UpBit: Scalable In-Memory Updatable Bitmap Indexing

Presenters: Yuxin Li, Jingyi Huang, Yizheng Xie, Taishan Chen

Background

Bitmap indexing

- Popular indexing technique for large data
- Widely applied in the industry

However,

- Storage requirements are **very** high without compression
- To reduce redundancy and improve read performance, we use compression and encodin

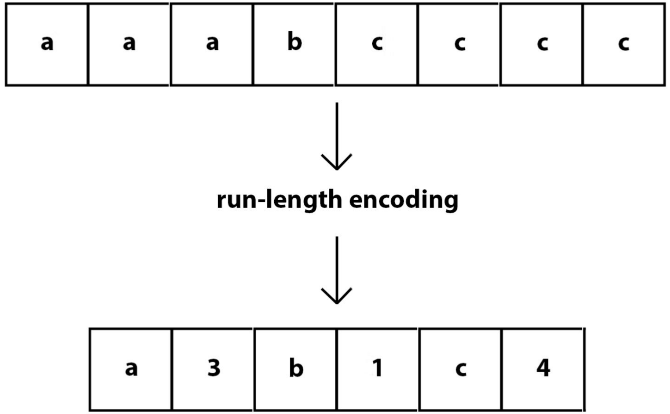
select * from T where X < 2

bitwise logical operation b1 OR b2

OID	X	bitmap index			
		=0	=1	=2	=3
1	0	1	0	0	0
2	1	0	1	0	0
3	3	0	0	0	1
4	2	0	0	1	0
5	3	0	0	0	1
6	3	0	0	0	1
7	1	0	1	0	0
8	3	0	0	0	1
		b_1	b_2	b_3	b_4

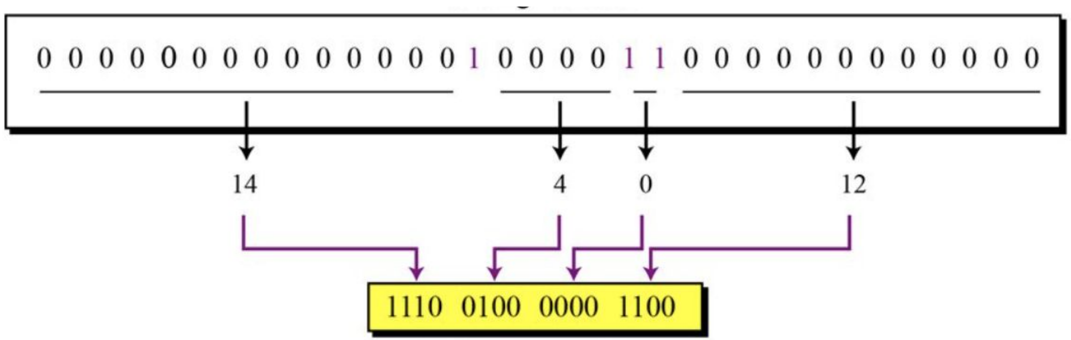
Figure 1. A sample bitmap index.

Run-length encoding (RLE)



- Simplest method of compression

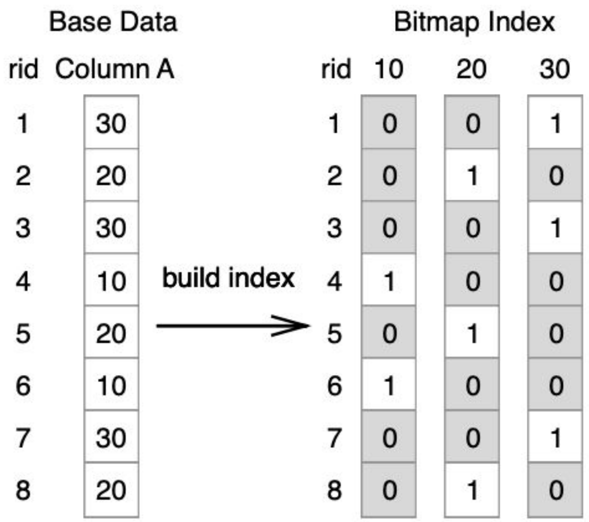
- Replace consecutive repeating occurrences of a symbol by “symbol + num of occurrences”



- This method can be more efficient if the data users only 0s and 1s in bit patterns and **one symbol is more frequent than the other**

Background

Read-optimized bitmap indexes



bitwise logical operation 10 OR 20

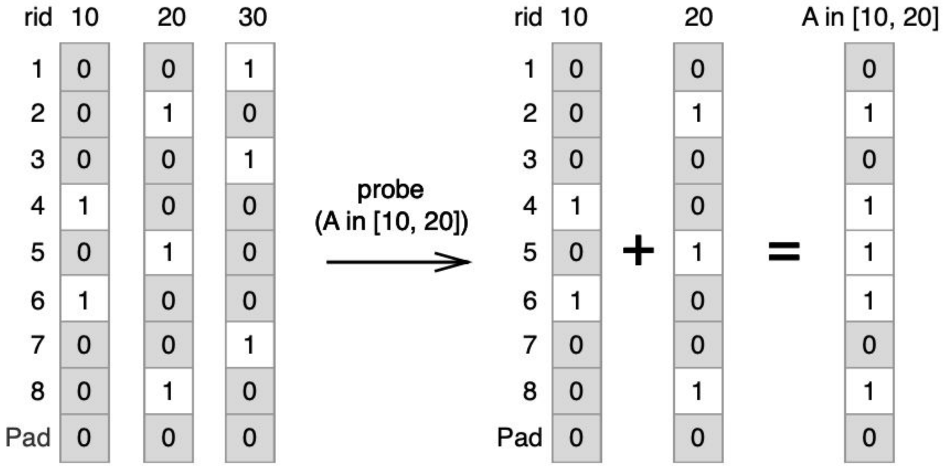
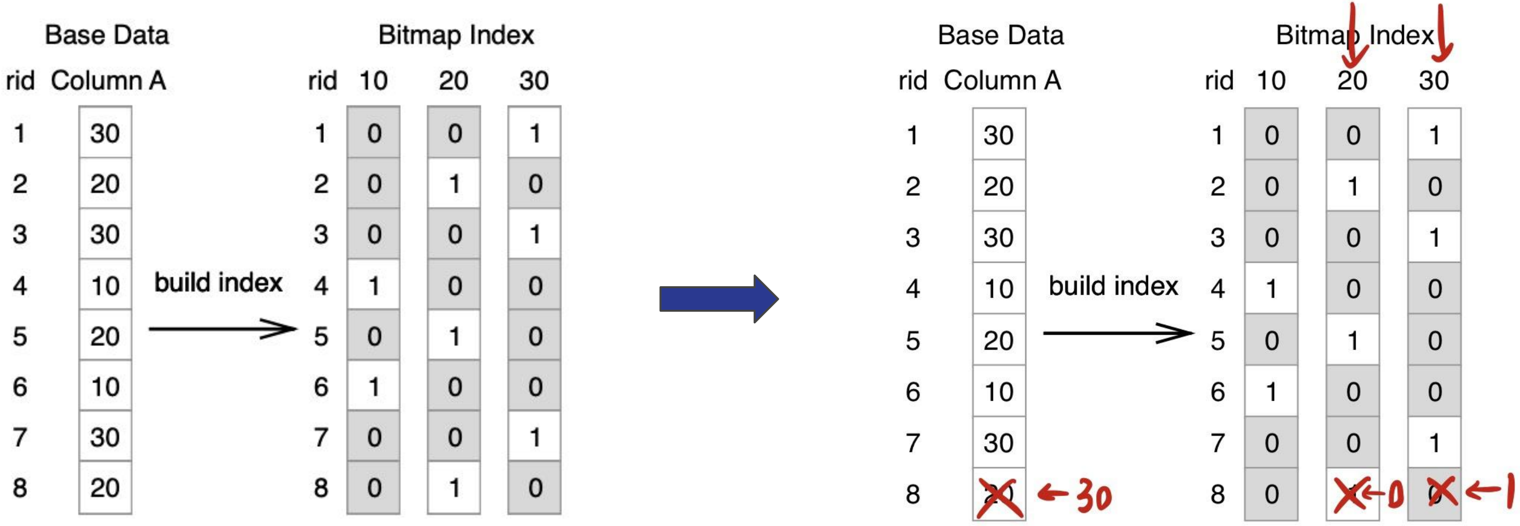


Figure 3: Searching a Bitmap Index for $A \in [10, 20]$.

Background

Bitmap indexes are not suitable for updates

- **In-place updates** caused expensive steps of decoding and encoding, why?



Update Conscious Bitmaps – Update

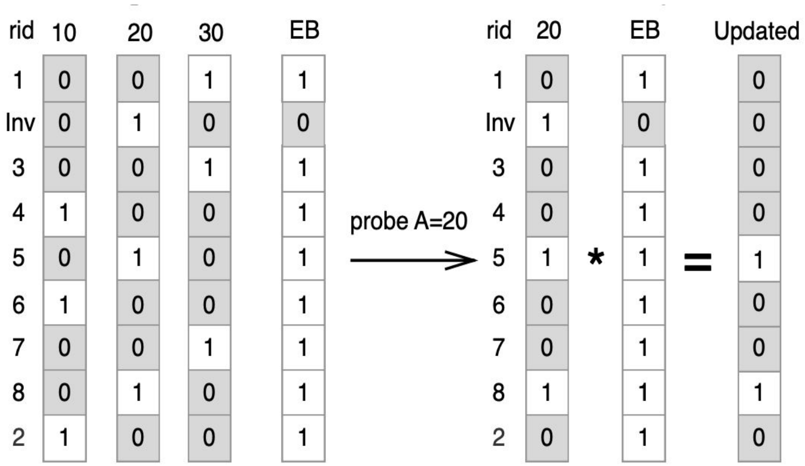
The state of art — Update Conscious Bitmaps (UCB)

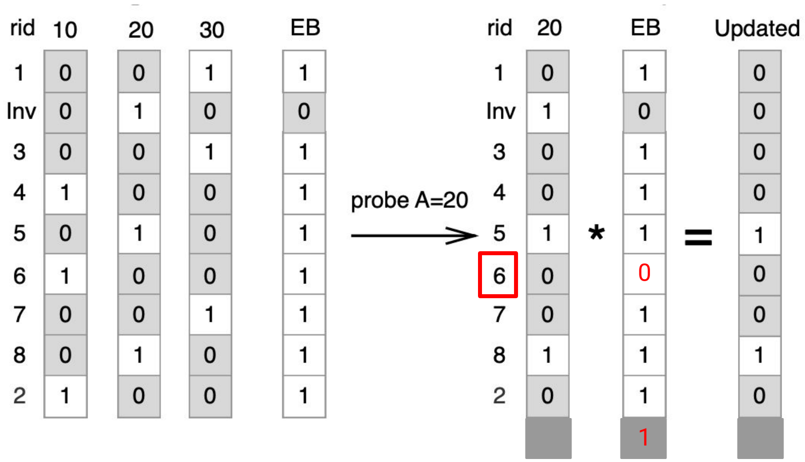
- Efficient deletes – delete then insert
- Existence bitvector (EB)
- Out of place update
- Avoided decoding + encoding value bitvectors at every update

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

Update Conscious Bitmaps – Read

Additional AND operation between value bitvector and EB

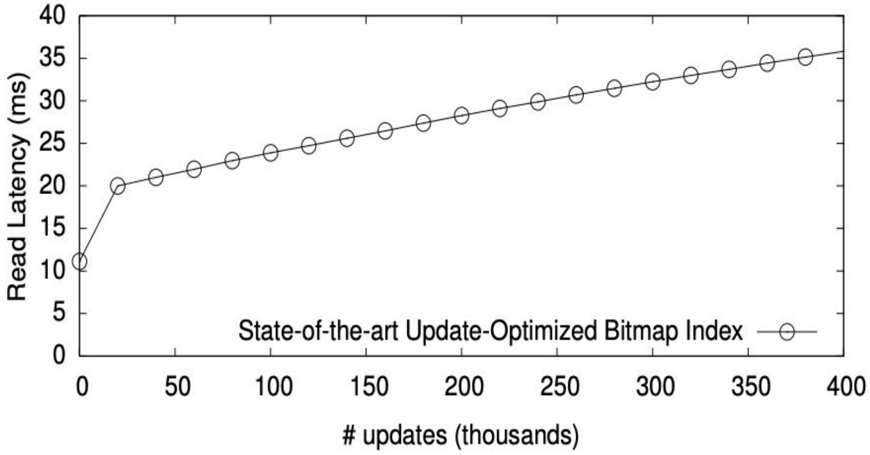


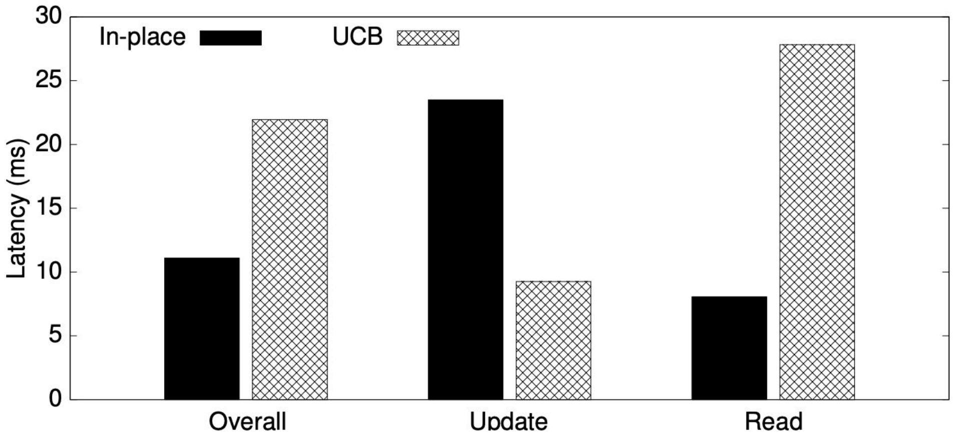


More updates and deletes – bitvectors are less compressible

Update Conscious Bitmaps

- When answering a read query
  - To perform AND operation, the **entire value bitvector & EB** needs to be decoded
  - UCB needs to consult a **translation table** for every invalidated row and do AND operation again





The problem: Scalability for Updates

Applications require support for **both efficient reads** and **updates**

Flaws with old approaches:

1. Read - optimized bitmap indexes designs are not suited for updates
2. Update optimized bitmap indexes (UCB)

In place update does not depend on past updates

- Drawback – Read performance does not scale with updates (As more updates arrive, read queries become increasingly more expensive. Why?)

Solution: UpBit

A scalable in-memory Updatable Bitmap index design

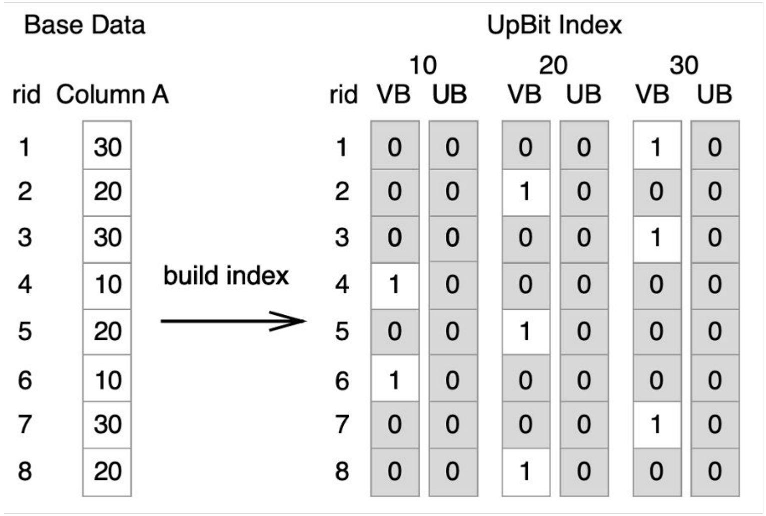
Two new design elements:

1. A corresponding update bitvector (UB) for every value bitvector
    a. Incoming update – UB contains both new and old value
    b. Periodically merged with values bitvectors then re-initialized (after exceeds the threshold)
2. Fence Pointers – direct access of compressed bitvector (any position)
    a. Avoid unnecessary decodings
    b. Enable efficient multi-threaded decoding of a bitvector

UpBit Design Patterns - Data Structure

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

- Update Bitvector
  - UpBit per value
  - Initialize: all 0s (Space)
  - Update: flip on bit
  - Current Value: XOR operation
- Value-Bitvector Mapping (VBM, Hashing)
- Counter of 1s
  - Avoid XOR
  - Trigger Merging

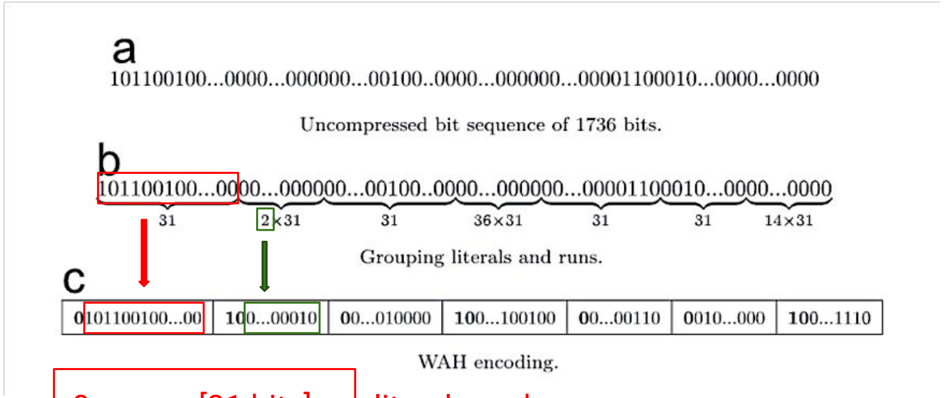


UpBit Design Patterns - Fence Pointer (1)

Problem: How to access bit B[k] efficiently?

Word Aligned Hybrid (WAH) Encoding

- Each 31 Not Encoded bits is a sequence
- Encoded bits are word-aligned
- Hybrid Encoding
  - For consecutives sequences of 0s or 1s: run-length encoding
  - For short sequence of mixed 0s and 1s: literal representation



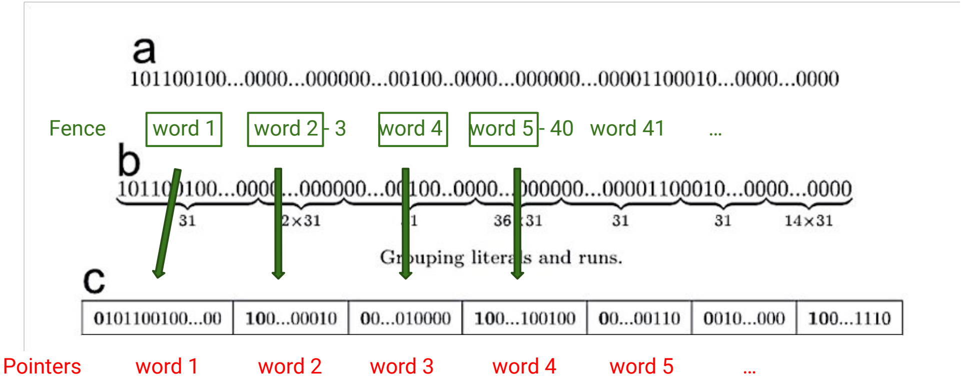
0xx...xxx [31 bits] — literal word

All 1s: 11xx...xxx [30 bits]
All 0s: 10xx...xxx [30 bits] — fill word

UpBit Design Patterns - Fence Pointer (2)

Fence Pointer (FP) Idea

- **Fence**: the position of not encoded word
- **Pointer**: the position of the encoded word that **starts with** the content of this not encoded word
- **Granularity g** on not encoded bitvector: Num of not encoded words between two fences
- Approximation



UpBit Design Patterns - Fence Pointer (3)

Steps: Building FP for bit vector V

```
comp_pos = 0
uncomp_pos = 0
last_uncomp_pos = 0
for each i ∈ {1,2,...,length(V_i)} do
    if isFill(V_i[pos]) then
        value, length += decode(V_i[pos])
        uncomp_pos += length
    else
        uncomp_pos++
    end if
    if uncomp_pos − last_uncomp_pos > THRESHOLD then
        FP.append(comp_pos, uncomp_pos)
        last_uncomp_pos = uncomp_pos
    end if
    comp_pos++
end for
```

Decoding

Append FP pairs

Implementation

- Array of **(umcomp_pos, comp_pos)** pairs
- **g** is the threshold when appending pairs
- **umcomp_pos** is the pos of not encoded word

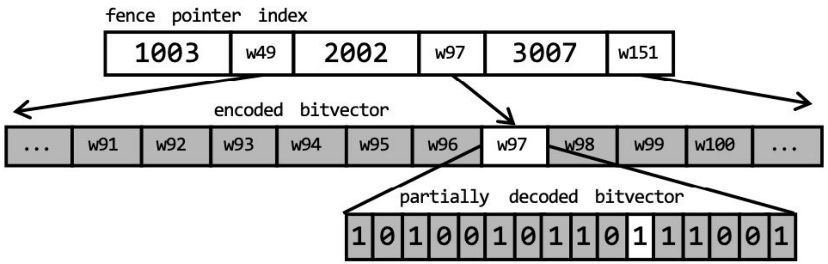
  comp_pos is the offset of encoded word

1003	w49	2002	w97		
uncomp_pos	comp_pos	uncomp_pos	comp_pos	…	comp_pos

UpBit Design Patterns - Fence Pointer (4)

Steps to Get Bit B[k] (Input: row k, bitvector B):

1. Use FP to get **nearest** position **pos**
2. While not found bit k:
    a. Decode word **w** at position **pos**
    b. If found: return value **val**
    Else: **pos ++**



Example 1: Searching row **62073**

Not encoded word: 62073/31 = **2002**

Bit Position: 62073 mod 31 = **11**

Decode **w97** and get position **11**

Example 2: Searching row **62150**

Not encoded word: 62150/31 = **2004**

Nearest not encoded word: **2002**

Bit Position: 62150 - 31*2002 = **88**

Decode **from w97** and get position **88**

UpBit Design Patterns - Scaling (Merging)

Step 1: Reuse result from XOR

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

```
V_i = V_i ⊕ U_i
comp_pos = 0
uncomp_pos = 0
last_uncomp_pos = 0
for each i ∈ {1,2,...,length(V_i)} do
    if isFill(V_i[pos]) then
        value, length+ = decode(V_i[pos])
        uncomp_pos+ = length
    else
        uncomp_pos++
    end if
    if uncomp_pos − last_uncomp_pos > THRESHOLD then
        FP.append(comp_pos, uncomp_pos)
        last_uncomp_pos = uncomp_pos
    end if
    comp_pos++
end for
U_i ← 0s
```

Step 2: Build FP

Algorithm 7: Merge UB of bitvector i.

Step 3: Re-initialize UB

Why Merging?

- **Less Compressible** UB
- Expensive **Decoding** and **XOR** Bitwise Operation

Merging Strategy:

- Maintain a **threshold T**
- When #1s > T: mark as “to be merged”
- Merge UB and VB to VB in the next search operation (Reuse)

UpBit Operations - Searching a Value

Steps (Input: Value **val**):

1. Retrieve **VB** and **UB** given **val** (VBM)
2. Check Counter
   a. All zeros: return **VB**
   b. Return **VB XOR UB**

	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

Probe A=20 →

	20				
rid	VB		UB		Updated
1	0		0		0
2	1		1		0
3	0		0		0
4	0	⊕	0	=	0
5	1		0		1
6	0		0		0
7	0		0		0
8	1		0		1

UpBit Operations - Deleting a Row

Steps (Input: Row k):

1. Find the Value **val** of row **k**
2. Retrieve **UB** given **val** (VBM)
3. Flip **UB** at position **k**: UB[k] = ¬UB[k]

	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

Question: How to find the value of row k?

UpBit Operations - Get Value of a Row

Steps (Input: Row k): **Parallel Reading**

For each Value **val** in range **[10, 20, 30]**:

1. Retrieve **VB** and **UB** given **val** (VMB)
2. Return **val** if **VB[k] ⊗ UB[k]**

Get Value of Row 2 →

	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

UpBit Operations - Updating a Row

Steps (Input: Row **k**, Value **val**):

1. Retrieve **UB_i** given **val** (VBM)
2. Find the old value **old_val** of row **k**
3. Retrieve **UB_j** given **old_val** (VBM)
4. Flip **UB_i** at position **k**: UB_i[k] = ¬UB_i[k]
5. Flip **UB_j** at position **k:** UB_j[k] = ¬UB_j[k]

	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

How UCB update?

- Delete (Invalidate) then Insert: need to keep row mapping
- Single EB: all updates require changing EB -> Less compressible

UpBit Operations - Inserting a Row

Steps (Input: Value **val**):

1. Retrieve **UB** given **val** (VBM)
2. If no padding space: extend **UB**
3. **UB.#elements ++**
4. **UB[#elements] ++**

Why UB?

Typically Smaller and more compressible

rid	10 VB	10 UB	20 VB	20 UB	30 VB	30 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

rid	10 VB	10 UB	20 VB	20 UB	30 VB	30 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

UpBit VS UCB

UpBit

- One UpBit per Value
- XOR Operation
- Partial Decoding using FP
- Scalability by Merging
- No Invalidation

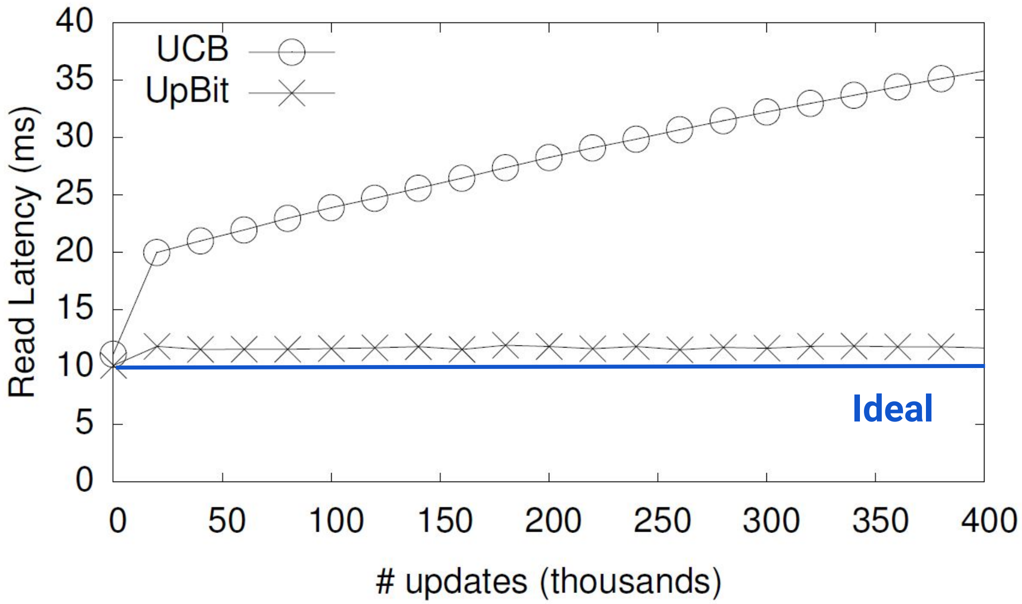
UCB

- Only One EB: Burden concentrated on one bitvector (less compressible)
- AND Operation
- Full Decoding: Poor performance on reading an arbitrary row
- Poor Scalability: Need to merge EB with each VB
- Invalidate Rows: Need to keep a mapping when a row is updated

Evaluation

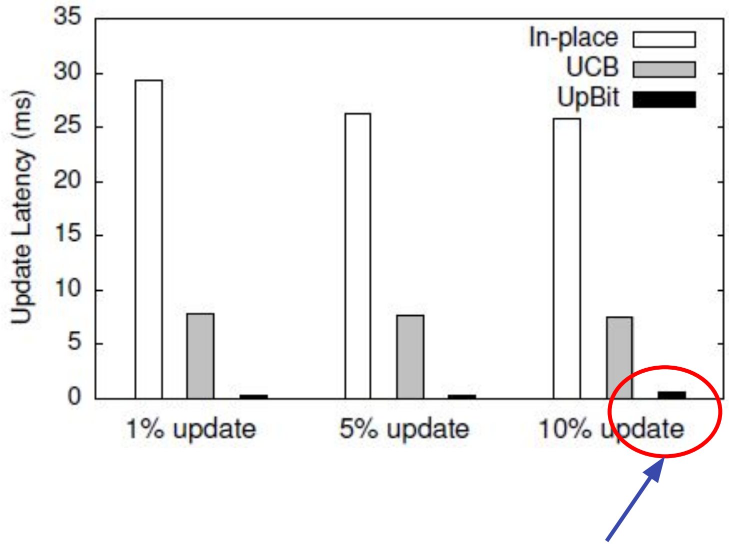
- Tested Approaches
  - In-place updates - Read-optimized Bitmap Indexing
  - UCB - Update-Conscious Bitmaps
  - UpBit
- Workloads
  - Synthetic data sets
  - Real-life data sets (Berkeley Earth dataset, TPC-H)
- Notations
  - ***n***: data size
  - ***d***: domain cardinality (i.e., # of different values)

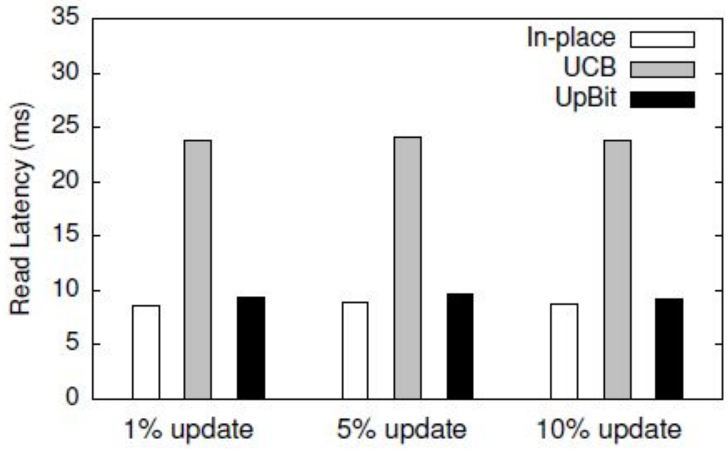
Stable read performance



- UpBit scales with the number of updates, limiting the size of UBs

Evaluation - Update and Read Performances



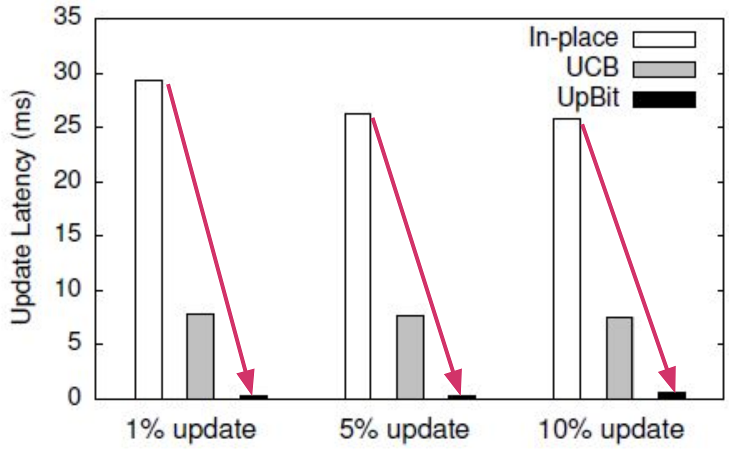


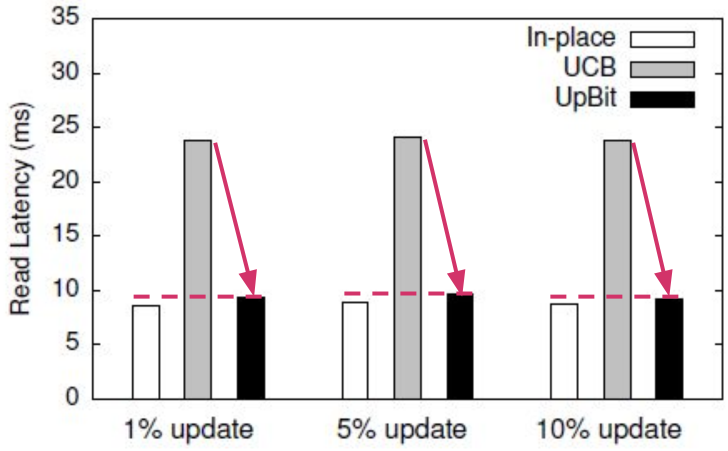
n = 100M
d = 100

What causes the increase in latency?

1. Merging
2. Lower compressibility compared to initial state

Evaluation - Update and Read Performances





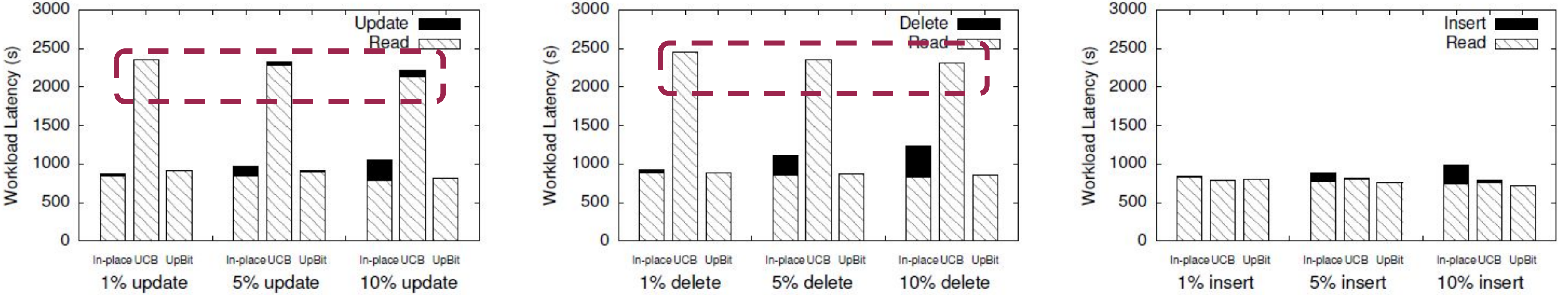
n = 100M
d = 100

- 15-29x faster than UCB
- 51-115x faster than in-place

- Only 8% read overhead over optimal
- 3x faster on read performance than UCB

Read latency pays off for update performance

Evaluation - Impact of Updates/Deletes/Inserts



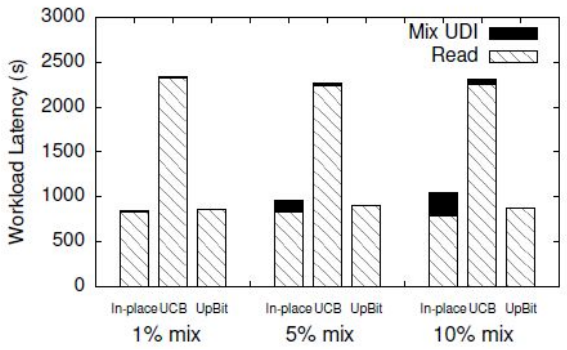
(a) UpBit vs. UCB vs. in-place for updates. (b) UpBit vs. UCB vs. in-place for deletes. (c) UpBit vs. UCB vs. in-place for inserts.

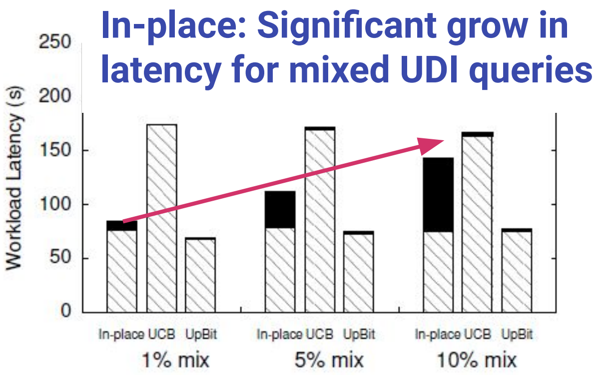
What may be the reason for the higher latency in update compared with delete?

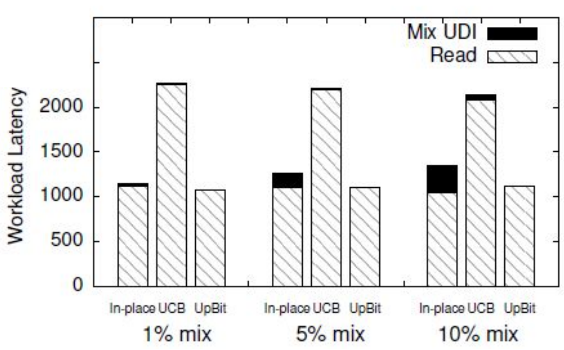
Synthetic data set

- ***n = 100M, d = 100***
- Workloads: 100k queries, 1%/5%/10% update/delete/insert

Evaluation - Scaling







d = 100
n = 100M

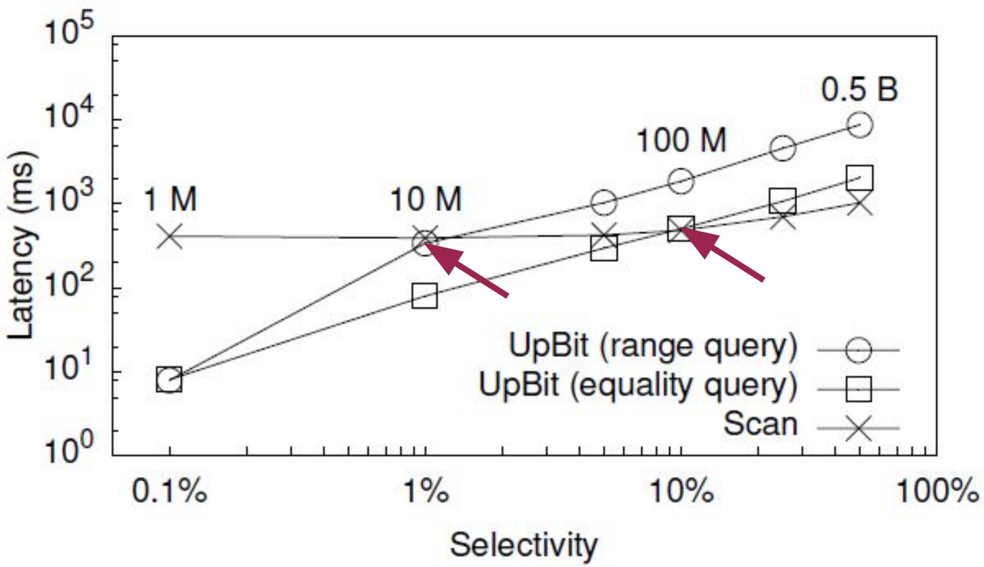
d = 1000
n = 100M

d = 100
n = 1B

Synthetic data set

- Workloads: 100k mixed queries

Analysis - UpBit vs. Scan



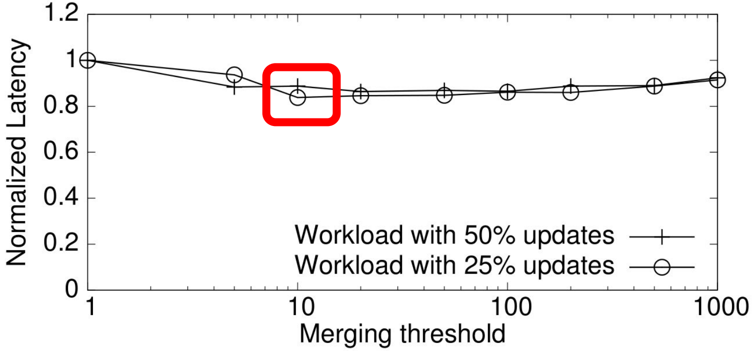
- UpBit outperforms fast scan at selectivity up to 1%

Selectivity: % of elements selected from a column

Tuning - **Merging Threshold**

More frequent merging

Better compression

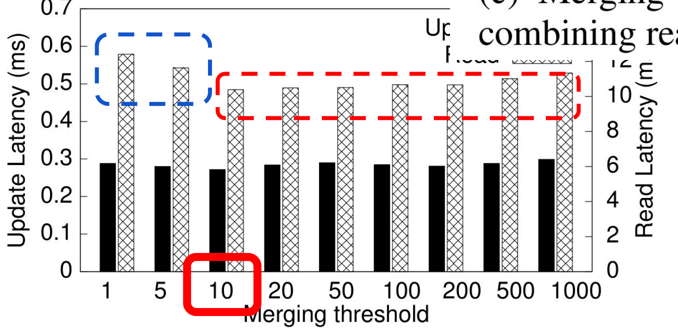


(c) Merging threshold for the overall workload combining reads and updates.

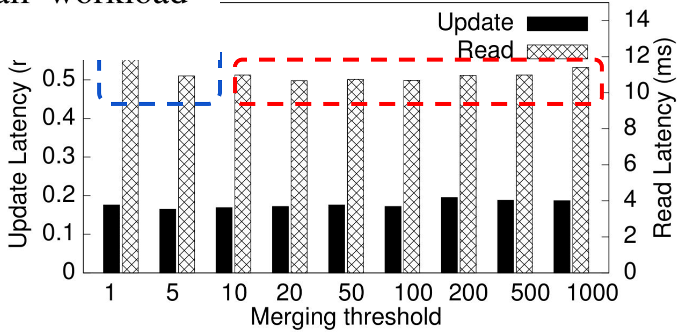
… but most time used for get old value, not very prominent!

Less frequent merging

Less compressible



(a) Read and update latency as a function of merging threshold for a workload with 20% updates.



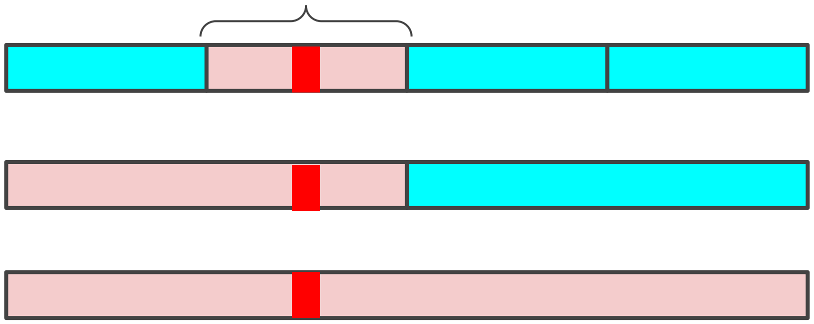
(b) Read and update latency as a function of merging threshold for a workload with 50% updates.

Tuning - **Fence Length**

More space overheads, less compressible — **Less** overheads, **more** compression

Fence Length

Faster decoding — Slower decoding



Need to experiment for different dataset.

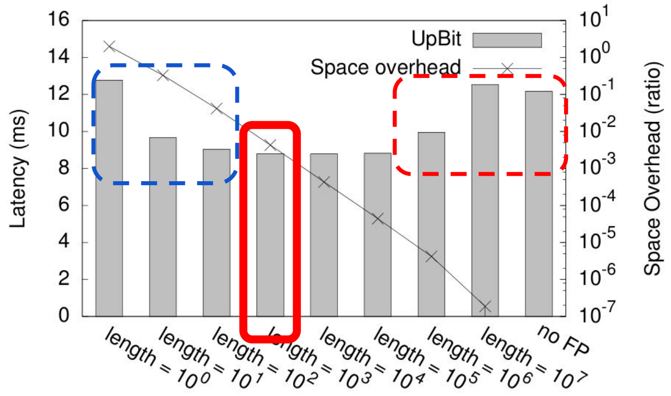
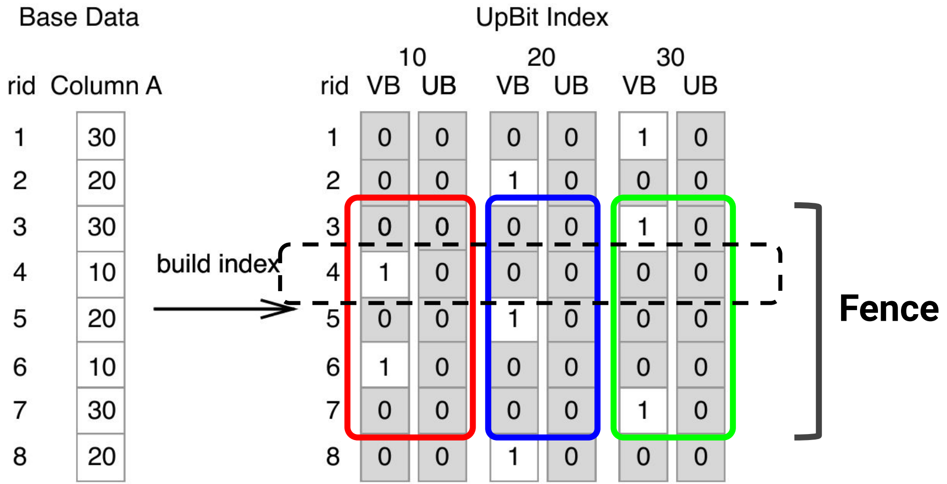


Figure 20: UpBit's optimal behavior needs fence pointers every 10^3-10^5 values having less than 0.5% space overhead.

Tuning - **Parallel Reading**



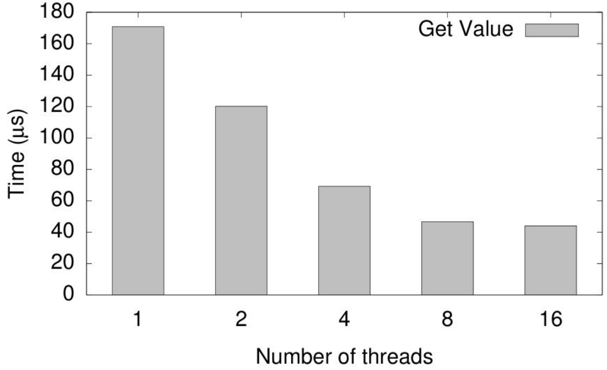


Figure 21: Bitvectors parallel scans scale with number of threads, leading to 3.9× improvement in *get_value*.

Impact and **Tradeoff**

In-place: In-place Update Bitmaps

UCB: Update Conscious Bitmaps

UpBit-FP: Fence Pointers, **no** Update Bitmaps

UpBit: FP **and** UB

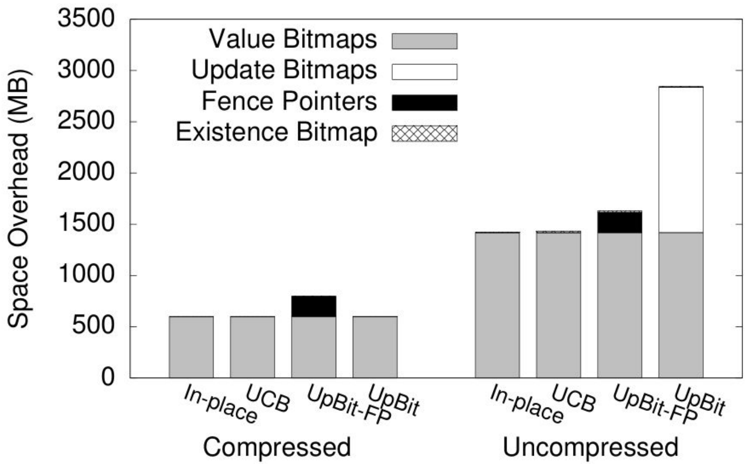


Figure 25: While UpBit auxiliary structures add a lot of raw space, when compressed the space overhead of UpBit is negligible.

Impact - **Size**

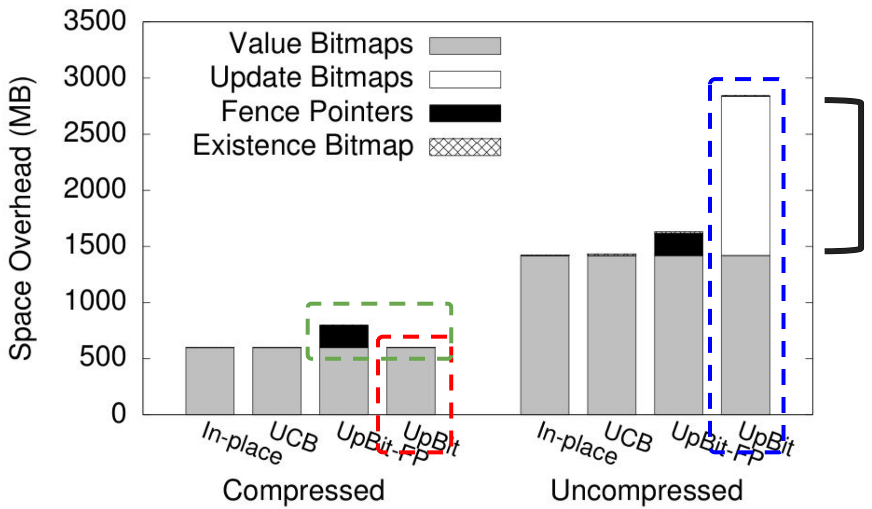


Figure 25: While UpBit auxiliary structures add a lot of raw space, when compressed the space overhead of UpBit is negligible.

UB can be efficient compressed

UB also makes bitmaps more compressible, less **uncompressed bits** so less **fence pointers**.

Impact - **Fence Pointer** only

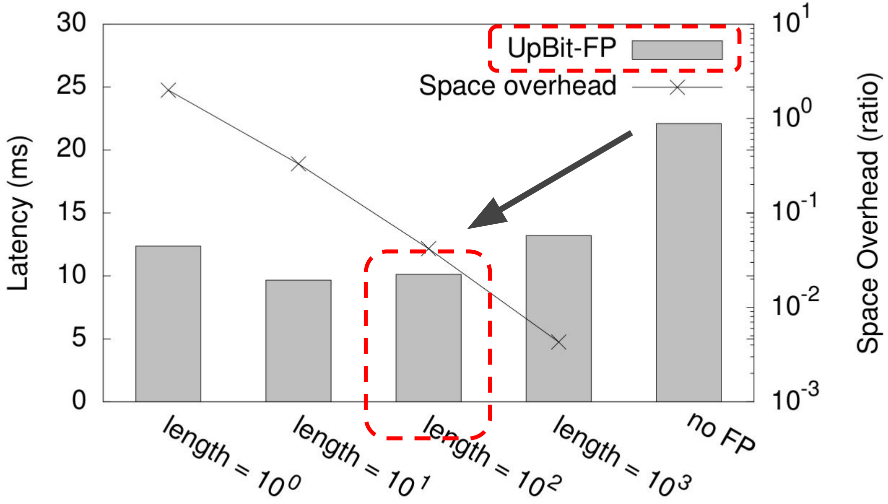


Figure 23: Fence pointers alone offer more than 2× better performance, having less than 10% space overhead.

Tradeoff - **Fence Length**

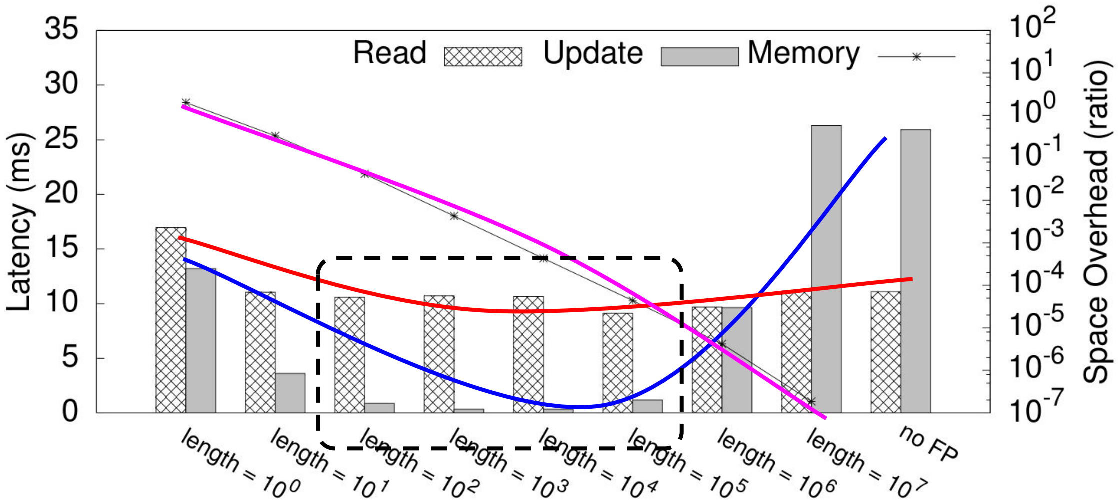


Figure 26: In order to decide fence pointer granularity we analyze the expected workload to get the best combination of performance and memory overhead.

Read

Update

Memory

Performance gain - **FP and UB**

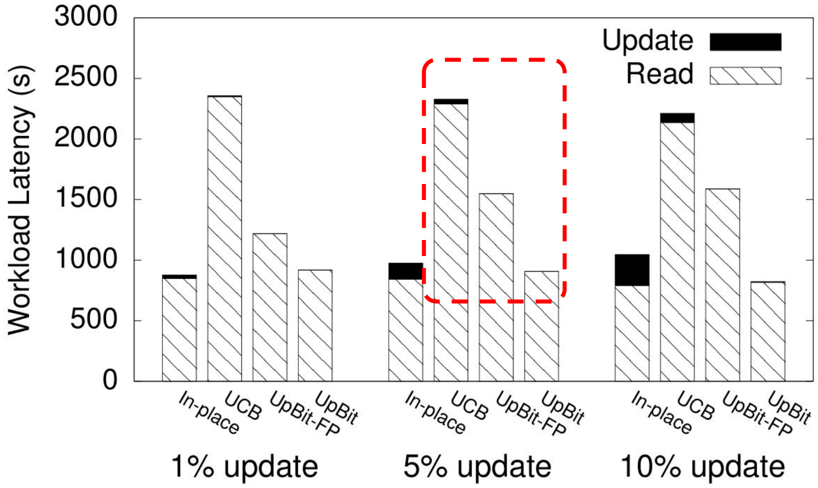


Figure 24: Both fence pointers, and update bitvectors contribute towards the overall performance gains of UpBit.

Experiment on TPC-H

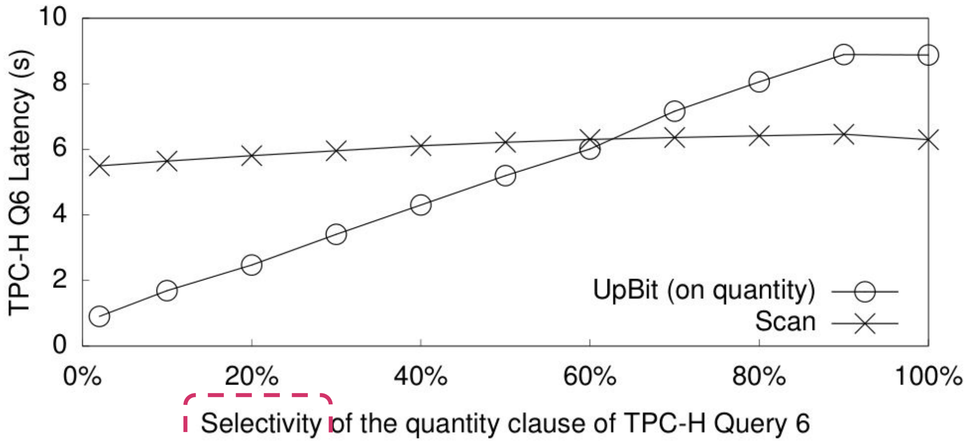


Figure 27: UpBit achieves significant benefits up to 6× compared to the scan-based execution of TPC-H Q6, when varying the selectivity of the l_quantity clause.

Less selectivity means **less values** fulfils requirements, and **less bitmap vectors** will be read.

Conclusion

Goals:

- ❑ Higher compressibility
- ❑ Efficient access to value
- ❑ Bounded cost of updates

Designs:

- ➔ Distributing the update overhead to multiple UBs
- ➔ Partial decoding with fence pointers
- ➔ Query-driven UB merging

Further Discussion: Concurrency Control

- What concurrency control strategy should we choose for UpBit?

Q & A

Thank You!

Any questions for us?