UpBit: Scalable In-Memory Updatable Bitmap Indexing

Manos Athanassoulis Zheng Yan[*] Stratos Idreos

HARVARD UNIVERSITY

[*]during an internship at DASlab

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

encoded bitvectors

Index Size

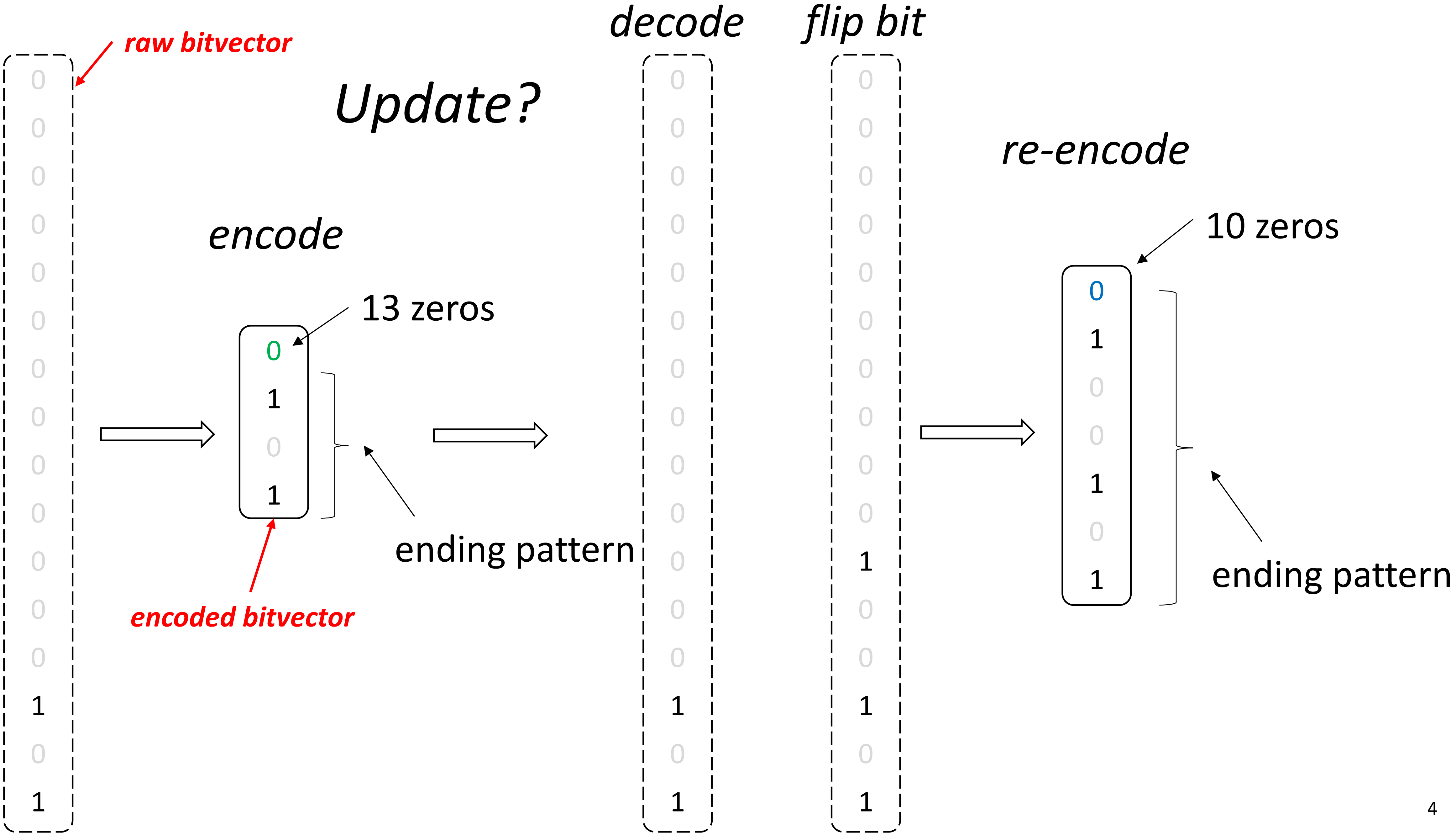
- Space-inefficient for domains with large cardinality
- Addressed by bitvector encoding/compression

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

but …

Updating encoded bitvectors is **very** inefficient

Update?


raw bitvector
encode
13 zeros
ending pattern
encoded bitvector
decode
flip bit
re-encode
10 zeros
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)

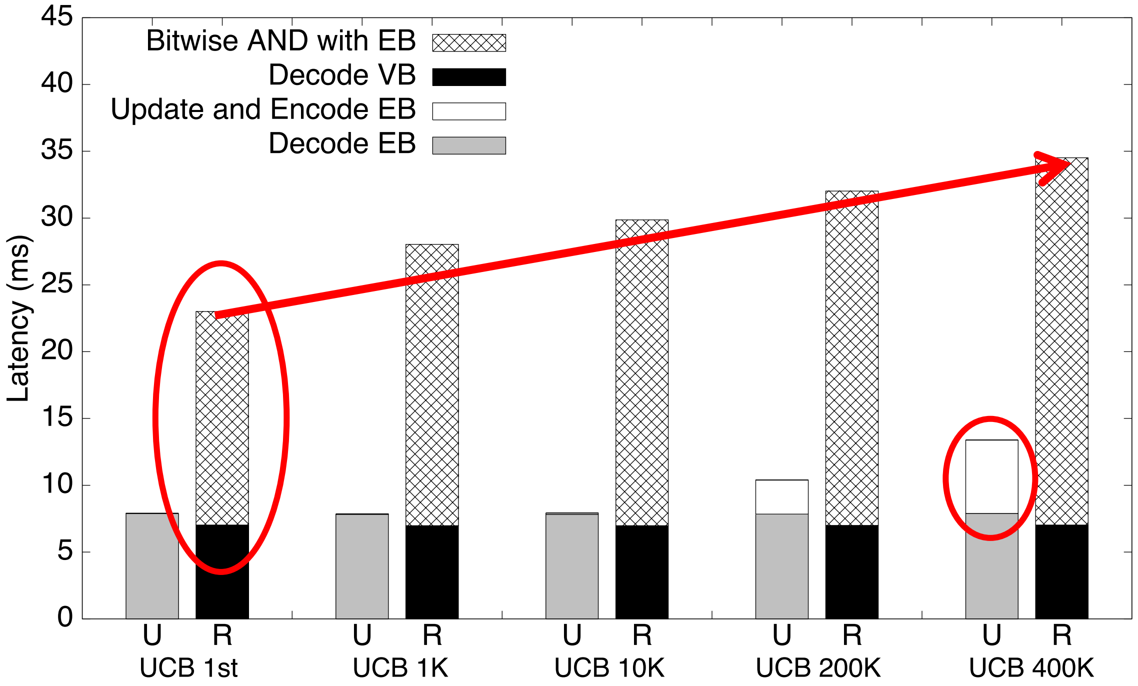
reads?
- bitwise AND with EB

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

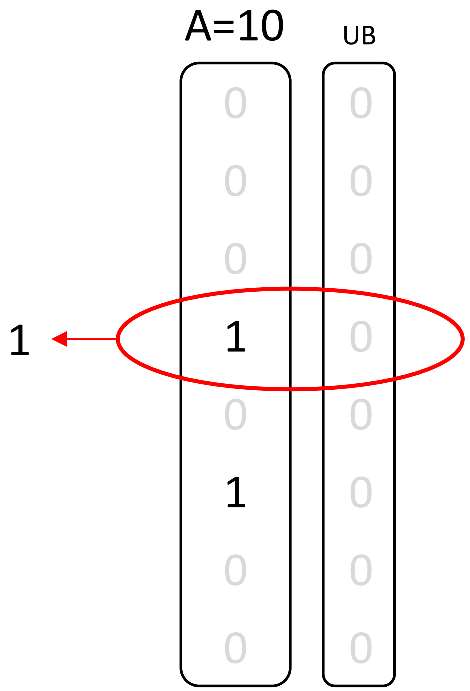
Bitmap Indexing for Reads & Updates

distribute update cost

efficient random accesses in compressed bitvectors

query-driven re-use results of bitwise operations

Design Element 1: update bitvectors

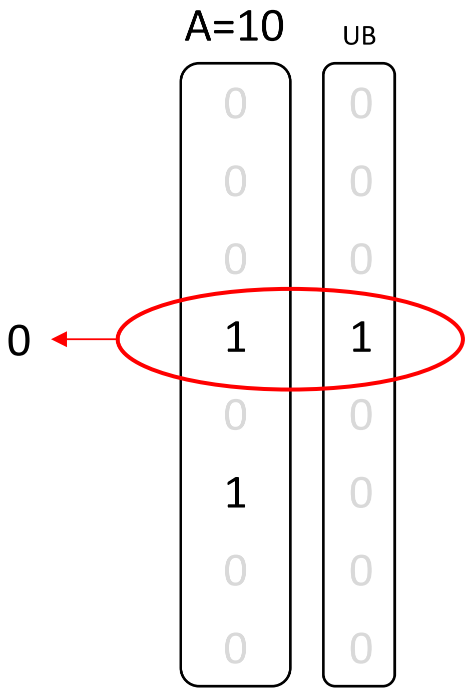


one per value of the domain
initialized to 0s

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

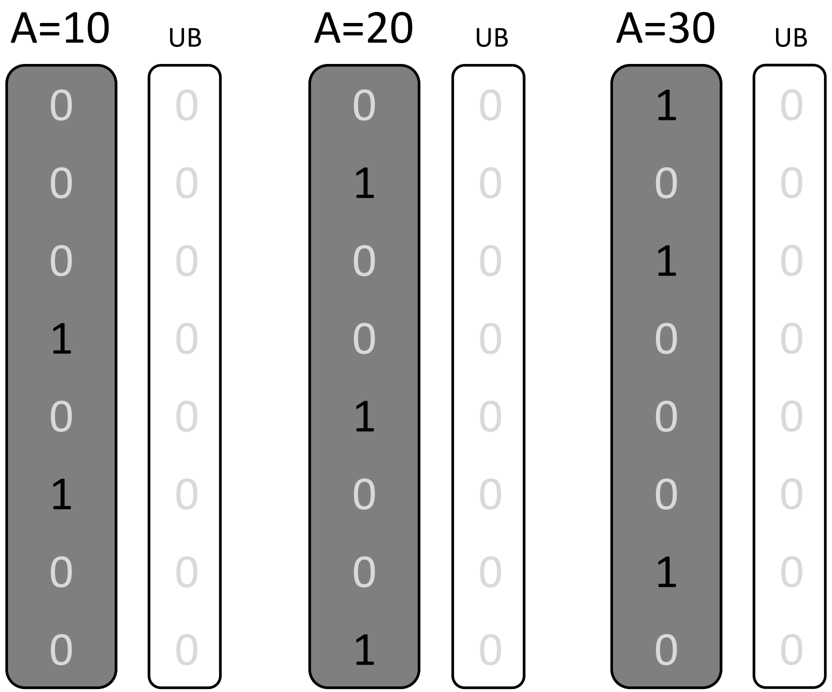
… distribute the update burden

Updating UpBit ...

... row 5 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 …



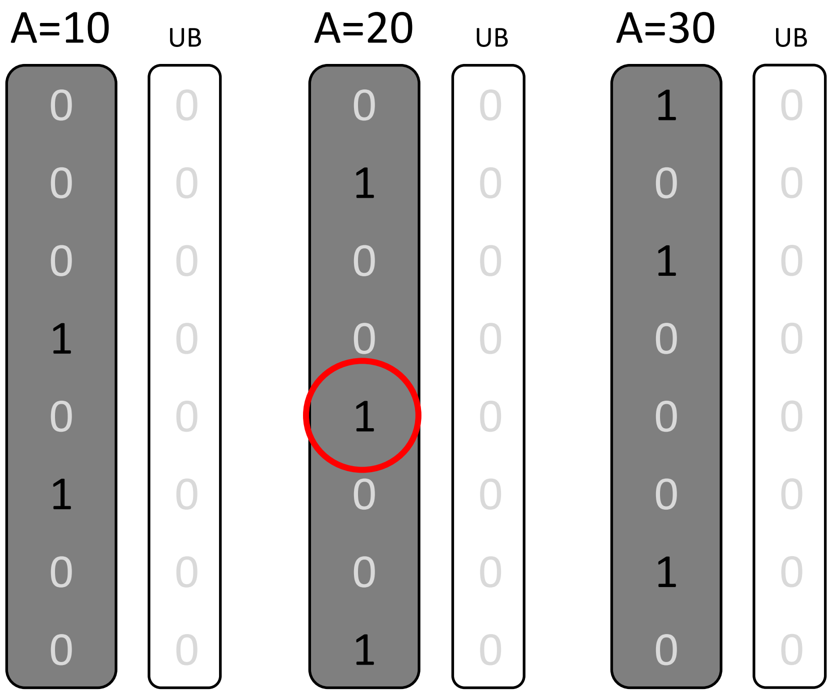
… row 5 to 10

1. find old value of row 5 (A=20)

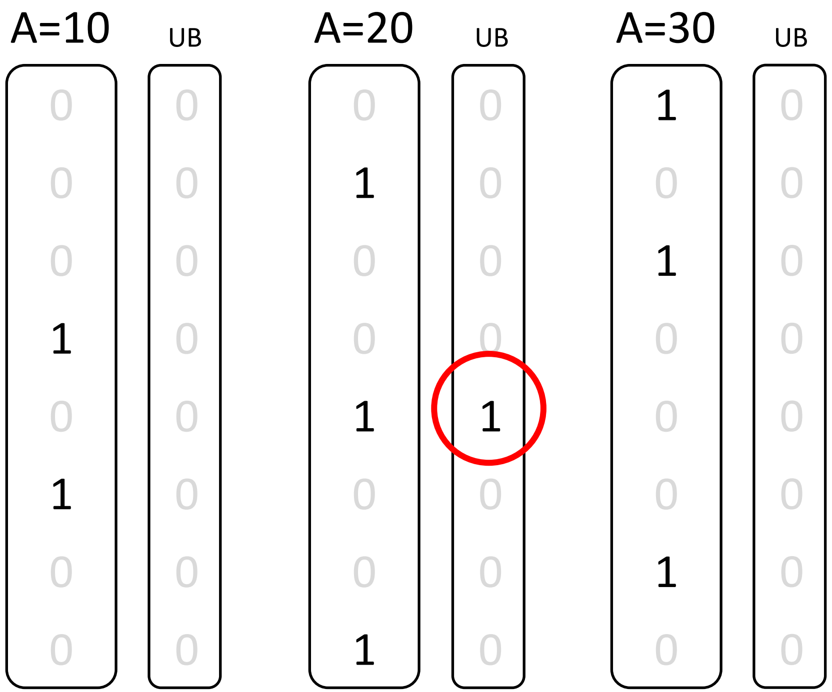
Updating UpBit …

… row 5 to 10

1. find old value of row 5 (A=20)



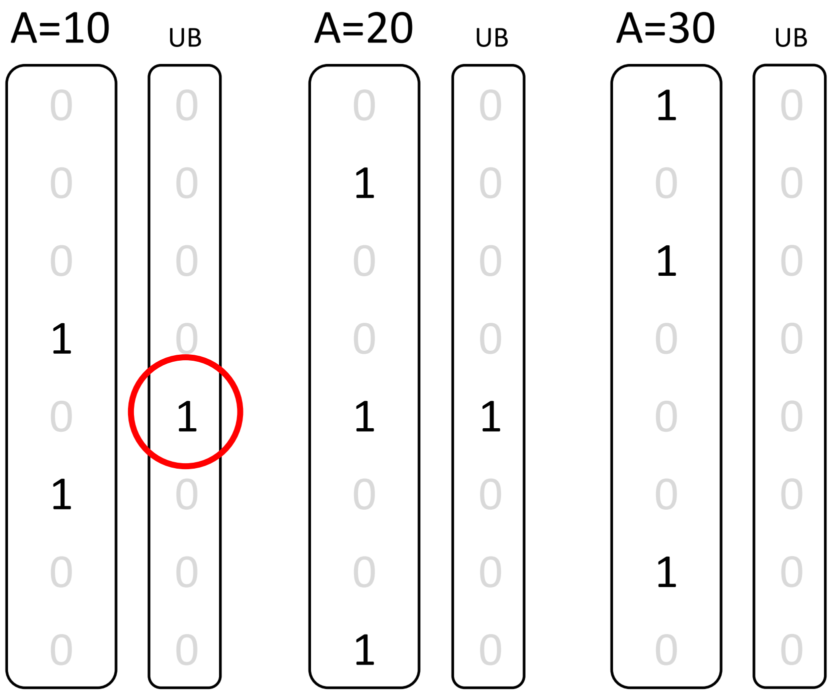
Updating UpBit ...



... row 5 to 10

1. find old value of row 5 (A=20)
2. flip bit of row 5 of UB of A=20

Updating UpBit …



… row 5 to 10

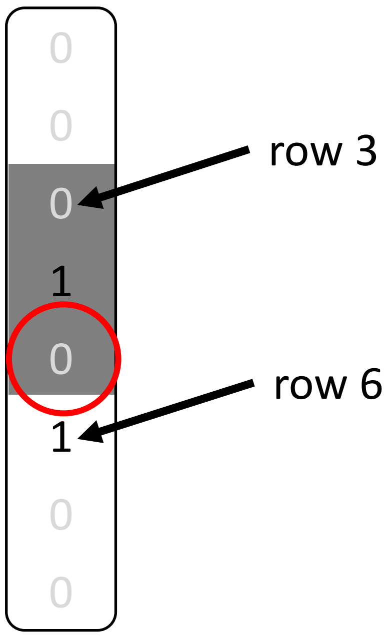
1. find old value of row 5 (A=20)
2. flip bit of row 5 of UB of A=20
3. flip bit of row 5 of UB of A=10

can we speed up step 1?

Design Element 2: fence pointers

efficient access of compressed bitvectors

fence pointers



we can find row 5 without decoding & scanning the whole bitvector

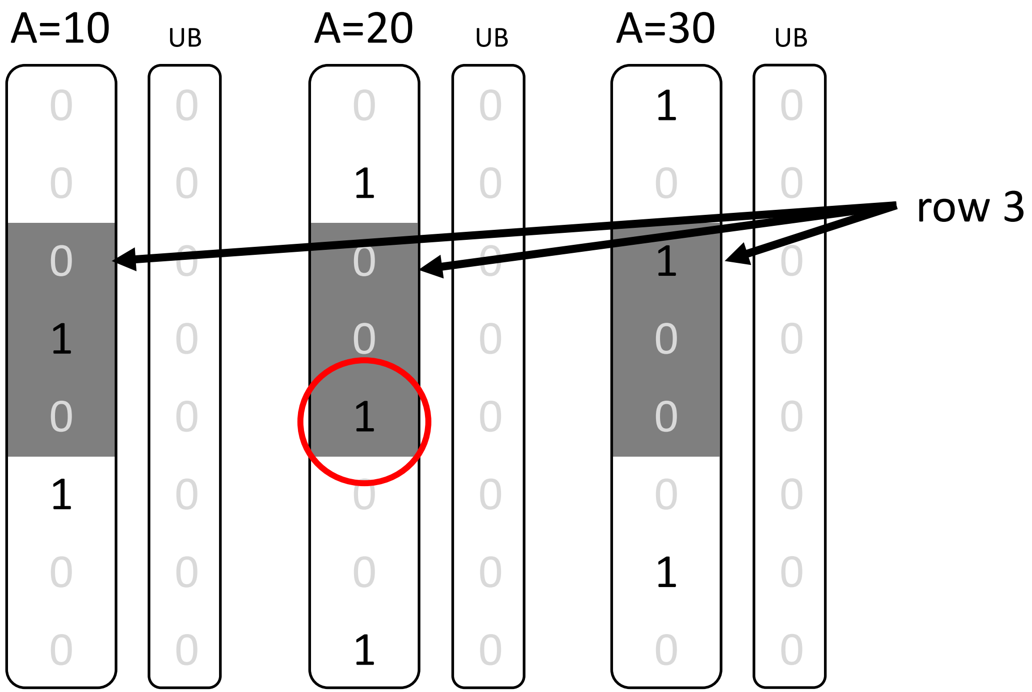
Updating UpBit …

… row 5 to 10

1. find old value of row 5 (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

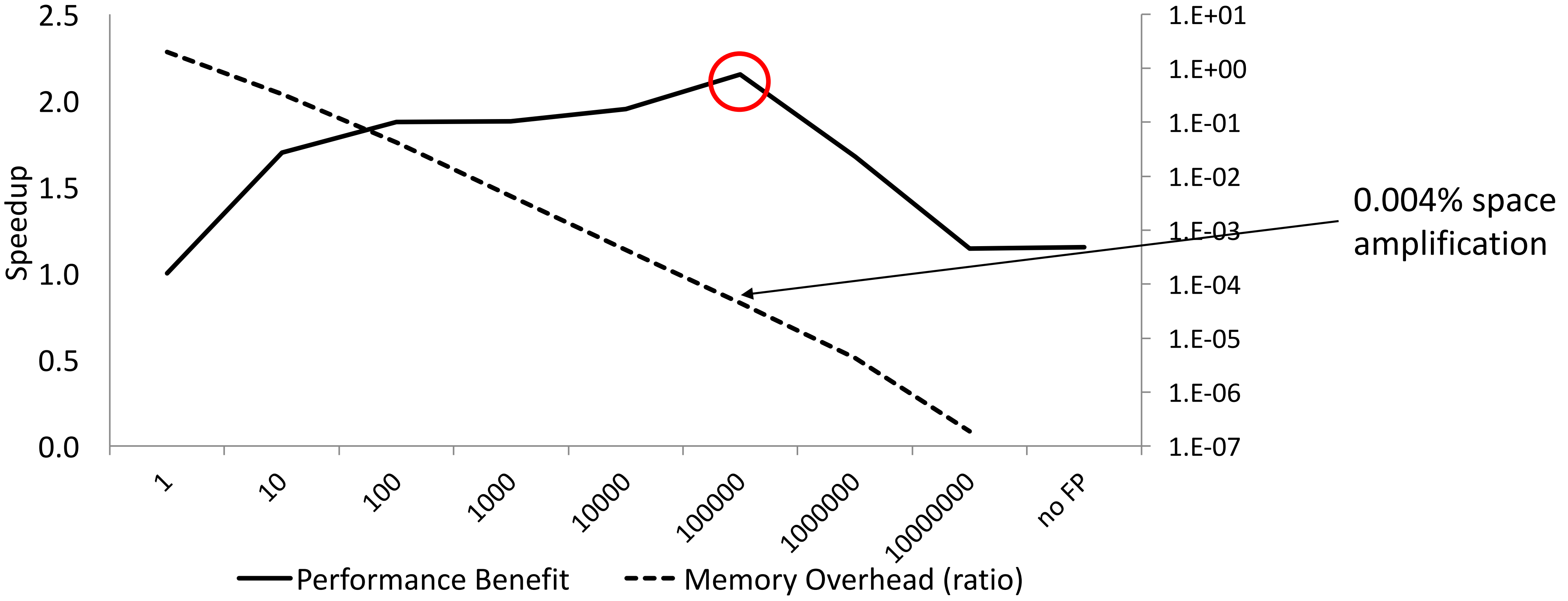
Updating UpBit (with fence pointers)…



… row 2 to 10

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

How dense should the *fence pointers* be?



fence pointers every 10^5 entries

Querying

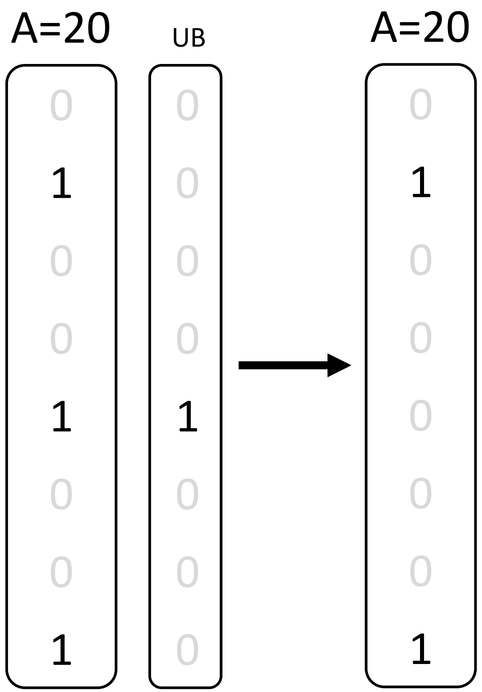
Querying UpBit …

… 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	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

Querying UpBit ...



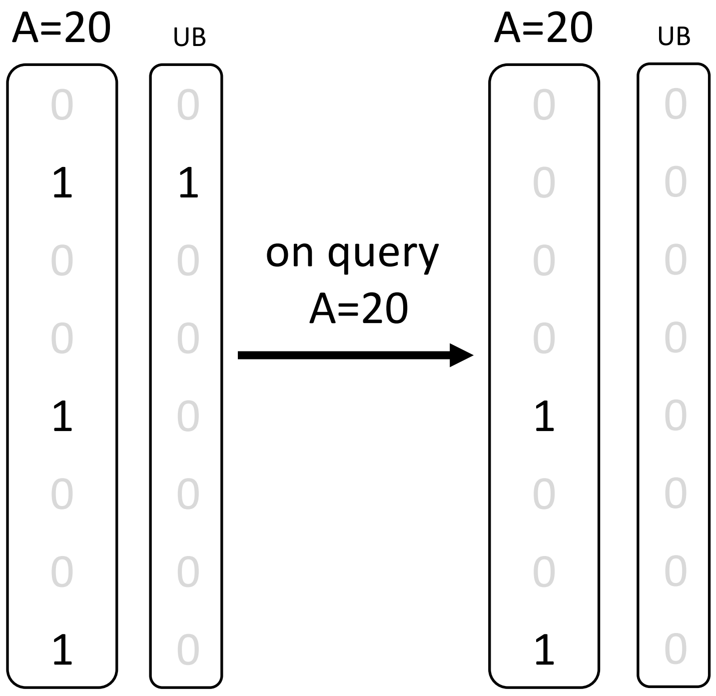
... A = 20

Return the XOR of A=20 and UB

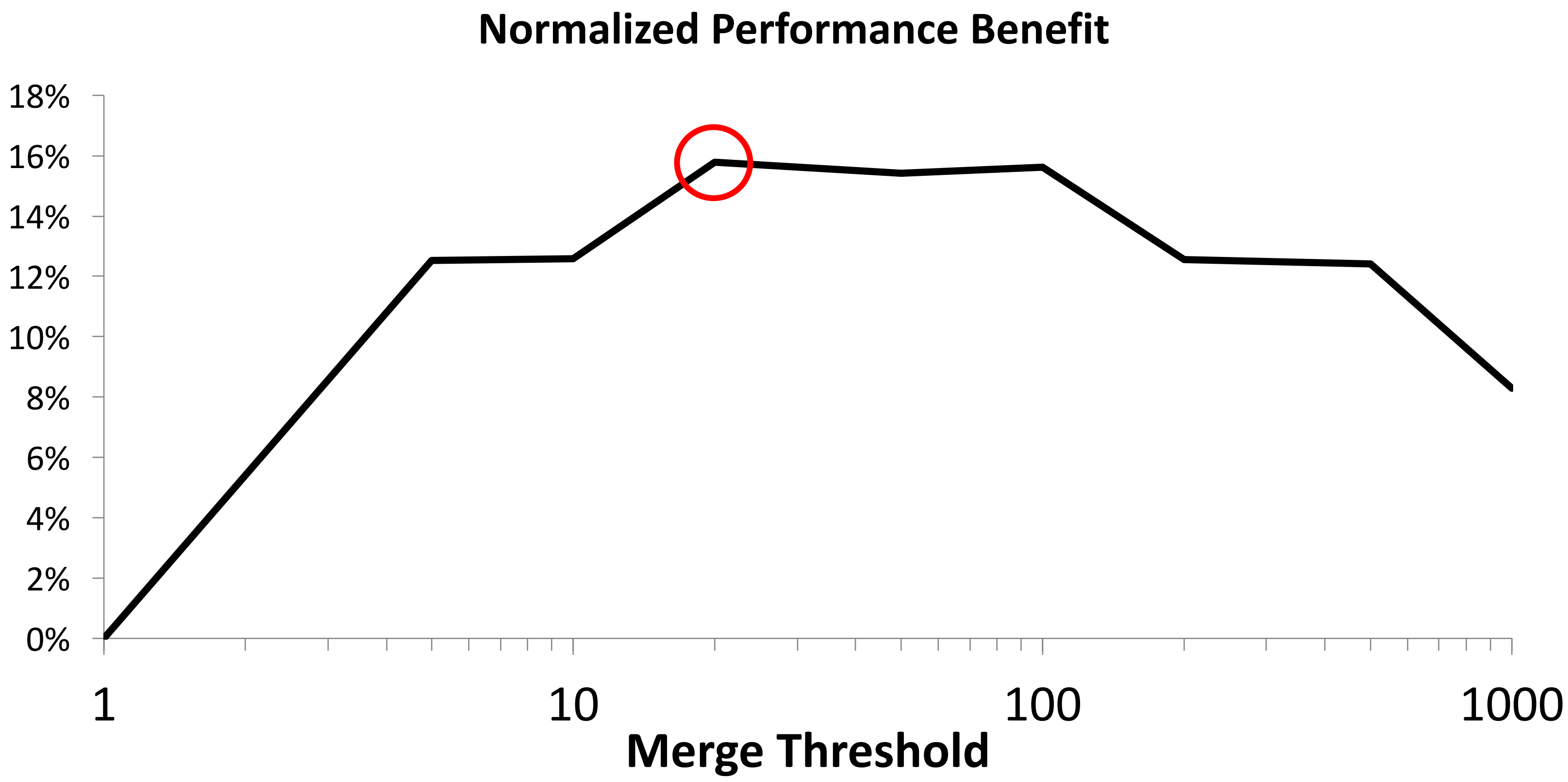
can we re-use the result?

Design Element 3: query-driven merging

maintain high compressibility of UB
query-driven merging



How frequently to merge UB back to VB?



merge back when 20 updates have been recorded

UpBit

update bitvectors

fence pointers

query-driven UB merging

Goals

Faster updates

Faster reads
(than update-optimized)

Fast overall latency

Experiments

Synthetic data

- ***n***: # tuples
- ***d***: # domain values (cardinality)
- ***q***: # queries
- ***u***: % updates in the workload

TPC-H SF 100 data

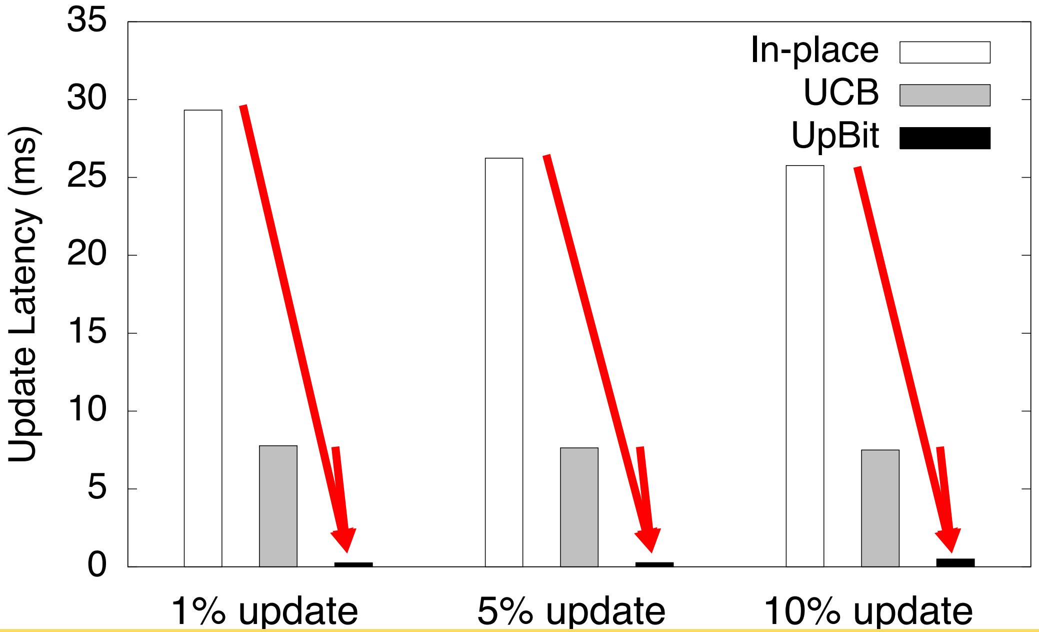
Prototype C++ Implementation of UpBit using FastBit

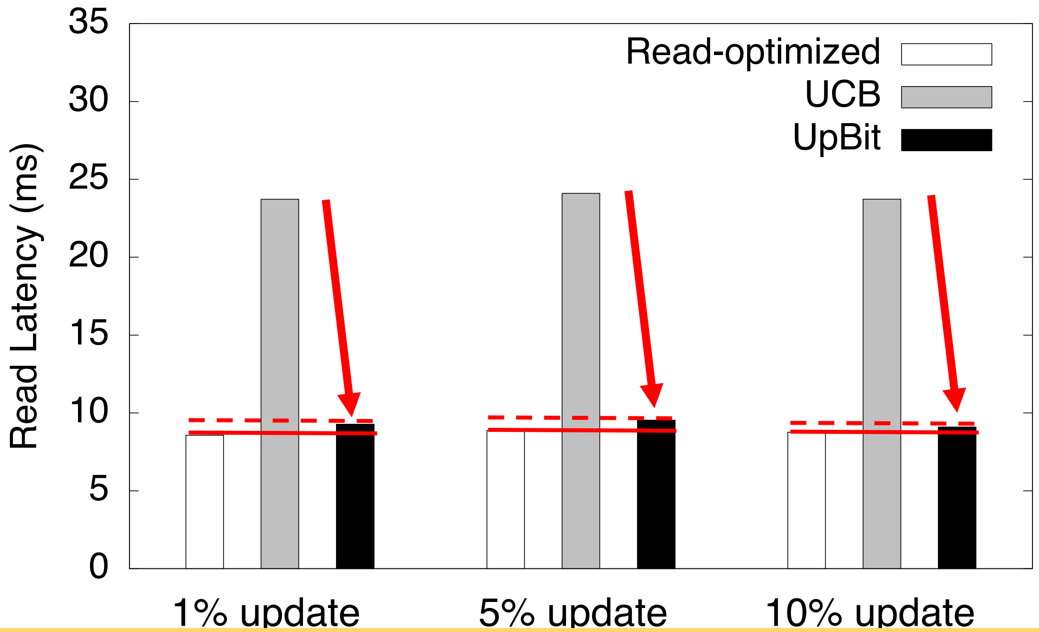
Integrated in a prototype column-store system

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

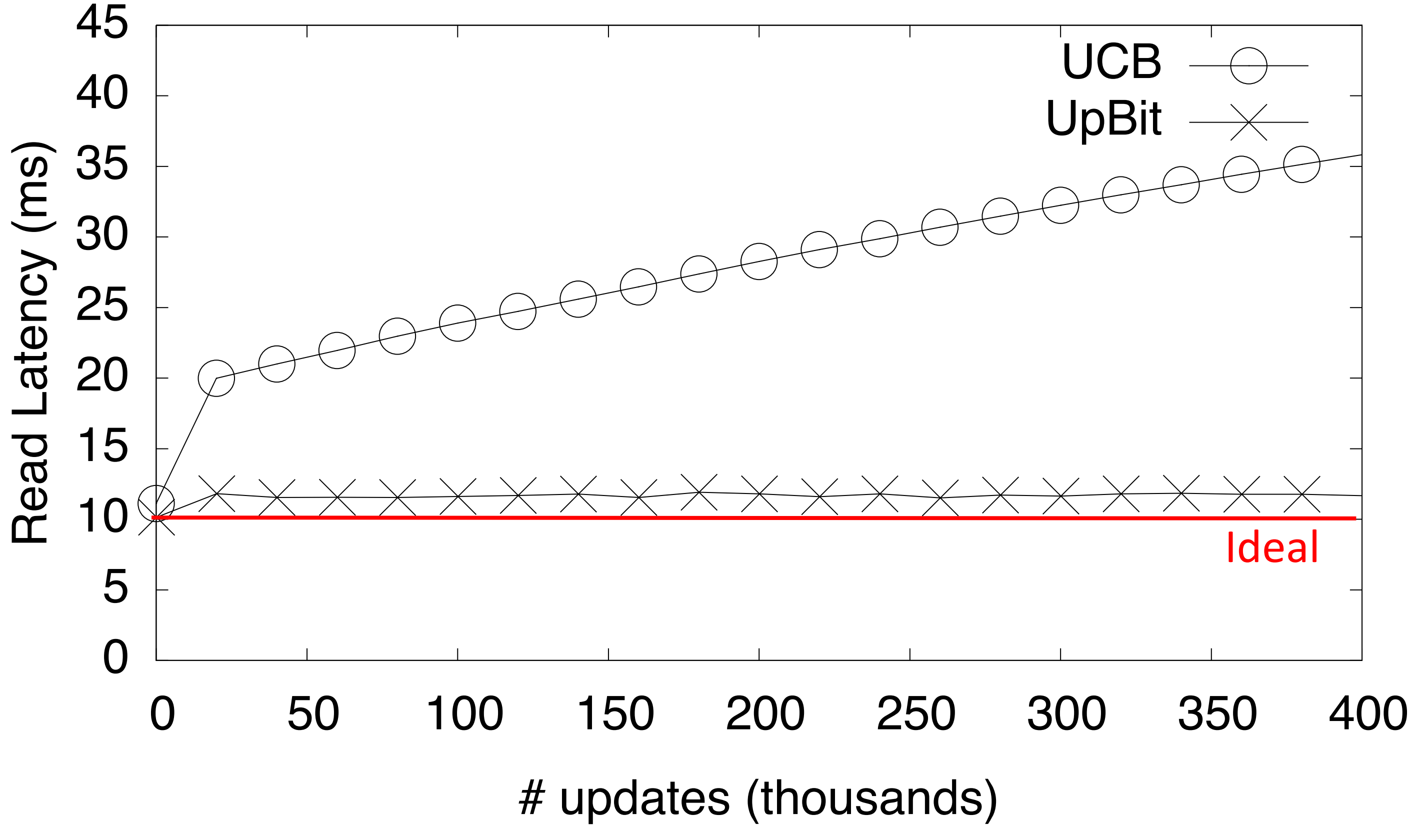
51-115x faster than in-place

3x faster reads than UCB

UpBit offers robust reads

n=100M tuples, d=100 domain values

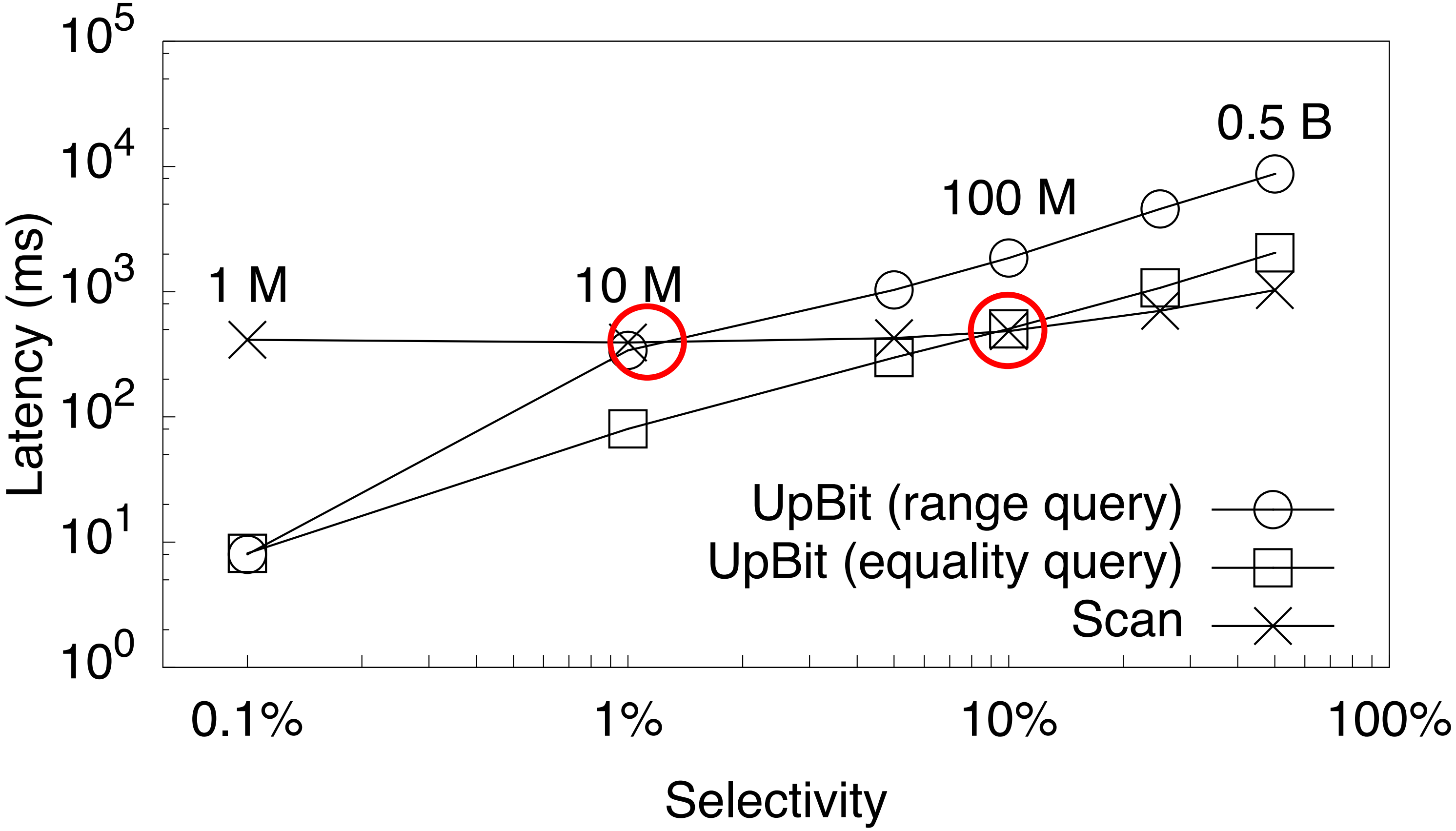
50%/50% update/read queries



scalable read latency

UpBit as a general index: UpBit vs. Scan

n = 1B, d = 1000 distinct domain values (range)
n = 1B, d varies for equality: 1000,100,10,1



Scan
tight for-loop
SIMD
multi-core

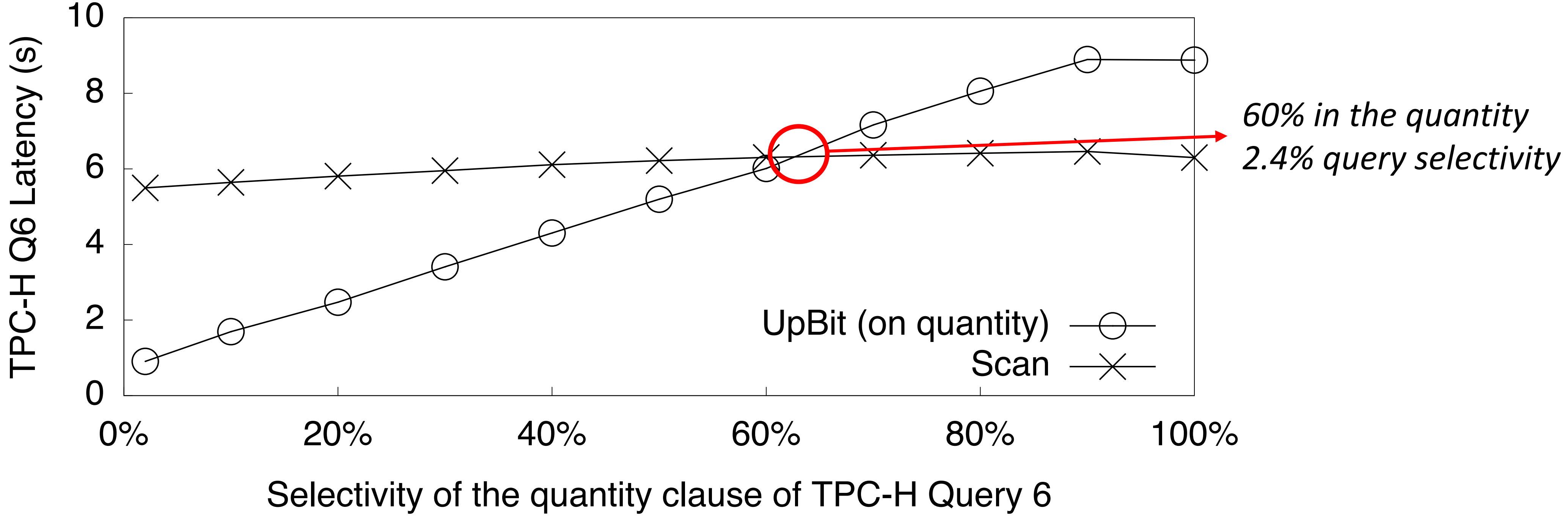
Equality query
all qualifying tuples have the same value (always 1 bitvector)

Range query
qualifying tuples may have many value (bit-OR bitvectors)

break-even: 10% for equality queries and 1% for range queries

UpBit as a general index: UpBit vs. Scan TPCH Q6

TPCH SF 100 (overall query selectivity 0.08% - 3.9%)



SELECT sum(l_extendedprice * l_discount) as revenue
FROM lineitem
WHERE l_shipdate >= date '[DATE]'
AND l_shipdate < date ' [DATE] ' + interval '1 ' year

an update-aware bitmap index is a viable *general-purpose index*

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!

DASlab

HARVARD UNIVERSITY

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