UpBit: Scalable In-Memory Updatable Bitmap Indexing

Guanting Chen, Yuan Zhang
2/21/2019

BACKGROUND

Some words to know

Bitmap: a bitmap is a mapping from some domain to bits

Bitmap Index: A bitmap index is a special kind of database index that uses bitmaps

Bitmap Index

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

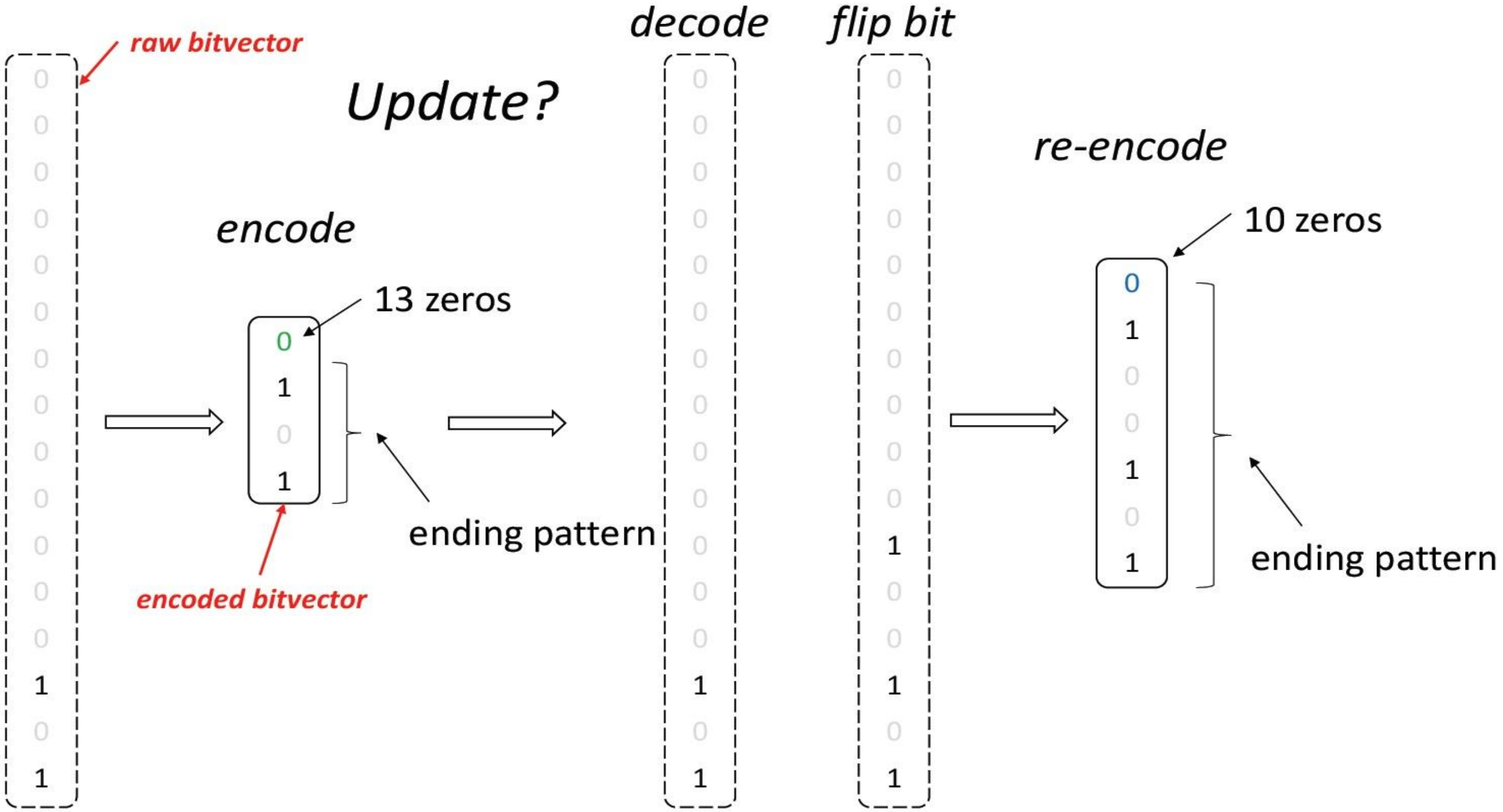
- Typically a single bit per row
- One bitvector per value
- Advantage: Fast read for equality and range queries.
- Need to be compressed for space-efficient

Keep Bitvectors Small

- Bitvectors contain redundancy
- Reduce redundancy
- And improve read performance
- So we have compression and encoding!

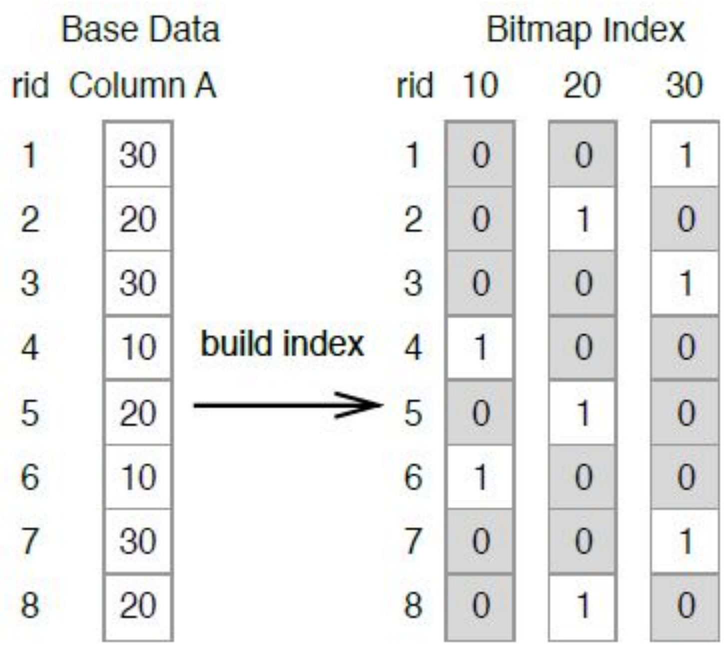
How to compress/encode?

One way:



THE PROBLEM

Traditional Bitmap Index



Base Data

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

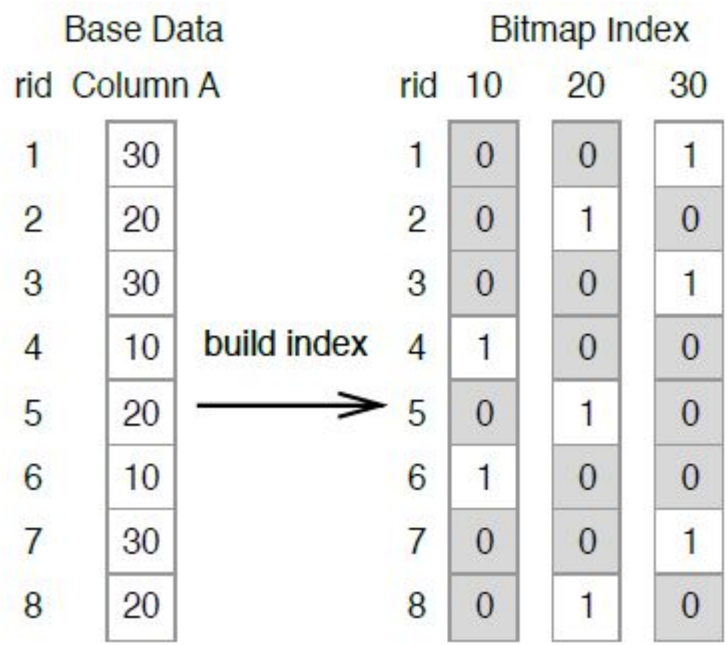
build index

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

- Read-optimized
- Bitvectors are encoded

Traditional Bitmap Index



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

What if we want to update?

- Costly decoding whole bitvectors
- Re-encoding updated bitvectors

Do not offer efficient updates!!!

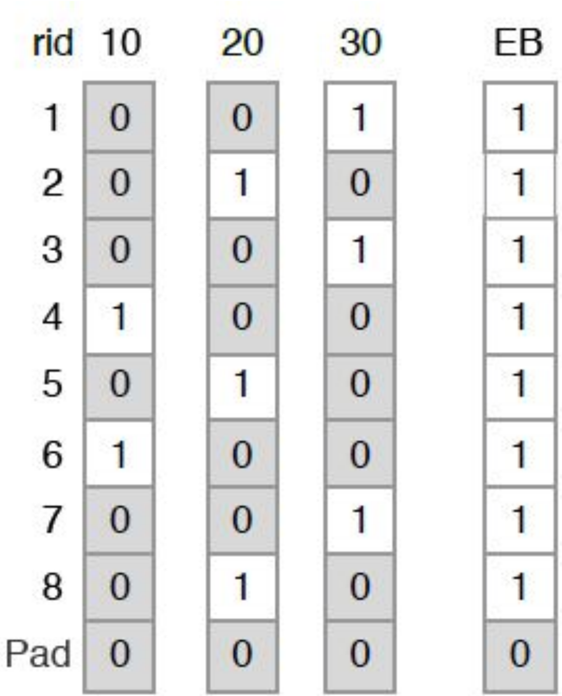
To Solve The Problem

Bitmap indexing should deliver both:

- Good READ performance
- Efficient UPDATE!!!

POSSIBLE SOLUTION

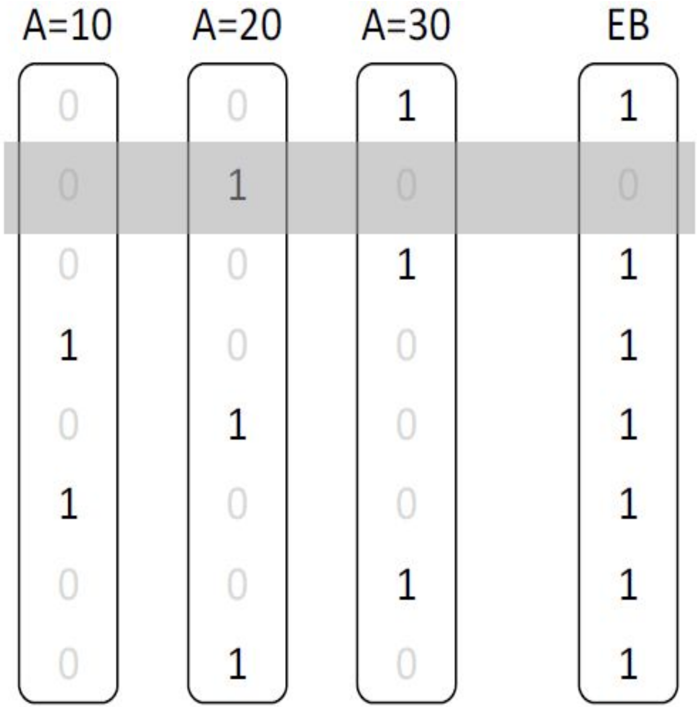
UCB: Update Conscious Bitmap



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

- State-of-the-art update-optimized
- Using existence bitvector(EB): indicate bits are valid or not

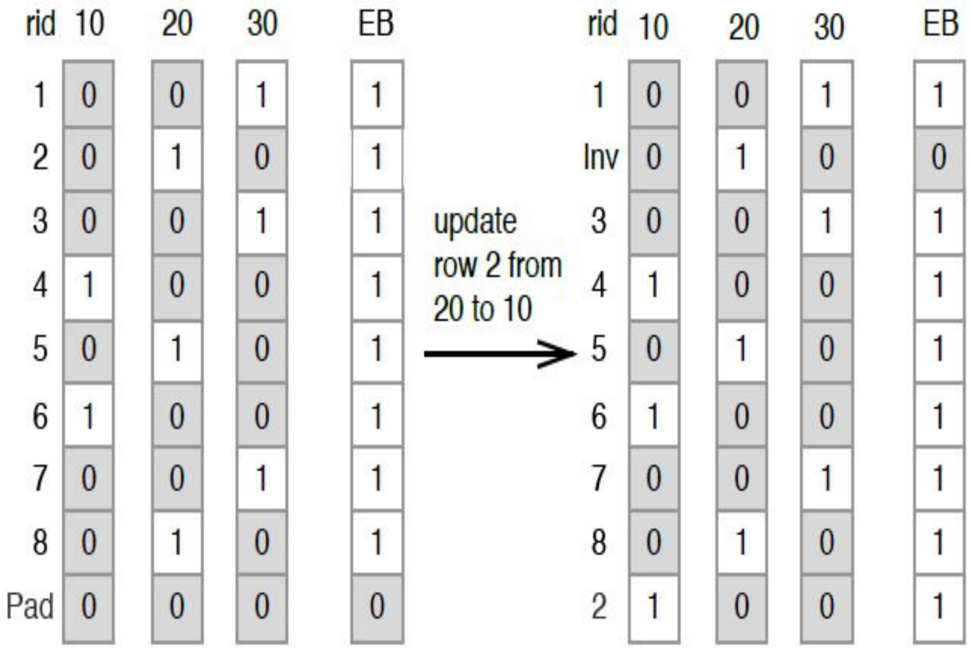
UCB Advantages



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

- Efficient deletes by invalidation

UCB Advantages



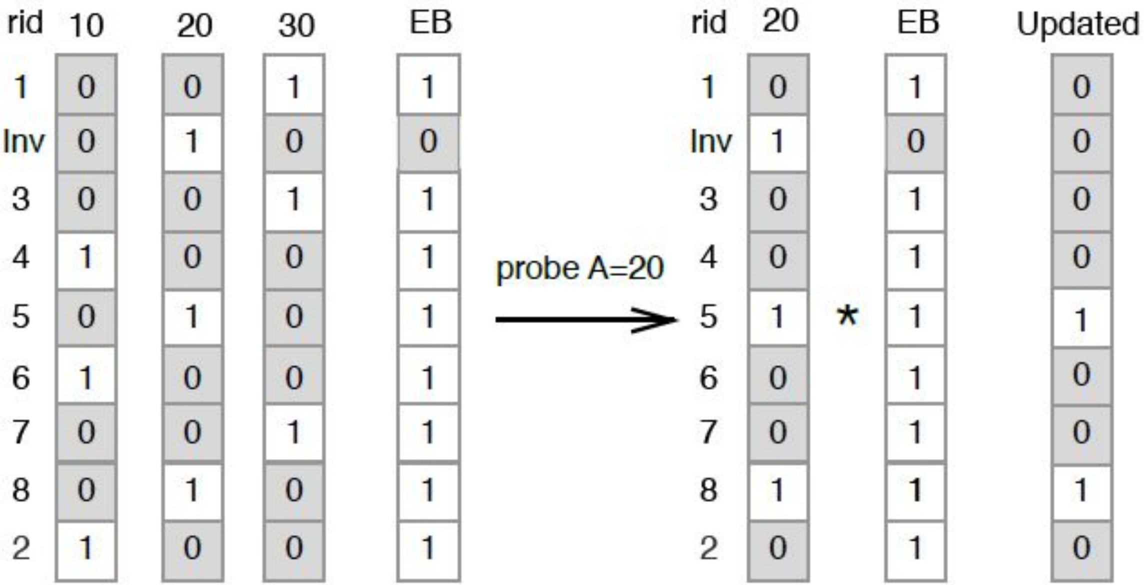
rid	10	20	30	EB
1	0	0	1	1
2	0	1	0	1
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
Pad	0	0	0	0

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

- Faster updates by deleting then appending

UCB Read



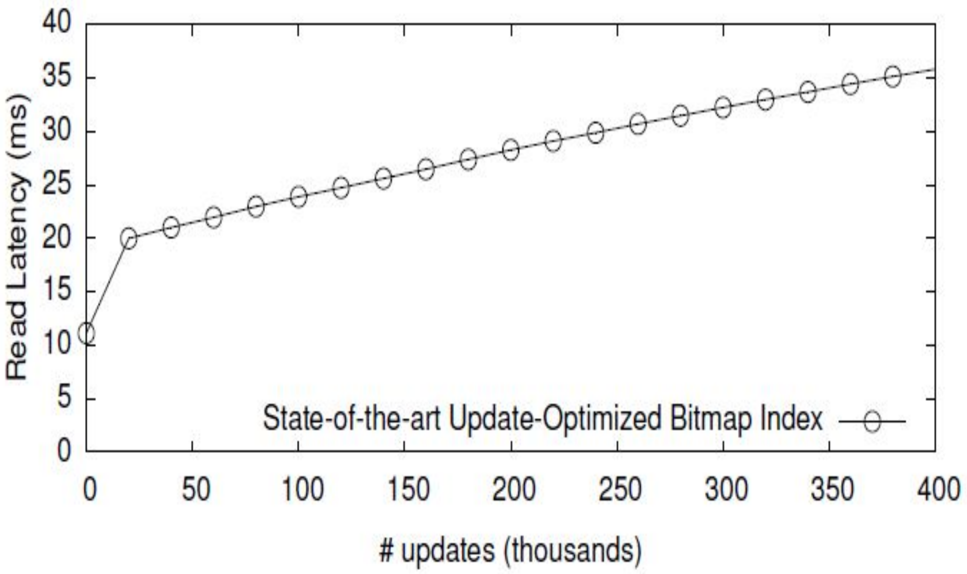
rid	10	20	30	EB
1	0	0	1	1
Inv	0	1	0	0
3	0	0	1	1
4	1	0	0	1
5	0	1	0	1
6	1	0	0	1
7	0	0	1	1
8	0	1	0	1
2	1	0	0	1

rid	20		EB	Updated
1	0		1	0
Inv	1		0	0
3	0		1	0
4	0		1	0
5	1	*	1	1
6	0		1	0
7	0		1	0
8	1		1	1
2	0		1	0

(b) Probe for value B using UCB.

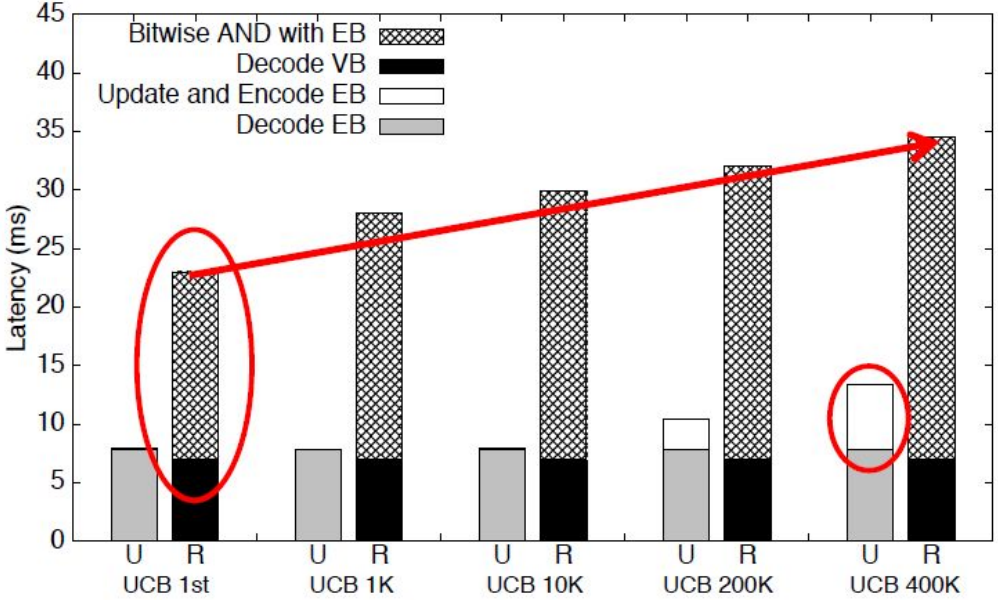
- Read: bitwise AND with EB

UCB Limitations



- More updates, read becomes more expensive

Why?



- Repetitive bitwise operations
- Single auxiliary bitvector

WE STILL HAVE THE PROBLEM !

The Bitmap Index Should:

- Distribute update cost
- Efficiently access compressed bitvectors
- Re-use results of bitwise operations

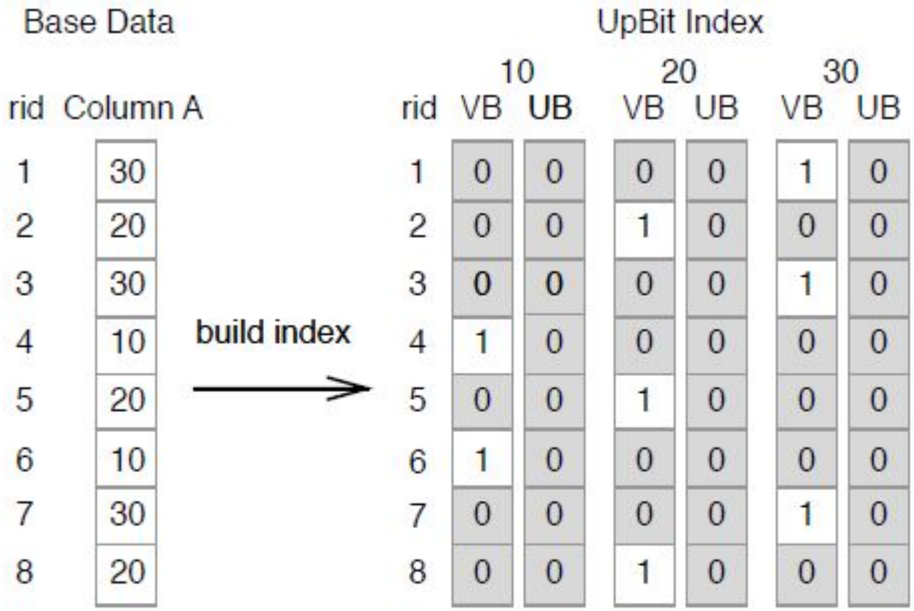
A BETTER SOLUTION

UpBit: Updatable Bitmap Indexing

UpBit

Offer efficient updates without hurting read performance!!!

UpBit



Base Data

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

build index

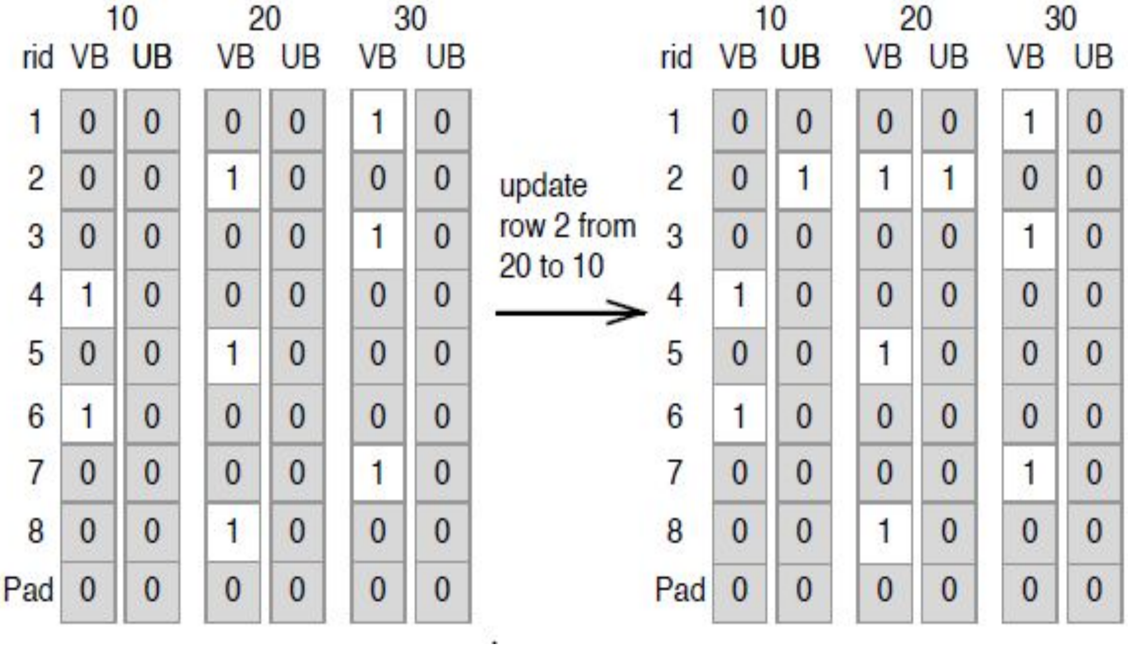
UpBit Index

	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

Maintain update bitvectors(UB):

- One per value
- Initialized to 0s
- Every update flips on a bit on UB
- Double the amount of uncompressed data
- Sparse, compressed size is small(only small ones)

UpBit: Update



	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

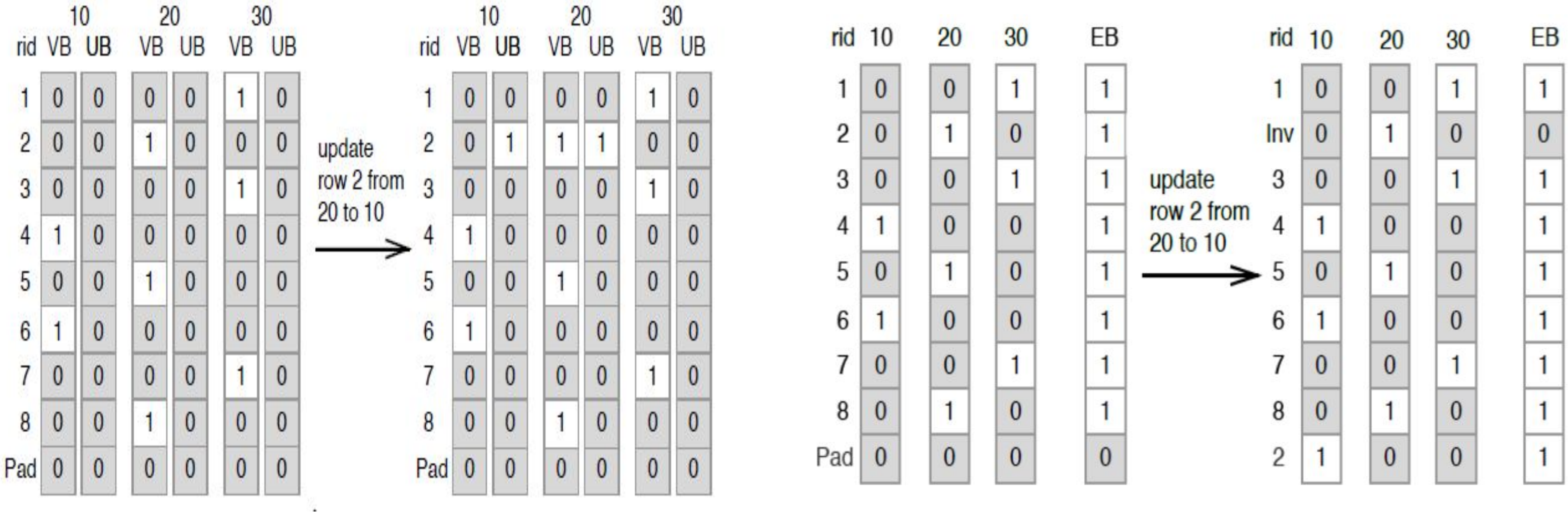
update row 2 from 20 to 10

	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

Three steps to update:

1. Find old value of row 2(20)
2. Flip bit of row 2 of UB of 20
3. Flip bit of row 2 of UB of 10

UpBit VS UCB: Update



	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

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

UpBit:

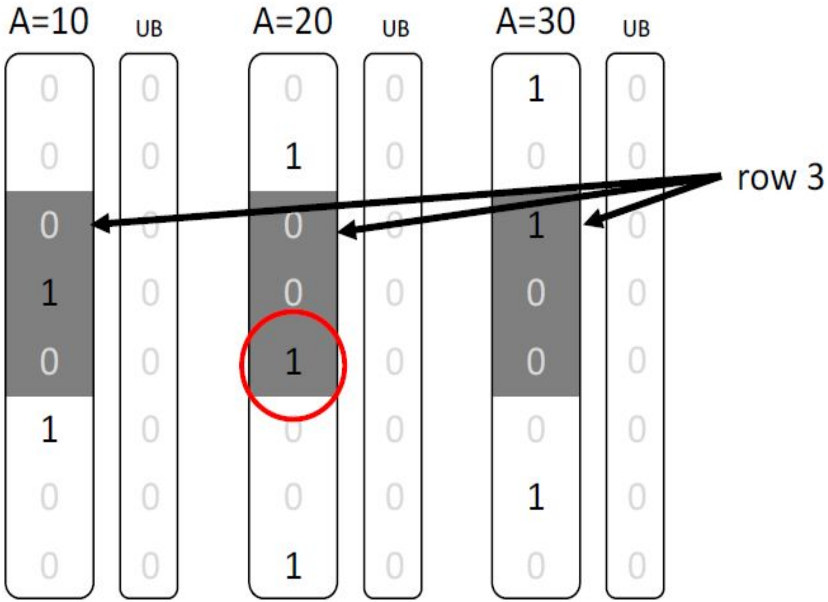
- No single bitvector(EB) receives all updates
- Distribute the update burden to multiple UB

UpBit: Update

Will be faster if we speed up step 1!

How?

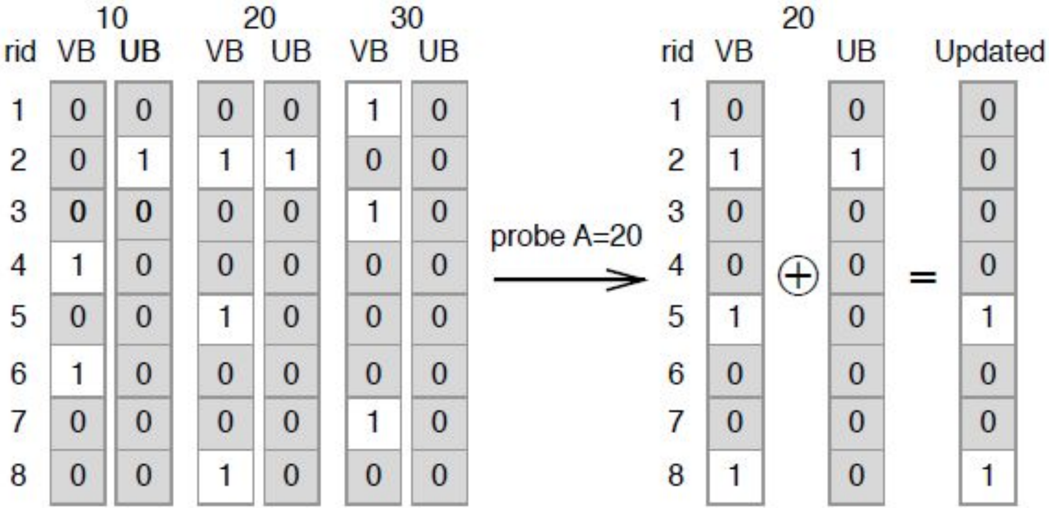
UpBit: Retrieve Value Of A Row



Using fence pointer:

- Avoid decoding entire bitvector
- Decode only a small part of the bitvector
- Efficiently retrieve a value

UpBit: Read



	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

(c) Search for a single value with UpBit.

Return the XOR of VB and UB

UpBit: Read

Can we re-use the result?

How?

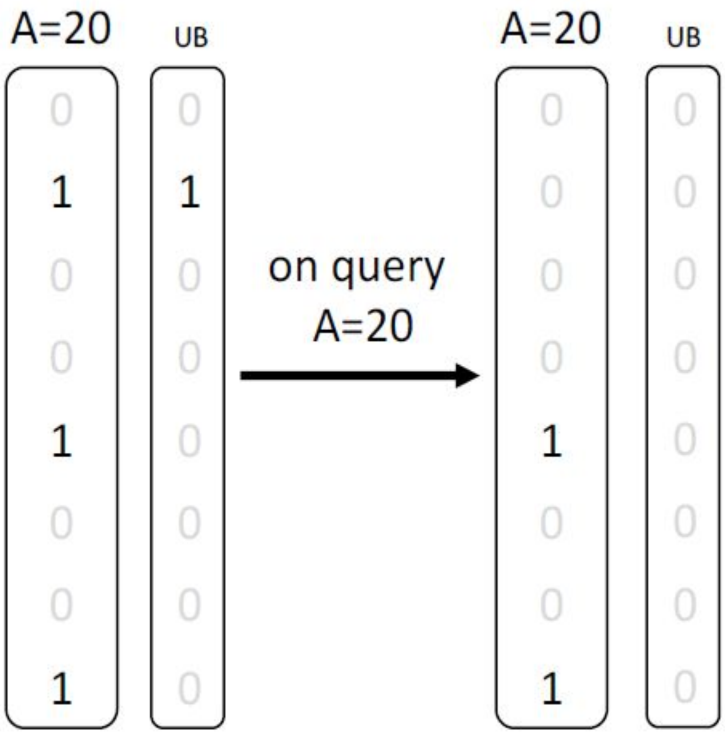
UpBit: Merge

Why merge?

- Accumulated operations lead to less compressible UB
- More expensive bitwise operations and decoding
- Need to maintain high compressibility of UB

How and When to merge?

UpBit: Merge



Merge periodically:

- Maintain a threshold based on # updates

Query-driven merge:

- “query then merge”
- Use the result of XOR
- Update VB using the result
- Set UB to zeros

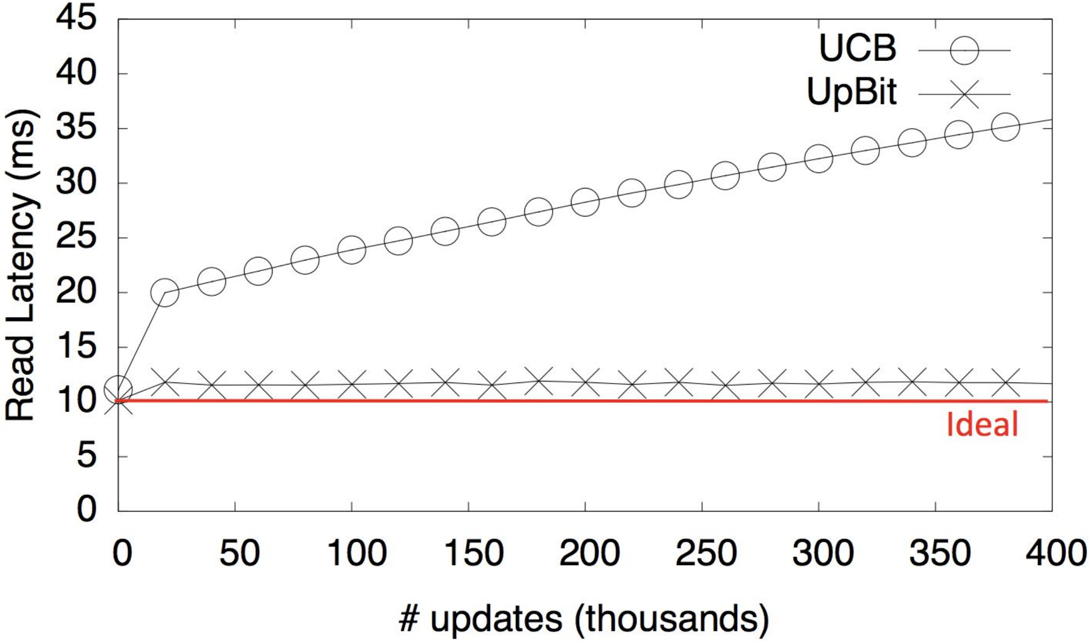
UpBit VS UCB: Read

UCB:	UpBit:
Bitwise AND between VB and EB	Bitwise XOR between VB and UB
Decode and encode entire bitvector	Partially decode and encode bitvector
No merge(merge EB means merge with all VB)	Merge to maintain compressibility
Read performance does not scale with updates	Read performance scales with updates ?

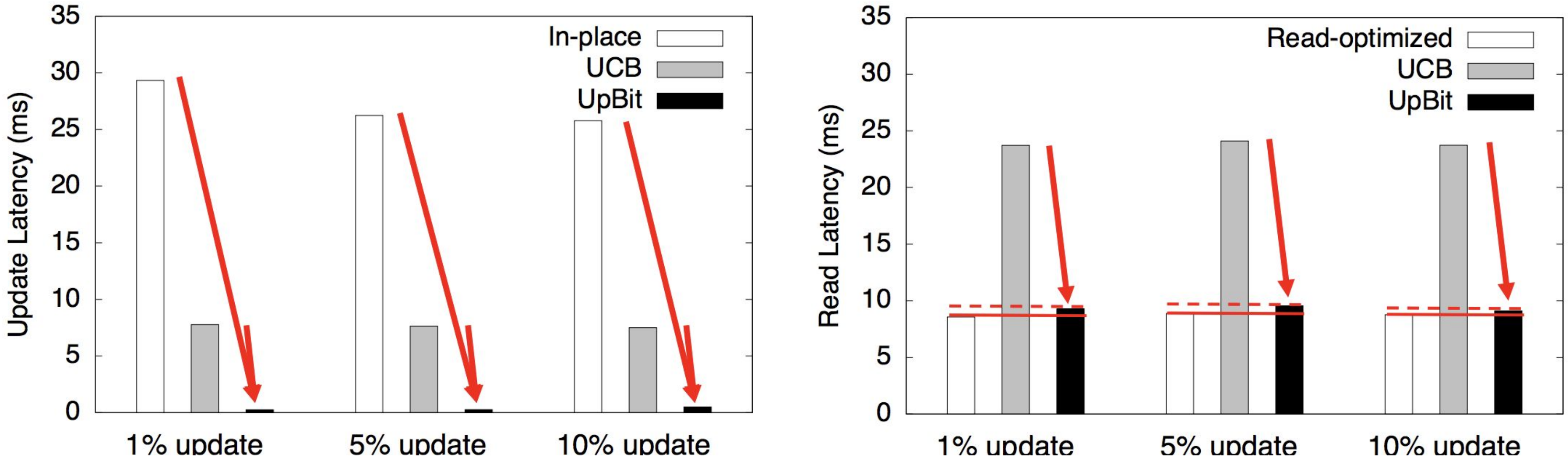
BENCHMARKING

Scalability

- 100M values of real life data set
- 100 unique values of domain
- 100k operations of query mix



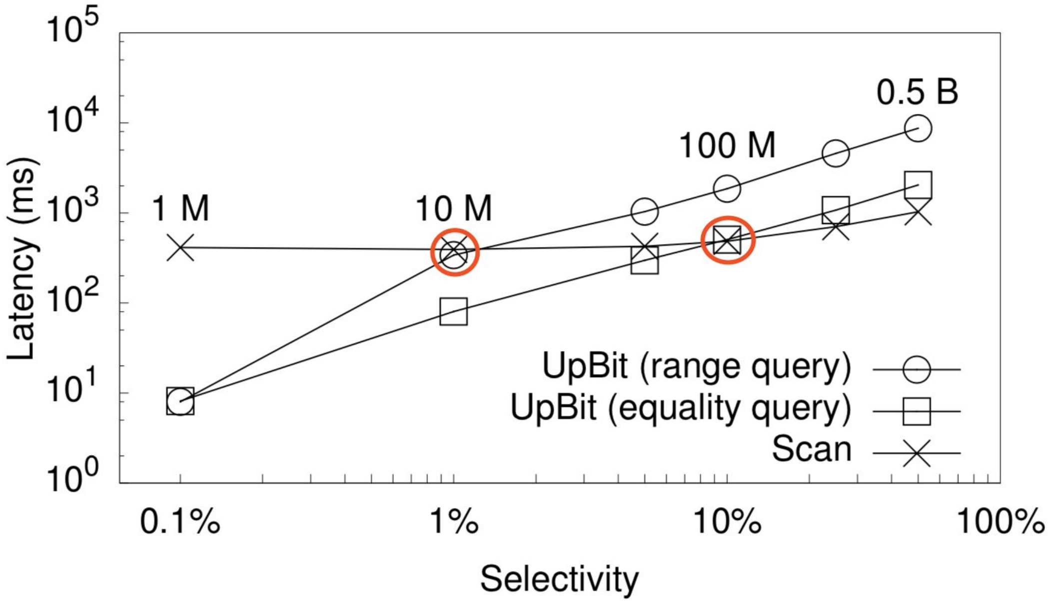
Update & Read Performance



- Fence pointer enables fast read & update
- UpBit has 8% read overhead at most due to XOR operations

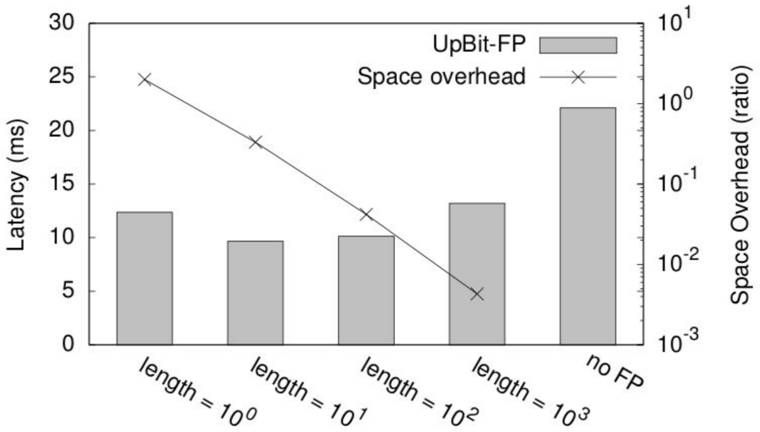
UpBit vs Scan

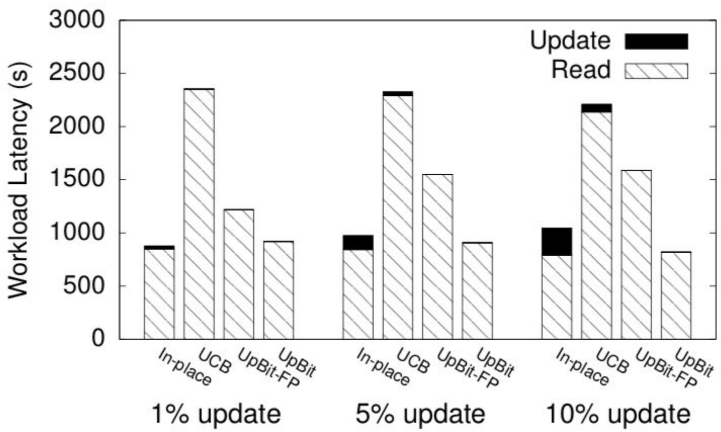
- Compared with a fast scan, UpBit is faster for range queries with up to 1% selectivity.



Design element - Fence Pointer

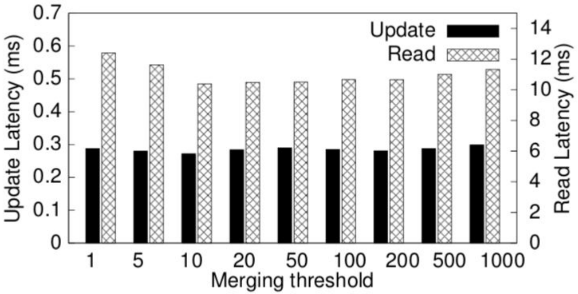
UpBit-FP: Using fence pointer and only **ONE** update bitvector(like UCB's EB)



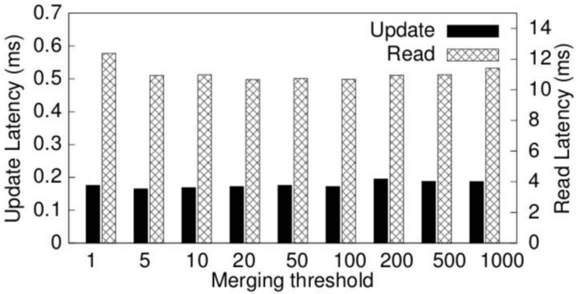


- Fence pointer alone - NOT maintaining COMPRESSIBILITY
- Updates bitvectors needs fence pointer to amortize their cost

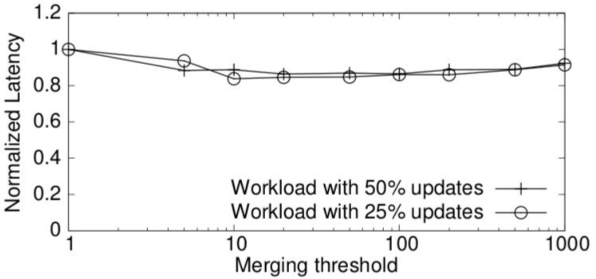
Design element - Merging Threshold



20% updates workload



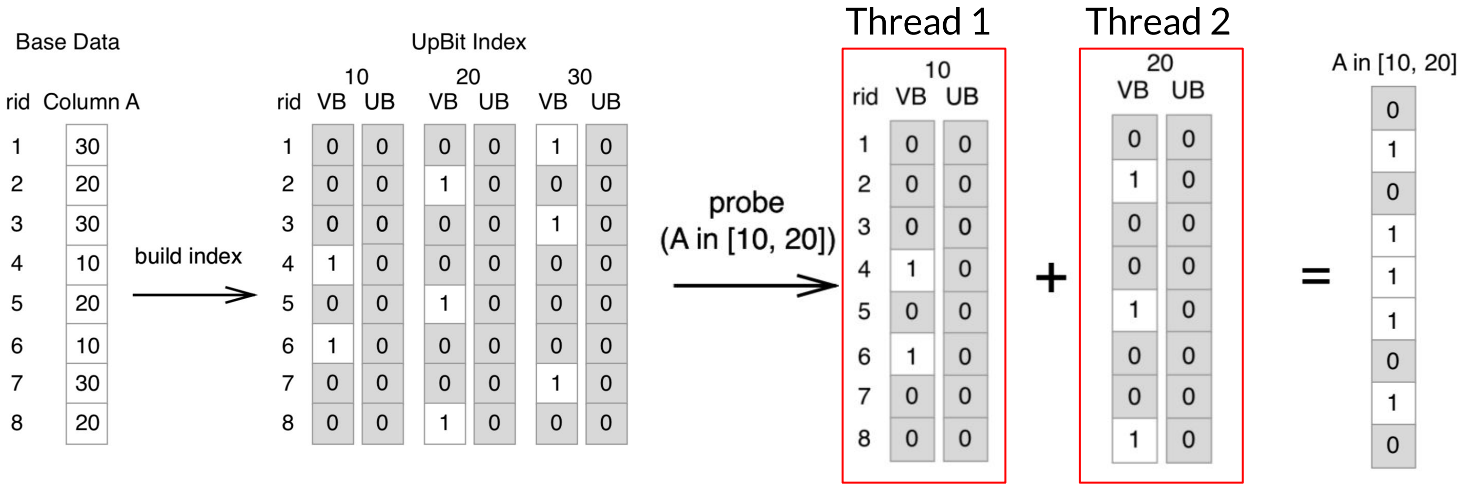
50% updates workload



overall comparison

- Threshold of 10 updates(bits set to 1) leads to fastest workload execution

Further improvement - Parallelism



Base Data

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

UpBit Index

	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

Thread 1

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

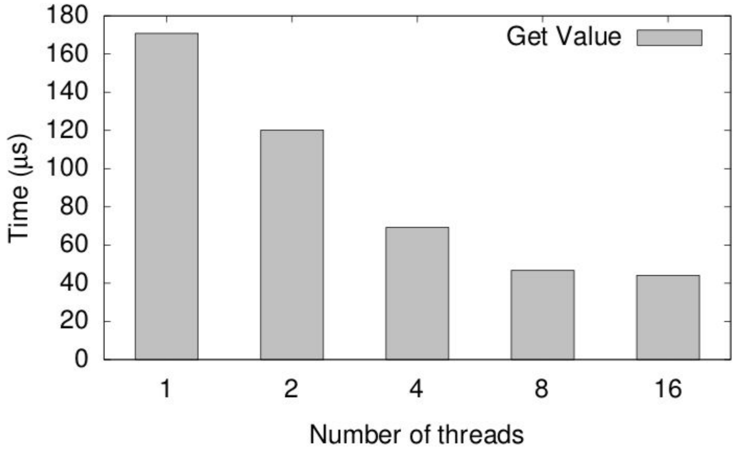
Thread 2

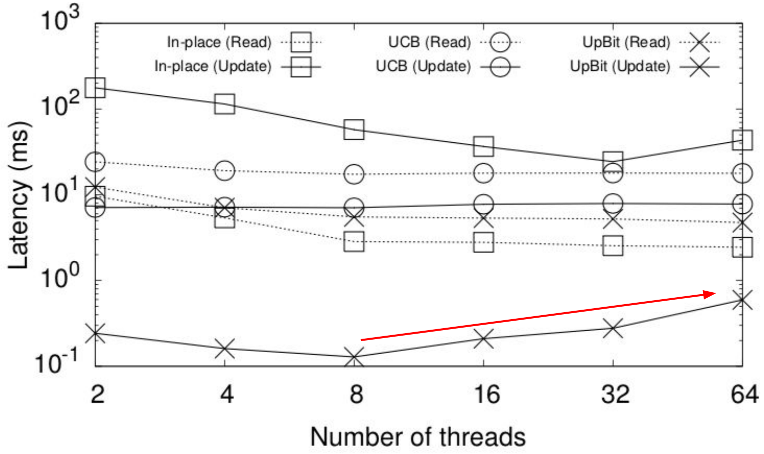
20	
VB	UB
0	0
1	0
0	0
0	0
1	0
0	0
0	0
1	0

A in [10, 20]
0
1
0
1
1
1
0
1
0

- Each pair of VB & UB are actually decoupled and domain-isolated
- Which means they can be queried in parallel

Further improvement - Parallelism





- Performance UpBit(Update) does not scale with the number of threads after 8

CONCLUSION

Design goals

- Multiple Update Vectors → distribute update cost → maintain more compressible UBs
- Fence pointer → partial decoding → efficient retrieval of a value at arbitrary position
- Query-driven UB merging → keep maintaining high compressibility

Pros

- **On the surface:**
    - **Straightforward design idea, clear illustration**
      (easy to understand)

- **Underneath:**
    - **Interesting details and effects**
      (the way it distributes update cost and maintains compressibility is really cool!)

    - **Concrete pseudocode**
      (better understanding of logical implementation underneath the design)

Cons

- Does not cover the marterialization of BitMap index
- What if Read-Optimized BitMap index also employs fence pointer?
- What about different cardinality?

THANK YOU