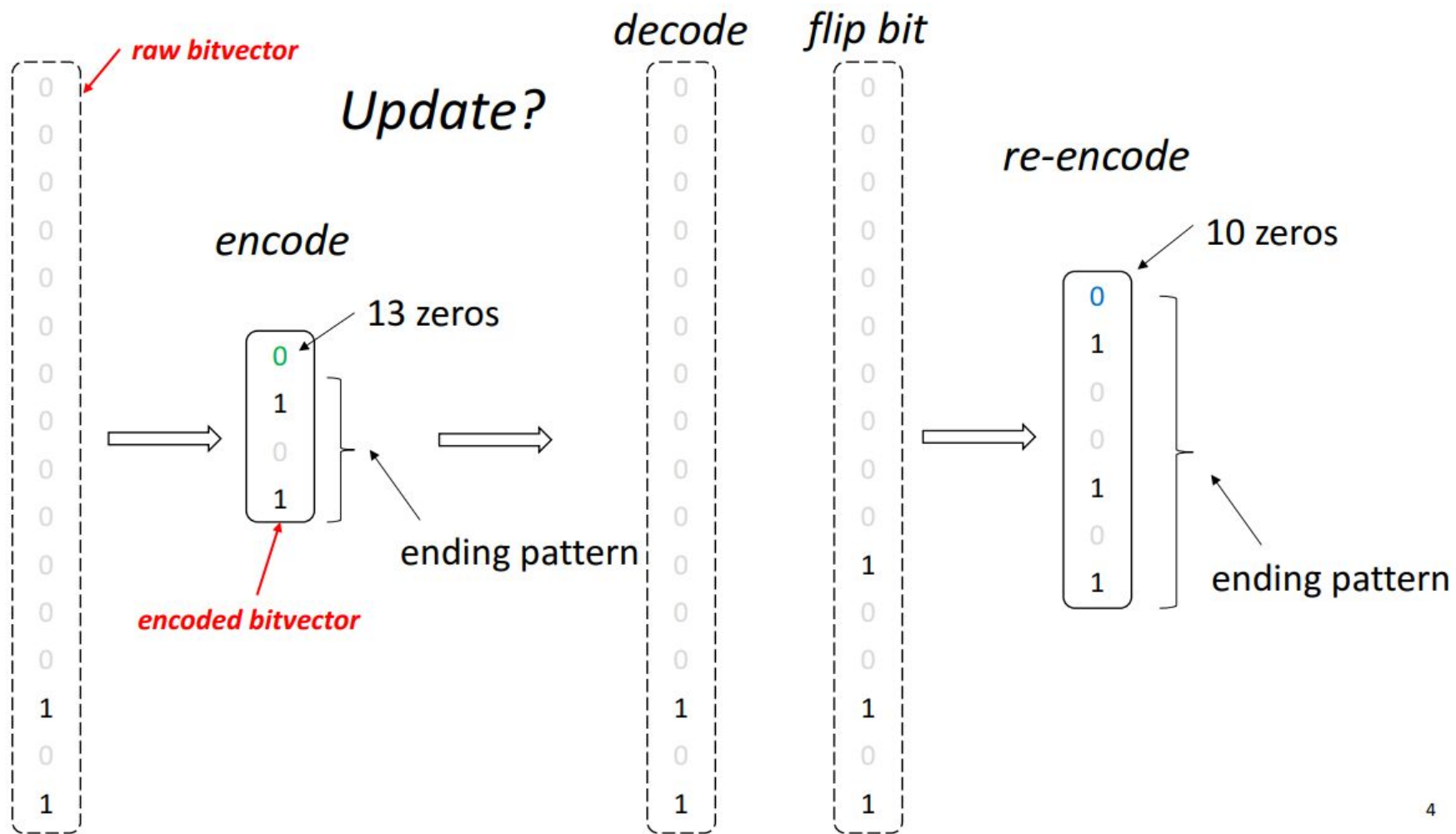
- Bitmap Indexing
  - Read 
  - Update 
  - Memory 
- Why?
  - Compression
    - {en,de}coding
  - Fast Bitwise Operations
    - c.f. SIMD



# Basic Bitmap Design

- Bitmap Indexing
  - For each **distinct value** in the data domain, we have a **bit vector**
- Operation cost
  - read: 1 decode
  - update: 1 decode + 1 encode

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	1
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

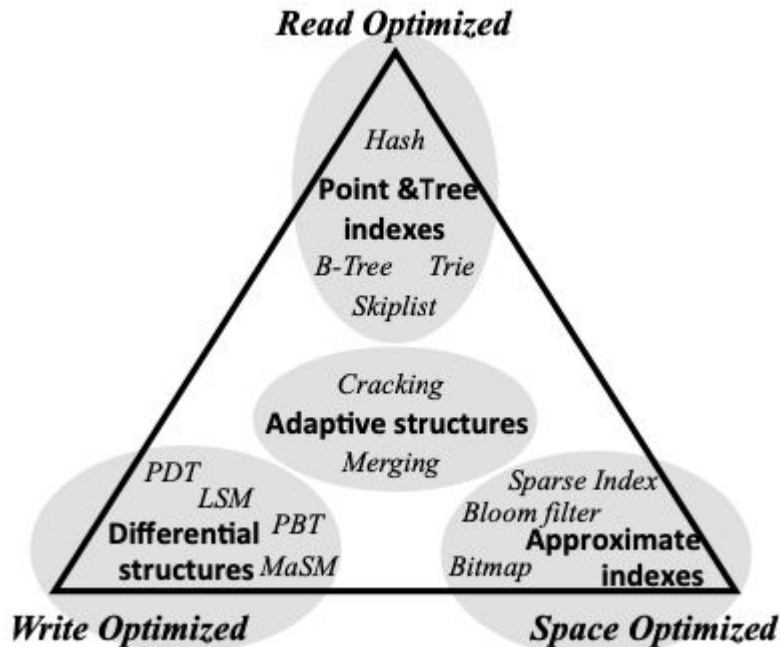




# Recall: RUM Conjecture

What we want:

- shift more to left
- auxiliary data structs
  - reduce friction of R/W
  - distribute load: currently have single bit vector
    - better compression
- Still space efficient
  - periodic compaction/merging





# Problem

How do we make bitmaps efficient for both reads and updates?



# Solution

Update Conscious Bitmaps (UCB)





# Update Conscious Bitmaps (UCB)

Update Conscious Bitmaps (UCB), SSDBM 2007

A=10	A=20	A=30	EB
0	0	1	1
0	1	0	1
0	0	1	1
1	0	0	1
0	1	0	1
1	0	0	1
0	0	1	1
0	1	0	1

Main Component: **Existence Bitvector (EB)**

The **existence bitvector** determines whether or not the entire row is **valid** or **invalid**.

**Instead of updating** the data in-place, we will now **append the new version** to the end.

# UCB Update

rid	10	20	30	EB
1	0	0	1	1
2	0	1	0	1
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
Pad	0	0	0	0

update  
row 2 from  
20 to 10

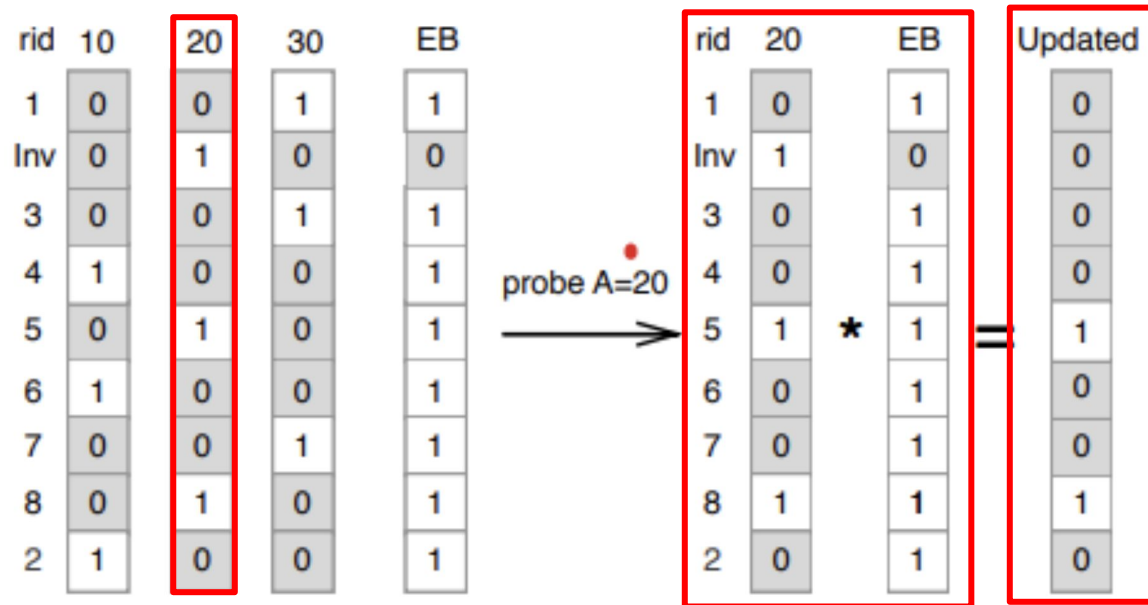
rid	10	20	30	EB
1	0	0	1	1
Inv	0	1	0	0
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
2	1	0	0	1

Mark for deletion then  
append

(a) Update value of second row from 20 to 10 using UCB.



# UCB Search





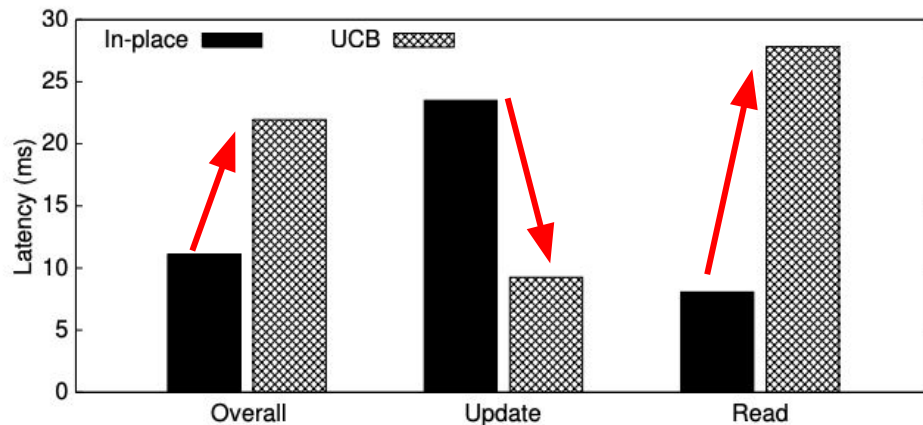
Bitwise AND  
between VB and EB

(b) Probe for value B using UCB.



# MAJOR FLAW

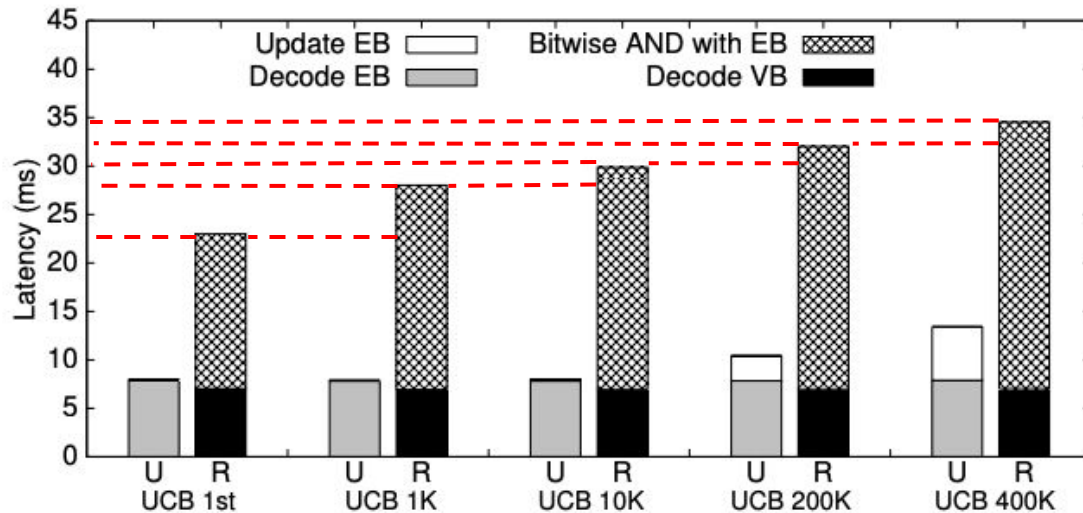
- More Update → queries = less compressible
- Now the situation is reversed
  - Read 
  - Update 





# Why does latency increase if we have more updates?

- Cost of  $\text{VB} \wedge \text{EB}$  increases as more rows becomes **invalidated**
- Increased average read latency





# How can we address this issue?



**SOLUTION: UPBIT**

Distribute the update cost

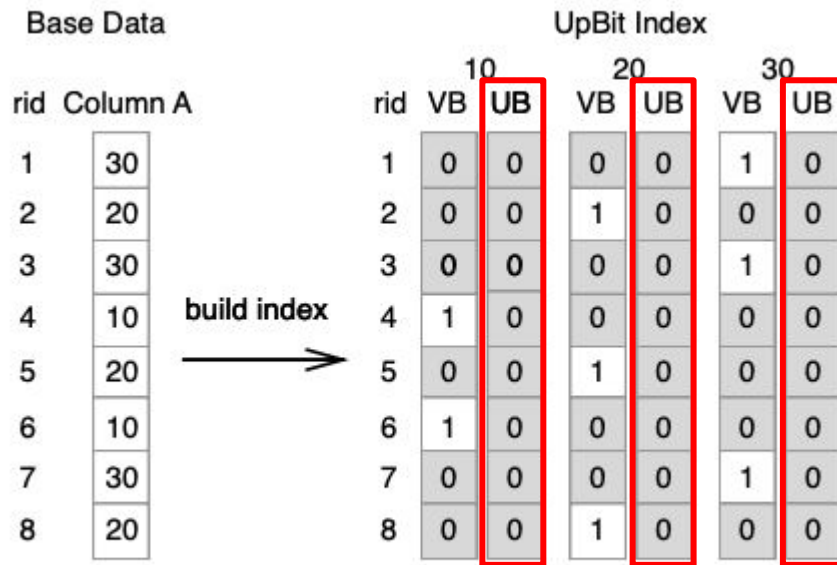
Efficiently access certain portions of the compressed bitvector





# Update Bitvectors (UB)

- Instead of having **only one EB**, we now have a **UB for each value of the domain**
- Each value in UB is initialized to 0
- Now the actual value is calculated using  $VB \oplus UB$
- Keep a counter of bits set to 1 in each UB, if 0 skip XOR altogether



(a) The internals of UpBit.

## Example: Updating

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

update  
row 2 from  
20 to 10



	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	1	1	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

1. Use Value Bitvector-Mapping (VBM) to find the correct row.
2. Flip bit of row 2 of UB of A = 20
3. Flip bit of row 2 of UB of A = 10



## Example: Deletion

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

→  
delete row 2

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

## Example: Insertion

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

→  
insert value 20

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
9	0	0	1	0	0	0



# Problem

We still need to decode the entire bitvector before we can obtain the value of a specific row.

Can we improve this?

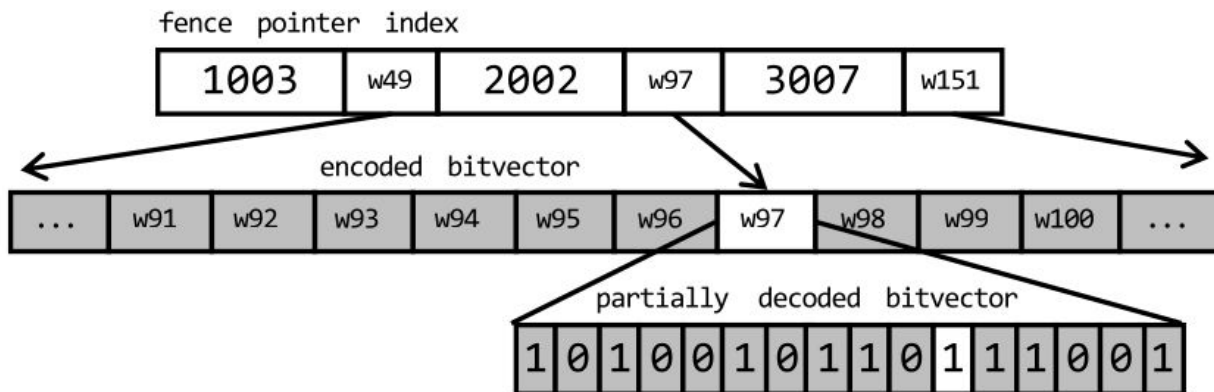


# Solution

Fence Pointers

# Fence Pointers

- Enables **efficient partial decoding** close to any part of the bitvector.
- map: unencoded\_word -> encoded\_word
  - decode only the word





# Example: Fence Pointers Walkthrough

Operation:

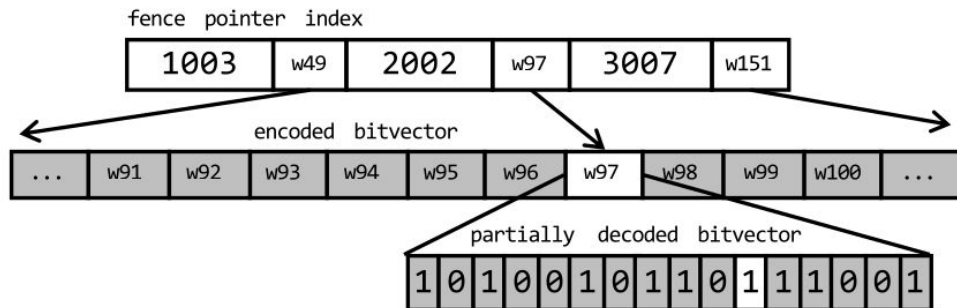
- FIND bit 62073

Math:

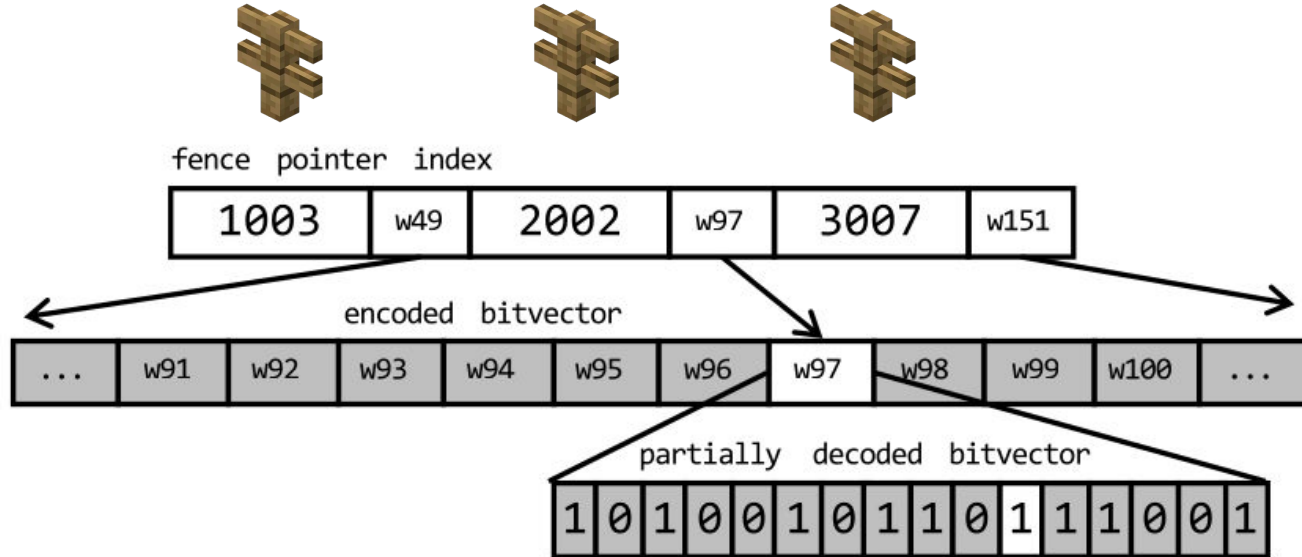
- unencoded word =  $62073 / 31 = W2002$
- pos =  $62073 \% 31 = 11$

Fence:

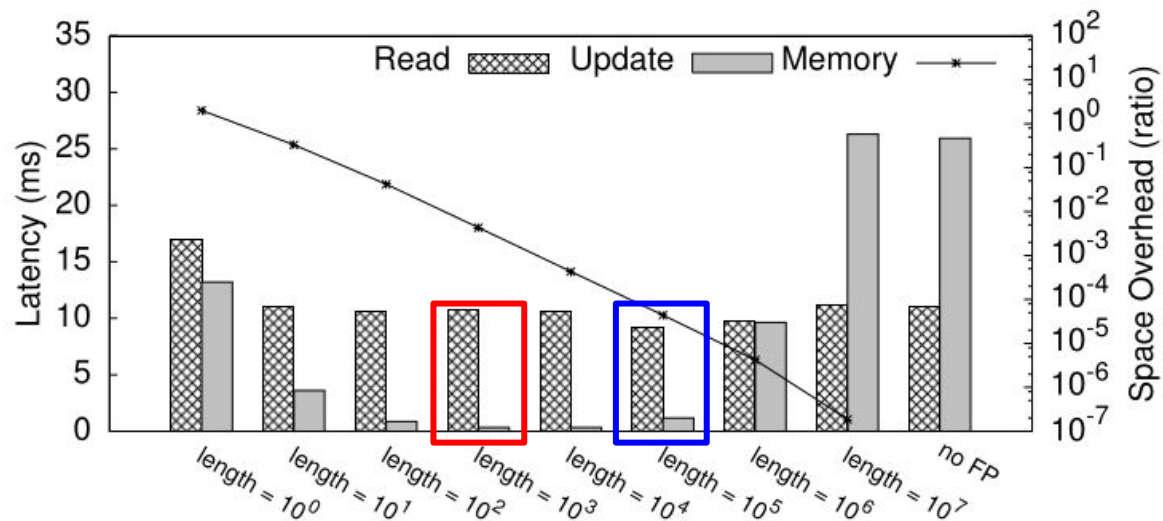
- W2002 -> w97 (encoded word)



Q: Find bit 62150?  $62150/31=W2004$



# Fence Pointer Granularity



- triangular relationship
  - c.f. RUM
  - not an equilateral triangle!
- tune granularity based on expected workload
  - min write  $10^3$
  - min read  $10^5$





# Problem

As updates stack, UB becomes less compressible.

How do we fix this?



# Solution

Merging



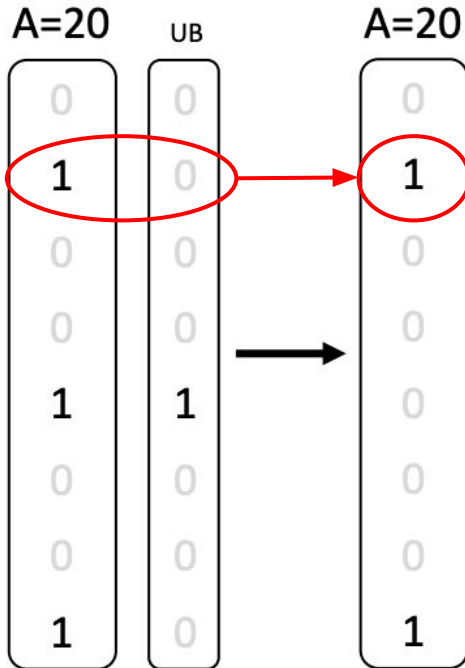
# Merging

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

1. **Merge** each **UB** with the corresponding **VB** **once we reach a certain threshold** of updates.
2. Once the threshold is reached we mark that bitvector as **"to be merged"**.
3. The **next time we perform a search** using that bitvector we **update the VB as well** since we are already calculating  $VB \oplus UB$ .
4. Once finished UB is **reset back to all 0s**.



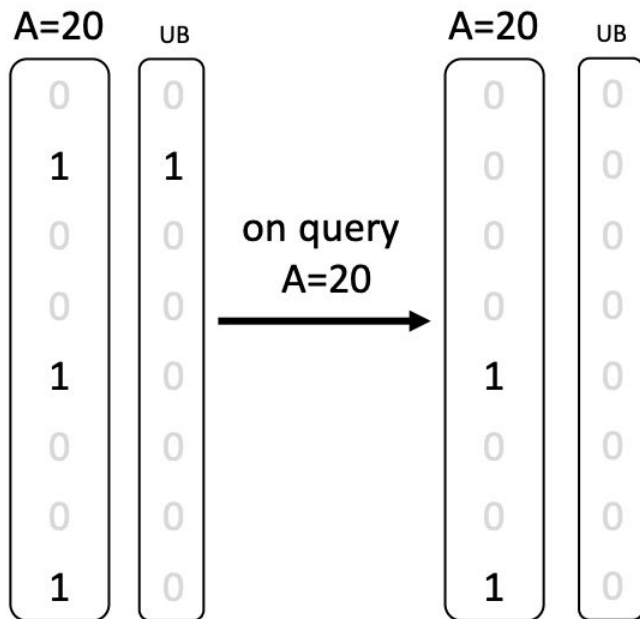
# Merging



Here we perform an XOR between the VB and UB and set the new VB equal to that result



# Merging



Merging is only performed once we receive a query using that VB in order to reduce overhead, since we are already performing that calculation in the search



## Why doesn't this work with UCB?

- Since we split the work of the existence bitvector between multiple update bitvectors, we only need to XOR a portion of the existence bitvector amount.
- If we try to apply this method to UCB we will need to merge the existence bitvector with every single value bitvector, instead of only a portion of it.



# Experimental analysis



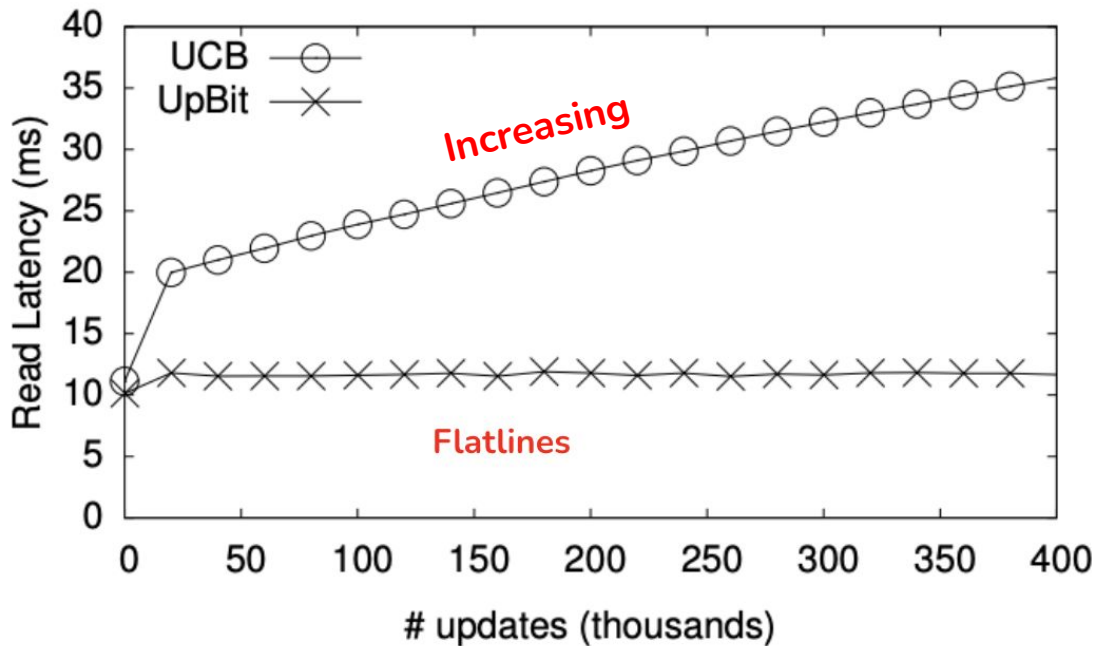


# Experimental Analysis

- **Synthetic Data Variables:**
  - $n$  = # of tuples/ size of dataset
  - $d$  = cardinality of domain/
- **Distribution of data**
  - Uniform distribution
  - Zipfian distribution: generates skew
- **Standalone implementation in C++ built upon the Fastbit bitvector**

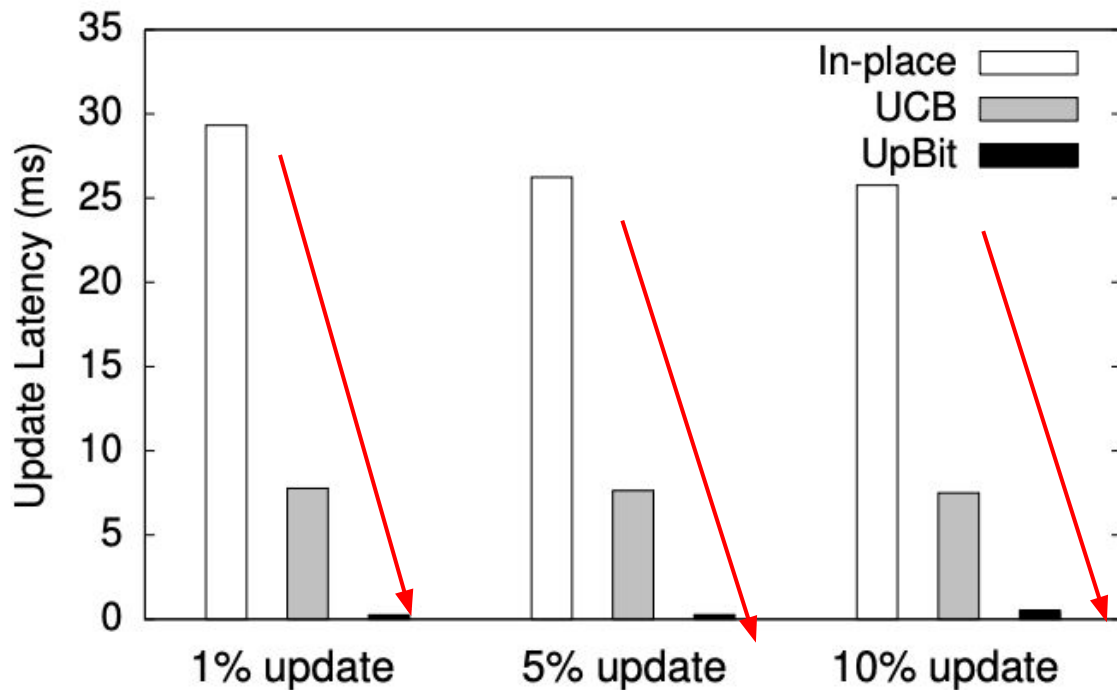


# Scalable read



- Most important limitation of UCB
- **UCB** does not provide this
  - EB becomes less compressible
- **UpBit** does!
  - Recall: distributed UBs, better compressibility

# UpBit supports fast updates



$n = 100M, d = 100$

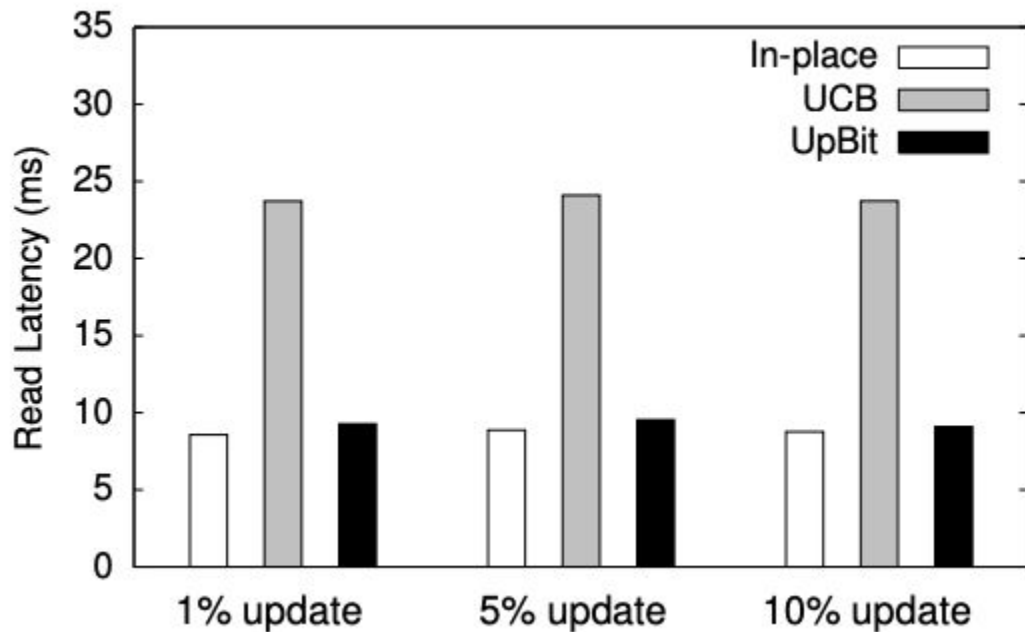
query mix of 100k operations

UCB is 3.43-3.77x faster than in-place

UpBit is 51-115x faster than in-place



## UpBit vs. read-optimized



$n = 100M$ ,  $d = 100$

query mix of 100k operations

UpBit outperforms UCB by 3x,  
but loses to read-optimized  
indexes by 8%.



# Multithreading

Why so much faster?

- distributed UB
- note: serial execution
  - no locking for protected updates in these experiments

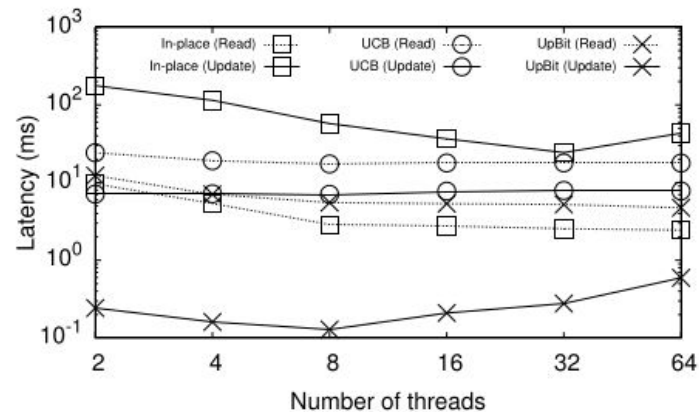
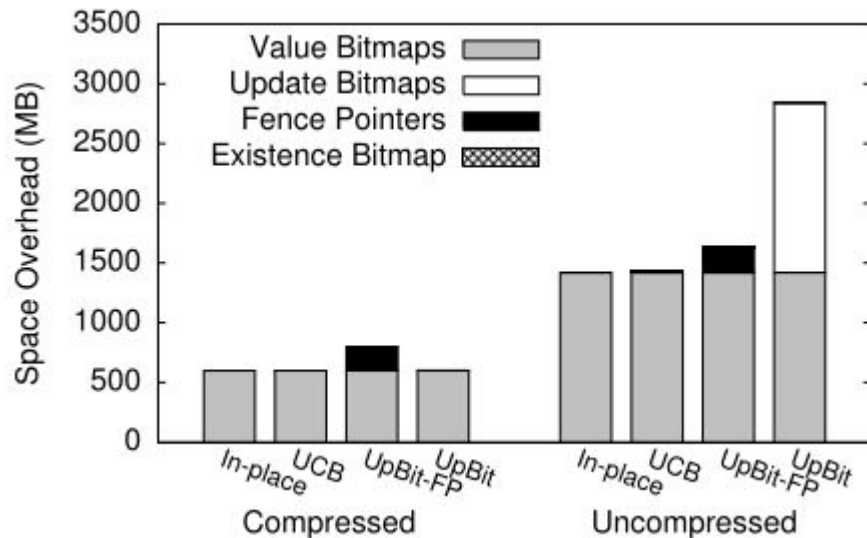


Figure 22: Updates with UpBit are two orders of magnitude faster than other approaches and scale for up to 8 threads.



# Space overhead: auxiliary structs

- Compressed very close
- Notice:
  - good compression on UB





# Design Choices (how it differs from traditional system)



1. Distribute the update cost
  - a. **Update Bitvectors**
2. Efficiently access certain portions of the compressed bitvector
  - a. **Fence Pointers**



# Goals of the Architecture

1. Make bitmaps efficient for both read and update queries.
2. Avoid unnecessarily encoding and decoding the entire bitvector.
  - a. FP
3. Make the bitvectors more compressible as updates stack.
  - a. on-the fly merges during reads
  - b. query-driven absorption of updates





# Tuning knobs

1. UB-VB merging threshold
2. FP granularity
3. # threads





# Critics and Proponents





# Overheads

Can we justify the overheads UpBit imposes by using the results it achieves?



# Storage Overhead and Balance

*“UpBit achieves efficient and scalable updates, while allowing for comparable read performance, having up to 8% overhead.”*

- 8% is justified? How?
- Effects of skewness?



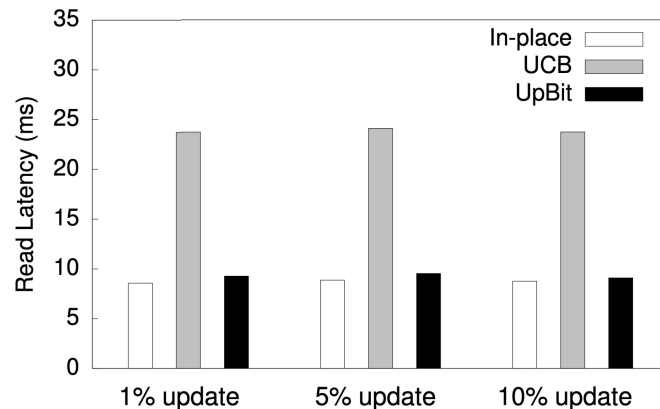
# Storage Overhead and Balance (Response)

8% is justified? How?

- **Significant** improvements in update performance (51x-115x)
  - overall beneficial for any sort of mixed workload

Effects of skewness?

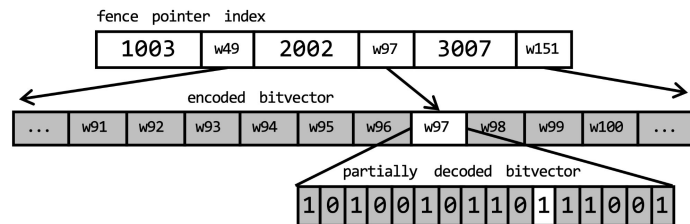
- Empirically, overhead remains stable at 8%
  - overhead due to XOR operation + fence pointers
  - merges mitigate UB growth
  - fence pointers scale well with data





# Impact of Fence Pointer Overheads

- Fence pointers to **minimize** decoding overhead
- **Direct** access
- Is the space overhead **worth** it?





## Impact of Fence Pointer Overheads (Response)

- Space overhead in full UpBit: 0.5% (fence pointers) + 15% (update bitvectors)
- Mainly from update bitvectors, so they are worth it.



# **Resilient to Change?**

## **Robust?**



# Resilience to Different Scenarios

Performance in **OLAP** (range queries) vs. **OLTP** (short queries with point reads + random infrequent updates)?





# Resilience to Different Scenarios (Response)

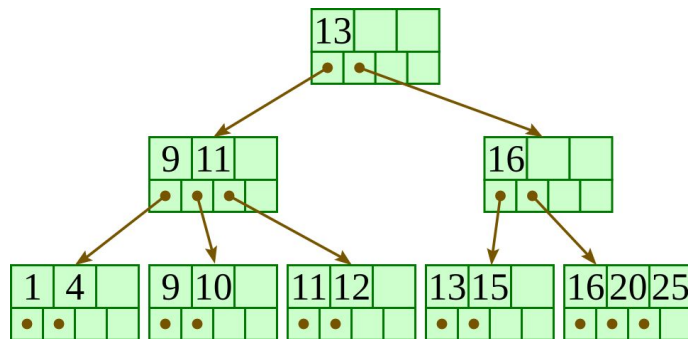
Performance in OLAP vs. OLTP?

- *UpBit incurs 8% read overhead vs. in-place updates*
  - *detrimental for point reads in OLTP*
- *Bitmaps are typically designed for OLAP workloads*
  - *optimized for (selective) range queries & updates*
  - *take advantage of efficient bitvector operations on columns*



# UpBit vs B+ Tree as Index

- **UpBit** is great! -efficient queries and space efficiency
- Why are **B+ trees** used then?





# UpBit vs B+ Tree as Index (Response)

Considering the benefits of UpBit, why are B+ trees so predominantly used as indexing structures over bitmaps considering how space efficient they are and how easily we can perform queries?

- Use cases are different
- Bitmaps are for low-moderate cardinality
- Sparse data - selectivity queries
- Very good for OLAP queries



# **Design Decisions (UpBit vs UCB)**



## EB vs UB

- Seperate update vectors, helps **parallelism**
- What about one **combined** structure of update vectors?
- **Drawbacks?**
- A **hybrid** approach?

rid	UpBit Index					
	10		20		30	
	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0



## EB vs UB (Response)

Combined structure? Hybrid approach?

- *Major complexity increase*
- *Might lose/ complicate parallelism*
- *What is the benefit?*
  - *Would need to build & evaluate to see any significant benefits*



# Update-only Workloads

- What if **only** updates, no reads?
- Would **UCB** be the same?



## Update-only Workloads (Response)

What if a workload just has updates? Would UCB work as good UpBit in this case considering it no longer has to pay the penalty of high read cost when updates are high which was its primary disadvantage.

Initially maybe, Eventually No - less compressible





## Delete efficiency

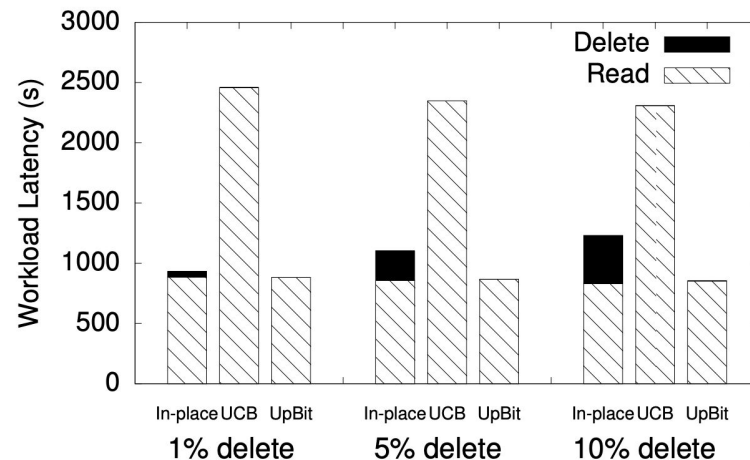
- Deletes are **fast** in UpBit
- **Slow** in UpBit due to EB becoming dense
- A **change** to UCB to make deletes as fast?

# Delete efficiency (Response)

- *UCB “delete” inefficiency stems from reads*
- *With many deletes, EB becomes dense*

A change to UCB to make deletes as fast?

- *Not simply— problem is fairly core to UCB*





## AND vs XOR

- UCB -> AND
- UpBit -> XOR
- Why is UpBit faster?



## AND vs XOR (response)

We know reading in UCB requires an AND between the value and existence bitvector, but UpBit uses XOR for reading. Why is UpBit faster? Could the choice of operation make a difference in terms of time?

- Traditional hardware XOR faster
- New hardware ~ no difference
- EB can quickly become dense whereas UB remains sparse by design
- Operating on sparse is



# **Tuning the Design (Frequency of XOR and fence pointers)**



## Timing the XOR

- Non-ideal merge frequency of UB and VB?
- Automate merge frequency → Use ML?
- Workload variance changes ideal “threshold” and frequency of fence pointers - do we assume constant workload? Skews?



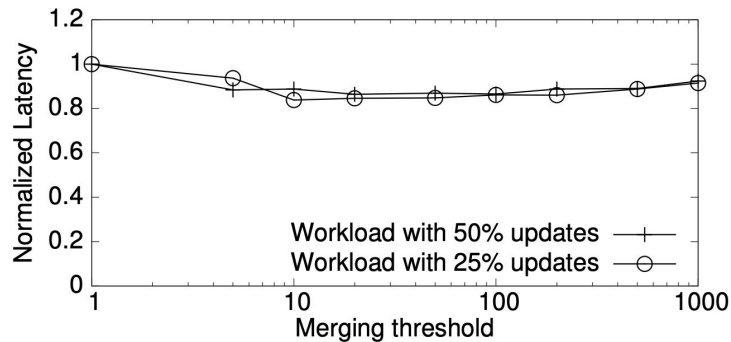
# Timing the XOR (Response)

What if the merging of the update bitvector (UB) and the value bitvector (VB) is not ideal?

- *Low frequency -> dense UB*
- *High frequency -> unnecessary operations*

Could we make the merge tuning automated?

- *Threshold value is fairly tolerant*
- *Dynamic threshold adjustment would need to be worth any overhead*





# Optimal Value of 'T'

**merge (index:  $UpBit$ , bitvector:  $i$ )**

---

```
1:  $V_i = V_i \oplus U_i$ 
2:  $comp\_pos = 0$ 
3:  $uncomp\_pos = 0$ 
4:  $last\_uncomp\_pos = 0$ 
5: for each  $i \in \{1, 2, \dots, length(V_i)\}$  do
6:   if  $isFill(V_i[pos])$  then
7:      $value, length+ = decode(V_i[pos])$ 
8:      $uncomp\_pos+ = length$ 
9:   else
10:     $uncomp\_pos++$ 
11:   end if
12:   if  $uncomp\_pos - last\_uncomp\_pos > THRESHOLD$  then
13:      $FP.append(comp\_pos, uncomp\_pos)$ 
14:      $last\_uncomp\_pos = uncomp\_pos$ 
15:   end if
16:    $comp\_pos++$ 
17: end for
18:  $U_i \leftarrow 0s$ 
```

---

**Algorithm 7:** Merge UB of bitvector  $i$ .

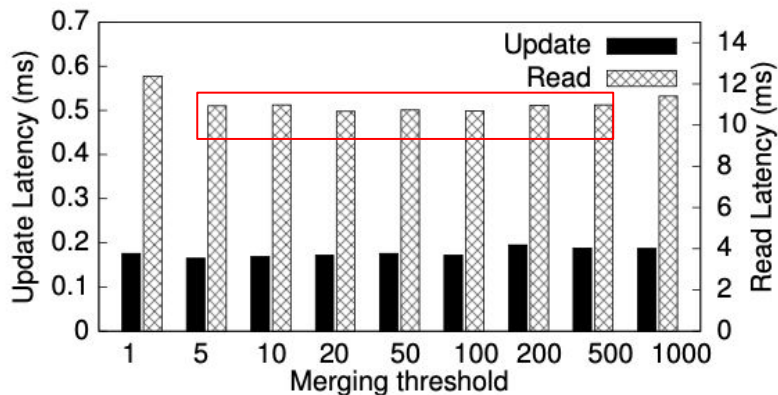
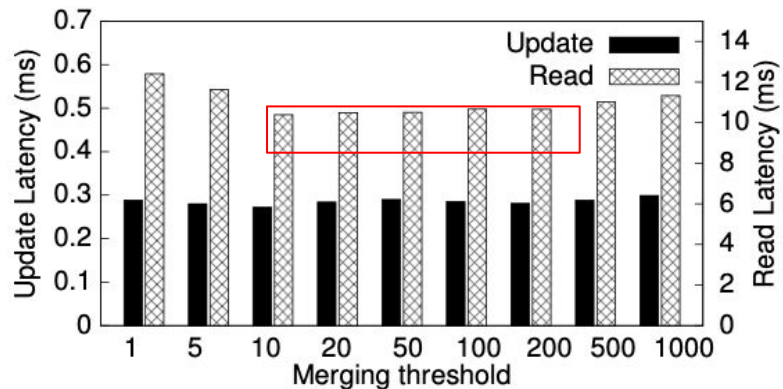
Performance degradation with bad values of merge frequency - 'T'?





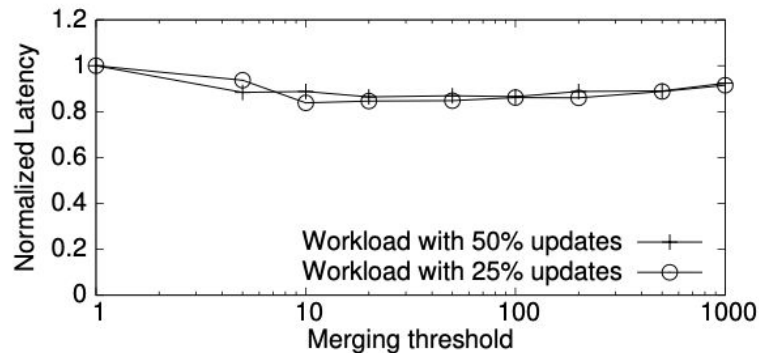
## **Optimal Value of 'T' (Response)**

# Optimal Value of merge frequency 'T'



Graph remains nearly constant

Theoretically, we do not expect this. Why?

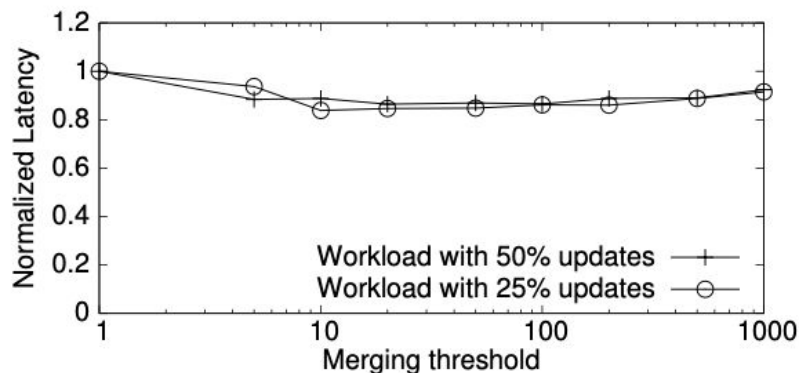




## Optimal Value of 'T' (Response)

Why the constant behavior in graph?

- Slight increase in latency after  $T=10$  (Paper identifies 10 as optimal)
- XOR performed at word level (hardware is very fast for XOR and remains almost constant for word level so does not matter if UB has extra 1s)
- UB just has to be relatively sparse
- WAH decompression does not scale linearly with number of updates
- Data suggests UB accumulation is not as significant for performance
  - evaluate wider range of workloads (very high update frequency might be more sensitive)





# Fence Pointer Frequency

- Granularity behavior of fence pointers?
- Generalized value for different workloads
- Any other factors that impact
- Can we use offline heuristics to calculate



# Fence Pointer Frequency (Response)

Random access → more granular | Scans → less granular fence pointers

Storage overhead  for too many fence pointers

Update  → maintaining cost might increase

Compressibility  → coarser granularity might work

Heuristic approach → maybe used like fence pointer every 128 words or 256 words but fine-tune based on exp results

Hardware might affect cost



# Fence Pointer Overhead

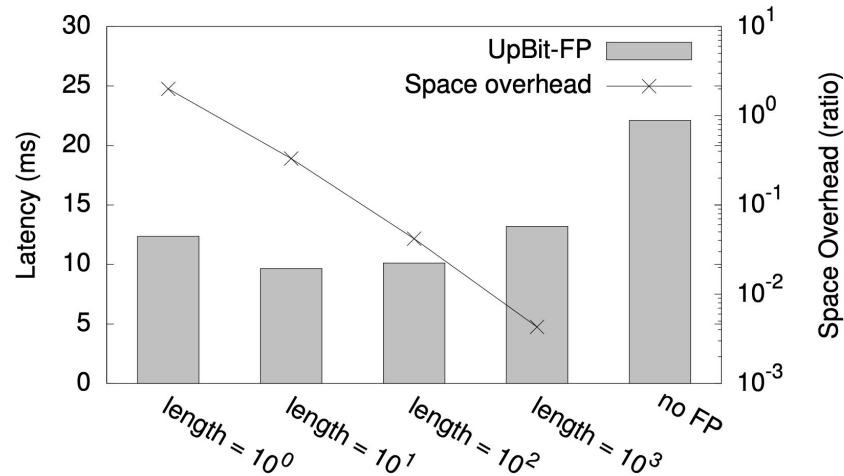
We observe that fence pointers can indeed help to decrease average latency by 2.29x, requiring, however, significant space overhead of about 15%. Almost the same benefit (2.18x), can be achieved for only 4% space overhead

- Fence pointer overhead reduce from 15% to 4% but performance reduced from 2.29x to 2.18x in the Experiments.
- Is this a bias?

# Fence Pointer Overhead (Response)

Why did the fence pointer overhead reduce from 15% to 4% when performance only reduced from 2.29x to 2.18x in the Experiments?

- *granularity & overhead in log scale*
- *sweet spot between no FPs and max FPs (effectively uncompressed)*
- *finer FP granularity -> diminishing rewards*





# **Alternative Design Tradeoffs**





# Background XOR Processing

- Perform XOR of UB and VB as a background process
- Eliminate the potential overhead of READ?



# Background XOR Processing (Response)

- Stale data - Might be delay between update and observed change
- Overhead of Synchronization
- Continuous CPU/ Memory use
- Increased complexity



# Using Byte-Aligned Bitmap Compression

- Why underlying **Fastbit** design?
- Why use **Word-Aligned Hybrid** and not **Byte-Aligned Bitmap Compression** (better space efficiency)?



# Using Byte-Aligned Bitmap Compression (Response)

Why did UpBit use the underlying Fastbit design?

- *FastBit “state-of-the-art”*
- *performant, optimized system*

Why use Word-Aligned Hybrid and not Byte-Aligned Bitmap Compression?

- *Words in WAH compression align with system architecture*
  - *better performance*
- *variable-length vs. fixed-length*
  - *space/performance tradeoff*



## Summary

- *Paper proposes robust design for bitmaps use case*
- *Extremely detailed experiments*
- *Details impact of each design separately*
- *Claims scalability but demonstrates mostly vertical scalability (size) but not concurrency as much (talks about threads)*
- *CUBIT addresses concurrency (also locking with delta records)*
- *Could be tested with workloads targeting only one part of data for updates*
- *potential comparison - bitmaps that just recreate the structure for updates (less heavy update workloads)*



**Thank you!**