CS 561: **Data Systems Architectures**

class 14

General-Purpose Bitmap Indexes

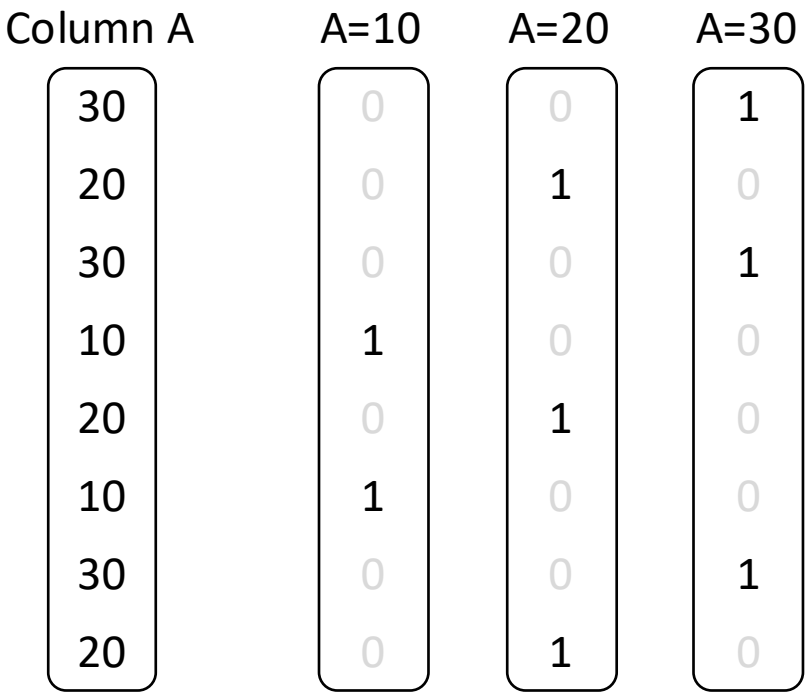
Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

Administrativia

- Mentor-meetings?
- Mid-semester project report on 3/22
- Guest Lecture by Prashant Pandey (Northeastern University) on 3/26
  "From Filters to Hash Tables: Rethinking Core Data Structures for Scalable Performance"

Bitmap Index Basics



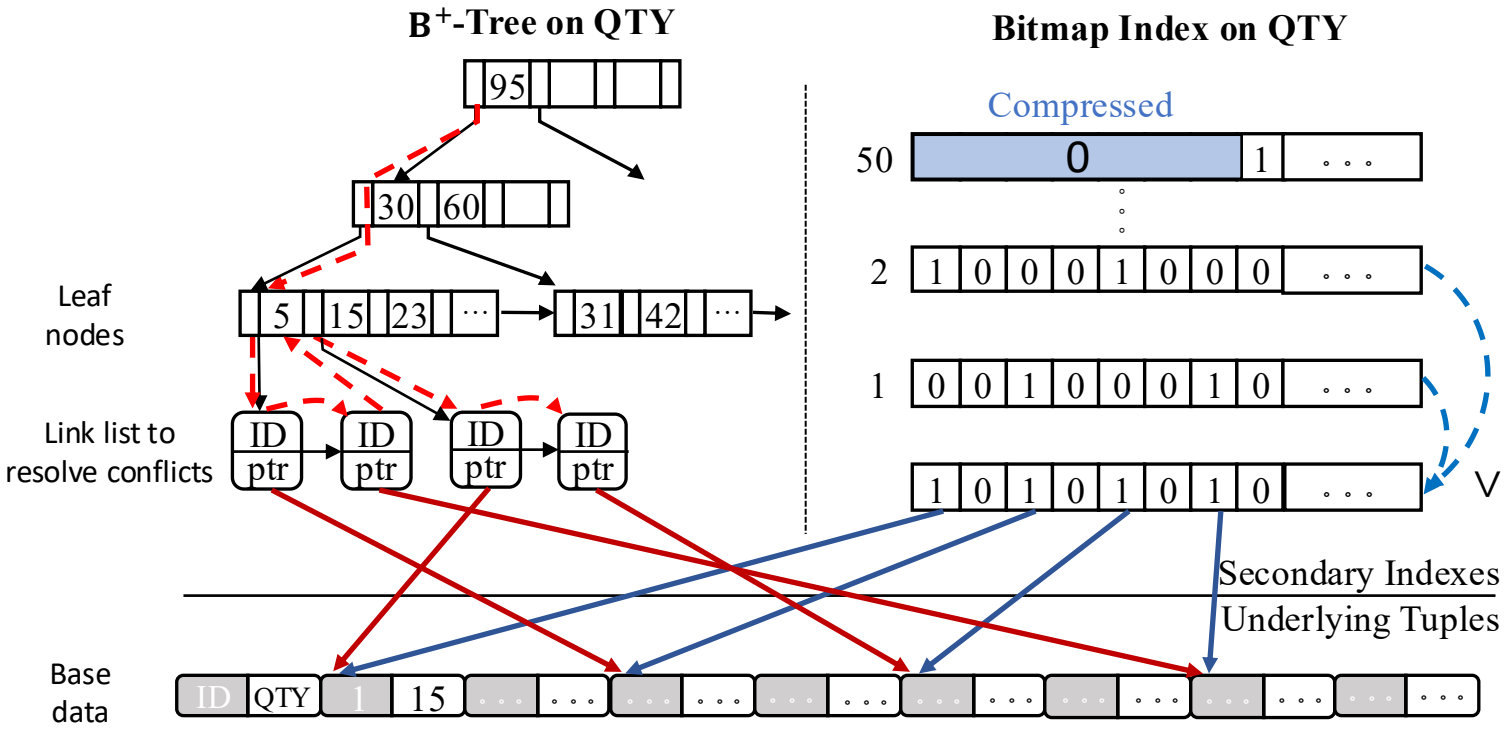
Specialized indexing

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

Bitvectors

- ☑ 0s are highly compressible
- ☑ Efficient bitwise operations (e.g., OR/AND)
- ☑ Sequential access minimizes cache and TLB misses

Bitmap Indexes as Secondary Indexes

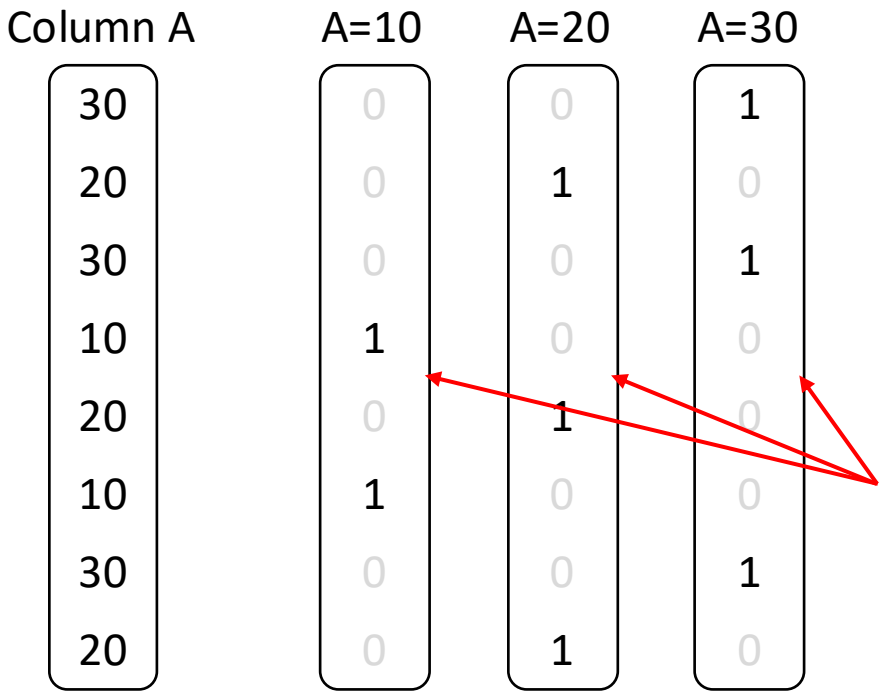


Queries of the form “QTY < 3” by using either a BTree or a bitmap index on a column with 50 distinct values

Tree-based Indexes	Bitmap-based Indexes	
✗	✓	Query result is clustered
✗	✓	Compact memory footprint
✗	✓	Composition-friendliness for multi-column indexes

Updatability

Bitmap Indexing Limitation (Updatability)



Index Size

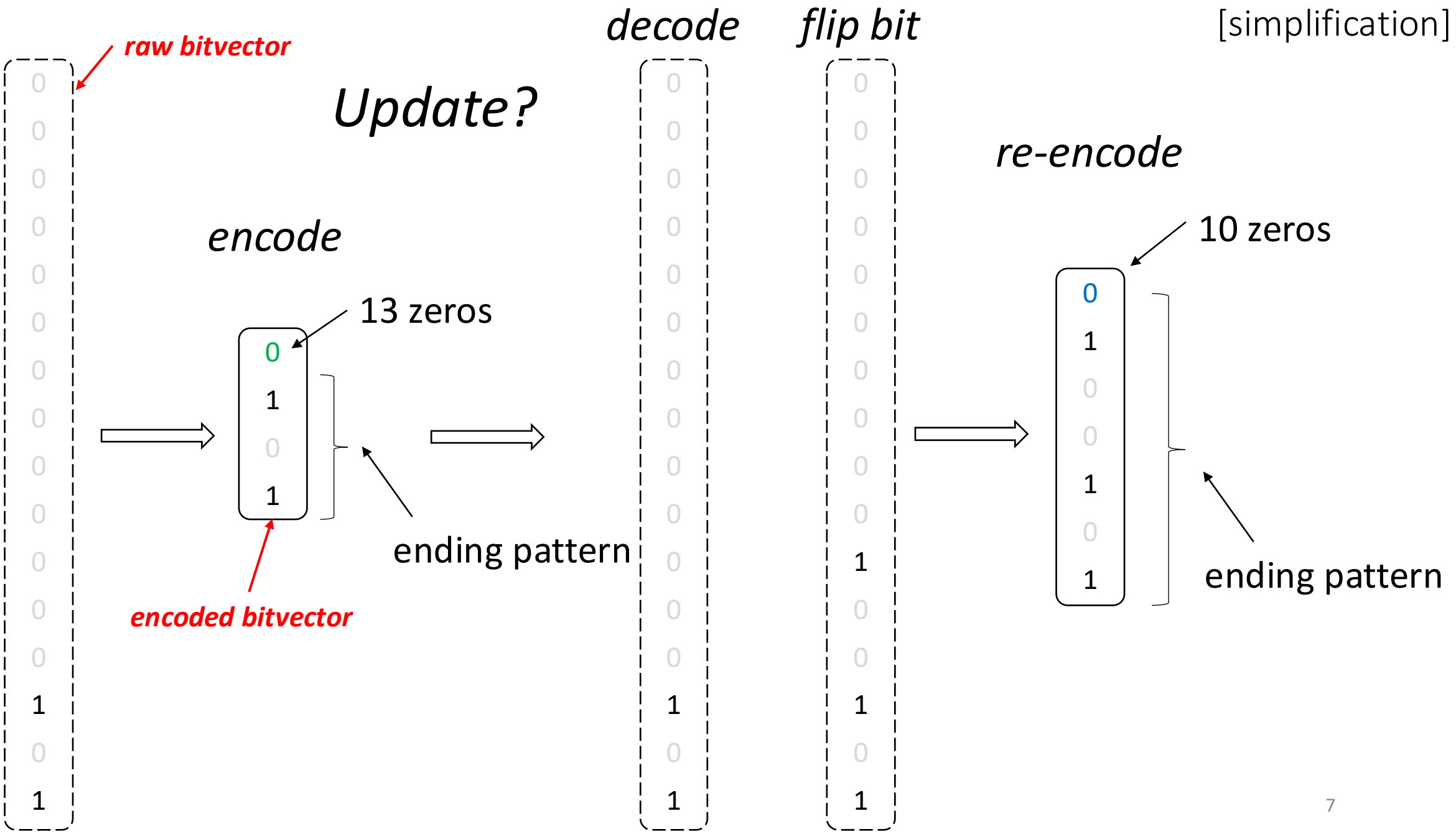
☑ Addressed by bitvector encoding/compression

core idea: *run-length encoding (RLE) to compactly store long runs of 1s or 0s*

but …

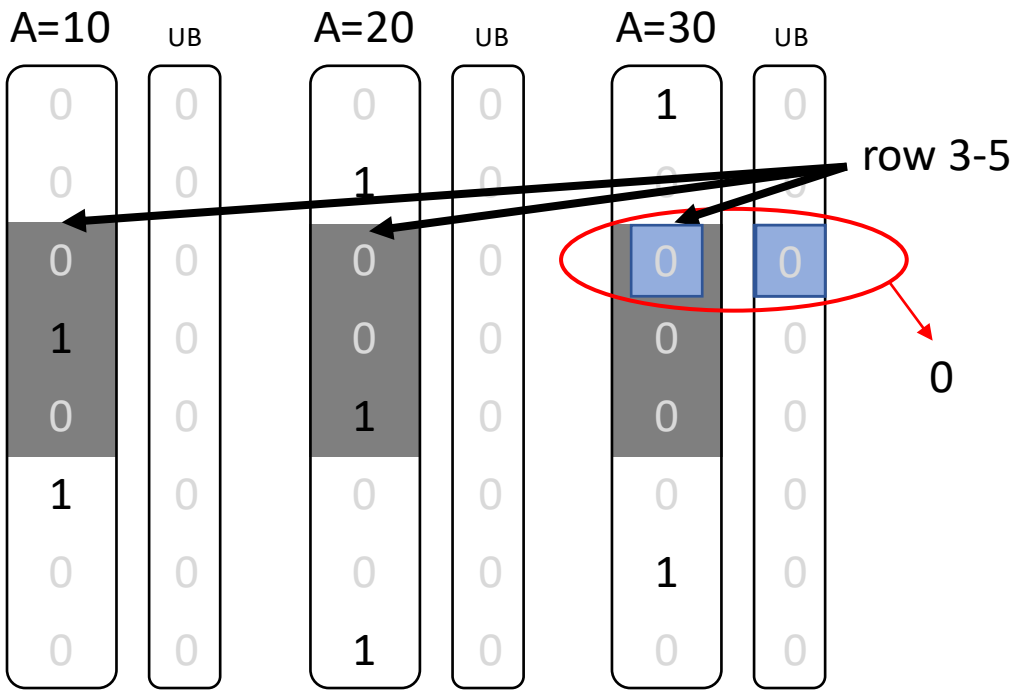
☒ Updating encoded bitvectors is **very** inefficient

[simplification]



Prior Work: UpBit

Updatable Bitmap Indexing (**UpBit**), SIGMOD'16



UB: one for each value bitvector and is initialized to 0s

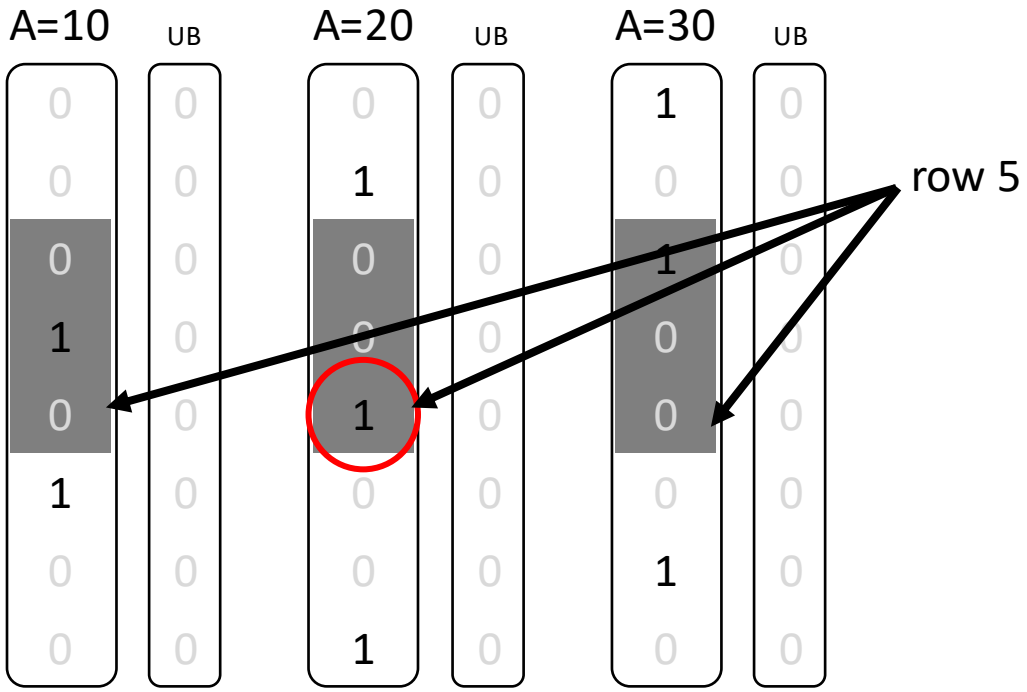
Updates/Deletes: only flip bits in UB

Query: XOR the corresponding <VB, UB> pair

Fence pointers: to quickly fetch bits in compressed bitvectors

Merge: UB is lazily (query-driven) merged with VB

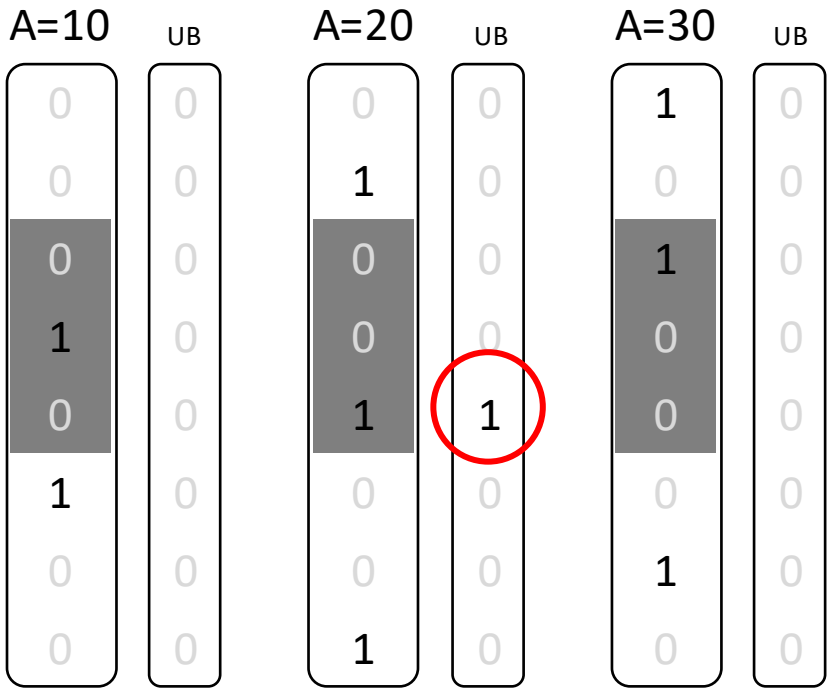
Updating UpBit …



… row 5 to 30

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

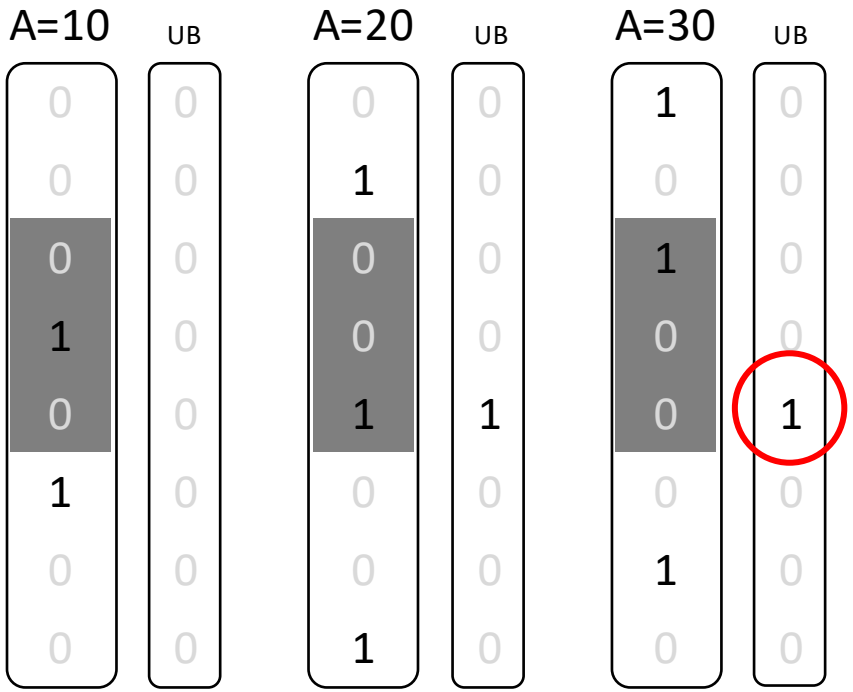
Updating UpBit …



… row 5 to 30

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

Updating UpBit …



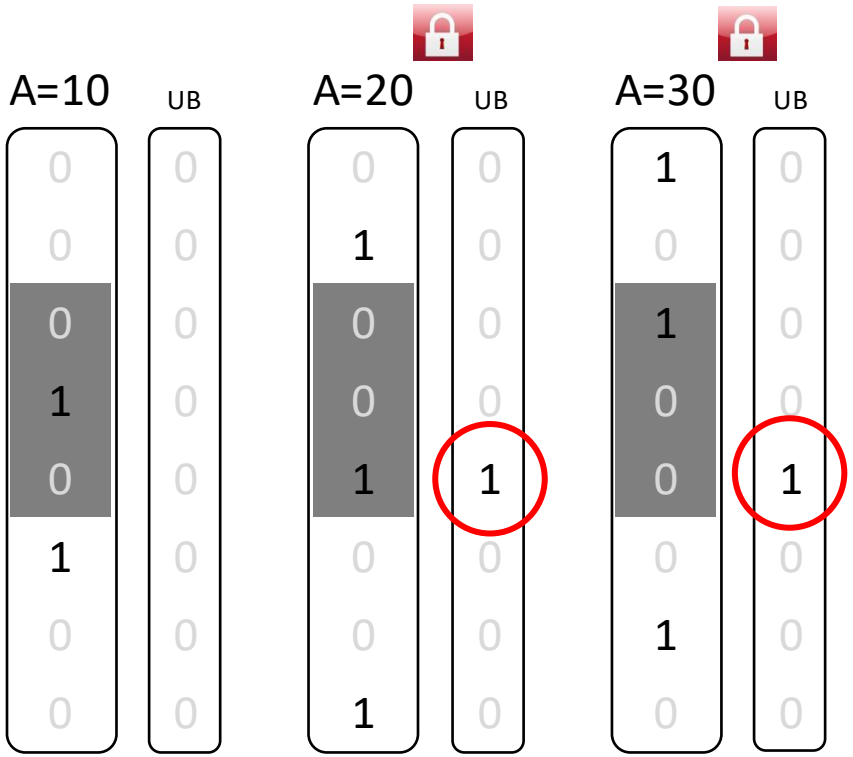
… row 5 to 30

1. find old value of row 5 (A=20)
   using fence pointers
2. flip bit of row 5 of UB of A=20
3. flip bit of row 5 of UB of A=30

Redirect updates to UBs

Fence pointers for efficient bit access

Scalability Issue

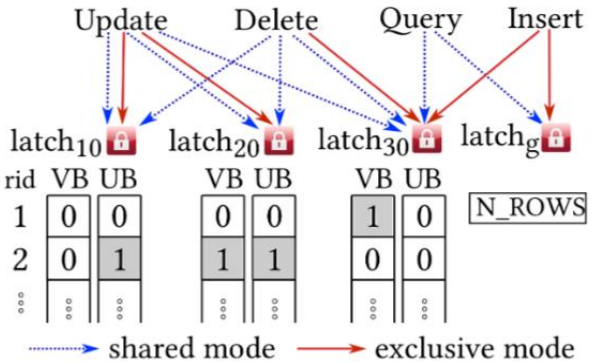


… row 5 to 30

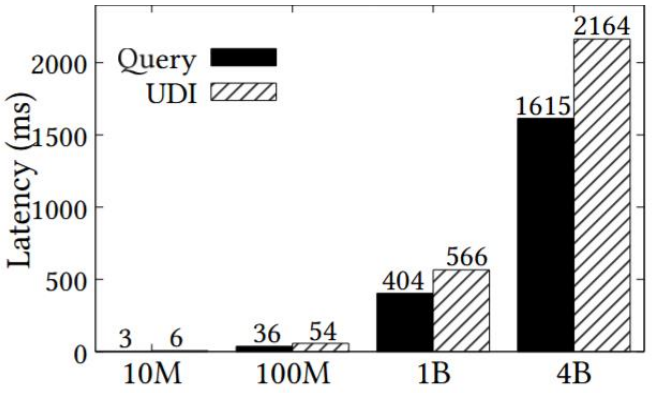
Acquire latches for 20 and 30 to update UBs

Updates are expensive (decode-flip-encode)

Queries are blocked, leading to long tail latency



Parallelized UpBit.



Mean query and UDI latency.

Does not scale on modern multicore systems

Goal

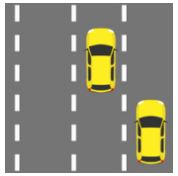
Concurrent Updatable Bitmap Indexing to overcome the key limitation of bitmap-based secondary indexes

- **Queries are wait-free** with guaranteed completion
- **Updates are lock-free** and avoid blocking each other

Concurrent Updatable BITmap Indexing (CUBIT)



Out-of-place update mitigates synchronization complexity

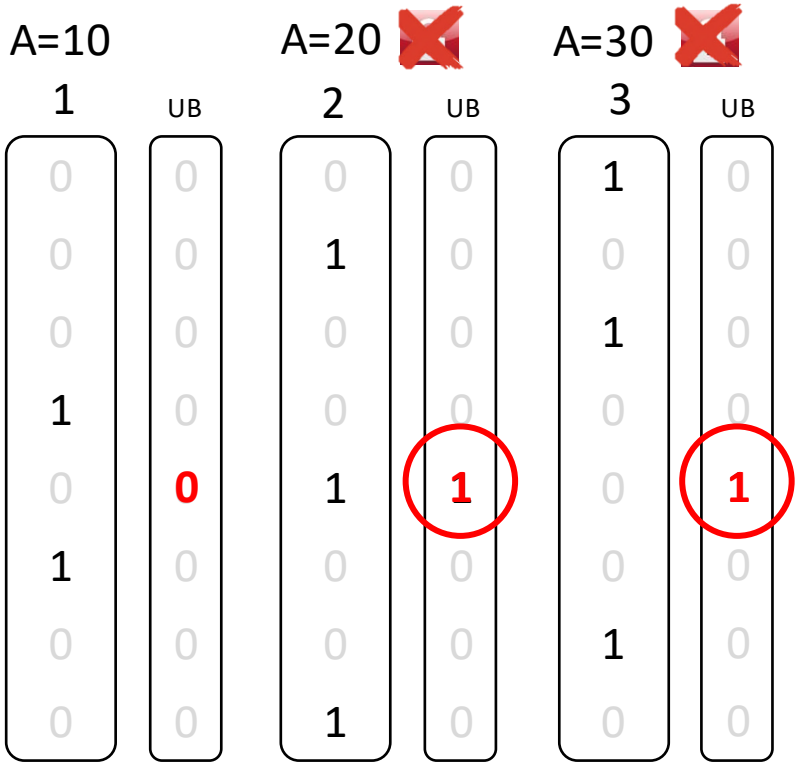


Multi-versioning enables parallel execution of queries and updates

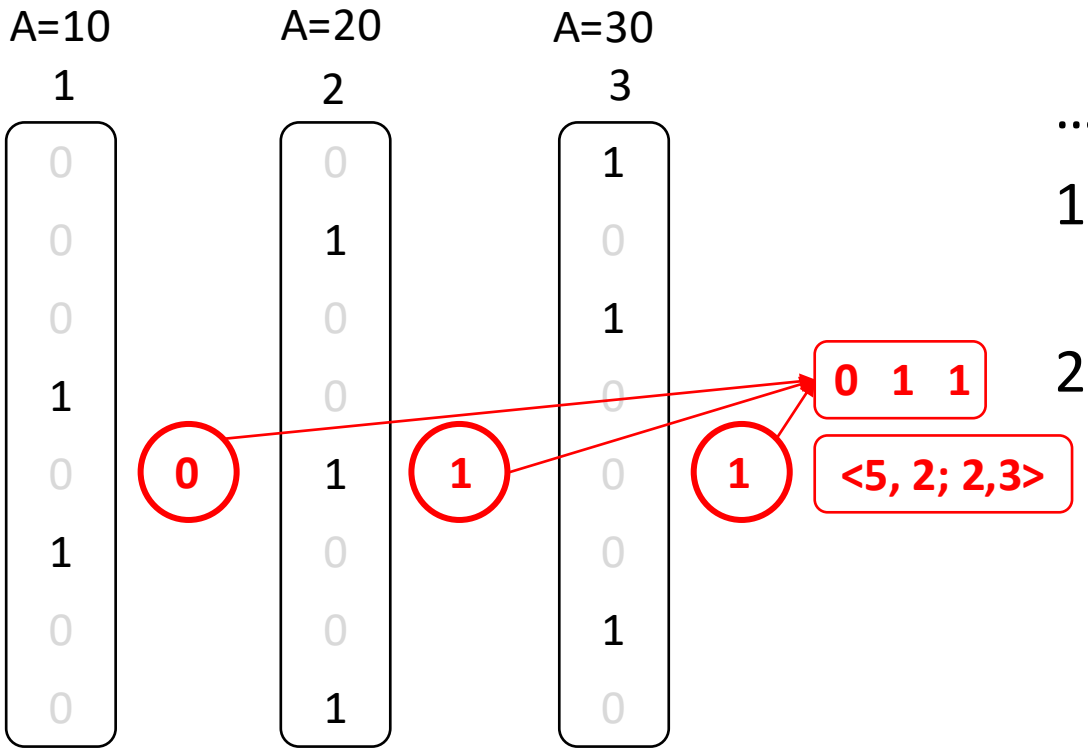


Bitvector Segmentation exploits parallelism on multicore systems

Design Element 1: Out-of-place Updates


A=10
A=20
A=30
1
UB
2
UB
3
UB
0
0
0
1
0
1
0
0
0
0
0
0
1
0
0
0
0
0
1
0
0
0
0
0
1
0
1
0
0
0
1
0
1
0
0
0
1
0
0
0
0
0
1
0
0
0
0
0
1
0
0

Design Element 1: Out-of-place Updates



… row 5 to 30

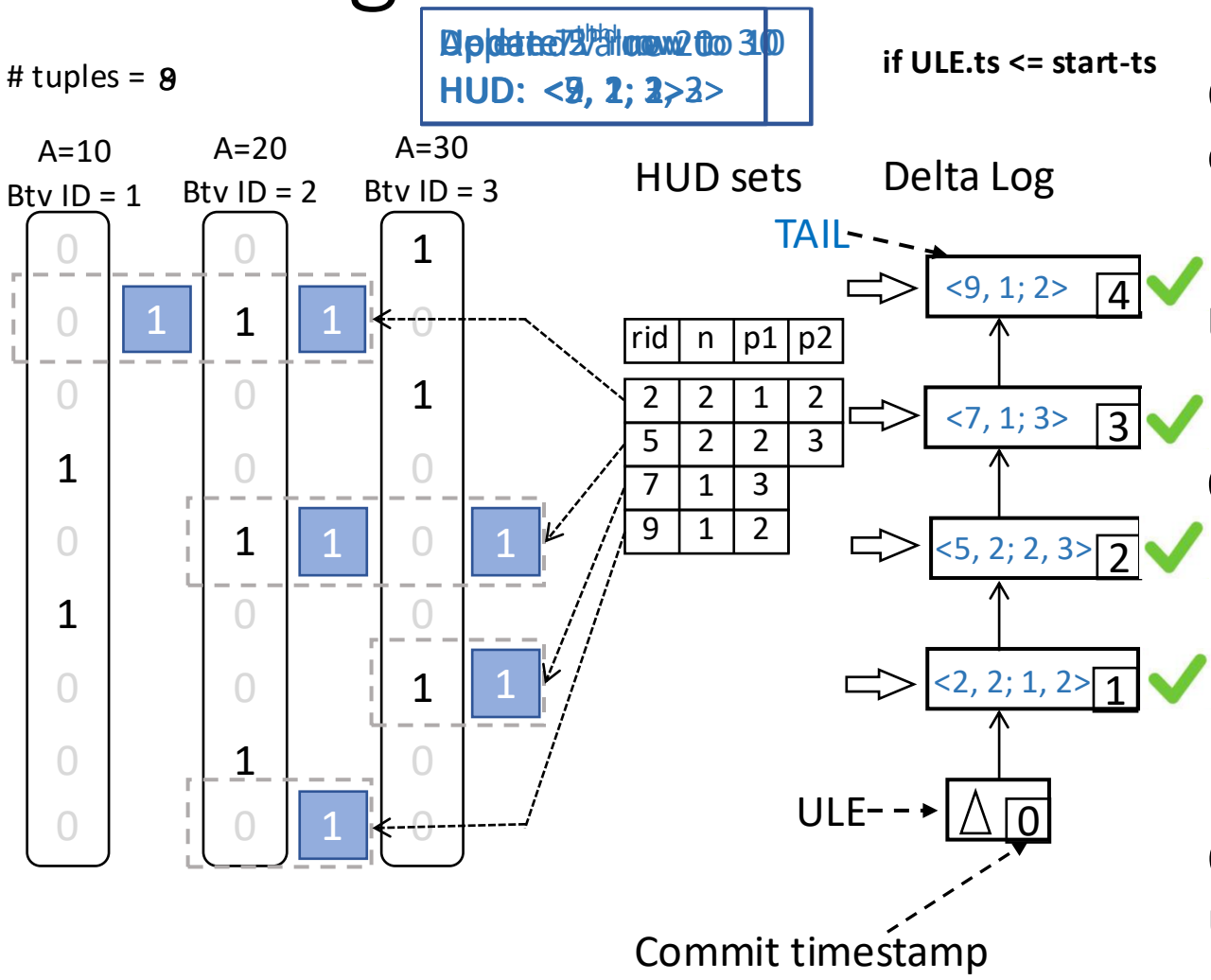
1. would flip two bits in bitvectors 2 and 3
   - Contention on multiple locks
2. CUBIT stores Horizontal Update Deltas (HUDs)
   - Use a lock-free data structure to handle updates
   - Conceptually the horizontal bit-array
   - Compactly stored as list of positions of the form <row_id, n_ones; p1, p2, …>

With high probability HUDs have up to 2 bits set for any domain cardinality

Compactly store update information that was scattered among VBs

Make update operation atomic

Design Element 2: MV with Timestamping



CUBIT stores HUDs in **UDI Log Entries (ULEs)** with commit timestamps

Per-bitmap (value) Delta Log contains linked ULEs

UDIs

- Commit by appending ULEs at TAIL

Queries

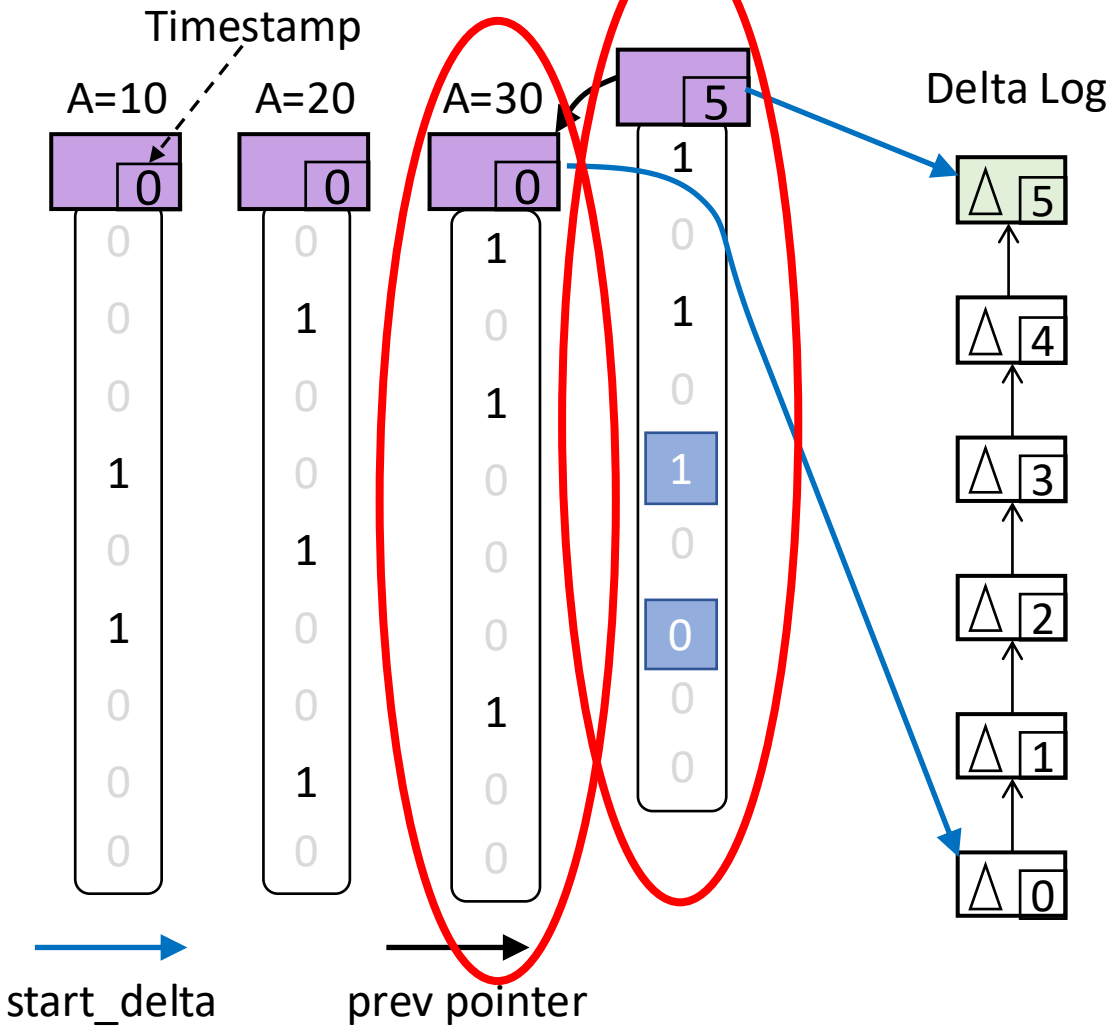
- Retrieve start-ts when started (e.g., 4)
- Fetch HUD sets from Delta Log
- Merge HUD sets into bitvectors on demand

Queries generate new bitvector versions with updated timestamps (used for merge operations)

Queries and UDIs are separated

Lightweight snapshot mechanism

Design Element 2: Merging Versions



Lazily merge HUDs into the bitvectors

Each **value** is associated with a **version chain**, with each version containing:

- Creation timestamp commit-ts
- A pointer start_delta to bypass the merged ULEs

Example: merging bitvectors on value 30

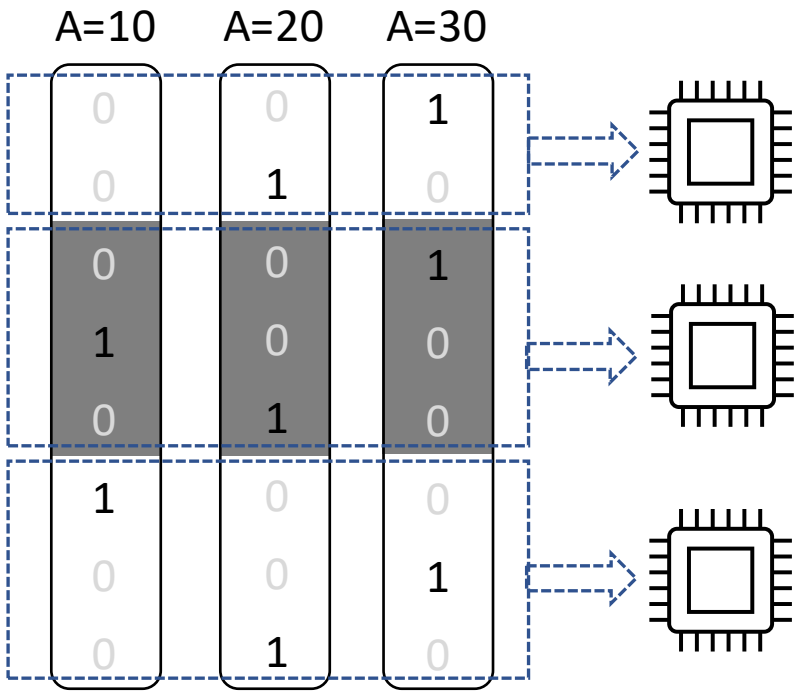
- Merge update information in HUDs into a private copy (with timestamp = 5)
- Point the new copy to a synthetic ULE, bypassing merged HUDs

By-product of queries

Accelerate subsequent queries

Design Element 3: Segmentation

Merge all bitvectors



Accelerate locating specified bits

Each bitvector is divided into multiple segments

Each segment is independently compressed

Already used in UpBit and Roaring

Moreover,

CUBIT's segmentation for parallel processing

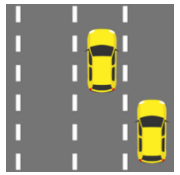
Each CPU core merges a group of segments across bitvectors

Intermediate results reside in cache

CUBIT



Out-of-place update



Multi-versioning



Bitvector Segmentation

Goals

Mitigate synchronization

Parallel queries and updates

Parallelism on multicores

Experiments

Synthetic data

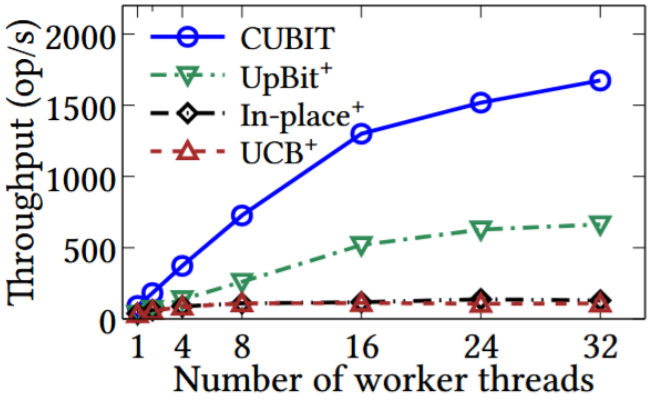
- ***n***: # tuples
- ***d***: # domain values (cardinality)
- ***q***: # queries
- ***u***: % updates in the workload

Prototype C++ implementation of CUBIT, UpBit, UCB, and In-place using FastBit

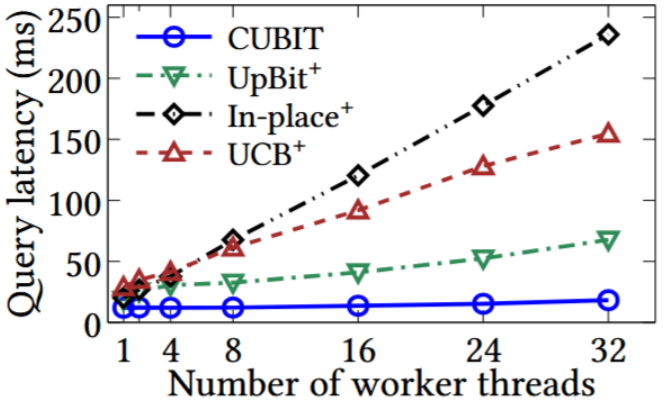
Integrated into a row-store prototype system (DBx1000) and a column-store system (DuckDB)

CUBIT Scales on Multicores

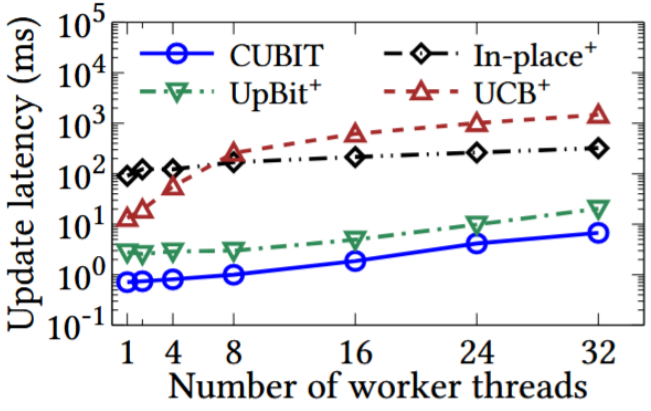
100M tuples with 100 cardinality, 10% UDI
Intel Xeon with 48 cores, 18*2MB LLC
192GB DRAM (indexes are in memory)



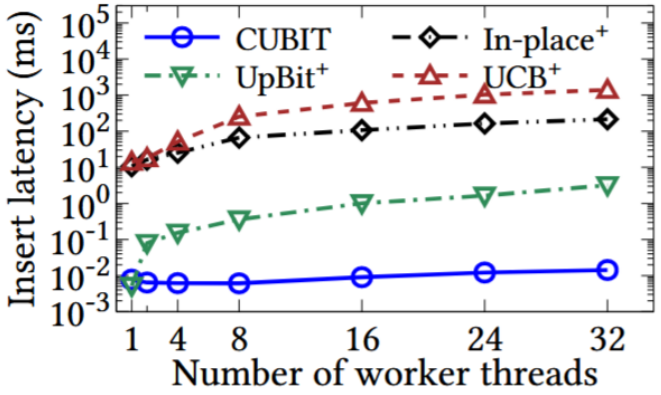
Overall throughput.



Query latency.



Update latency.



Insert latency.

Throughput

CUBIT outperforms UpBit (with per-bitvector latches) and UCB (with global latches)

Queries

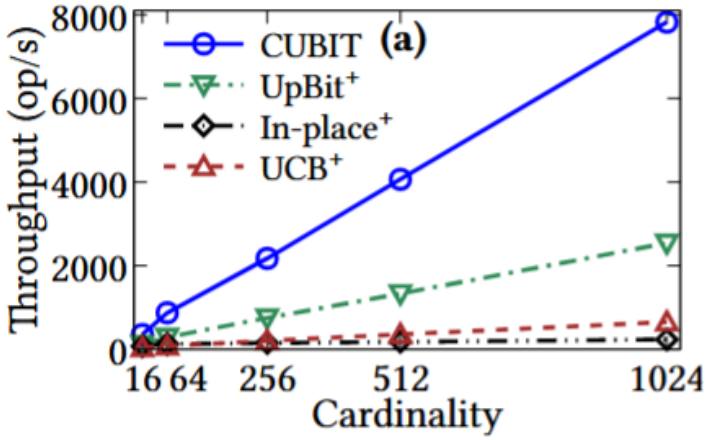
CUBIT provides wait-free queries

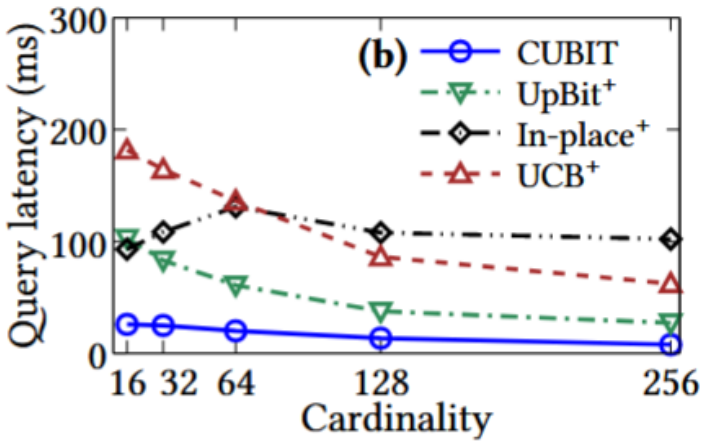
UDIs

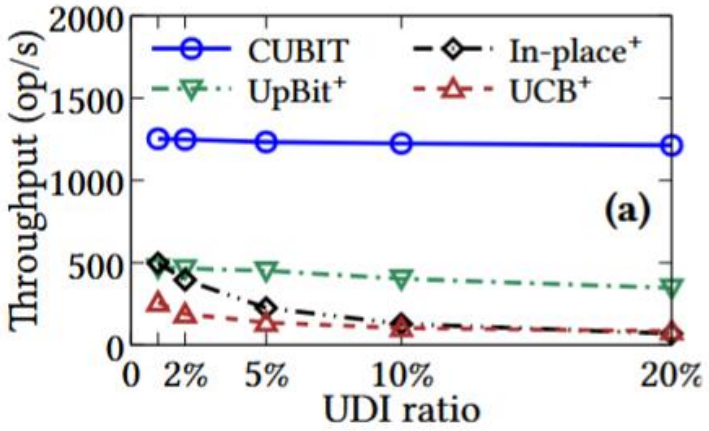
CUBIT commits by appending ULEs to Delta Log, which is more lightweight

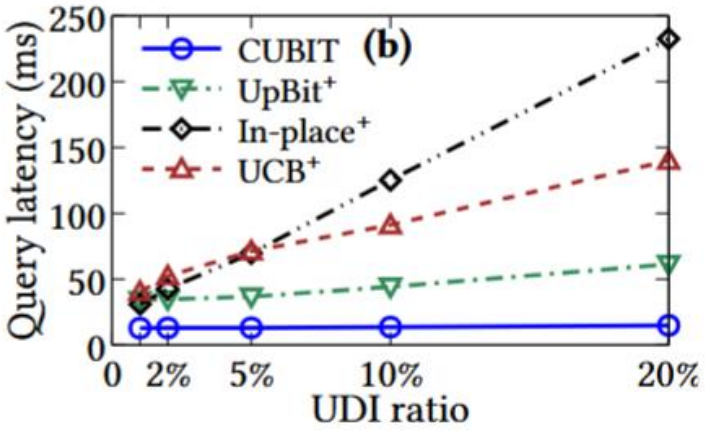
With 32 cores, 2.7x - 15.5x higher throughput and 3x – 224x faster updates

Sensitivity to Cardinality and Update Ratio









Same setup as before

Vary cardinality and UDI ratio separately

Increasing cardinality

Overall throughput increases as we have more compressible bitvectors and lower contention

Increasing UDI ratio

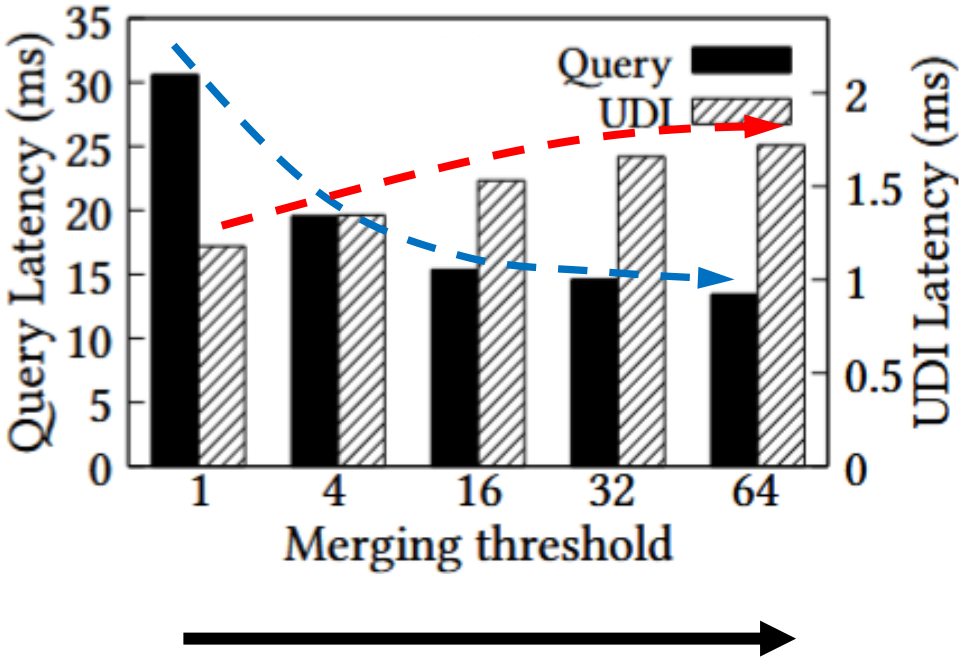
Increased contention in baselines

CUBIT is stable due to lock-free design

CUBIT's point query performance increases linearly with cardinality and is insensitive to UDI ratio

Tuning Merging Threshold

Same setup as before, but with 20% UDIs to trigger frequent merge



As merging threshold increases:

Query

Latency decreases because of the reduced frequency of generating new VBs

When the threshold is larger than 16, query latency plateaus because of the increased cost of traversing Delta Log

UDI

Latency increases slightly because of traversing the Delta Log

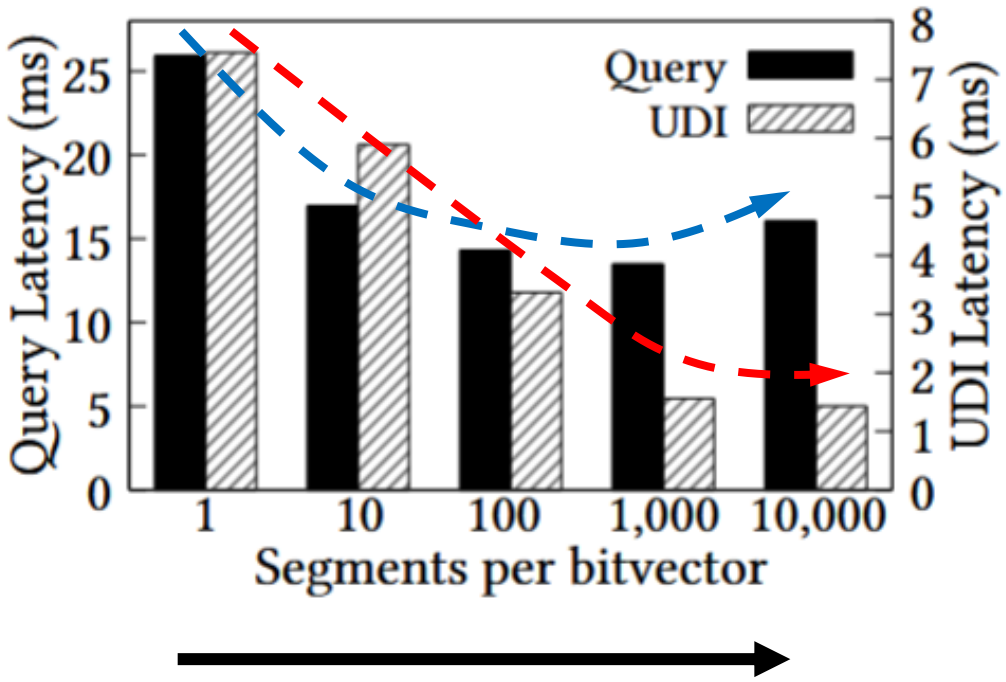
We set merging threshold to 16 in our experiments

Tuning Segmentation Granularity

Same setup as before

The size of each bitvector = 5MB

Two CPU cores



With a finer segmentation of the bitvector:

Query

Latency decreases by assigning segments to CPU cores more evenly

But increases beyond 1,000 segments due to segments traversal overhead

UDI

Always benefits

But plateaus after 5KB-sized segments

We divide each bitvector to 1,000 segments in our experiments with each containing 100K raw bit array (before compression)

CUBIT benefits OLAP

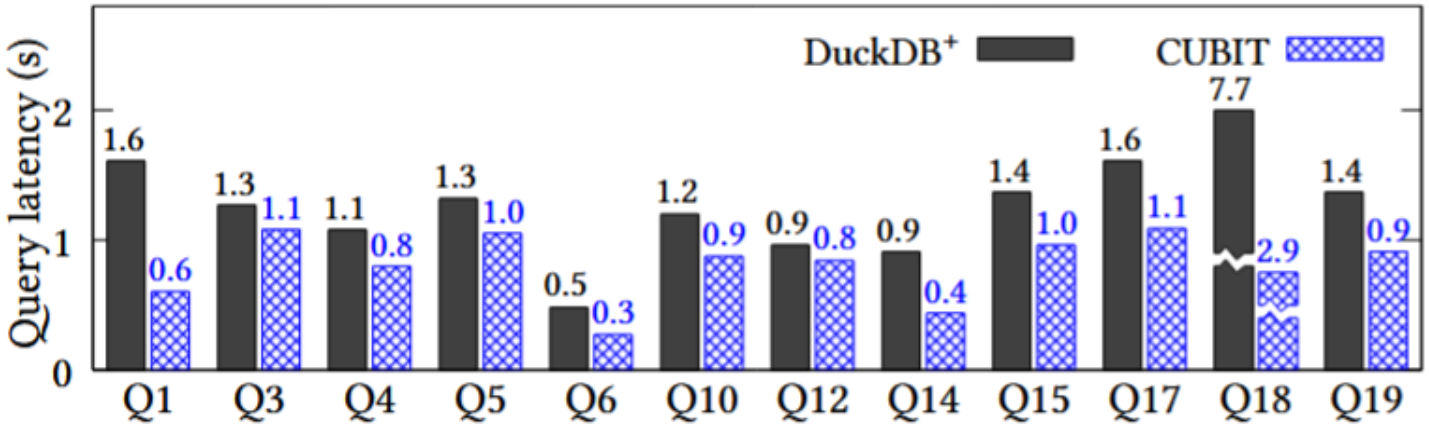
DuckDB with known choke points resolved

TPC-H SF 10 dataset

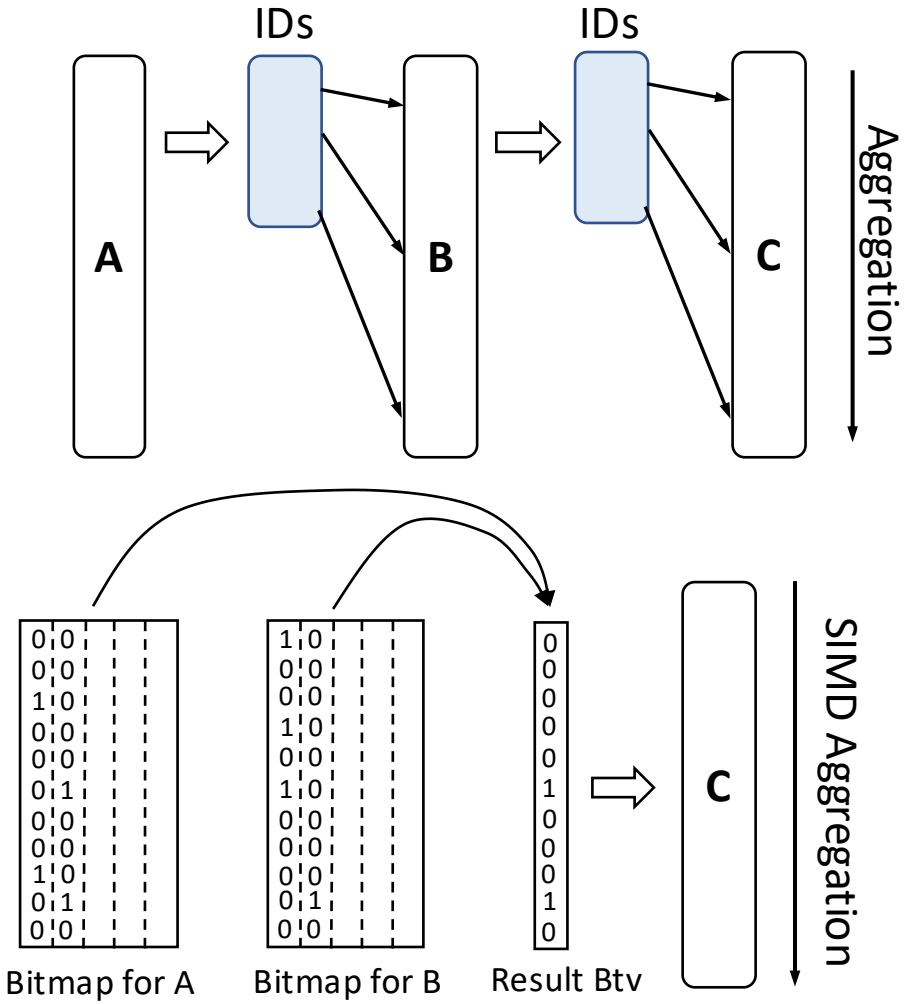
Single-core execution to focus on executor logic

Maintain CUBIT instances on attributes in WHERE/GROUP BY clauses, including

- l_quantity (cardinality = 50)
- l_shipdate (cardinality = 2,526)
- l_orderkey (cardinality = 15M)
- …



CUBIT benefits OLAP (Scan)



Select sum(C) **from** R **where** A > 5 **and** B < 10

For Scan-intensive queries:

Original DuckDB

1. scans two columns: *A and B*
2. Performs aggregation on qualifying values from *C*

CUBIT-powered Scan

1. performs logical ops on bitvectors from A and B
2. uses indexing result (clustered) to probe C

Benefit: reducing data read from storage

→ 1.6x query speedup

CUBIT-powered Scan probes less data

What about range queries?

(1) Barely compressible
(2) High update cost

		E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9
1	3	0	0	0	1	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	1	0	0
3	1	0	1	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	1	0	0	0
5	2	0	0	1	0	0	0	0	0	0	0
6	5	0	0	0	0	0	1	0	0	0	0
7	9	0	0	0	0	0	0	0	0	0	1
8	0	1	0	0	0	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	1	0	0
10	4	0	0	0	0	1	0	0	0	0	0
11	8	0	0	0	0	0	0	0	0	1	0
12	5	0	0	0	0	0	1	0	0	0	0

(a) (b) Equality Encoding (EE)

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	1	1
0	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1
0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	1	1	1

(c) Range Encoding (RE)

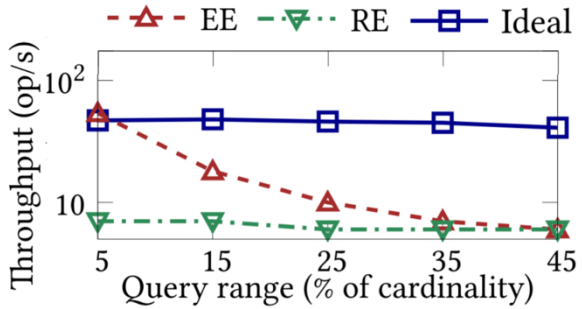
CUBIT mainly focuses on point queries:

- it adopts equality encoding scheme (EE)
- each (point) query reads one bitvector

However,

to support range queries (e.g., "A <= 8"),

- using EE leads to many ORs between bitvectors for high-cardinality attributes
- adopting range encoding faces storage and update challenges



Improving CUBIT to support range queries (e.g., encoding scheme)

Supporting Range Queries on High-Cardinality Attributes using Bitmap Indexing

Limitations with Equality Encoding (EE)

		E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9
1	3	0	0	0	1	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	1	0	0
3	1	0	1	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	1	0	0	0
5	2	0	0	1	0	0	0	0	0	0	0
6	5	0	0	0	0	0	1	0	0	0	0
7	9	0	0	0	0	0	0	0	0	0	1
8	0	1	0	0	0	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	1	0	0
10	4	0	0	0	0	1	0	0	0	0	0
11	8	0	0	0	0	0	0	0	0	1	0
12	5	0	0	0	0	0	1	0	0	0	0

The bit of the corresponding bitvector is set to 1, while other bits 0s.

✓ *Good for point queries (e.g., A = 5)*

✓ *Good for updates*

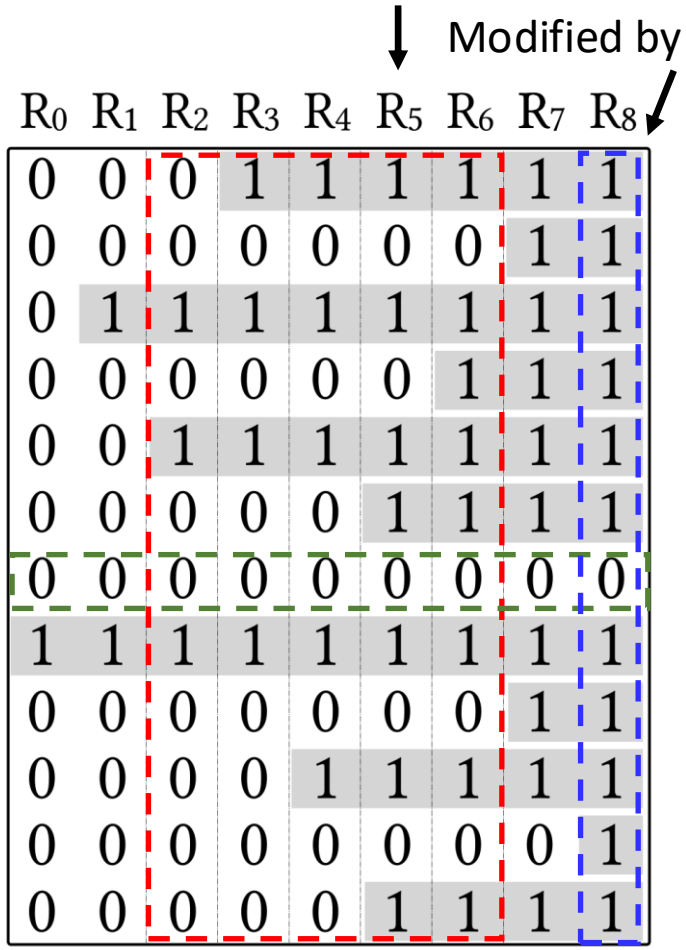
but …

✗ *Bad for range queries (e.g., A < 6)*

- For high-cardinality attributes, many (e.g., 1000+) bitvectors must be merged.

EE is effective for point queries, but is poor for range queries.

Limitations with Range Encoding (RE)



R(v, b): v <= b

For a tuple value **v**, if the value **b** of a bitvector satisfies a given predicate R(**v**, **b**), the corresponding bit of the vector is set to 1, otherwise 0.

Answer a range query by reading a bitvector
- E.g., query A < 6 is answered by reading R_5

☑ *Good for range queries*

but …

☒ *Bitvectors are barely compressible*

☒ *Updates flip bits in many bitvectors*

☒ *High contention on edge bitvectors*

RE performs well for range queries, but has several drawbacks

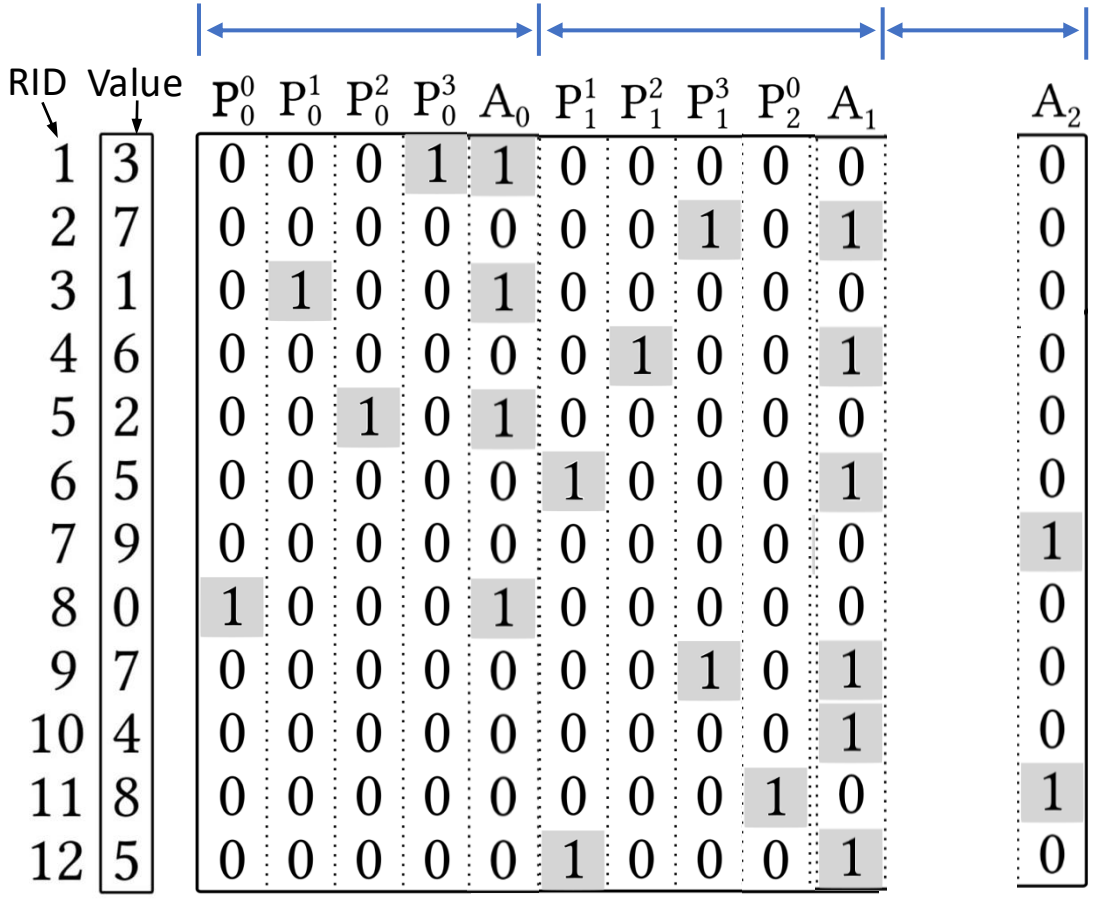
Goal

*A new **encoding scheme for efficient range queries** on frequently updated, high-cardinality attributes*

Solution: Group Encoding (GE)

- Fast range query performance like RE
- As space efficient as EE
- Match EE's update efficiency

Group Encoding (GE)



RID	Value	P_0^0	P_0^1	P_0^2	P_0^3	A_0	P_1^1	P_1^2	P_1^3	P_2^0	A_1	A_2
1	3	0	0	0	1	1	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	1	0	1	0
3	1	0	1	0	0	1	0	0	0	0	0	0
4	6	0	0	0	0	0	0	1	0	0	1	0
5	2	0	0	1	0	1	0	0	0	0	0	0
6	5	0	0	0	0	0	1	0	0	0	1	0
7	9	0	0	0	0	0	0	0	0	0	0	1
8	0	1	0	0	0	1	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	1	0	1	0
10	4	0	0	0	0	0	0	0	0	0	1	0
11	8	0	0	0	0	0	0	0	0	1	0	1
12	5	0	0	0	0	0	1	0	0	0	1	0

Point bitvectors (P_i):

- Assigns a distinct point bitvector to each value
- Similar to EE bitvectors (highly compressible)

Grouping:

- GE partitions the value domain into groups
- Maintains a cumulative bitvector for each group

Cumulative bitvectors (A_i):

- Aggregates all point bitvectors in each group
- Answer range queries efficiently

GE combines the advantages of EE and RE

Efficient Range Queries with GE

		P_0^0	P_0^1	P_0^2	P_0^3	A_0	P_1^0	P_1^1	P_1^2	P_1^3	A_1	P_2^0	P_2^1	A_2
1	3	0	0	0	1	1	0	0	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	0	1	1	0	0	0
3	1	0	1	0	0	1	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	0	1	0	1	0	0	0
5	2	0	0	1	0	1	0	0	0	0	0	0	0	0
6	5	0	0	0	0	0	0	1	0	0	1	0	0	0
7	9	0	0	0	0	0	0	0	0	0	0	0	1	1
8	0	1	0	0	0	1	0	0	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	0	1	1	0	0	0
10	4	0	0	0	0	0	1	0	0	0	1	0	0	0
11	8	0	0	0	0	0	0	0	0	0	0	1	0	1
12	5	0	0	0	0	0	0	1	0	0	1	0	0	0

(a) (b) Group Encoding (GE)

With EE: Merge a group of bitvectors

- E.g., for "X <= 4", merge 5 bitvectors

With GE: Merge a few cumulative bitvectors

- and a few point bitvectors in boundary groups
- E.g., for "X <= 4", merge 2 bitvectors

Assume: *Column cardinality = C* and *Group size = G*

Range query on GE (e.g., $X \leq v$) merges

- Cumulative bitvectors A_i with $i \leq \lfloor v/G \rfloor$,
- Point bitvectors in the last group

Significantly less than point bitvectors merged using EE

$$\text{"}X \leq v\text{"} := \begin{cases} \bigvee_{i=0}^{\lfloor v/G \rfloor} A_i & \text{If } (v+1) \bmod G = 0, \\ \left(\bigvee_{i=0}^{\lfloor (v-G)/G \rfloor} A_i\right) \vee \left(\bigvee_{i=0}^{v \bmod G} P^i_{\lfloor v/G \rfloor}\right) & \text{Otherwise.} \end{cases}$$

Leveraging cumulative bitvectors, GE provides fast range queries

Efficient updates with GE

		P_0^0	P_0^1	P_0^2	P_0^3	A_0	P_1^0	P_1^1	P_1^2	P_1^3	A_1	P_2^0	P_2^1	A_2
1	3	0	0	0	1	1	0	0	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	0	1	1	0	0	0
3	1	0	1	0	0	1	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	0	1	0	1	0	0	0
5	2	0	0	1	0	1	0	0	0	0	0	0	0	0
6	5	0	0	0	0	0	0	1	0	0	1	0	0	0
7	0	1	0	0	0	1	0	0	0	0	0	0	0	0
8	0	1	0	0	0	1	0	0	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	0	1	1	0	0	0
10	4	0	0	0	0	0	1	0	0	0	1	0	0	0
11	8	0	0	0	0	0	0	0	0	0	0	1	0	1
12	5	0	0	0	0	0	0	1	0	0	1	0	0	0

(a) (b) Group Encoding (GE)

Updating GE is efficient:

- Each update flips at most four bits
  - Two bits when the values are in the same group
- E.g., changing the 7th row to value 0 updates bits in P_0^0, A_0, P_2^1, A_2

Update cost matches EE

- EE updates two bits (in old and new bitvectors)

Orders of magnitude lower than RE

- RE updates on average *Cardinality/2* bits

By omitting redundant encoding, GE enables efficient updates

GE is Space Efficient

Data	N	Number of values
	C	Cardinality of the indexed attribute
	S_u	Size of uncompressed bitvector ($N/8$)
	r_{comp}	Compression ratio of point bitvector
	S_c	Size of compressed point bitvector (S_u/r_{comp})
RABIT	G	Group size of GE

The size of a GE bitmap index consists of:

- Compressed Point bitvectors : $C \cdot S_c$
- Decompressed Cumulative bitvectors: $\lceil C/G \rceil \cdot S_u$
- The total size of a GE bitmap index is $S(N, C, G) = C \cdot S_c + \lceil C/G \rceil \cdot S_u$

Our evaluation shows that

- $r_{\text{comp}} \approx C \cdot \alpha$ (uniform distribution, WAH compression)
- $\alpha \approx 0.016$ (Cardinality $C > 50$)

The additional memory overhead compared to its EE counterpart can be expressed as:

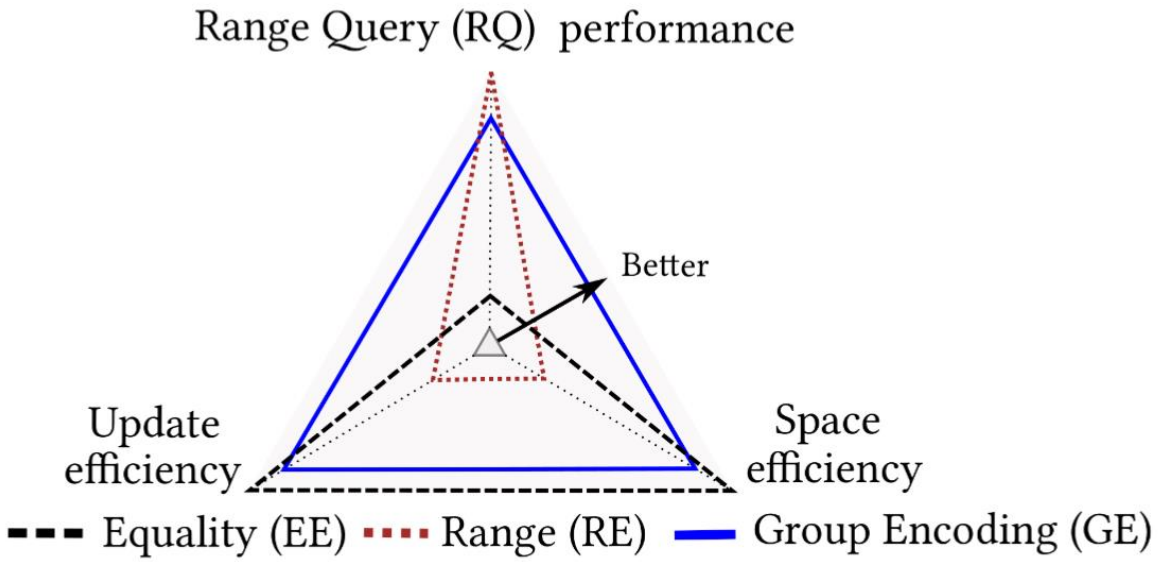
$$Overhead_{GE}(C, G) = \frac{\lceil C/G \rceil \cdot S_c \cdot r_{\text{comp}}}{C \cdot S_c} \approx \frac{r_{\text{comp}}}{G} \approx \frac{C \cdot 0.016}{G}$$

~26% when C/G = 16

By omitting redundant encoding, GE is as space efficient as EE

Goals of GE

- **Space efficient**
- **Update friendly**
- **Good for both point and range queries**



Experiments

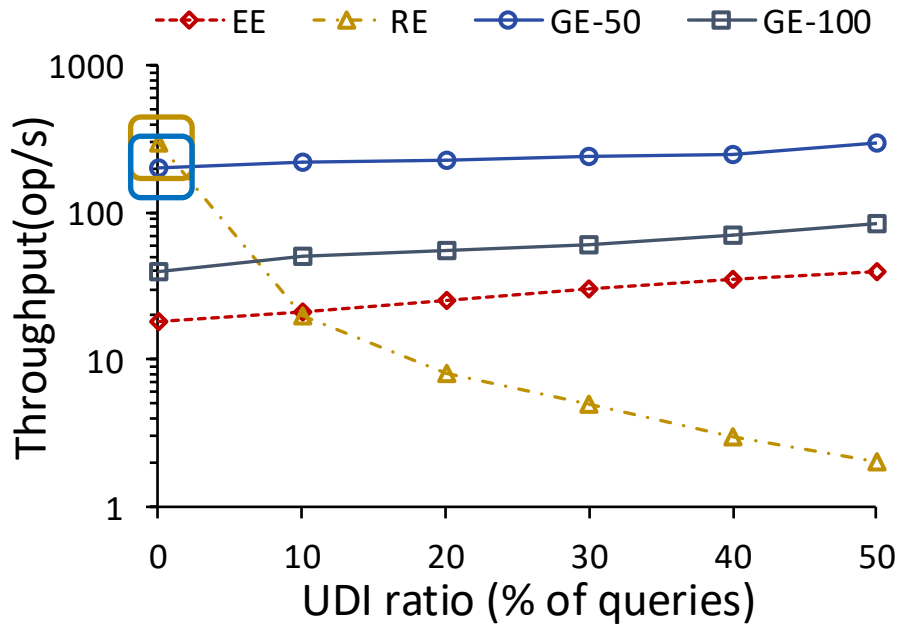
Prototype C++ implementation of UpBit (RE), CUBIT (EE), and RABIT (GE)

HTAP benchmark that runs TPC-C and -H transactions on TPC-H workloads (SF = 10)

Intel Xeon with 48 cores, 18*2MB LLC
192GB DRAM (indexes are in memory)

GE is Optimal for Range Queries

100M tuples with 1000 cardinality
GE in three group size (50, 100, 200)
Bitvectors are compressed using WAH



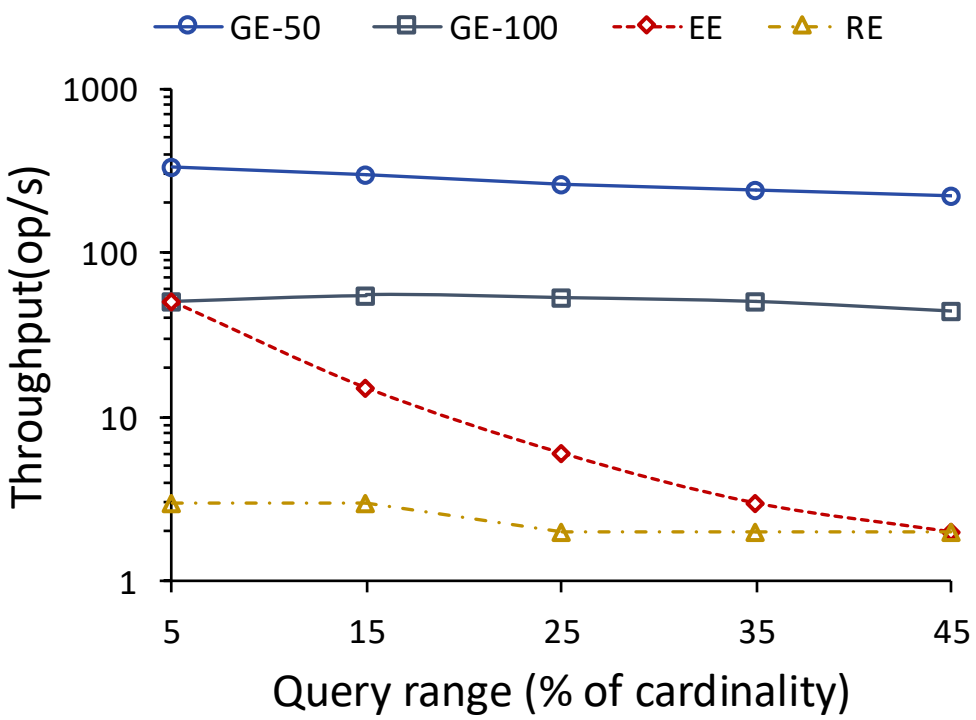
Varying UDI ratio

- For read-only workloads (0 UDIs), RE provides the highest performance
  - A proper Group size (50) makes GE as efficient as RE
  - EE is the slowest as it must OR bitvectors
- With UDIs, GE delivers superior performance, whereas RE performance drops significantly

With 20% UDIs, GE-100 is 2.7× (6.9×) faster, and GE-50 is 14× (37×) faster than EE (RE)

GE is Optimal for Range Queries

100M tuples with 1000 cardinality
GE in three group size (50, 100, 200)
Bitvectors are compressed using WAH



Varying query range

- GE experiences negligible degradation by leveraging cumulative bitvectors
- With small query range (5%), GE-100 degrades to EE, both ORing 50 bitvectors
- As query range increases, EE throughput drops significantly due to more bitvectors ORed
- RE achieves the lowest throughput due to the conflicts between UDIs and queries

When the query range reaches 45%, GE-50 outperforms the baselines by over 52×

RABIT (with GE) benefits OLAP

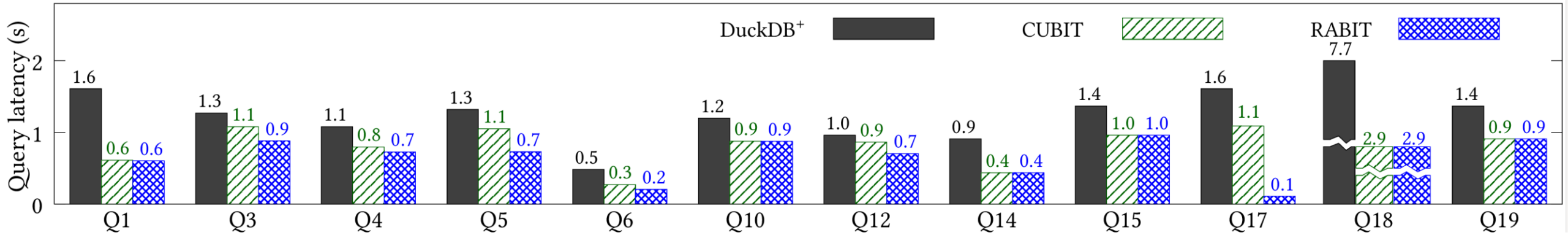
DuckDB (TPC-H SF 10 dataset)

Single-core execution to focus on executor logic

Use state-of-the-art CUBIT as bitmap indexing baseline

Maintain RABIT instances on attributes in WHERE/GROUP BY clauses, including

- l_shipdate (cardinality = 2,526, group size = 3 months)
- l_discount (cardinality = 10)
- l_quantity (cardinality = 50, group size = 5)



General-Purpose Bitmap Indexes

- Update-friendly via out-of-place updates
- Concurrency-aware via latch-free multi—versioning
- Range-Query-friendly via Group Encoding
  - A hybrid of Equality and Range Encoding

CS 561: **Data Systems Architectures**

class 14

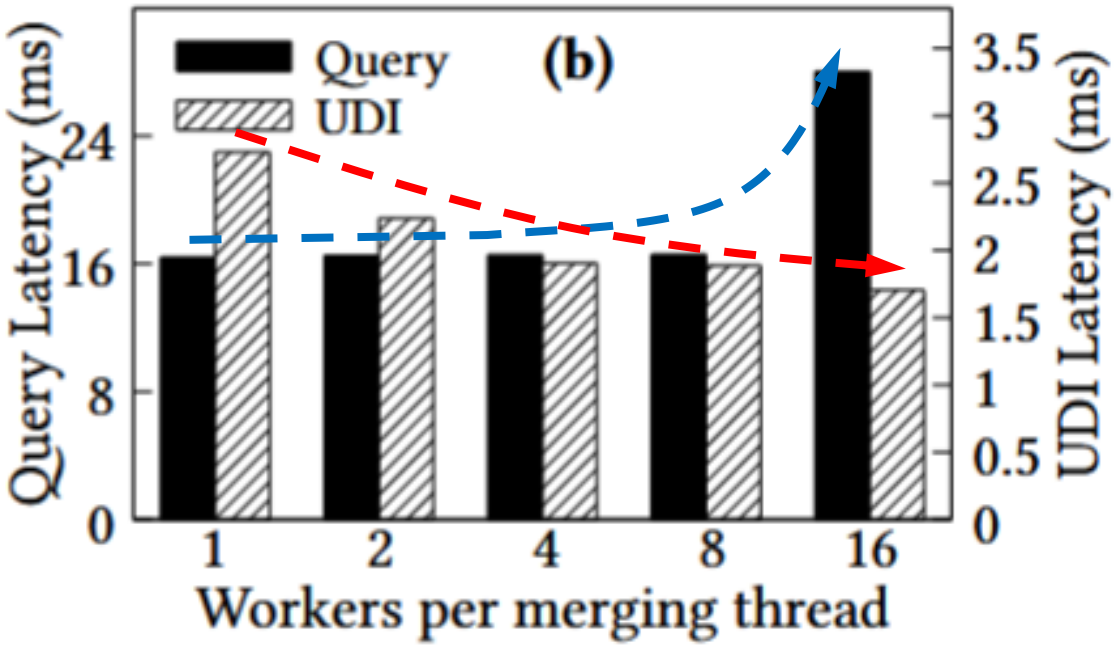
General-Purpose Bitmap Indexes

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

Backup Slides for Q&A

Tuning Number of Maintenance Threads



Same setup as before, but with 20% UDIs

As workers per merging thread increases:

Query

Latency plateaus when the ratio is less than 8.

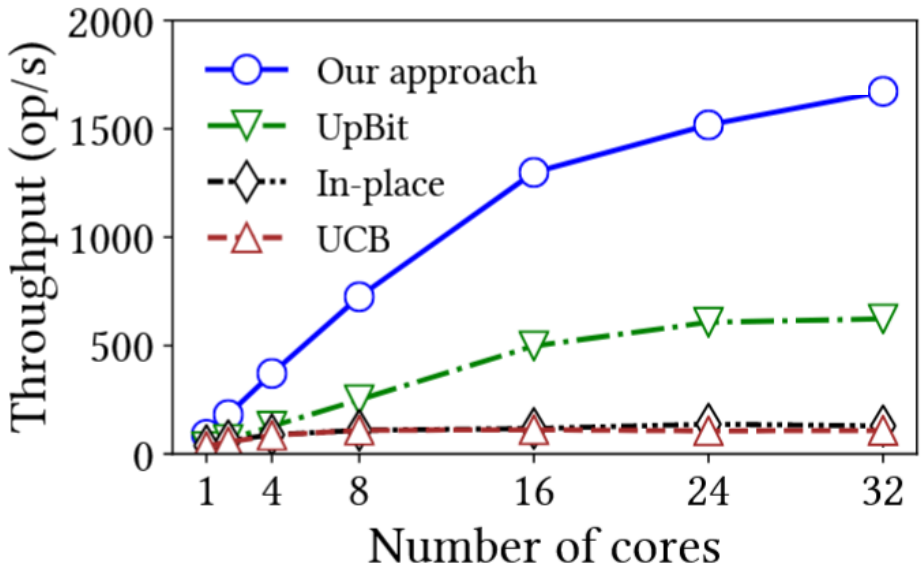
Later, performance decreases because maintenance threads become the bottleneck

UDI

Always benefit because of decreased contention on Delta Log

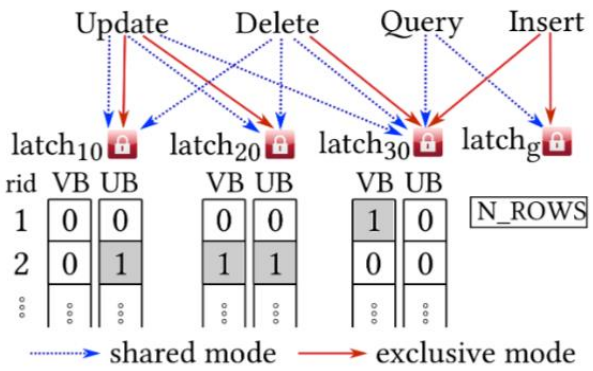
We set the ratio of worker/maintenance threads to <8

Prior Work: Evaluation Results

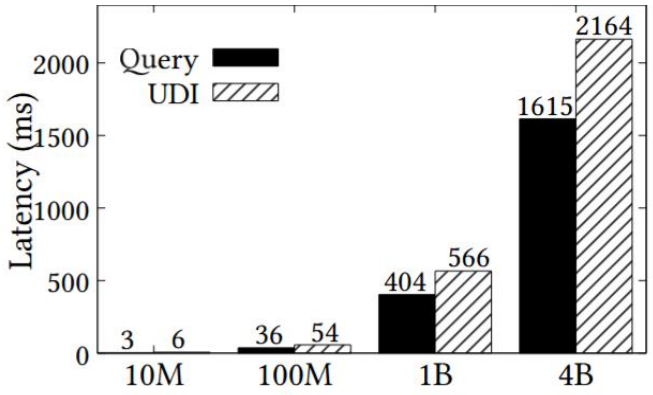


Query performance decreases for high concurrency when UDIs are in presence

why?



Parallelized UpBit.



Mean query and UDI latency.

(a) High contention on latches of <VB, UB> pairs

(b) Long critical sections

(c) Hot bitvectors (for skewed data)

Does not scale on multicores

RABIT (with GE) benefits OLAP

DuckDB (TPC-H SF 10 dataset)

Single-core execution to focus on executor logic

CUBIT relies on query-specific binning for range queries, making it less flexible

CUBIT-no-Binning suffers from high overhead due to OR operations on point bitvectors

Maintain RABIT instances on attributes in WHERE/GROUP BY clauses, including

- l_shipdate (cardinality = 2,526, group size = 3 months)
- l_discount (cardinality = 10)
- l_quantity (cardinality = 50, group size = 5)

