UpBit: Scalable In-Memory Updatable Bitmap Indexing

Manos Athanassoulis[1]   Zheng Yan[2]   Stratos Idreos[1]

[1]Harvard University   [2]University of Maryland

Indexing for Analytical Workloads

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

Specialized indexing

- ☑ Compact representation of query result
- ☑ Query result is readily available

Bitvectors

- ☑ Can leverage fast Boolean operators
- ☑ Bitwise AND/OR/NOT faster than looping over meta data

Bitmap Indexing Limitations

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

Index Size

☒ Space-inefficient for large domains

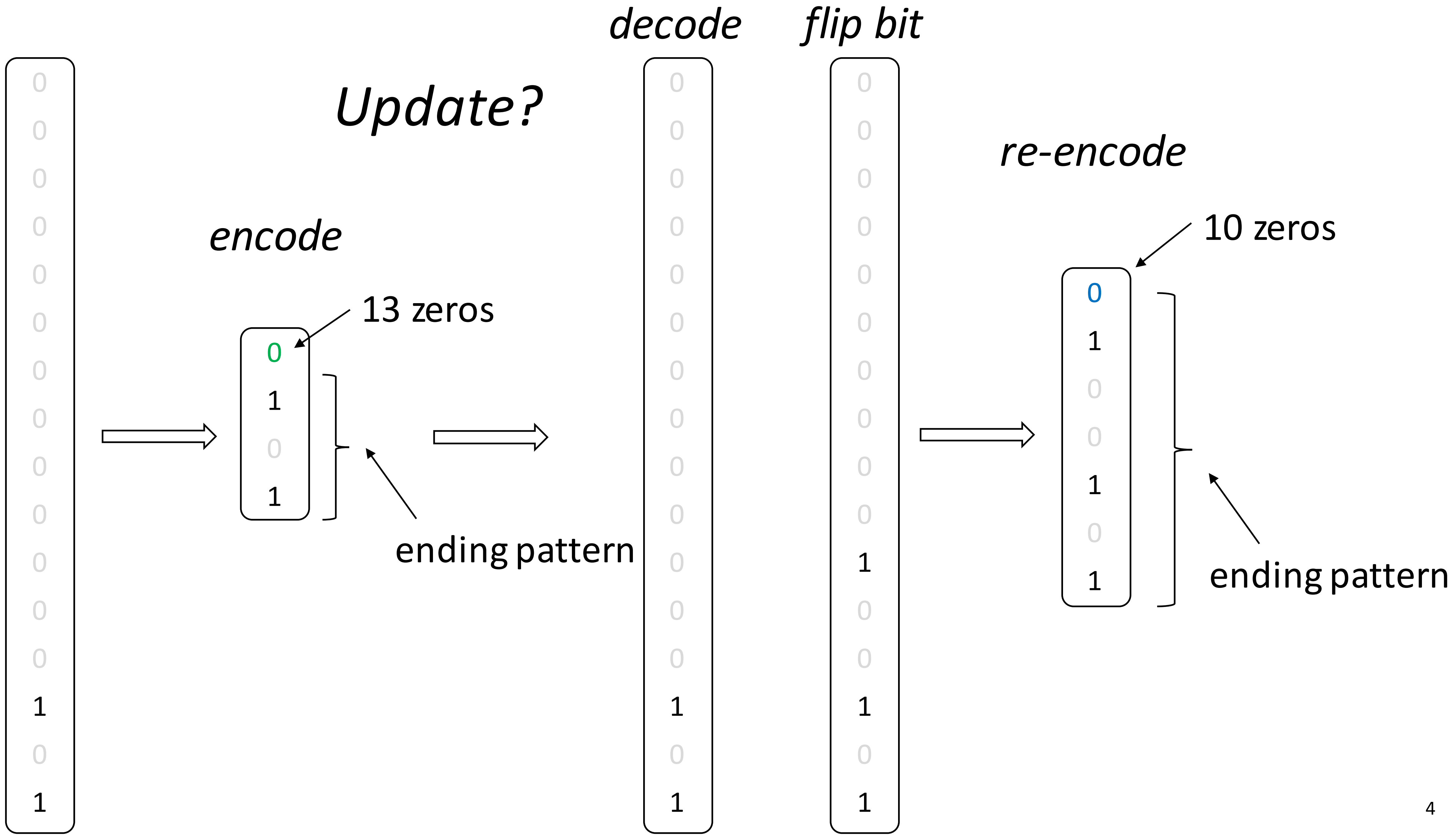
☑ Addressed by bitvector encoding/compression

core idea: *run-length encoding* in prior work

but …

☒ Updating encoded bitvectors is **very** inefficient

Update?


0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
encode
13 zeros
0 1 0 1
ending pattern
decode
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
flip bit
0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1
re-encode
10 zeros
0 1 0 0 1 0 1
ending pattern

Goal

Bitmap Indexing with efficient Reads & Updates

Prior Work: Bitmap Indexing and Deletes

Update Conscious Bitmaps (UCB), SSDBM 2007

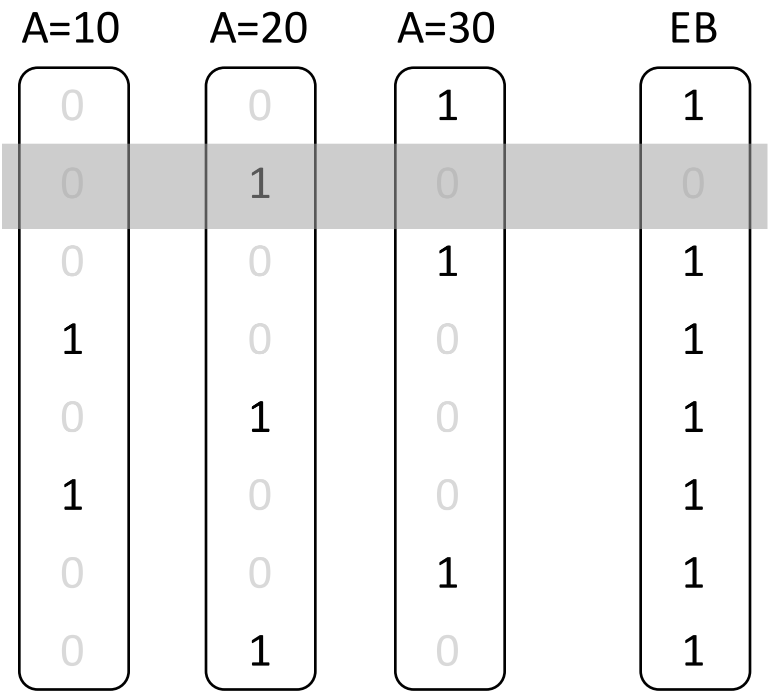
A=10	A=20	A=30	EB
0	0	1	1
0	1	0	1
0	0	1	1
1	0	0	1
0	1	0	1
1	0	0	1
0	0	1	1
0	1	0	1

efficient deletes by invalidation

existence bitvector (EB)

Prior Work: Bitmap Indexing and Deletes

Update Conscious Bitmaps (UCB), SSDBM 2007



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

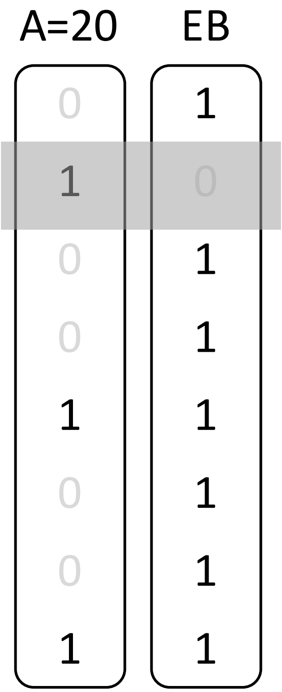
existence bitvector (EB)

reads?

bitwise AND with EB

updates?

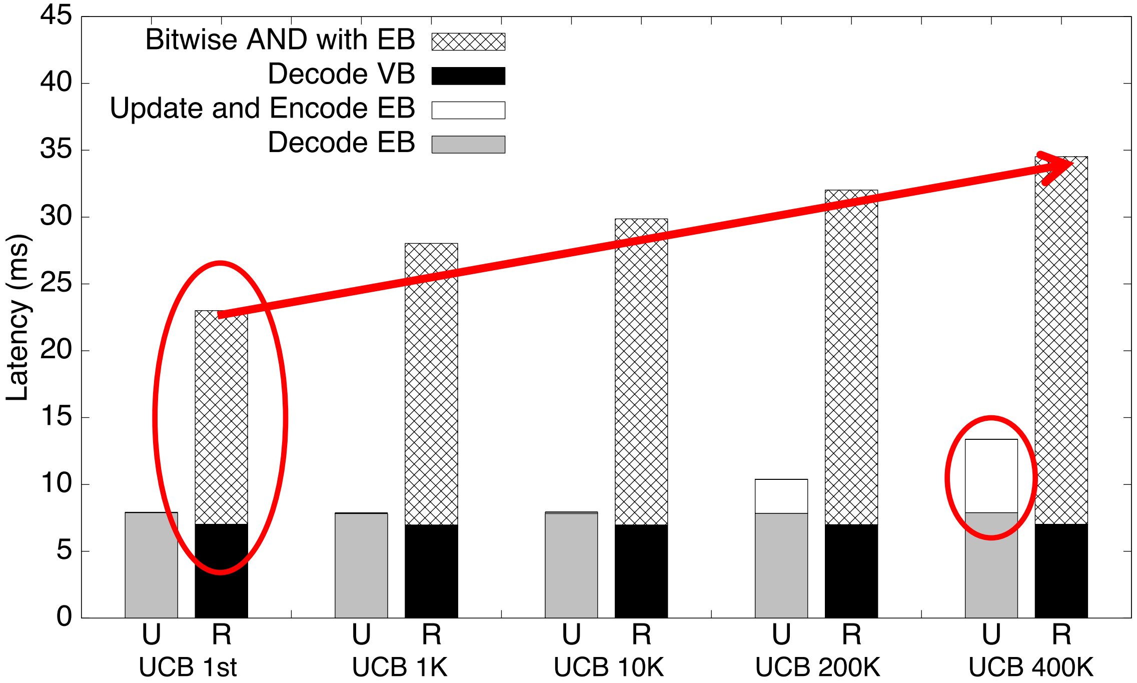
delete-then-append



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

Prior Work: Limitations

n=100M tuples, d=100 domain values, 50% updates / 50% reads



read cost increases with #updates

why?

bitwise AND with EB is the bottleneck

update EB is costly for >> #updates

UCB performance does not scale with #updates

single auxiliary bitvector

repetitive bitwise operations

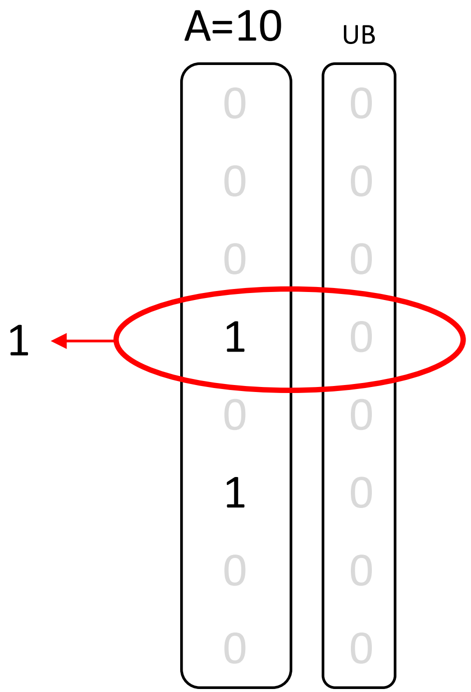
Bitmap Indexing for Reads & Updates

distribute update cost

efficient random accesses in compressed bitvectors

query-driven re-use results of bitwise operations

Design Element 1: update bitvectors

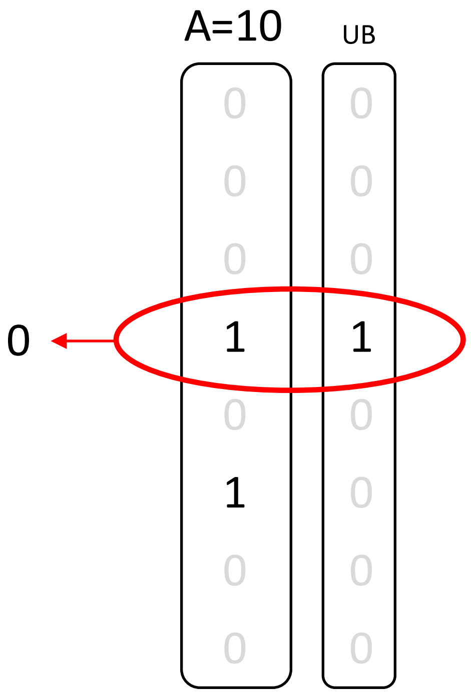


one per value of the domain
initialized to 0s

the current value is the XOR

every update flips a bit on UB

Design Element 1: update bitvectors



one per value of the domain
initialized to 0s

the current value is the XOR

every update flips a bit on UB

... distribute the update burden

Updating UpBit …

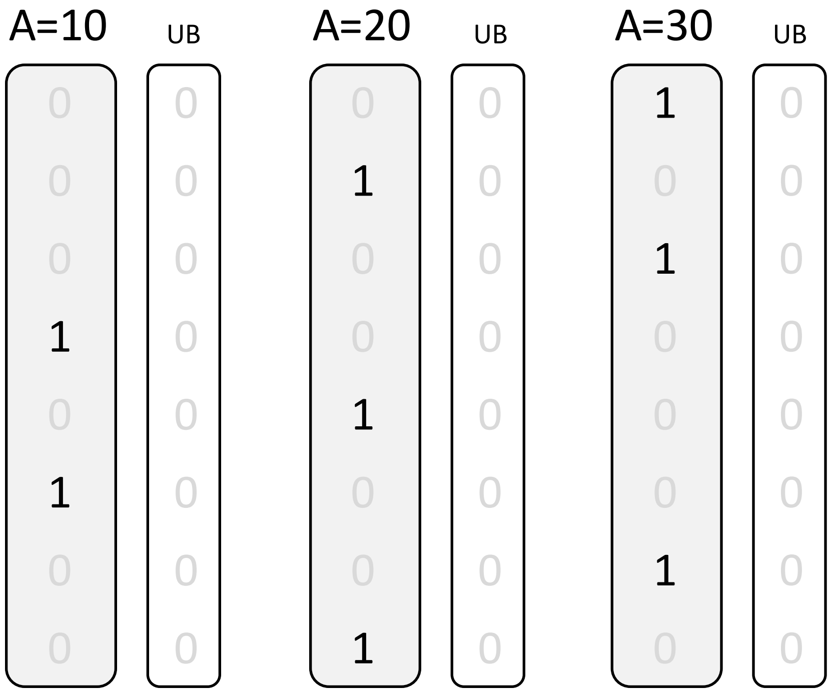
… row 2 to 10

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

Updating UpBit …

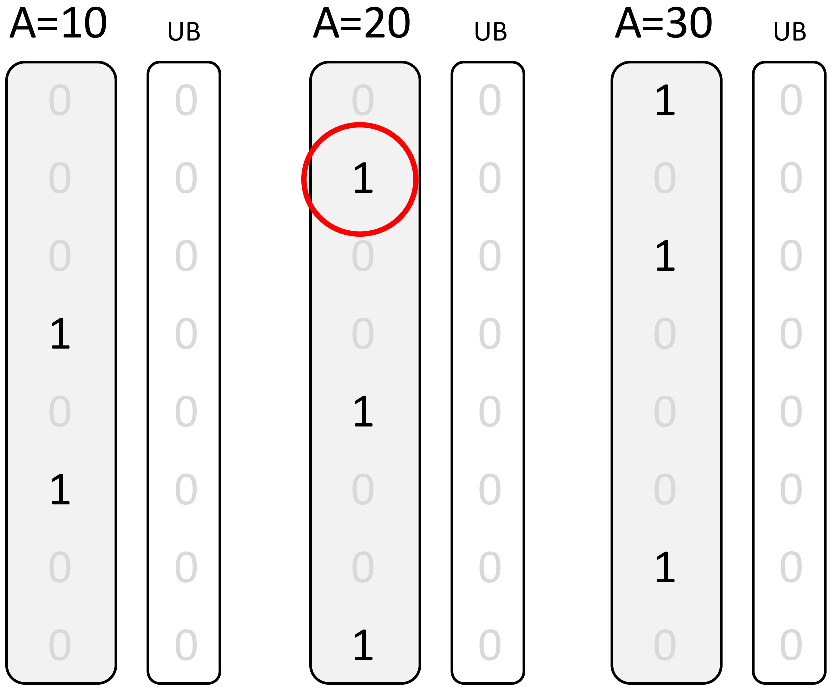
… row 2 to 10

1. find old value of row 2 (A=20)



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

Updating UpBit …

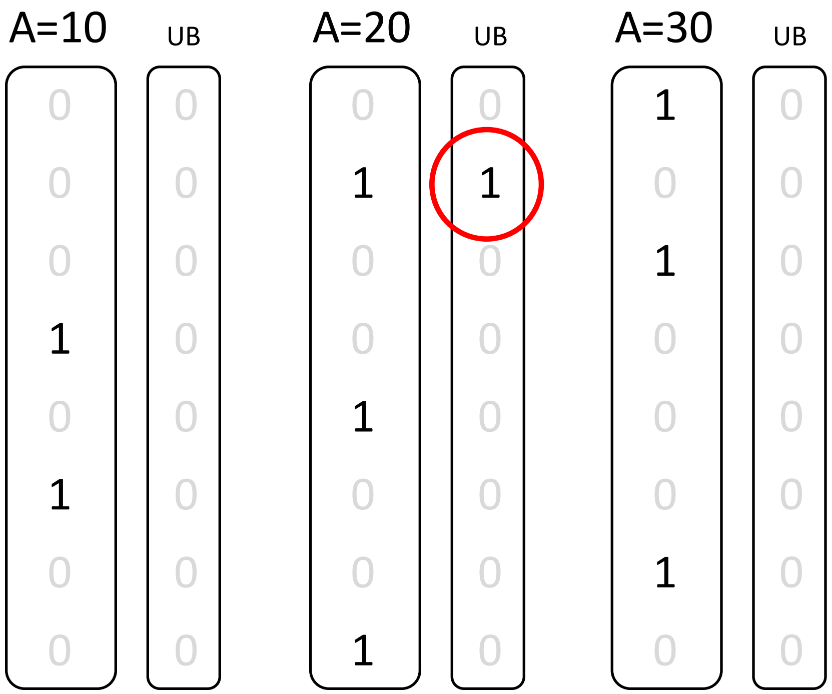


A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

… row 2 to 10

1. find old value of row 2 (A=20)

Updating UpBit ...



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

... row 2 to 10

1. find old value of row 2 (A=20)
2. flip bit of row 2 of UB of A=20

Updating UpBit ...

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

... row 2 to 10

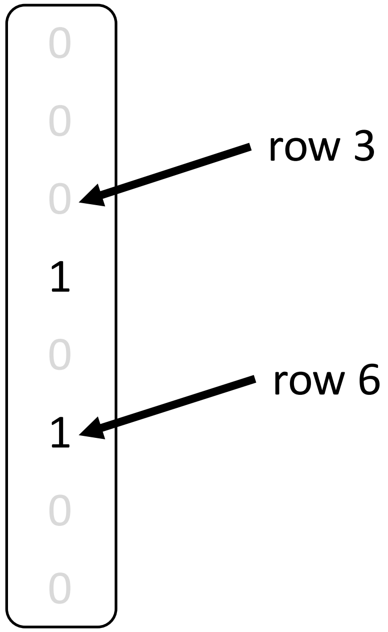
1. find old value of row 2 (A=20)
2. flip bit of row 2 of UB of A=20
3. flip bit of row 2 of UB of A=10

can we speed up step 1?

Design Element 2: fence pointers

efficient access of compressed bitvectors

fence pointers



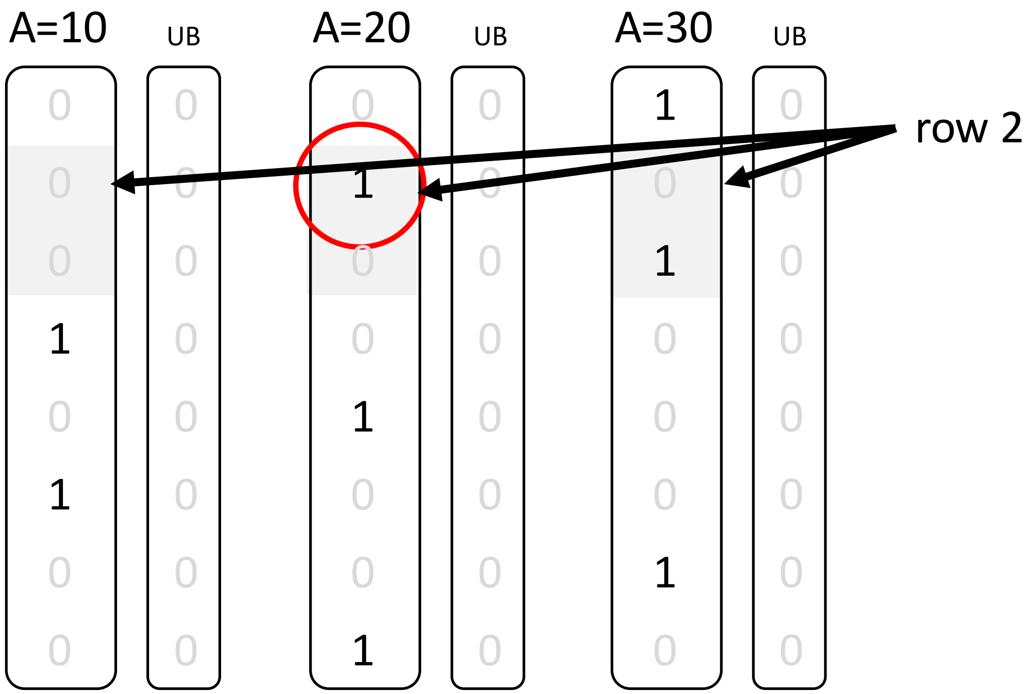
Updating UpBit …

… row 2 to 10

1. find old value of row 2 (A=20)

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

Updating UpBit (with fence pointers)…



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

… row 2 to 10

1. find old value of row 2 (A=20)
   using fence pointers

Querying

Querying UpBit …

… A = 20

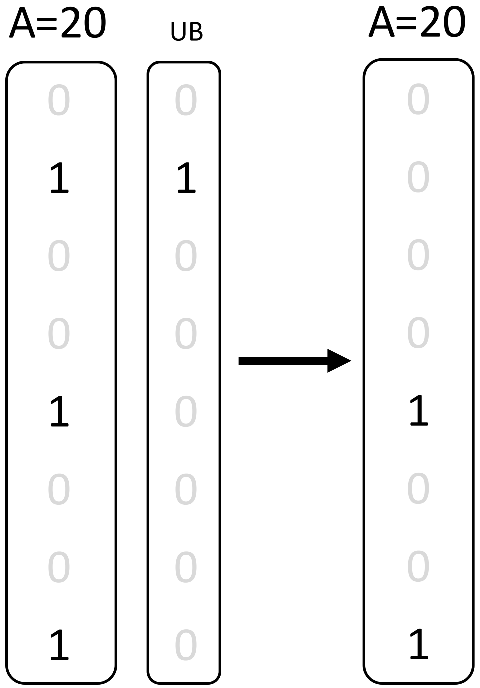
Return the XOR of A=20 and UB

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

Querying UpBit …

… A = 20

Return the XOR of A=20 and UB

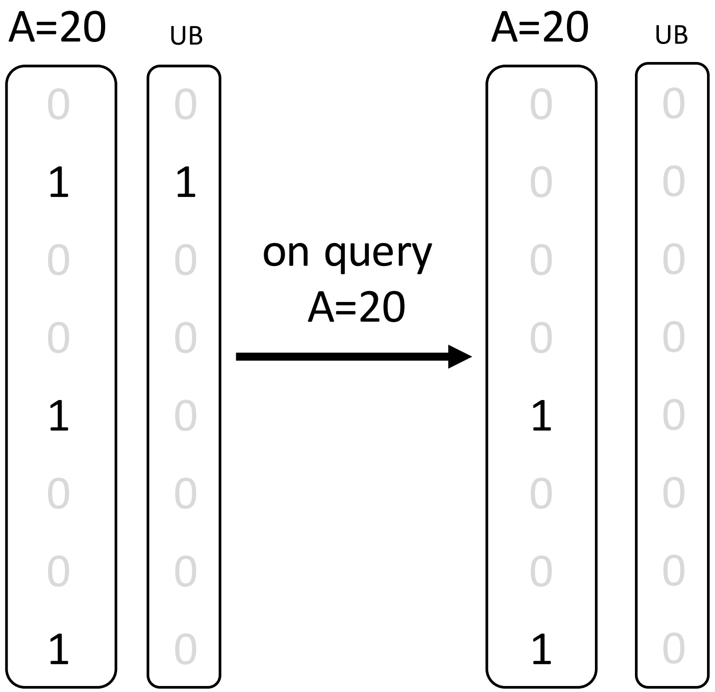


A=20	UB	→	A=20
0	0		0
1	1		0
0	0		0
0	0		0
1	0		1
0	0		0
0	0		0
1	0		1

can we re-use the result?

Design Element 3: query-driven merging

maintain high compressibility of UB
query-driven merging



A=20	UB
0	0
1	1
0	0
0	0
1	0
0	0
0	0
1	0

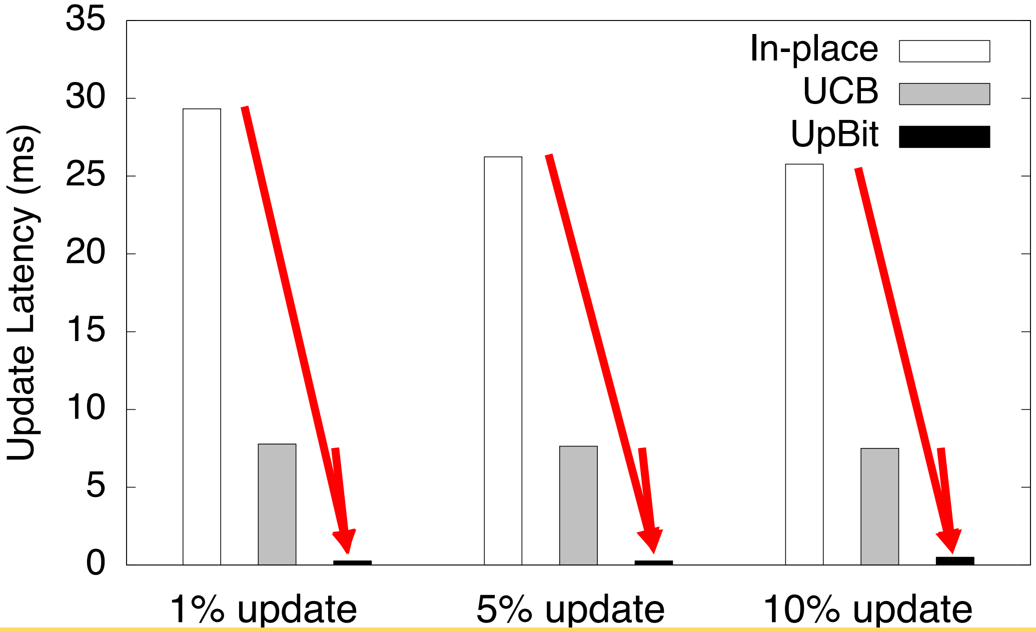
on query
A=20

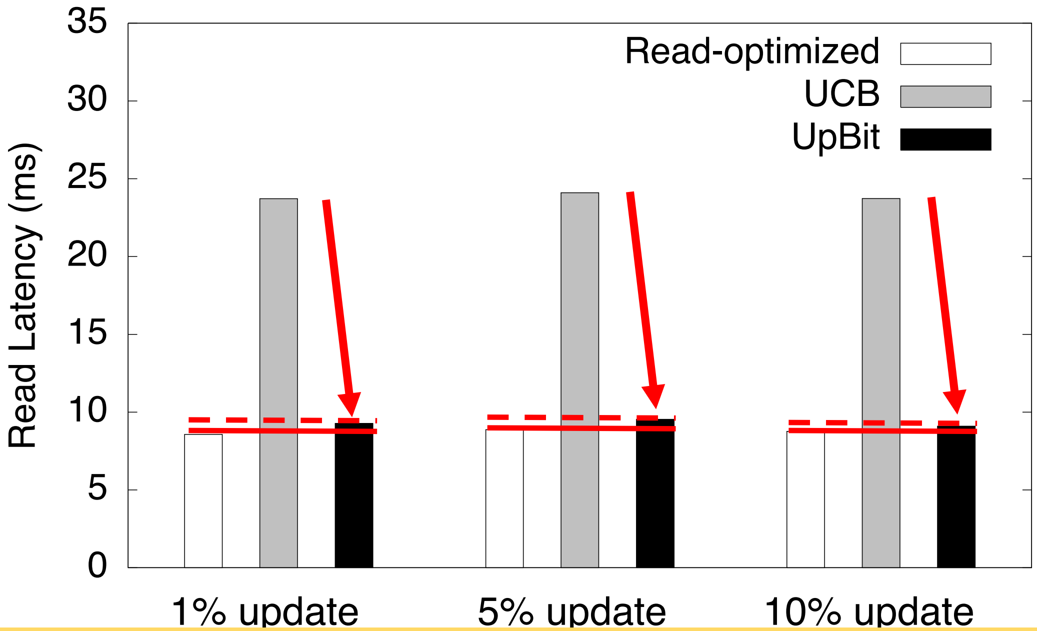
A=20	UB
0	0
0	0
0	0
0	0
1	0
0	0
0	0
1	0

UpBit supports very efficient updates

n=100M tuples, d=100 domain values

100k queries (varying % of updates)





updates: 15-29x faster than UCB

only 8% read overhead over optimal

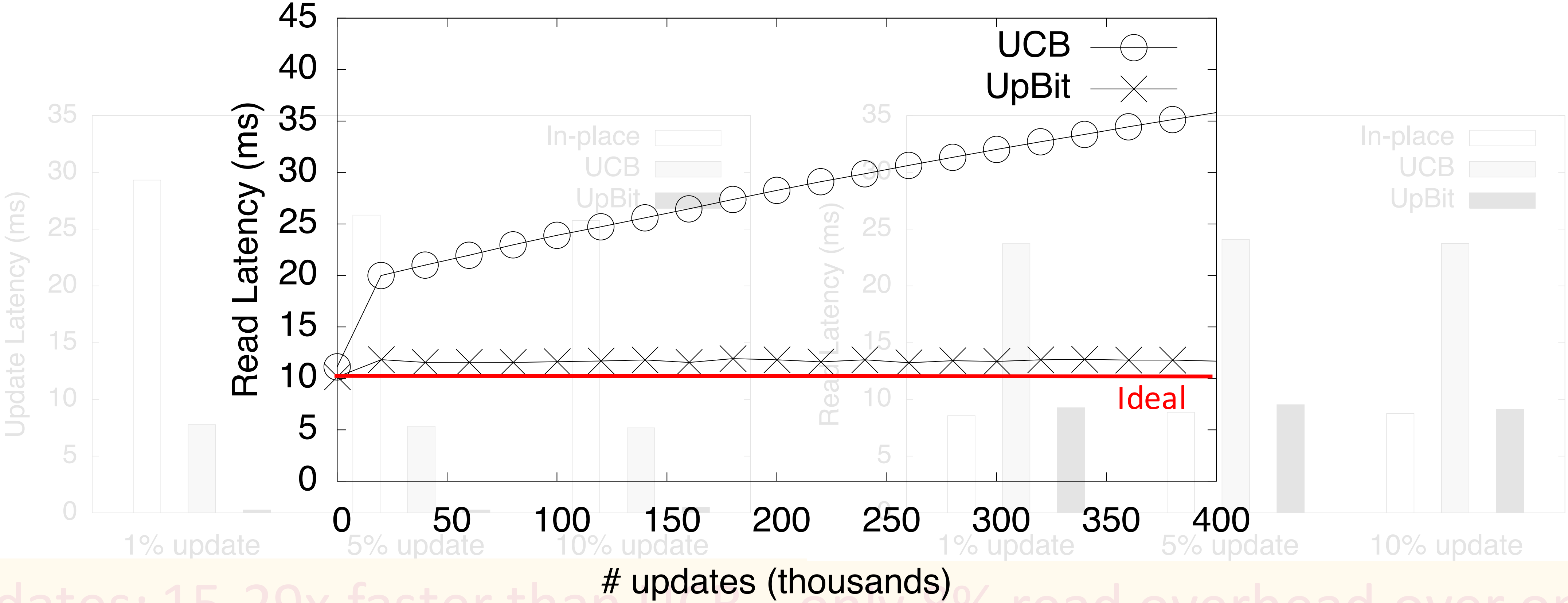
51-115x faster than in-place

3x faster reads than UCB

UpBit offers robust reads

n=100M tuples, d=100 domain values

50%/50% update/read queries



updates: 15-29x faster than UCB

only 8% read overhead over optimal

51-115x faster than in-place

3x faster reads than UCB

More in the paper …

Tuning: how frequent to merge UB to the index?

Tuning: what is the optimal granularity of fence pointers?

Optimizations: multi-threaded reads and updates

Performance: full query analysis (scientific data and TPCH)

UpBit: achieving scalable updates

distribute the update burden

update bitvectors

efficient bitvector accesses

fence pointers

avoid redundant bitwise operations

query-driven merging of UB

Thanks!

http://daslab.seas.harvard.edu/rum/