UpBit: Scalable In-Memory Updatable Bitmap Indexing

Dimitris Staratzis

Meet the authors

Manos Athanassoulis
Boston University

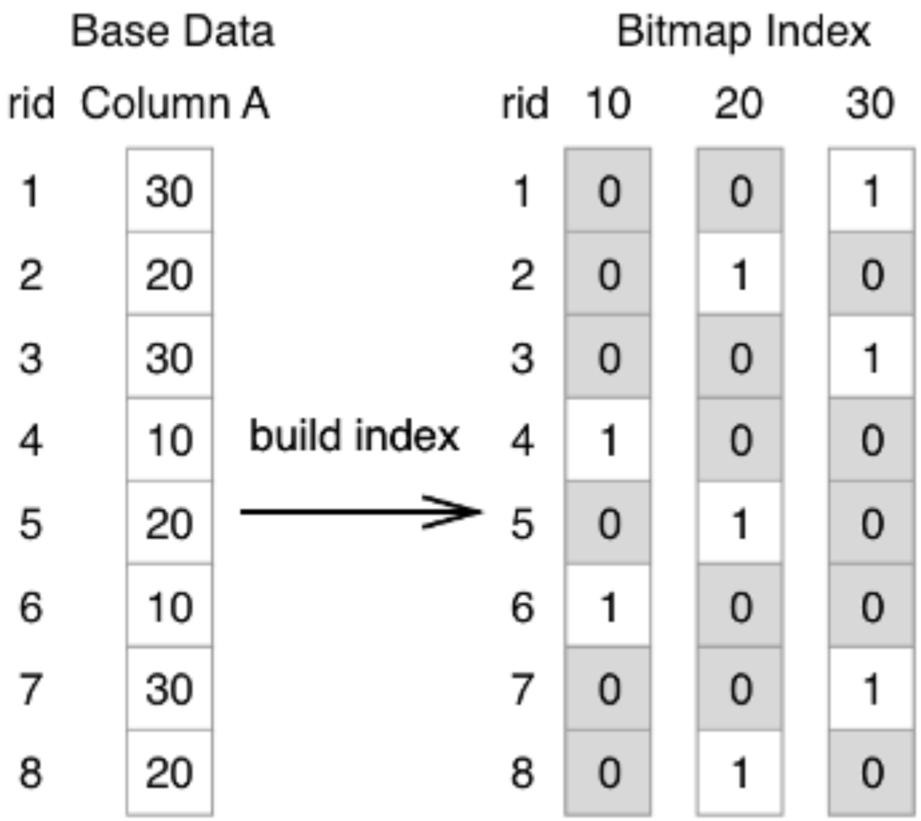
Zheng Yan
Facebook

Stratos Idreos
Harvard University

1. Background

(1)Bitmap Index Introduction

What is it?



Base Data

rid	Column A
1	30
2	20
3	30
4	10
5	20
6	10
7	30
8	20

build index

Bitmap Index

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

The domain of column A has d unique values which correspond to d value bitvectors $VB = \{V1,V2,...,Vd\}$

(1)Bitmap Index Introduction

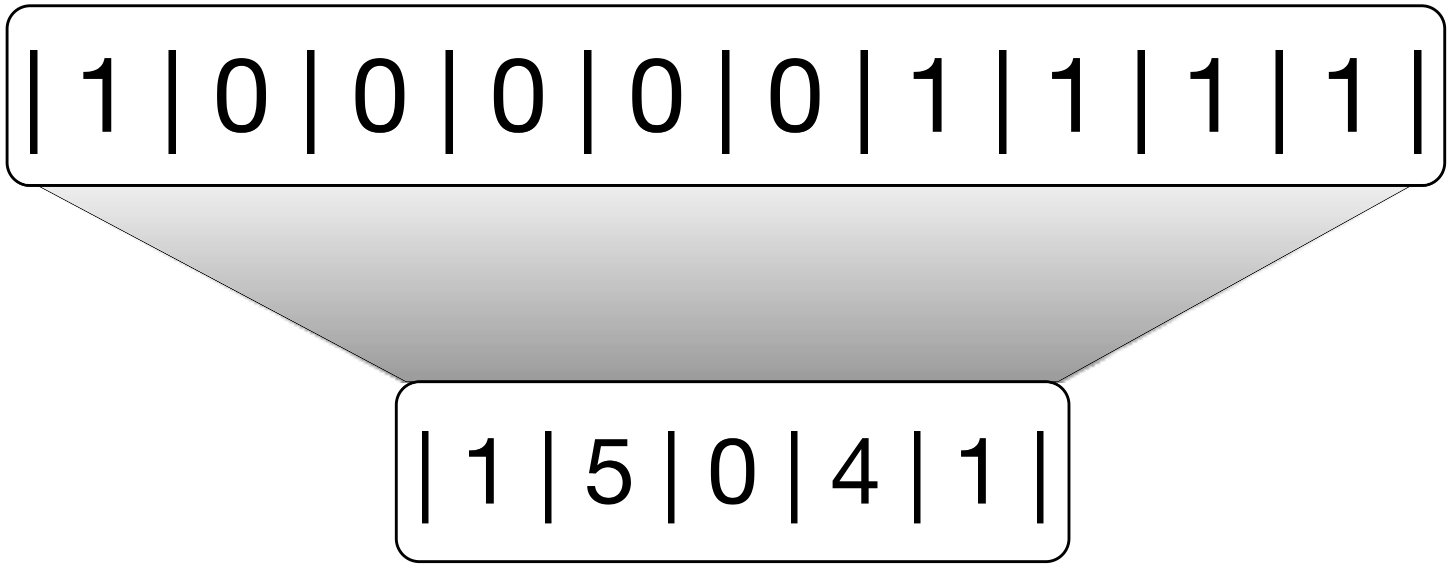
Why use it?

- Very fast ***equality*** and ***low selectivity*** queries
- Occupy relatively ***little space***
- Take advantage of ***parallelism***

Memory footprint

What is the cost of using it?

- To minimize storage requirements, we use compression.
- Typical example of Run-Length encoding:



| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

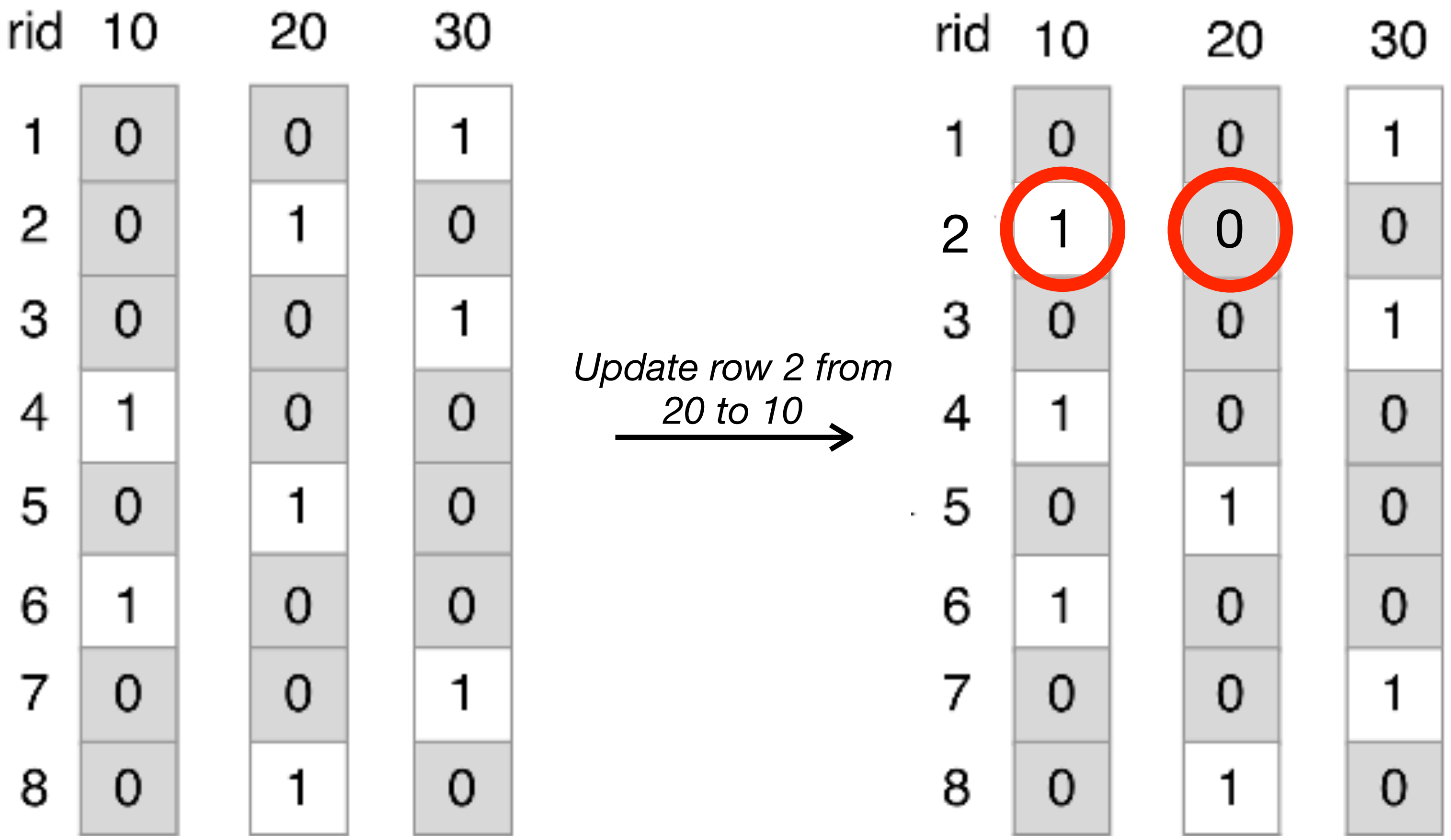
| 1 | 5 | 0 | 4 | 1 |

Very space efficient even for domains with large cardinality!

The Problem

Scalability for Updates

We need both good ***read performance*** and ***data freshness***.



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

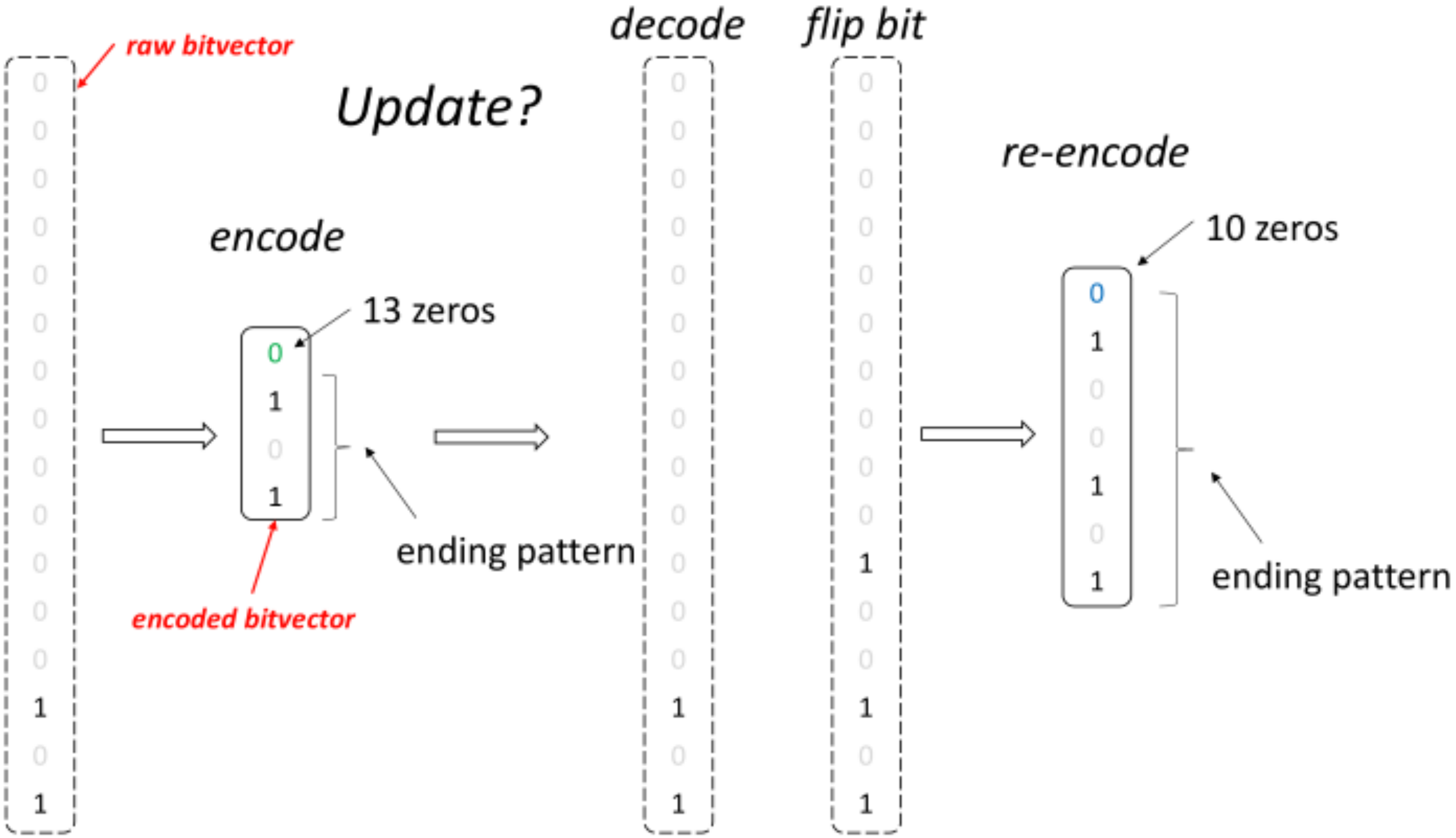
Update row 2 from 20 to 10 →

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

What is the problem?

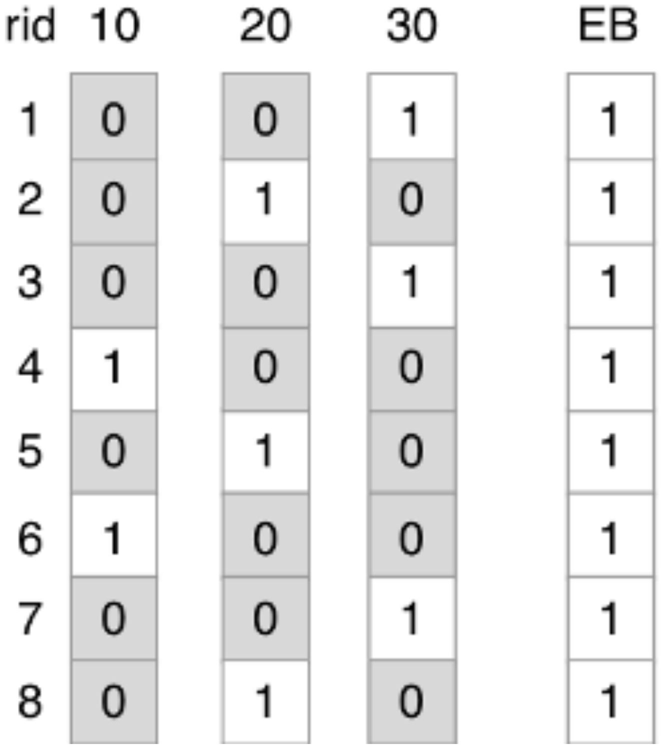
Updating bitvectors is very inefficient

Why?



Update Conscious Bitmaps (UCB)

What is the state of the art?

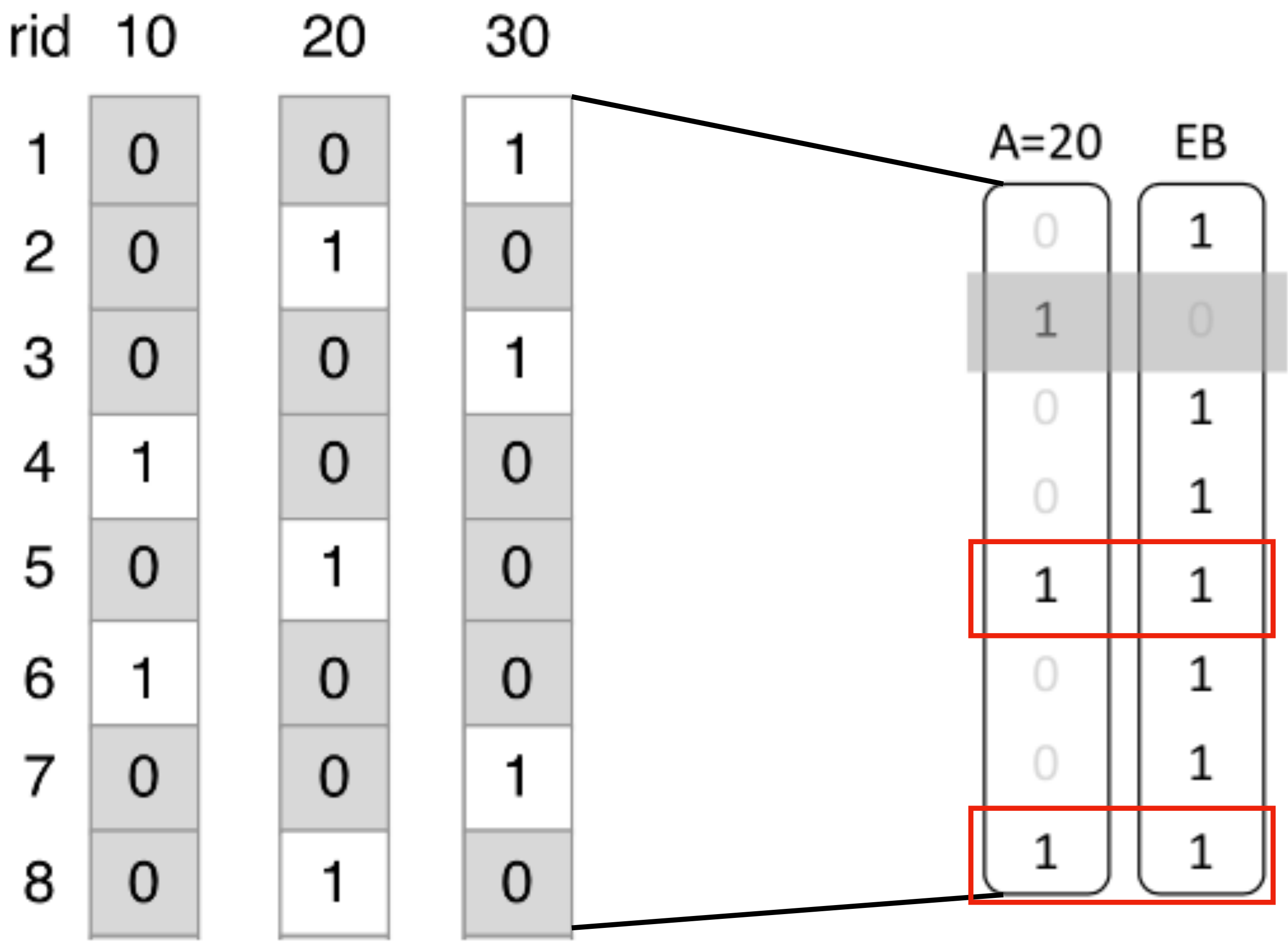


rid	10	20	30	EB
1	0	0	1	1
2	0	1	0	1
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1

- Core idea - Existence Bitvector (EB)
- EB is initialized with 1s

Update Conscious Bitmaps (UCB)

How to read?



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

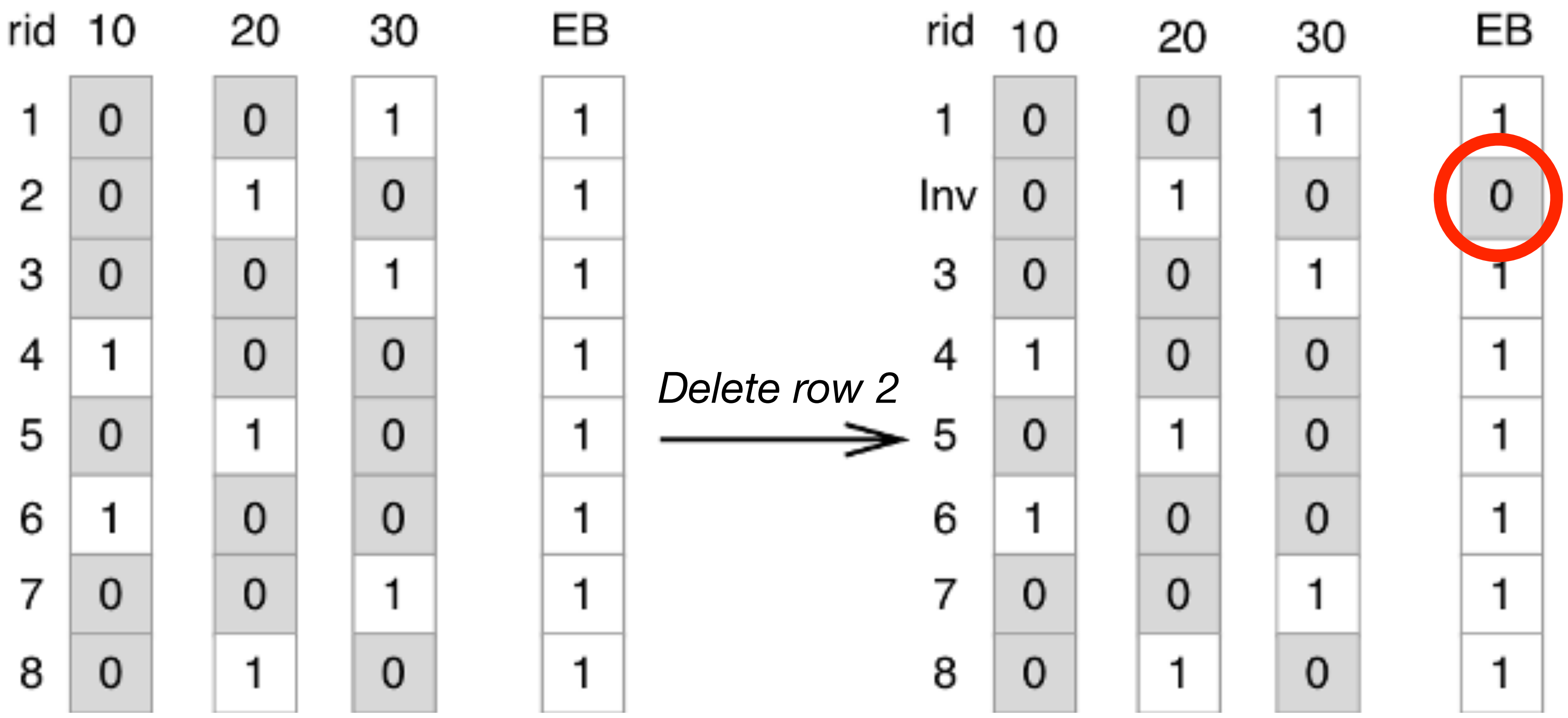
A=20	EB
0	1
1	0
0	1
0	1
1	1
0	1
0	1
1	1

- A bitwise AND between the VB and the EB is required

select * from table where columnA = 20

Update Conscious Bitmaps (UCB)

How deletes work in the state of the art?



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

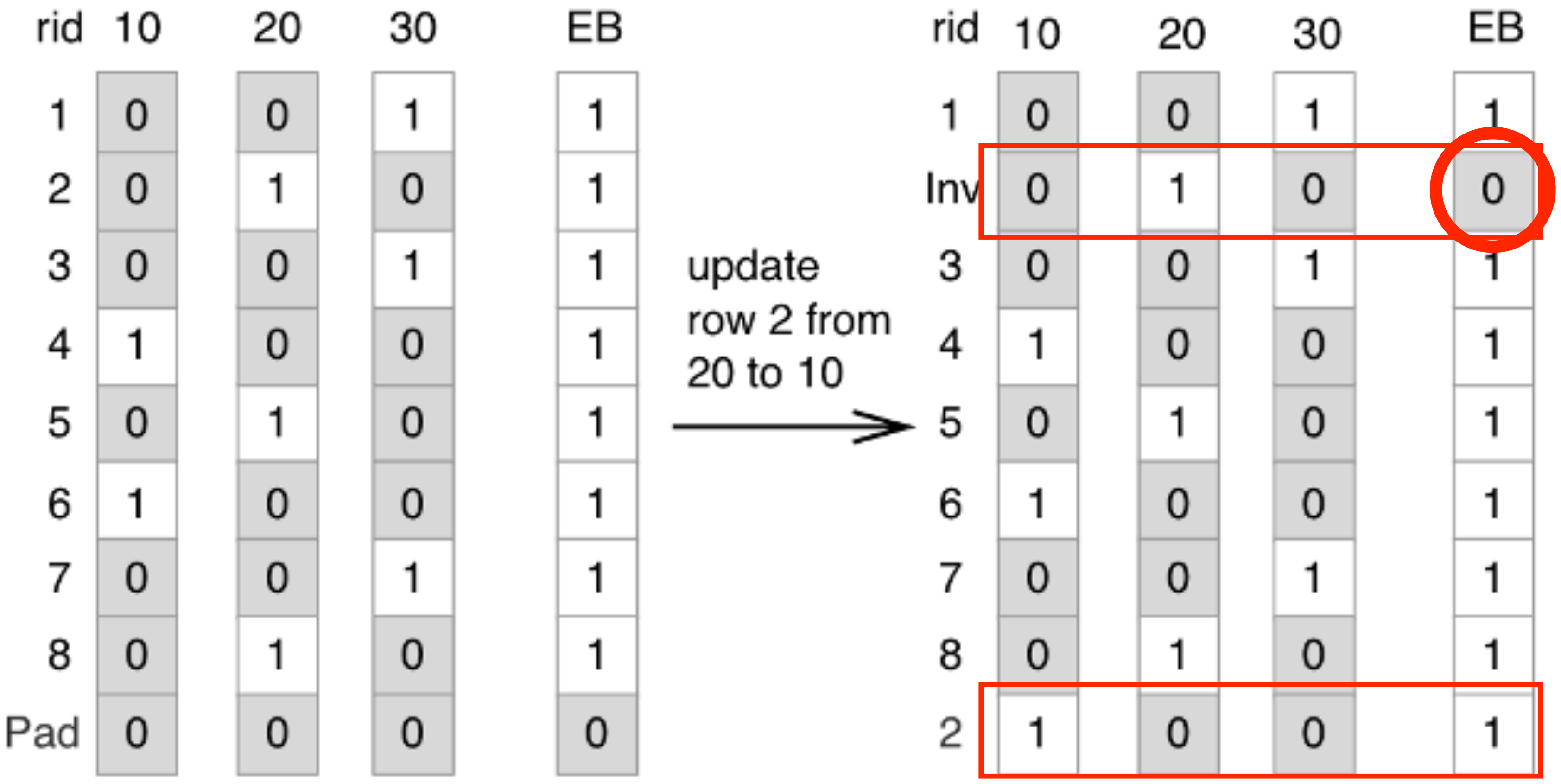
Delete row 2

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

Update Conscious Bitmaps (UCB)

How updates work in the state of the art?

First delete then append (Out-of-place)



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

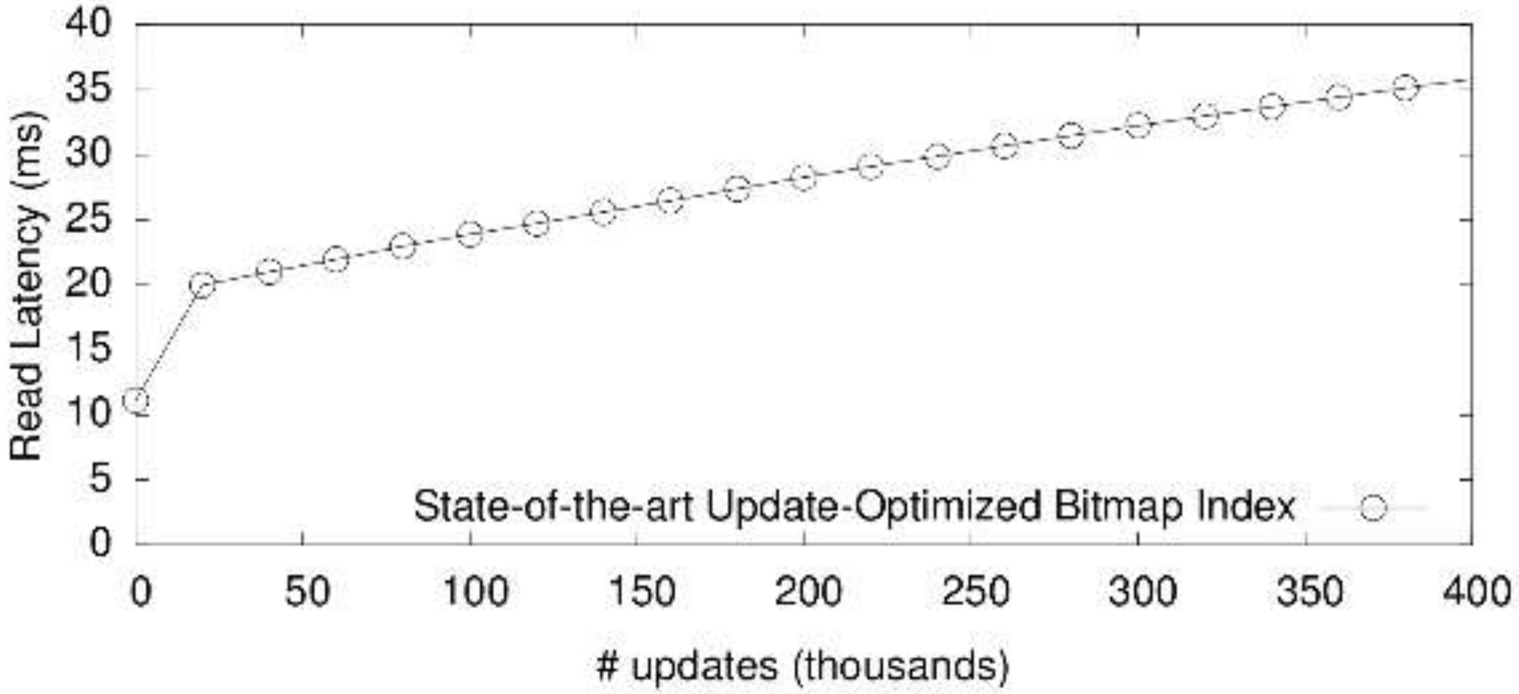
update row 2 from 20 to 10

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

Does UCB scale?

No!

Update Conscious Bitmaps (UCB)



As ***more updates*** arrive, read queries become increasingly ***more expensive***.

Why?

Update Conscious Bitmaps (UCB)

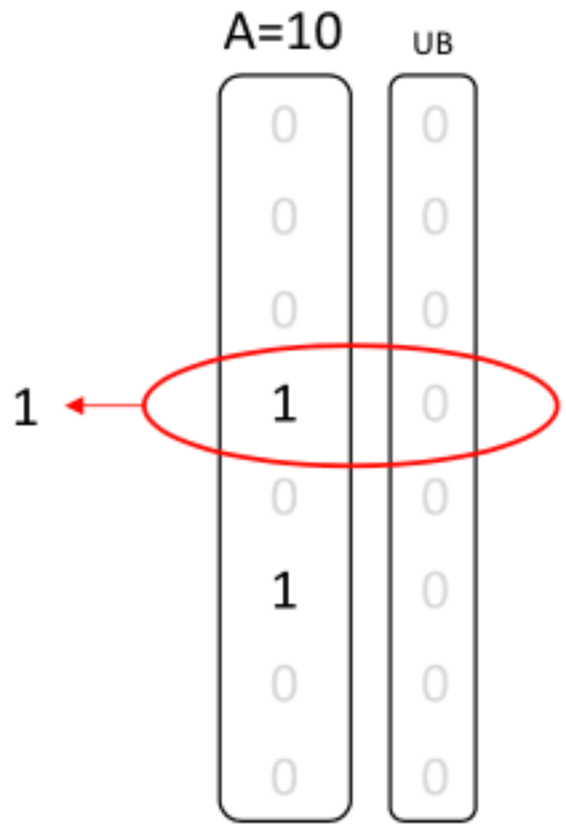
Why it does not scale?

- Updates/Deletes —> **Worse compressibility** of the bitvectors
- Need to **decode** and **re-encode**
- Need to map rowIDs with EB

2. The solution: UpBit

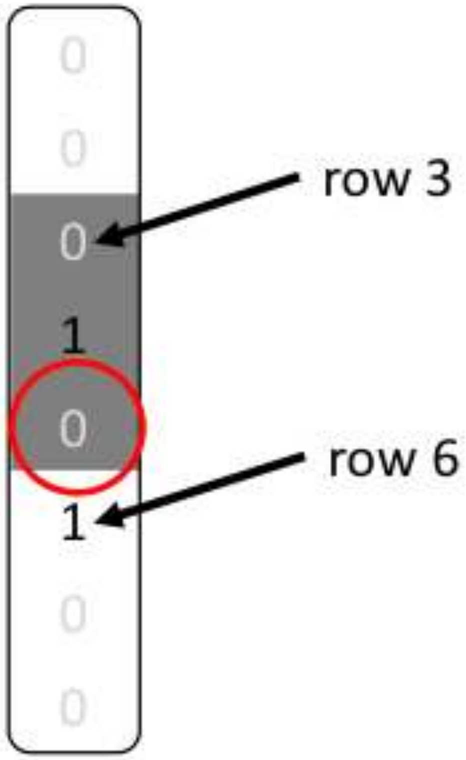
Scalable Updates in Bitmap Indexing

UpBit, 1st design element: Update Bitvector (UB)



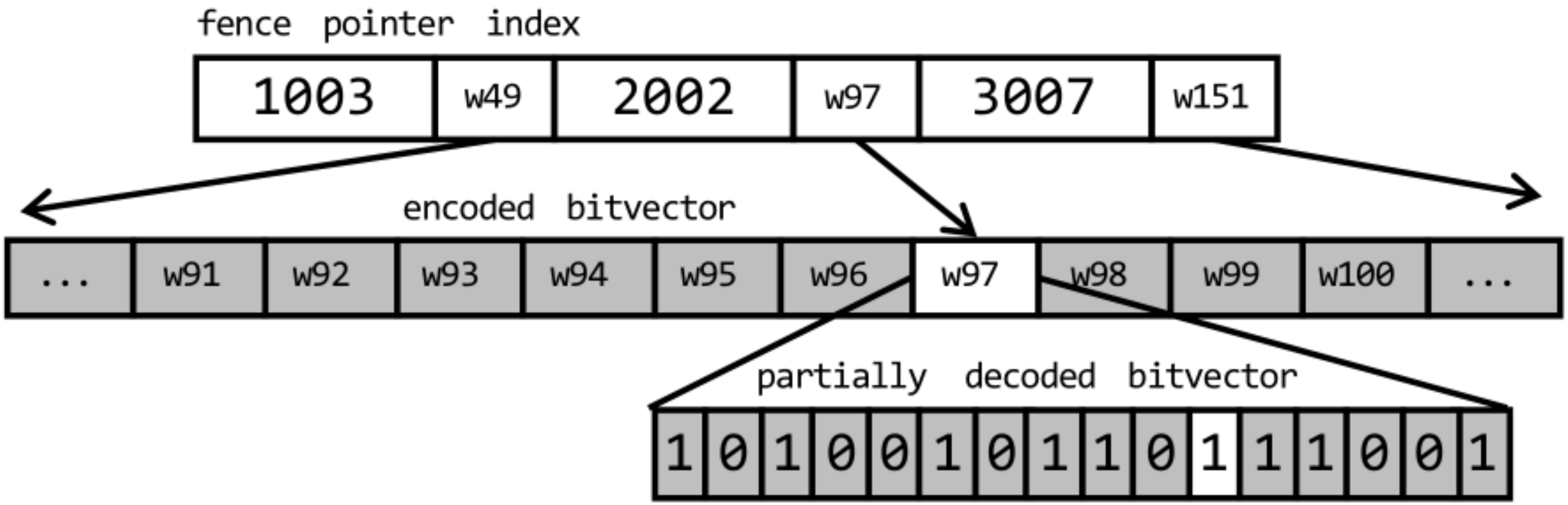
- Every update flips a bit
- The current value is the XOR
- Initialized to 0s
- One per value of the domain

UpBit, 2nd design element: Fence Pointers (FP)



- Efficient access to compressed bitvectors
- ***No need*** to decompress

Fence Pointers in Detail



fence pointer index

1003 | w49 | 2002 | w97 | 3007 | w151

encoded bitvector

... | w91 | w92 | w93 | w94 | w95 | w96 | w97 | w98 | w99 | w100 | ...

partially decoded bitvector

1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1

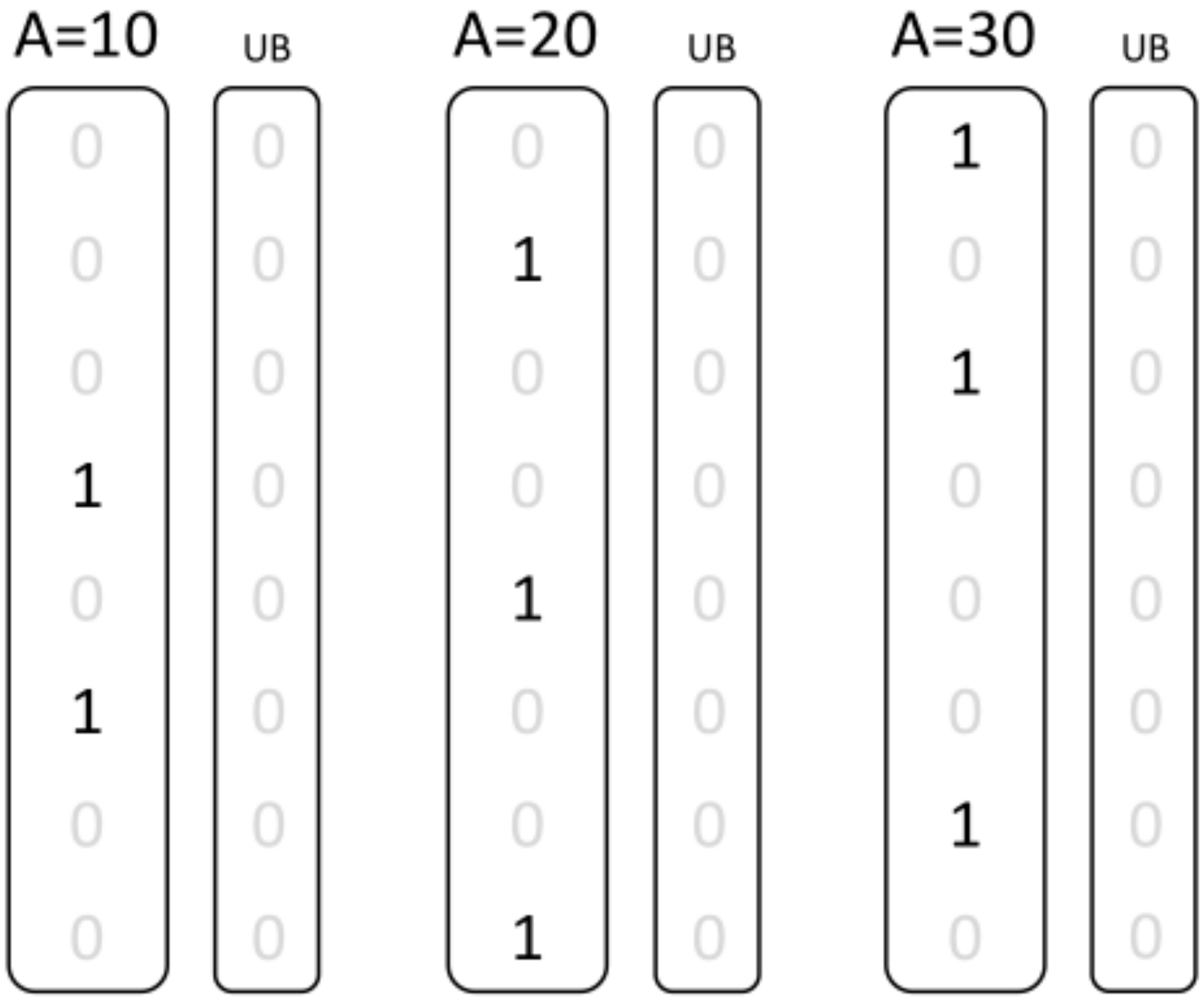
UpBit

Basic Operations

- **Updates**
- **Search**
- **Delete**
- **Insert**

UpBit - Update (1)

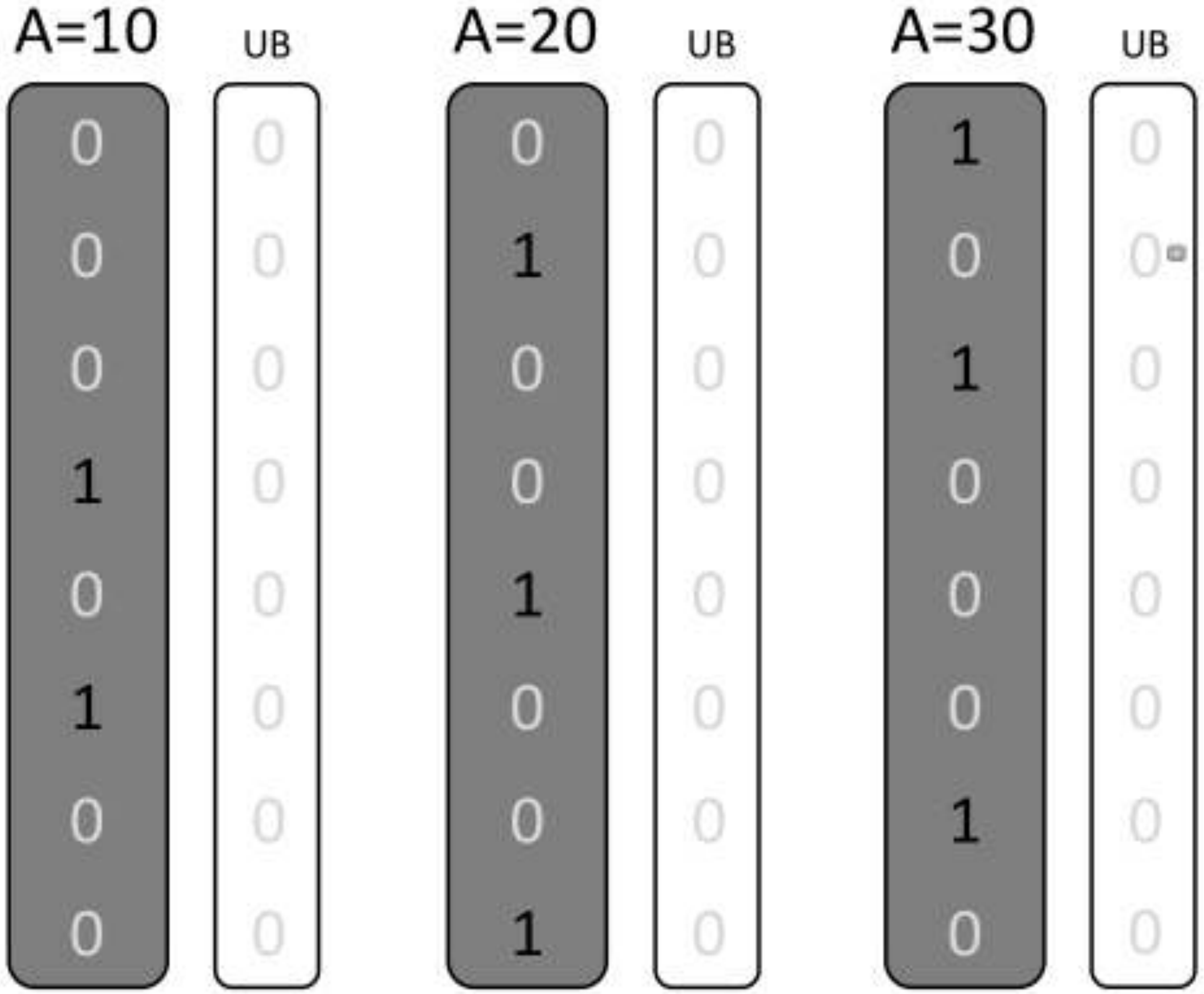
Update row 5 from 20 to 10



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	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

UpBit - Update (2)

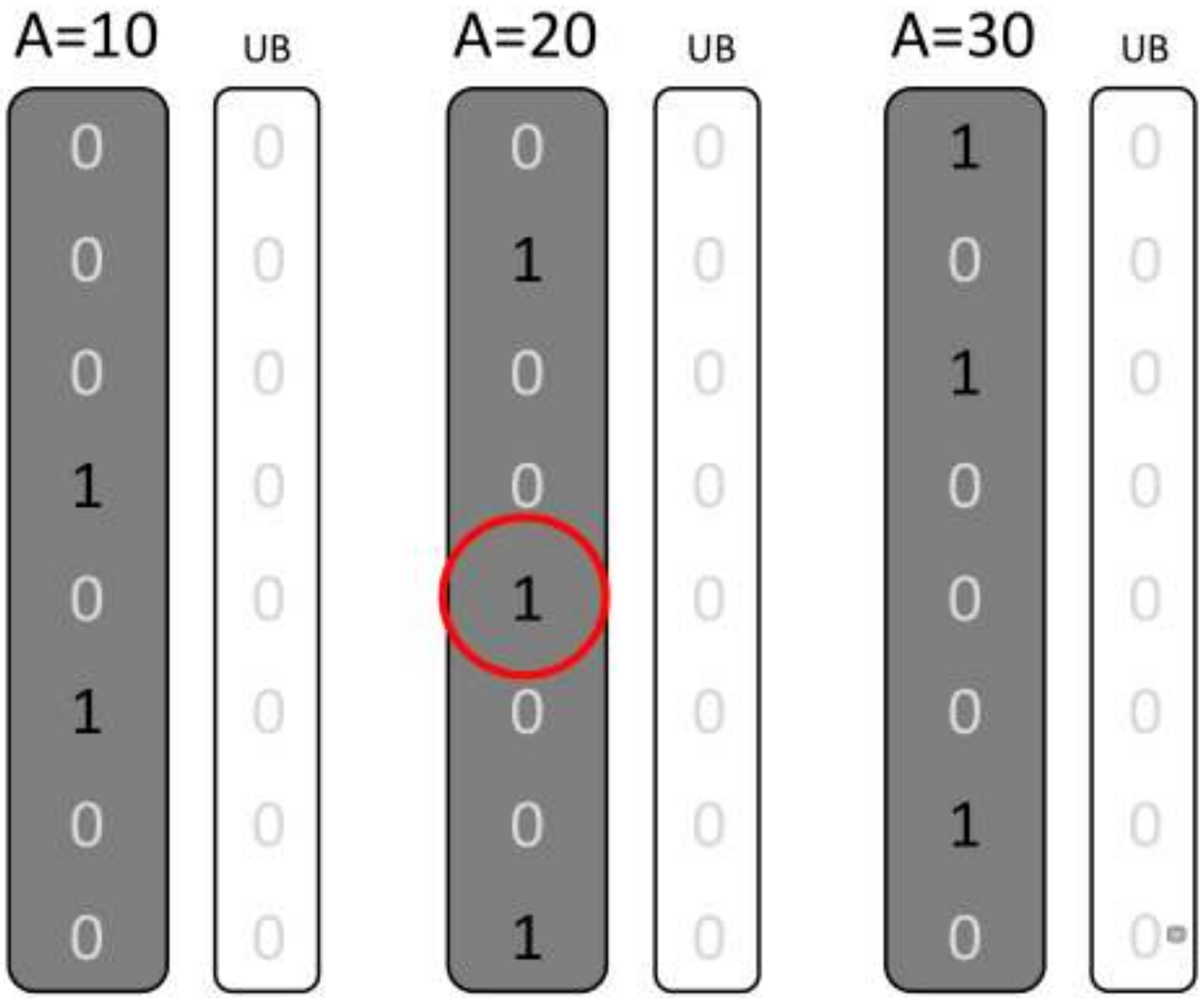
Update row 5 from 20 to 10



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	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

UpBit - Update (3)

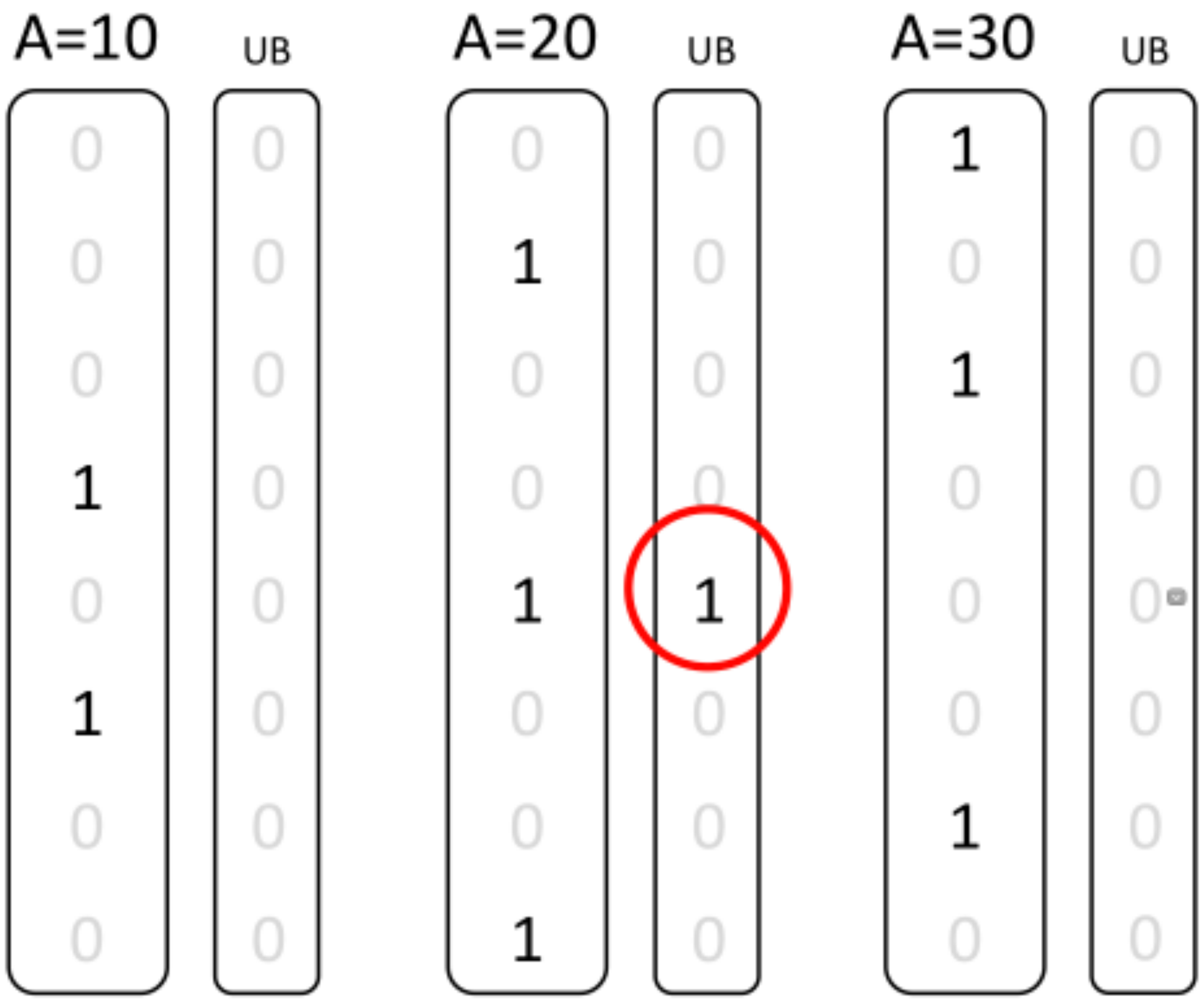
Update row 5 from 20 to 10



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	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

UpBit - Update (4)

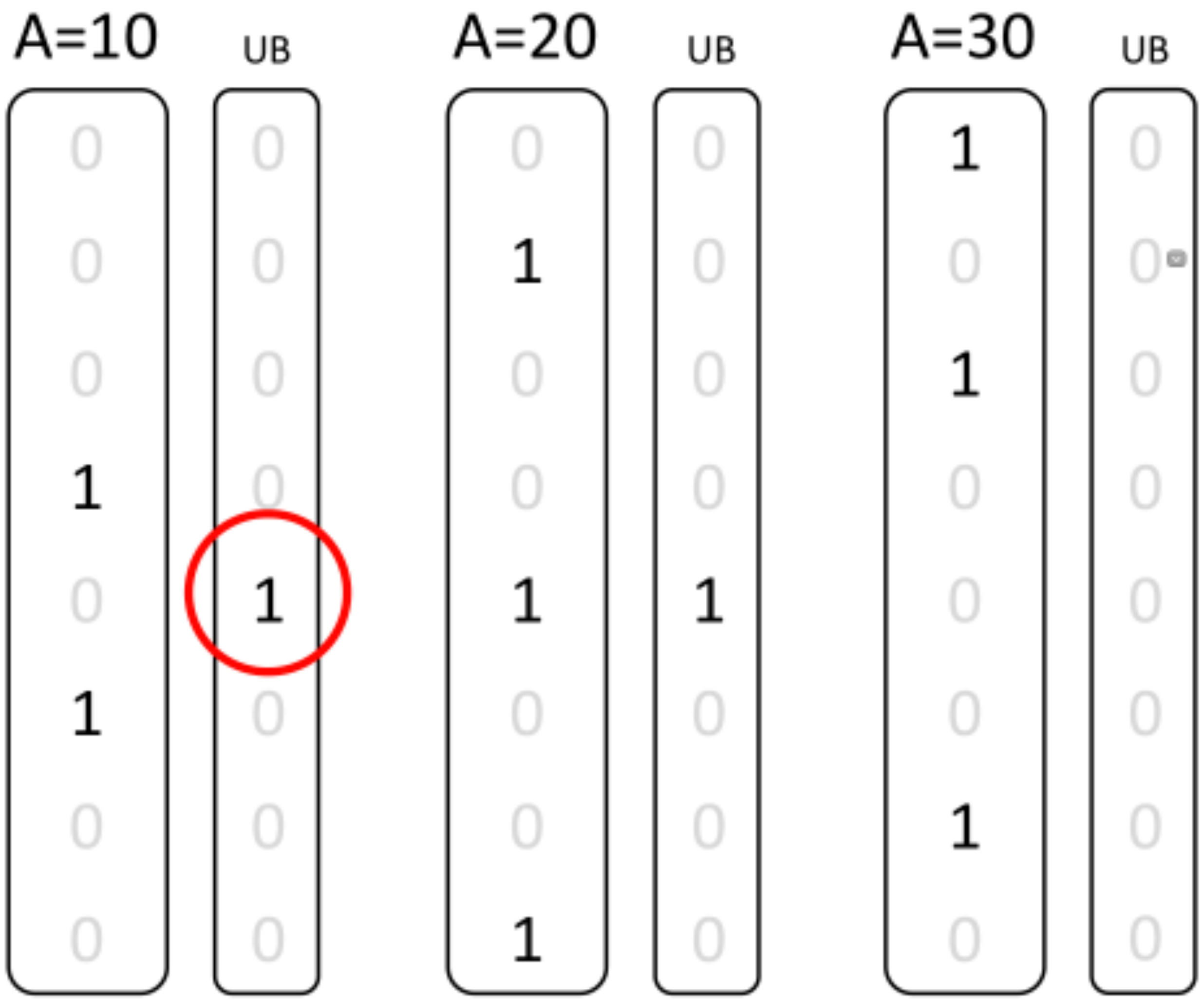
Update row 5 from 20 to 10



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	0	1	1	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

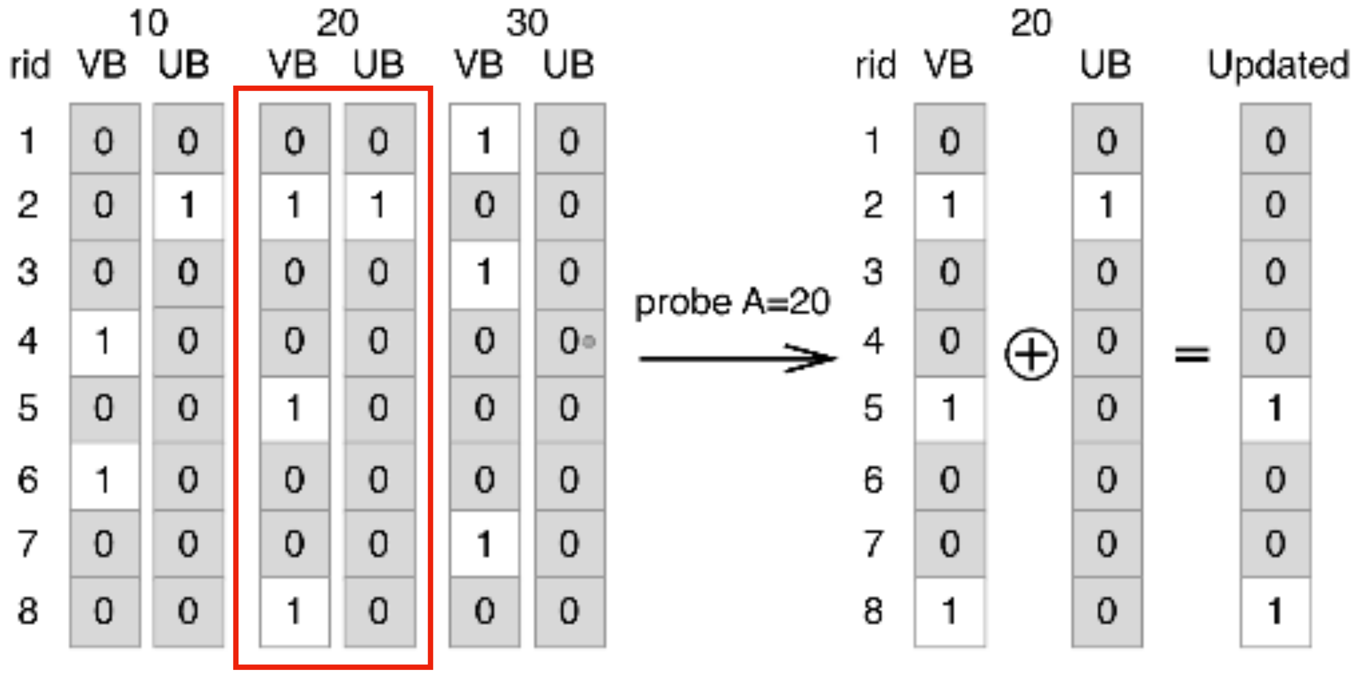
UpBit - Update (5)

Update row 5 from 20 to 10



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

UpBit - Query



	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

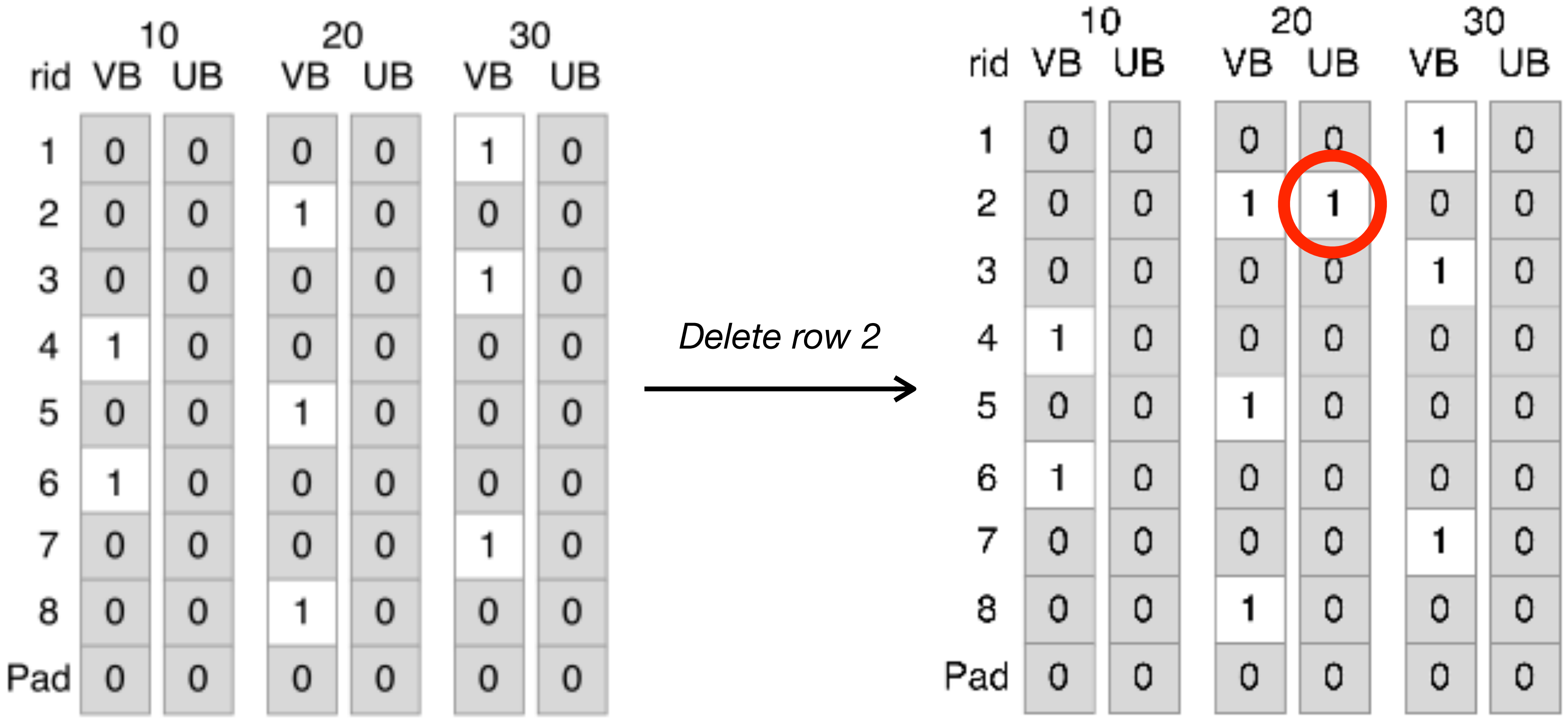
	20		
rid	VB	UB	Updated
1	0	0	0
2	1	1	0
3	0	0	0
4	0	0	0
5	1	0	1
6	0	0	0
7	0	0	0
8	1	0	1

select * from table where columnA = 20

1. Find the bitvector i that corresponds to val, using the VBM which links values to bitvectors

2. Perform bitwise XOR between Vi and Ui

UpBit - Delete row



	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

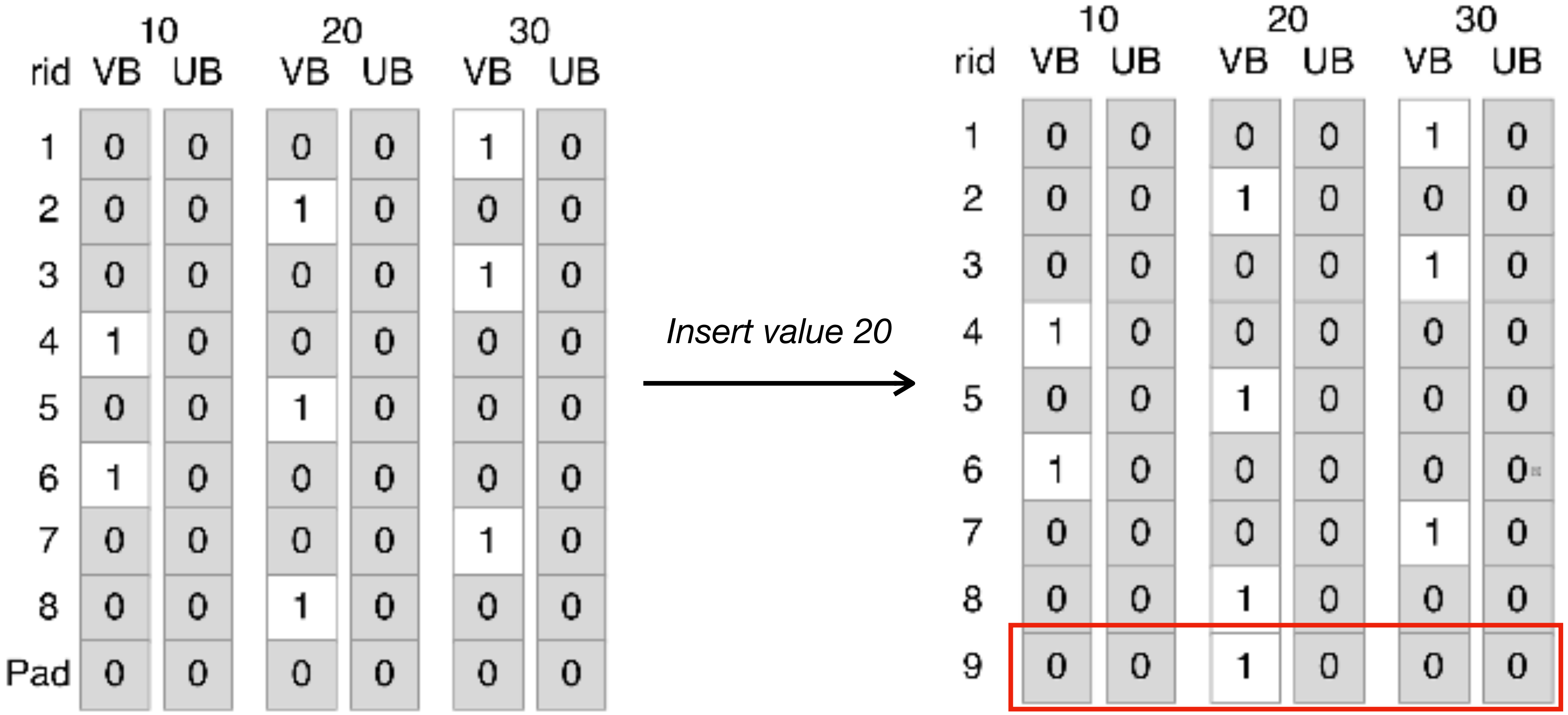
	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

1. We need to retrieve the value Bi of this row k

2. Find the update bitvector corresponding to this value Bi

3. Negate the contents of the selected update bitvector for row k

UpBit - Insert row



	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

1. We need to find the bitvector Bi corresponding to val (Ui)

2. Make sure enough padding space is available

3. We increase the Ui size by one element and we set the new bit equal to one on the Bi bitvector

Does UpBit scale?

Yes!

UpBit Scales

How?

Merge each UB with the corresponding VB

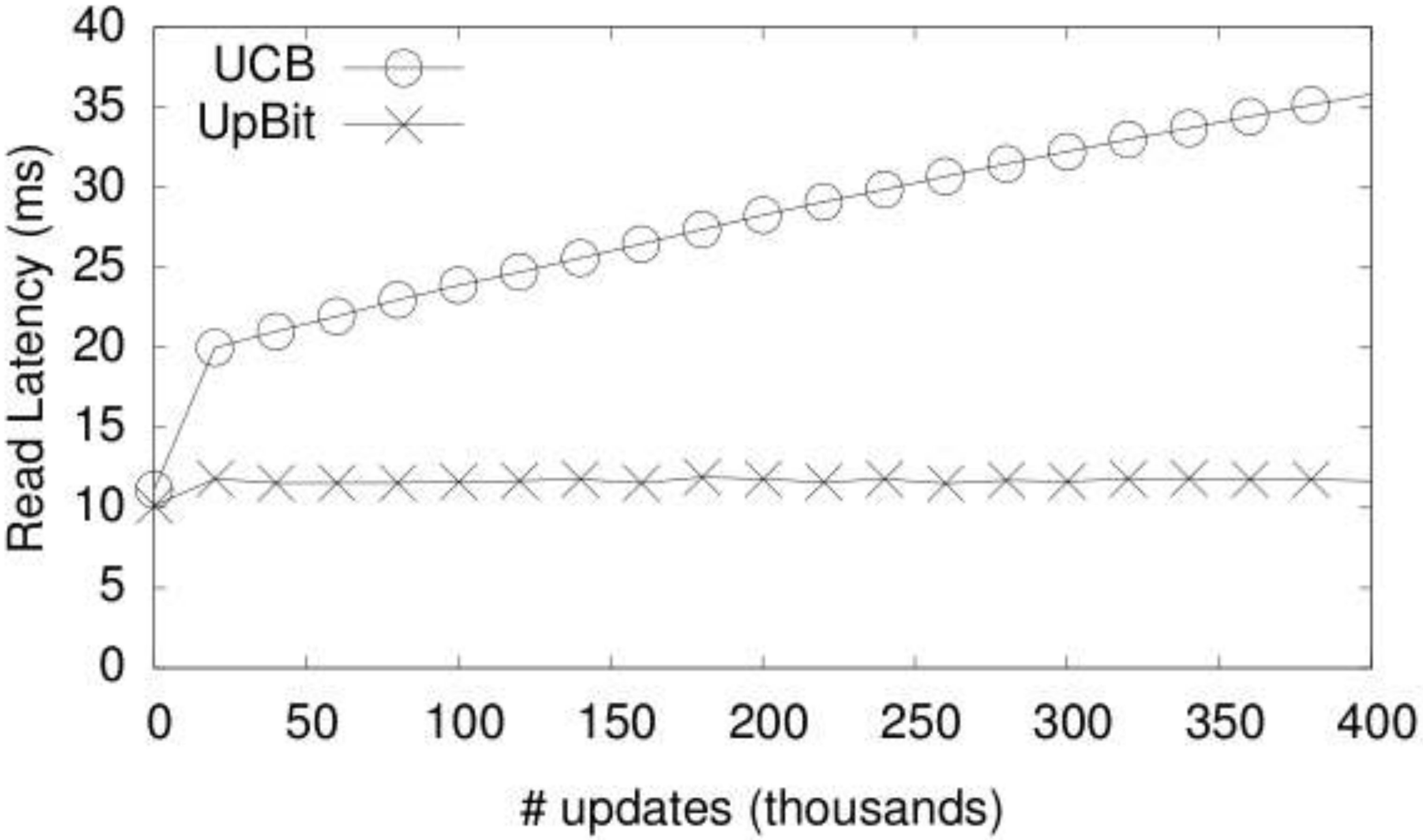
rid	VB		UB		Updated
1	0		0		0
2	1		1		0
3	0		0		0
4	0	⊕	0	=	0
5	1		0		1
6	0		0		0
7	0		0		0
8	1		0		1

When updates > T

- Mark UB as "to be merged”
- Reinitialize UB

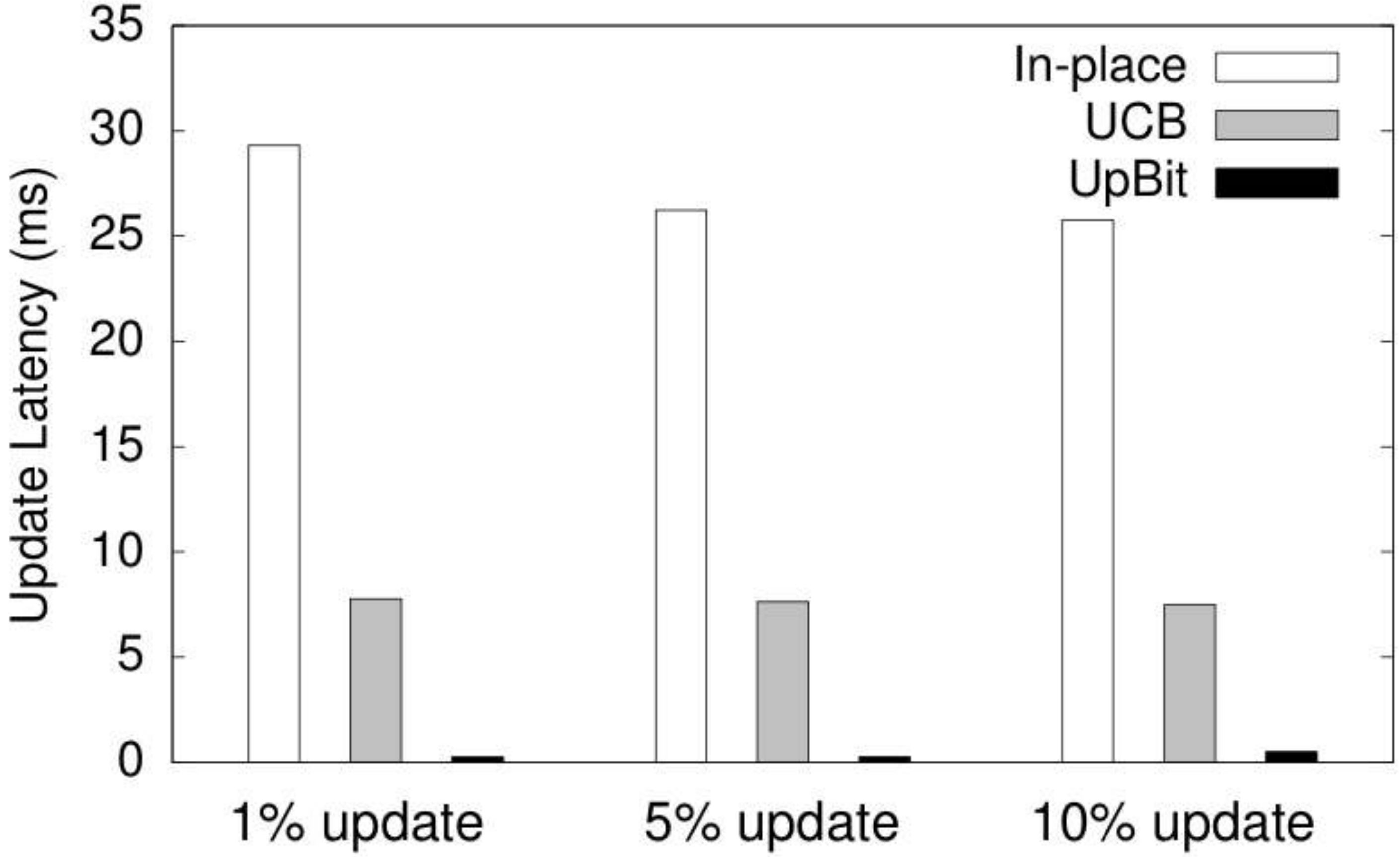
3. Experimental Results

Scalability



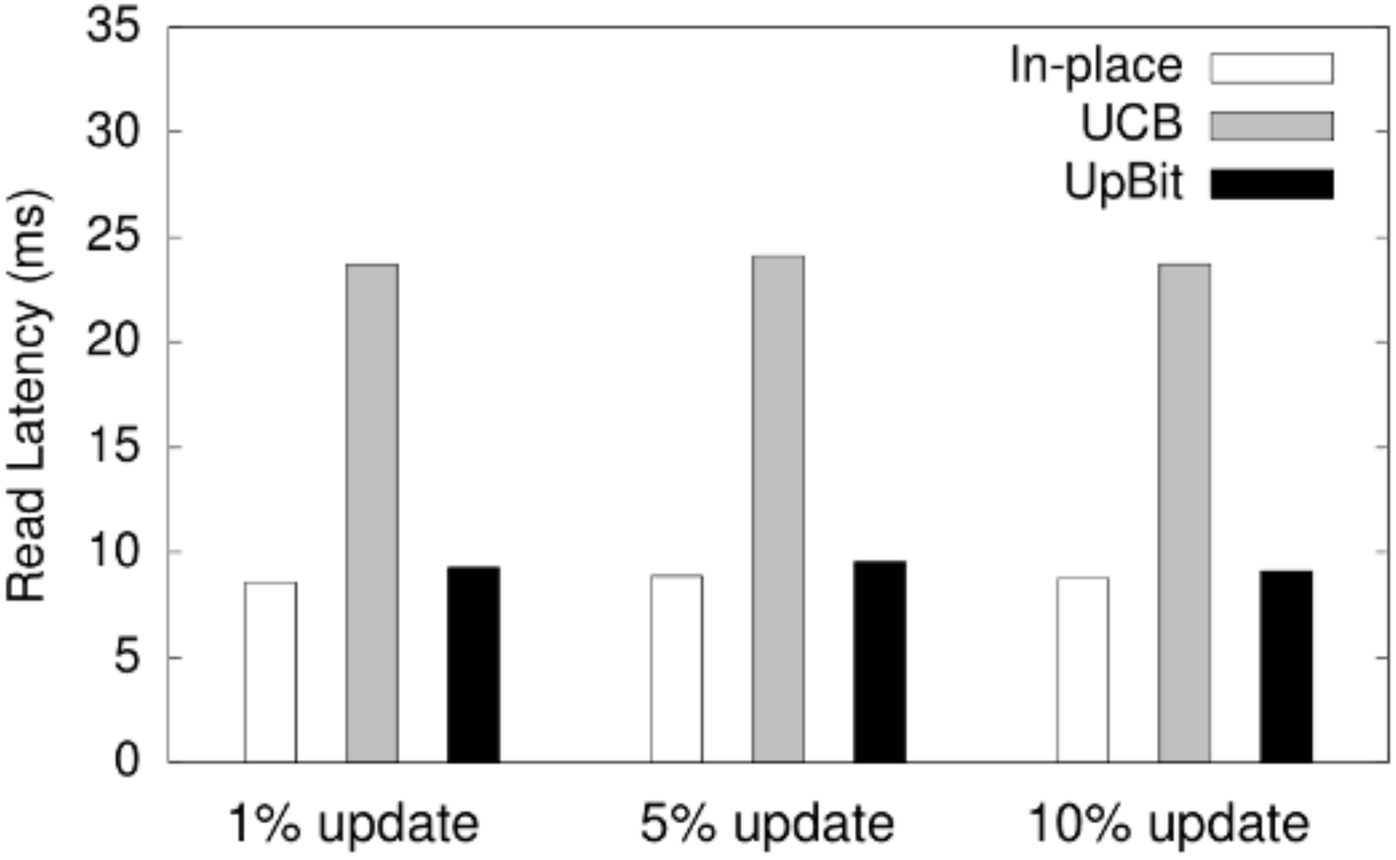
When stressing UpBit with updates, it delivers ***scalable*** read performance, addressing the most important limitation observed for UCB

Update Latency



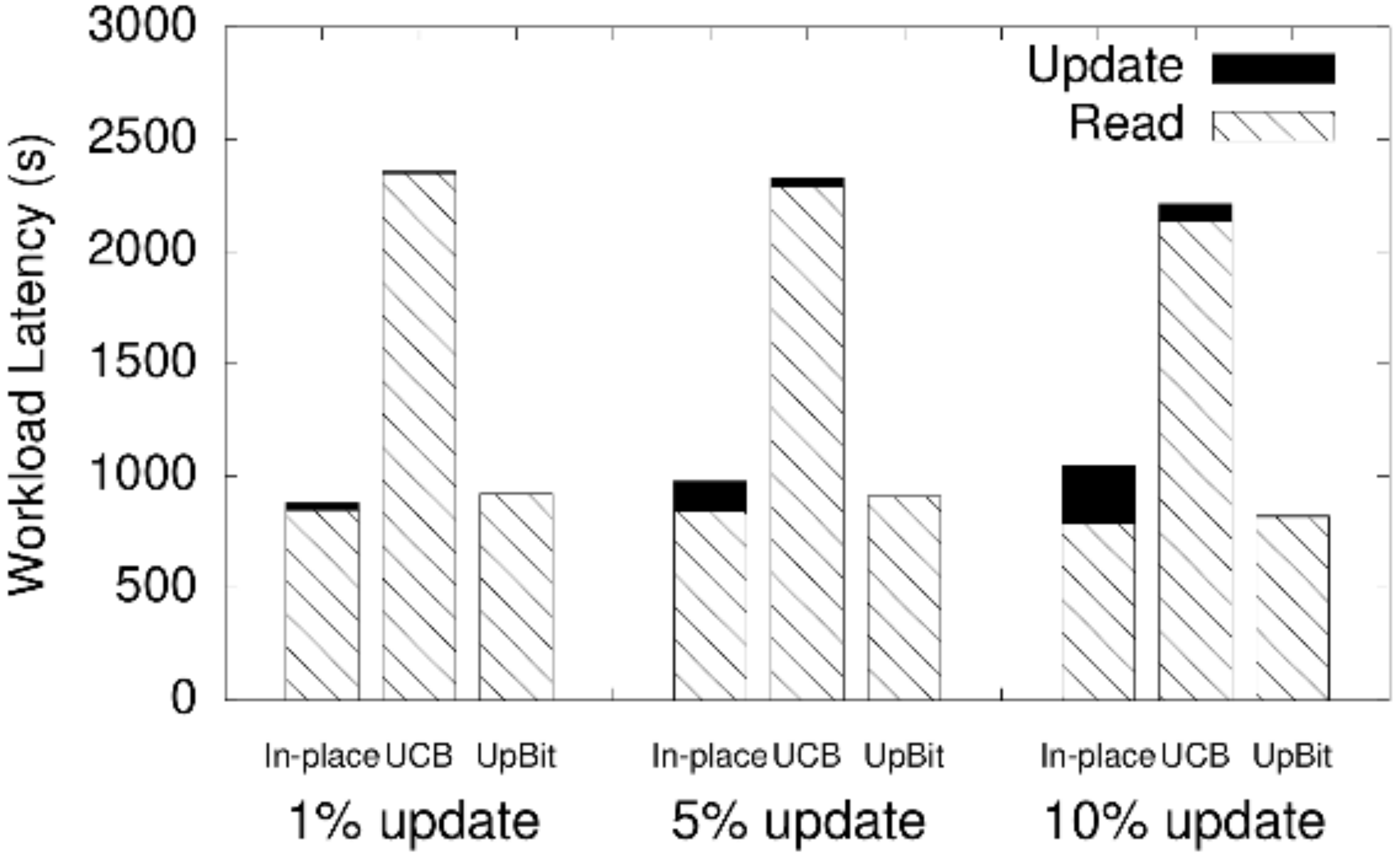
UpBit delivers 51 – 115× faster updates than in-place updates and 15 – 29× faster updates than state-of-the-art update-optimized bitmap index UCB.

Read Latency



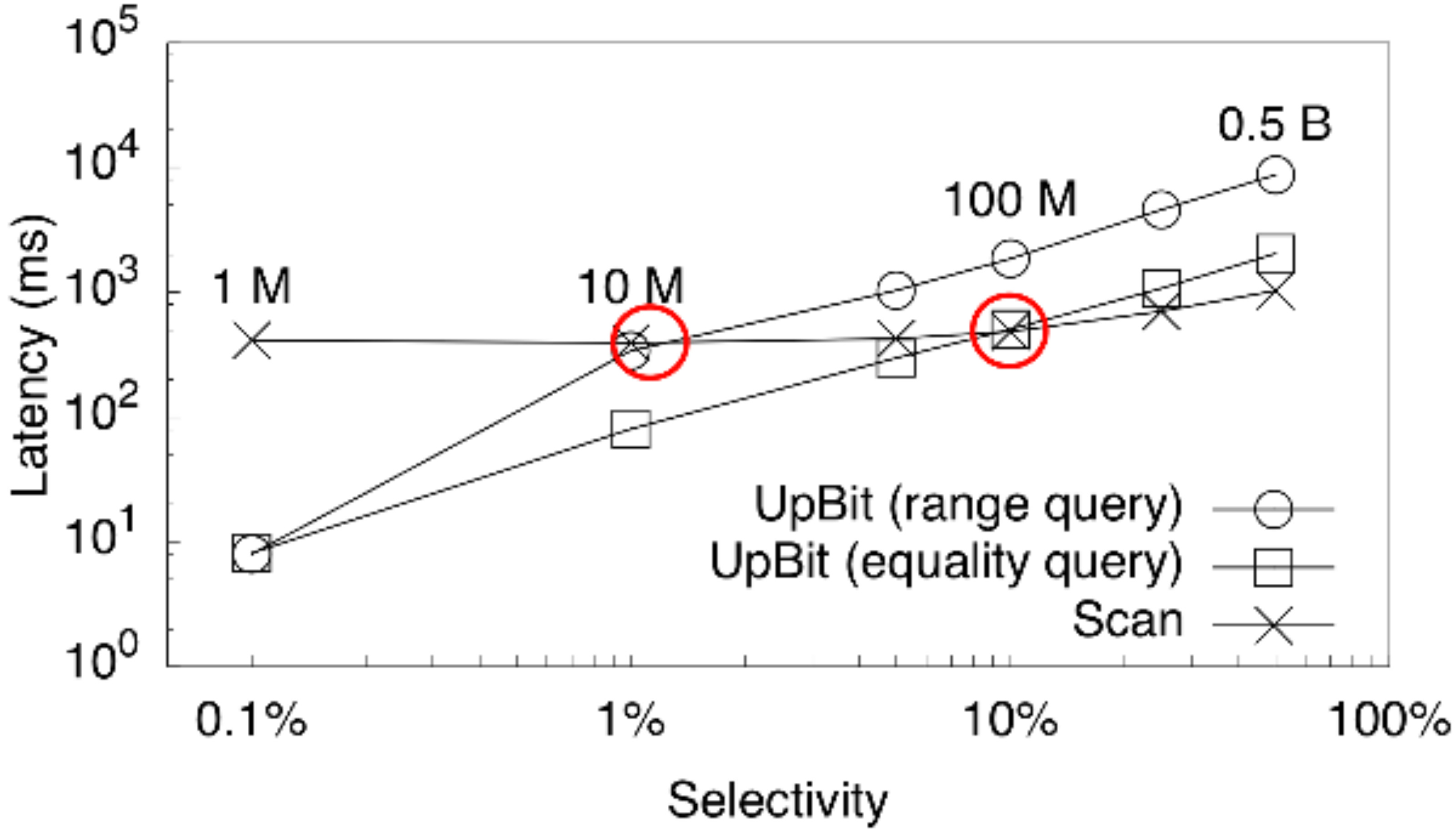
UpBit outperforms update optimized indexes by nearly 3× in terms of read performance while it loses only 8% compared to read-optimized indexes.

Workload Latency



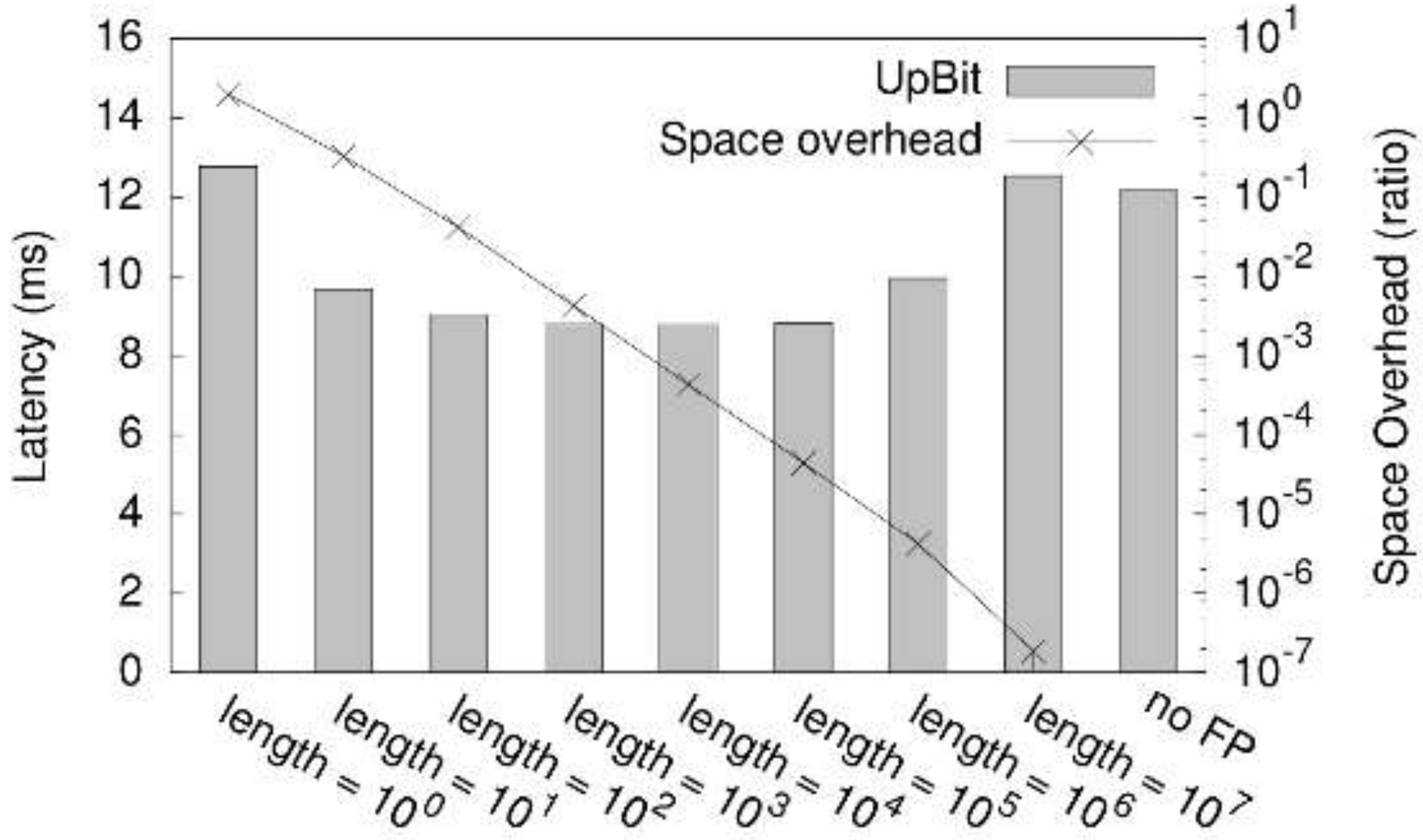
UpBit combines very low overhead on updates and very low reads.

Query Latency



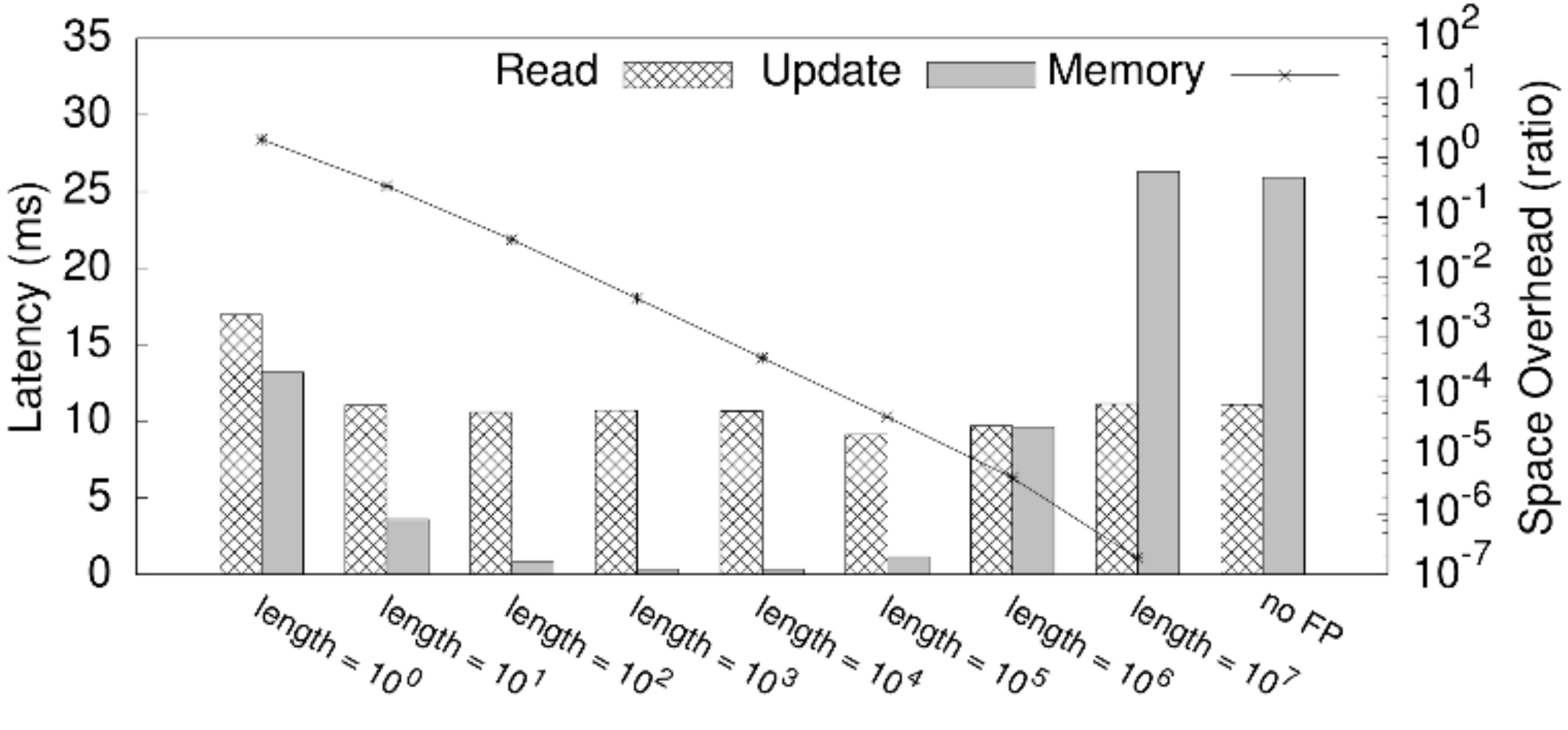
For low selectivity, UpBit is superior

FP Behavior (1)



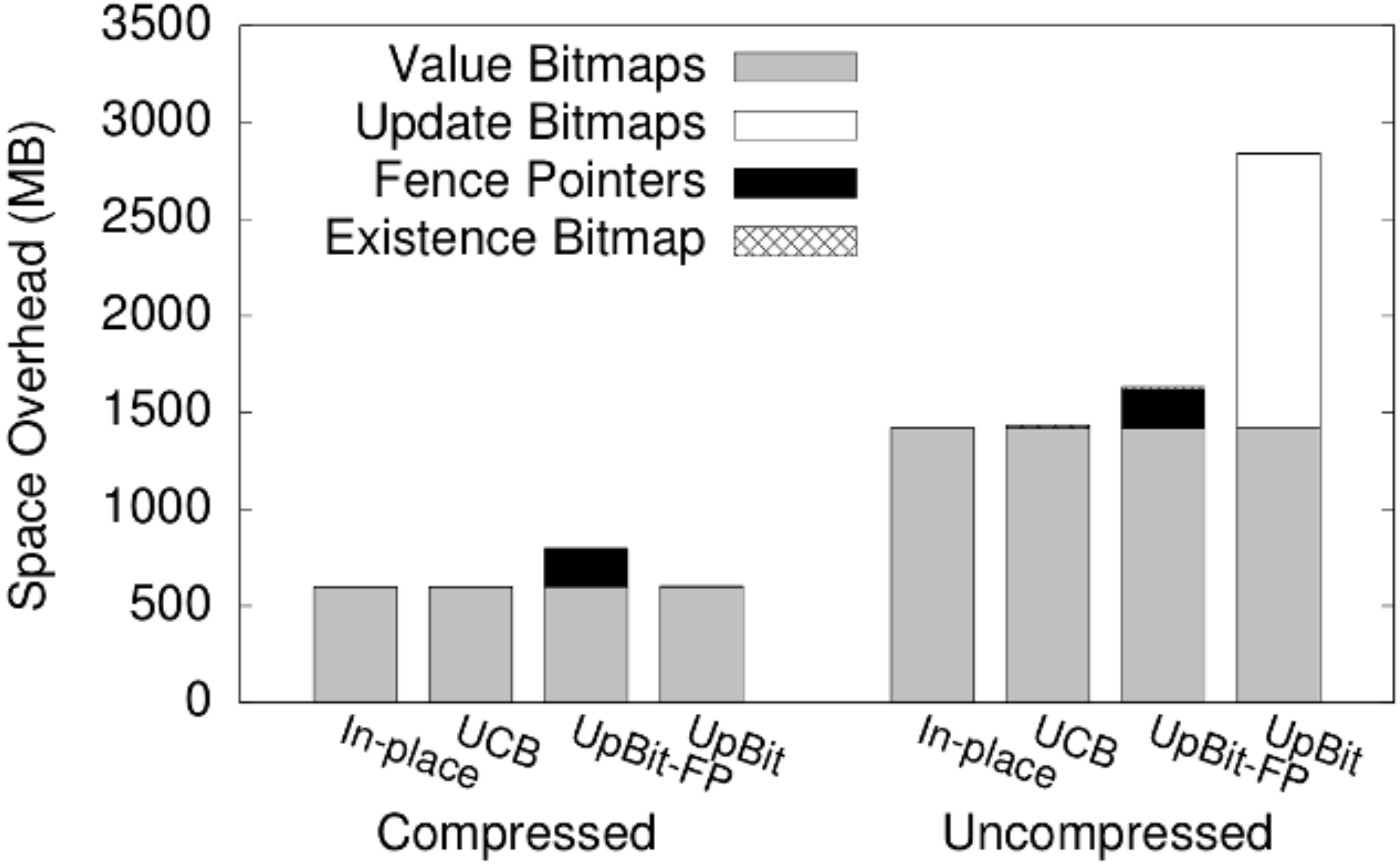
Optimal size: 10^3 - 10^5

FP Behavior (2)



Optimal size: 10^3 - 10^5

UpBit Space Overhead



Minimal when compressed!

Summary

- Bitmap Indexes are ***not efficient for updates***
- ***UCB*** improves this by ***introducing EB***
- UCB ***does not*** scale
- UpBit uses both ***UB*** and ***Fence Pointers*** to achieve ***scalability***

Thank you!