

Building General-Purpose Bitmap Indexes

A tale of UpBit, CUBit, RABIT, and beyond

Manos Athanassoulis

Keynote at ADMS @ VLDB 2026, Boston

Bitmap Index Basics

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

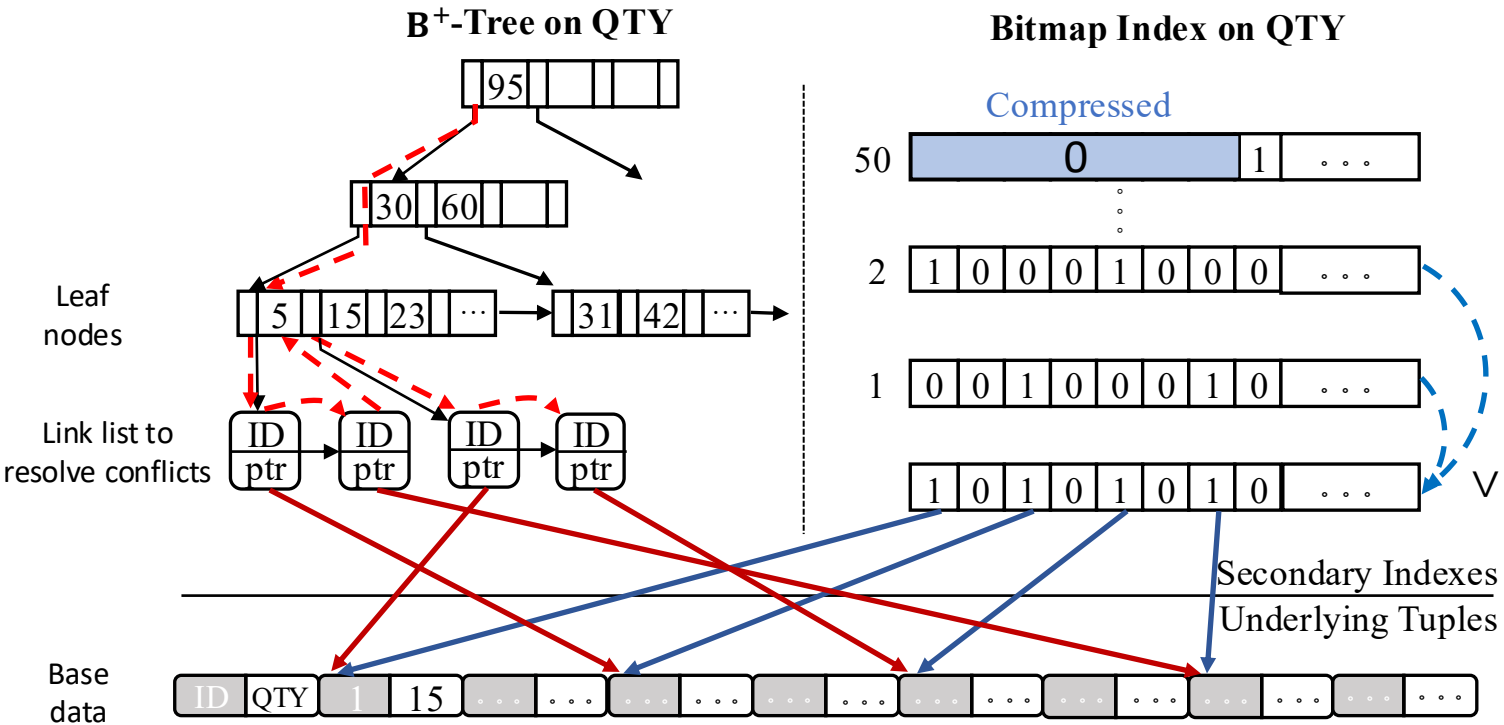
Specialized indexing

- ☒ Compact representation of 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

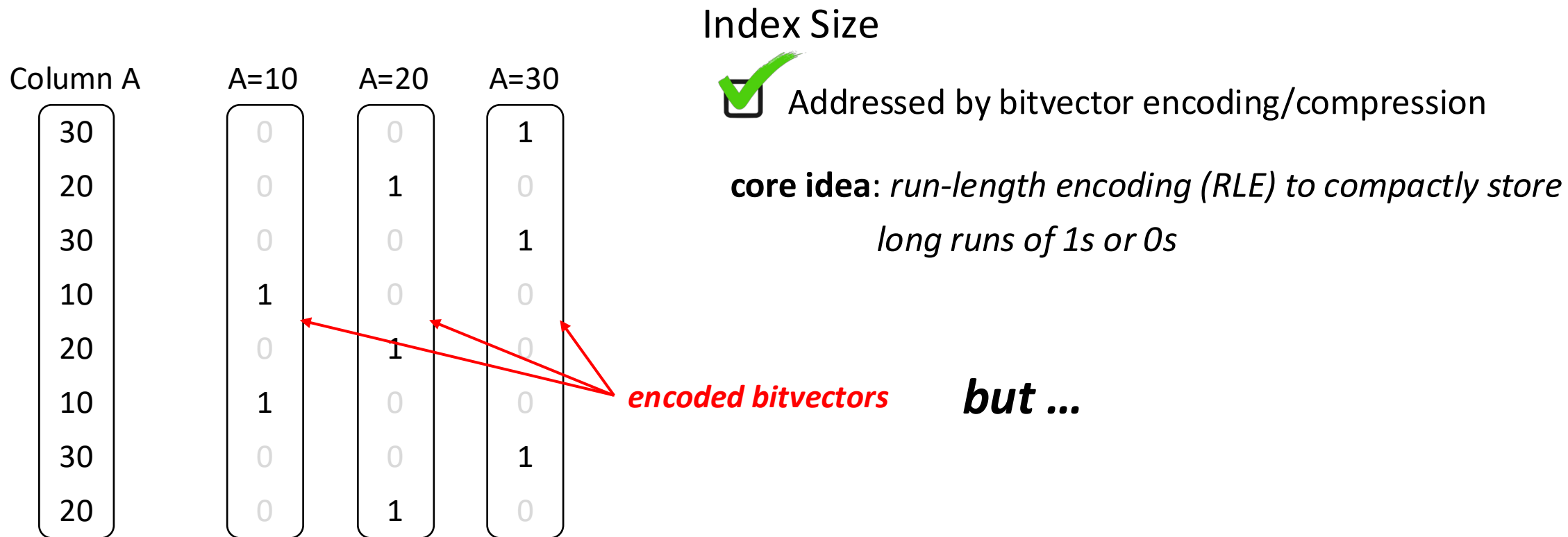


Query result is clustered

Compact memory footprint

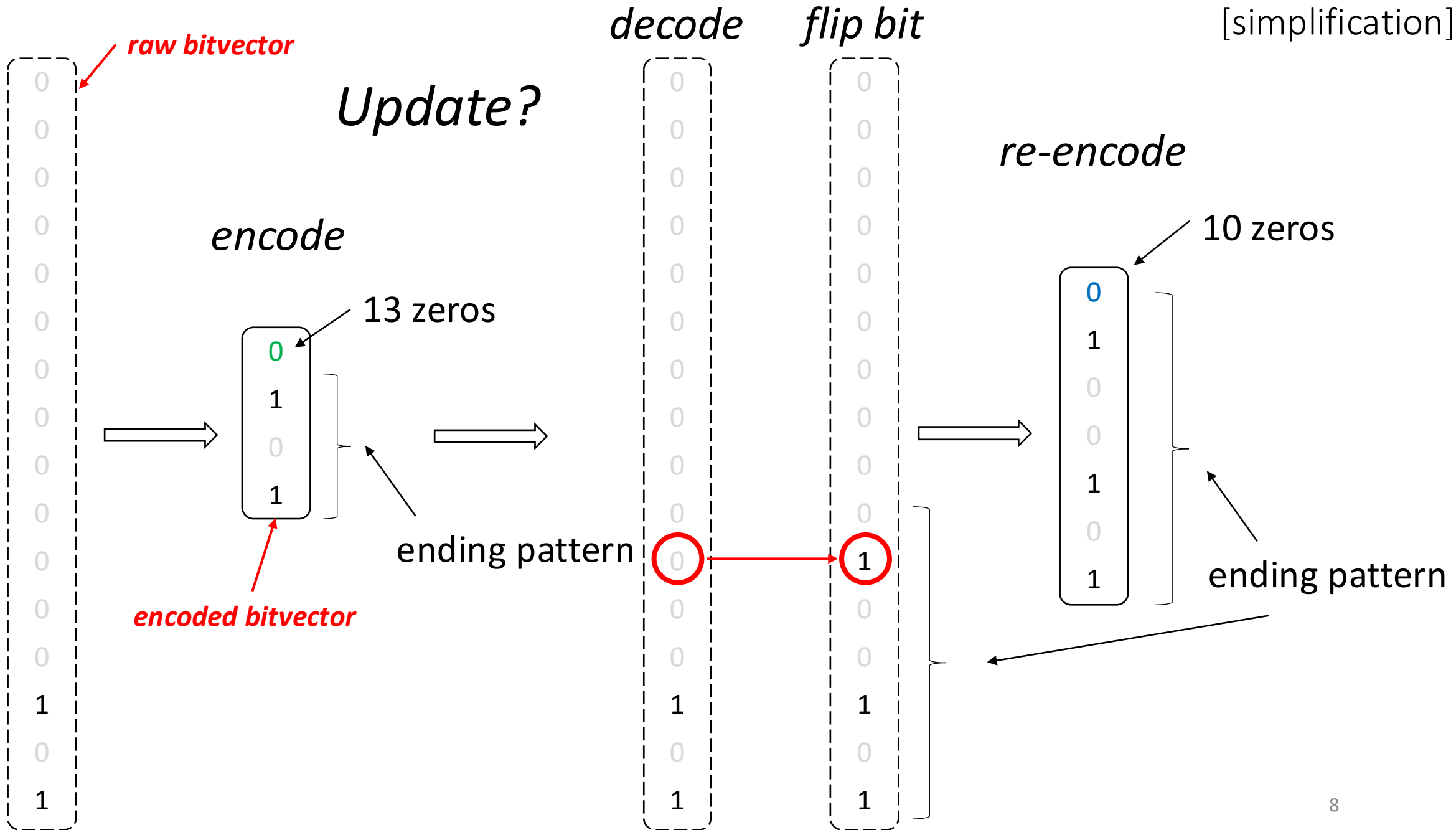
Composition-friendliness for multi-column queries

Bitmap Indexing Limitation (Updatability)



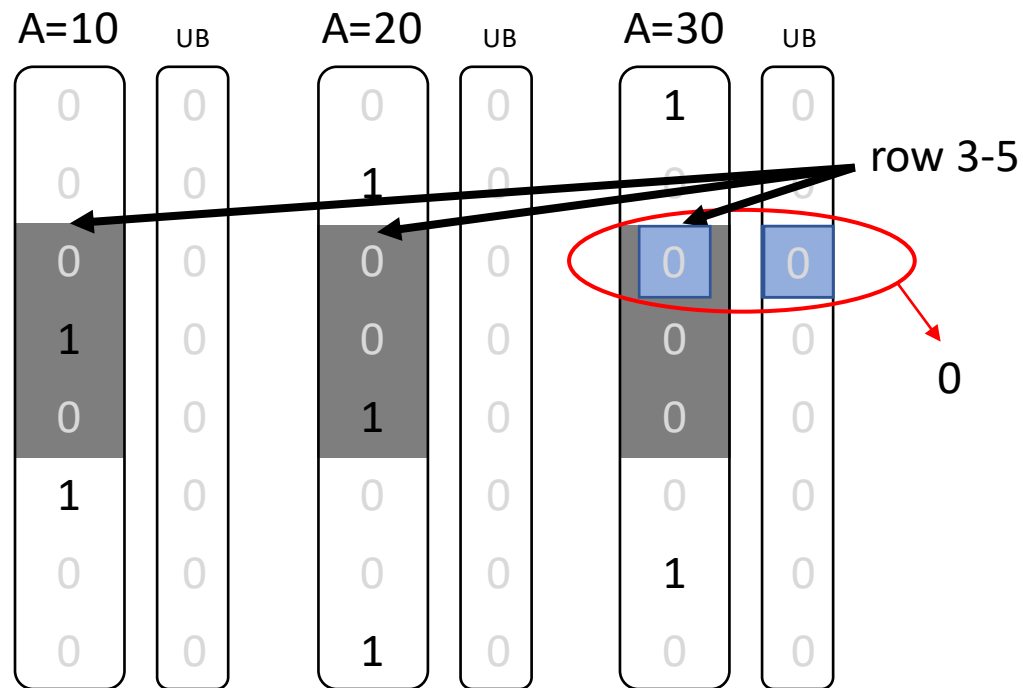
Updating encoded bitvectors is **very** inefficient

Updatability



Prior Work: UpBit

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



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

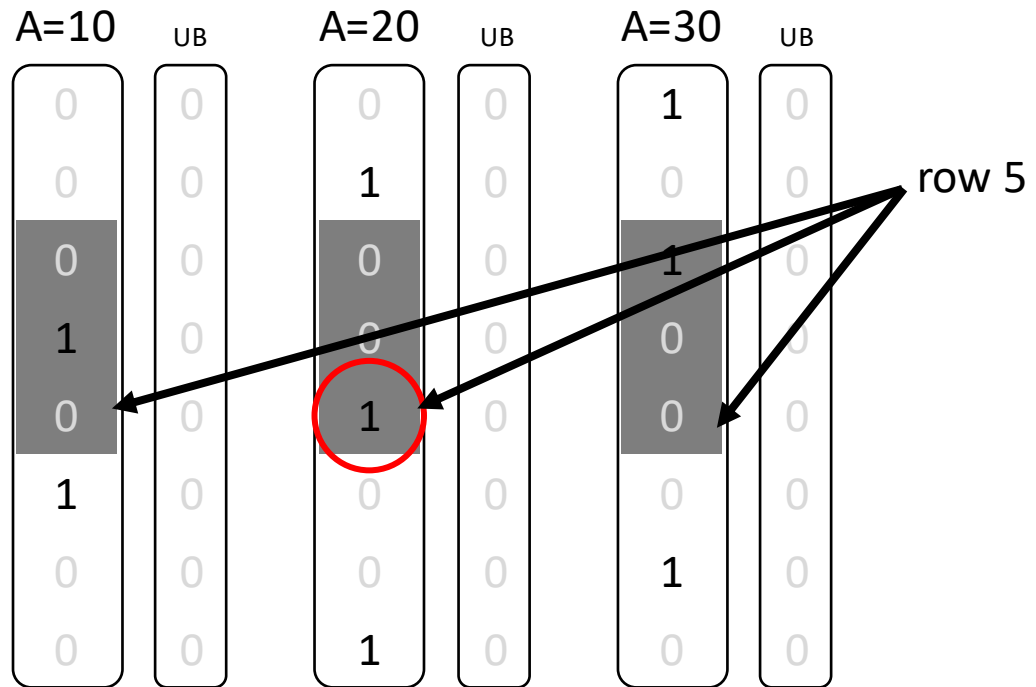
Updates/Deletes: only flip bits in UB

Query: XOR the corresponding $\langle \text{VB}, \text{UB} \rangle$ pair

Fence pointers: to quickly fetch bits in compressed bitvectors

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

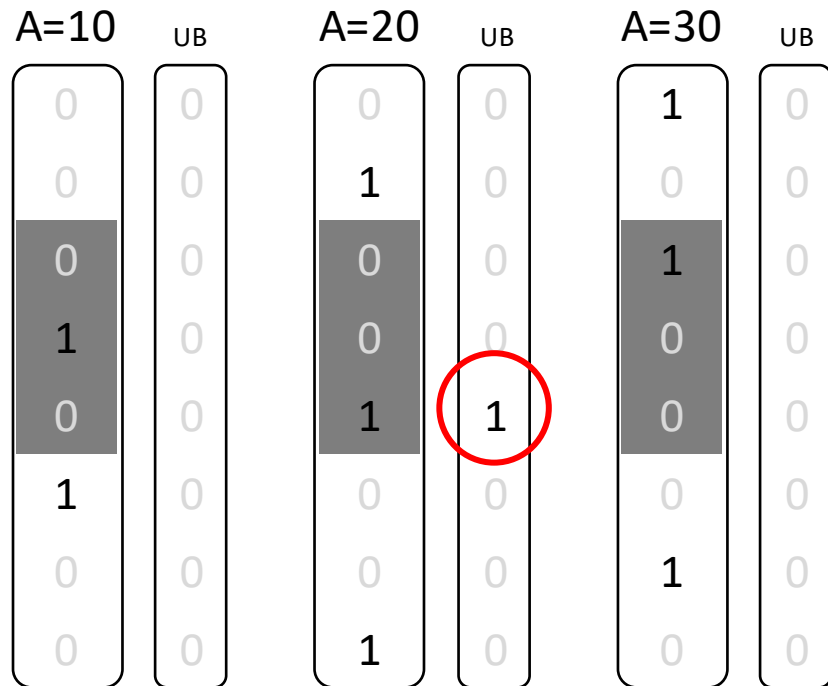
Updating UpBit ...



... row 5 to 30

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

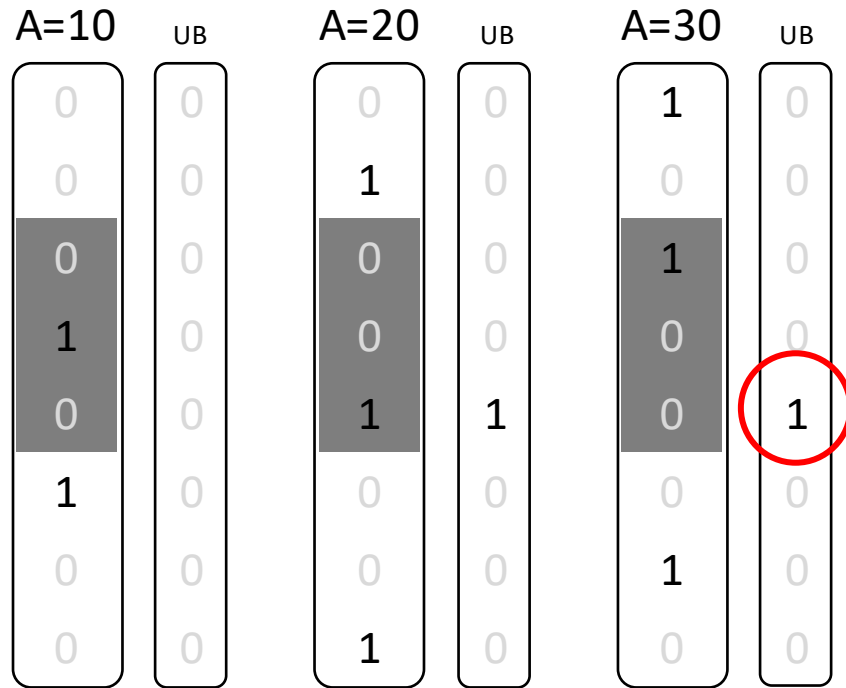
Updating UpBit ...



... 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

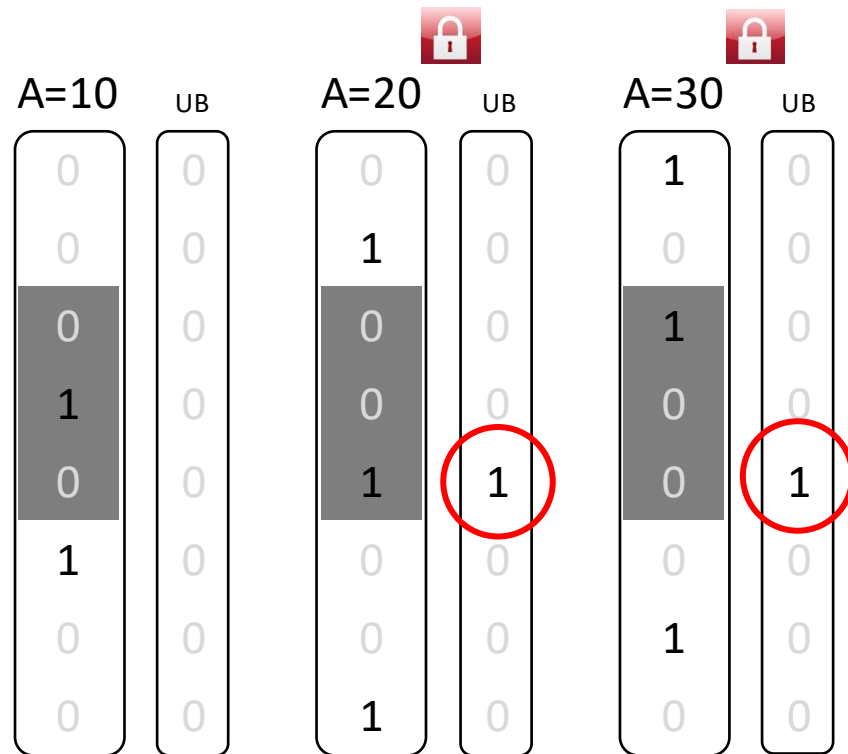
Updating UpBit ...



... 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

Scalability Issue



... row 5 to 30

Acquire latches for 20 and 30 **before updating UBs**

Update is **costly** due to **decode-flip-encode**

Queries are **blocked**, leading to **long tail latency**

Does not scale on modern multicore systems

Goal

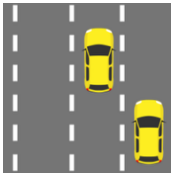
Concurrent Updatable Bitmap Indexing to overcome the key limitation of bitmap-based secondary indexes

- **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

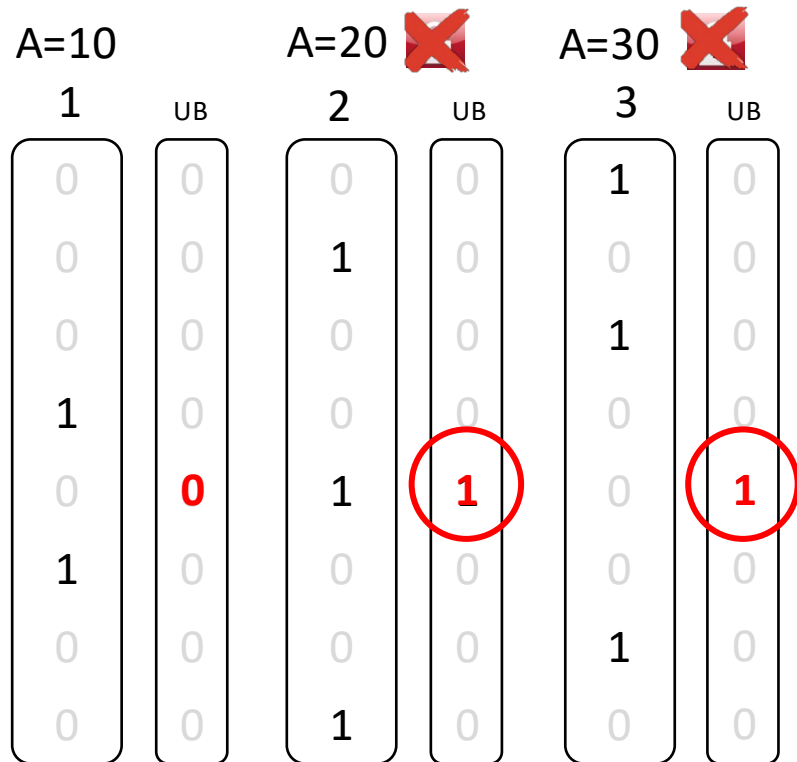


Multi-versioning enables parallel execution of queries and updates



Bitvector Segmentation exploits parallelism on multicore systems

Design Element 1: Out-of-place Updates



... row 5 to 30

1. would flip two bits in bitvectors 2 and 3

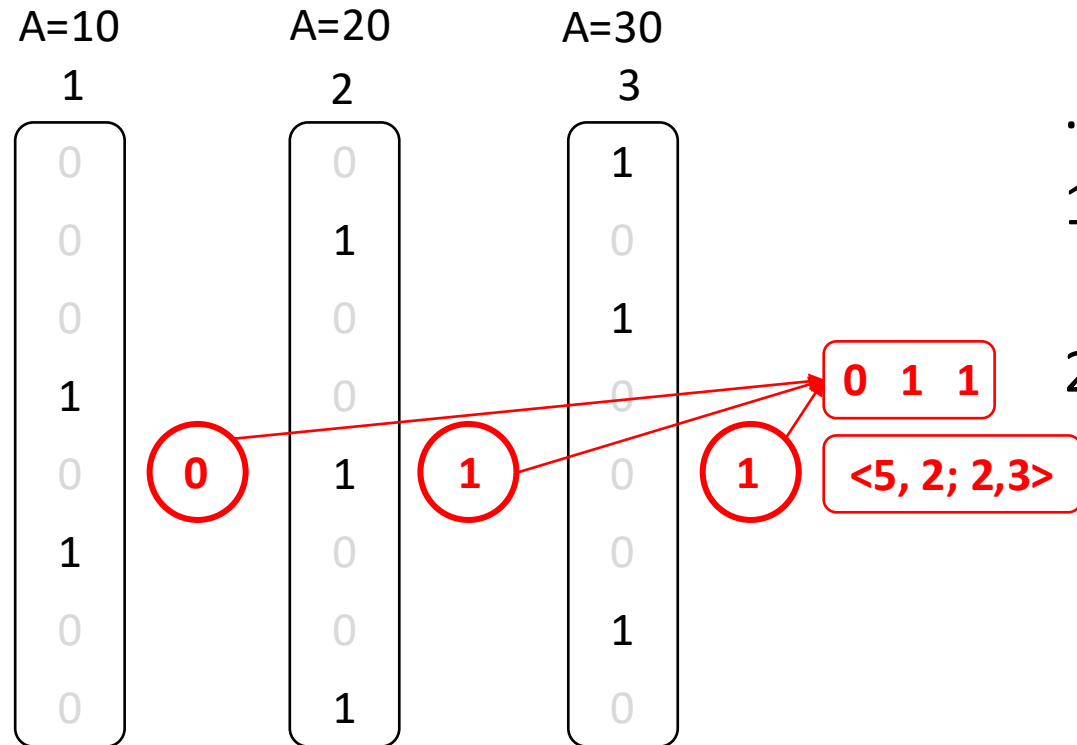
- Contention on multiple locks

Idea: Remove locks altogether!

2. CUBIT stores Horizontal Update Deltas (HUDs)

- Use a lock-free data structure to handle updates

Design Element 1: Out-of-place Updates



... row 5 to 30

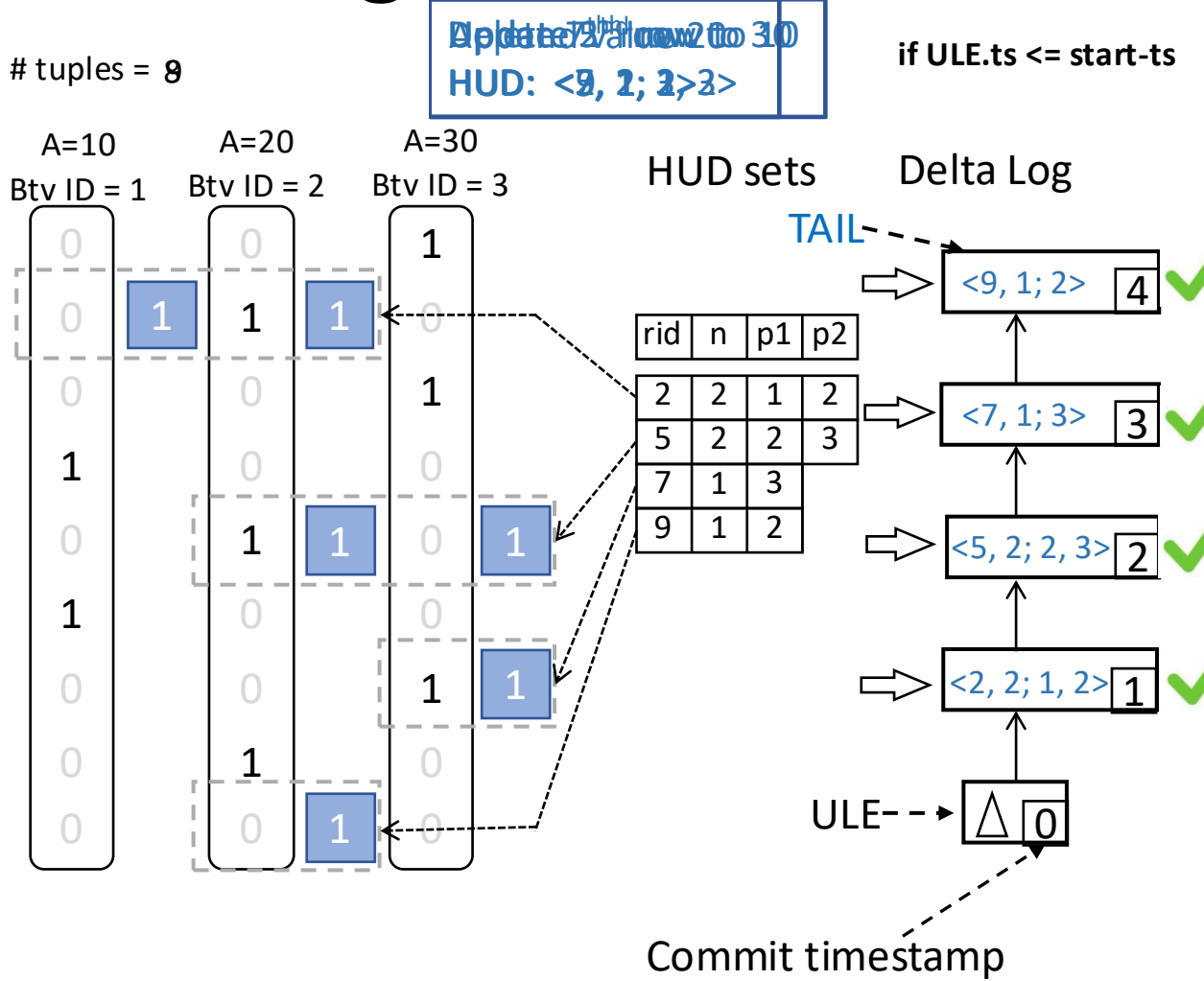
1. would flip two bits in bitvectors 2 and 3
 - Contention on multiple locks
2. CUBIT stores Horizontal Update Deltas (HUDs)
 - Use a lock-free data structure to handle updates
 - Conceptually the horizontal bit-array
 - Compactly stored as list of positions of the form <row_id, n_ones; p1, p2, ...>

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



HUDs in **UDI Log Entries (ULEs)** with commit tmstmp

Per-bitmap-index Delta Log contains linked ULEs

UDIs

- Commit by appending ULEs at **TAIL**

Queries

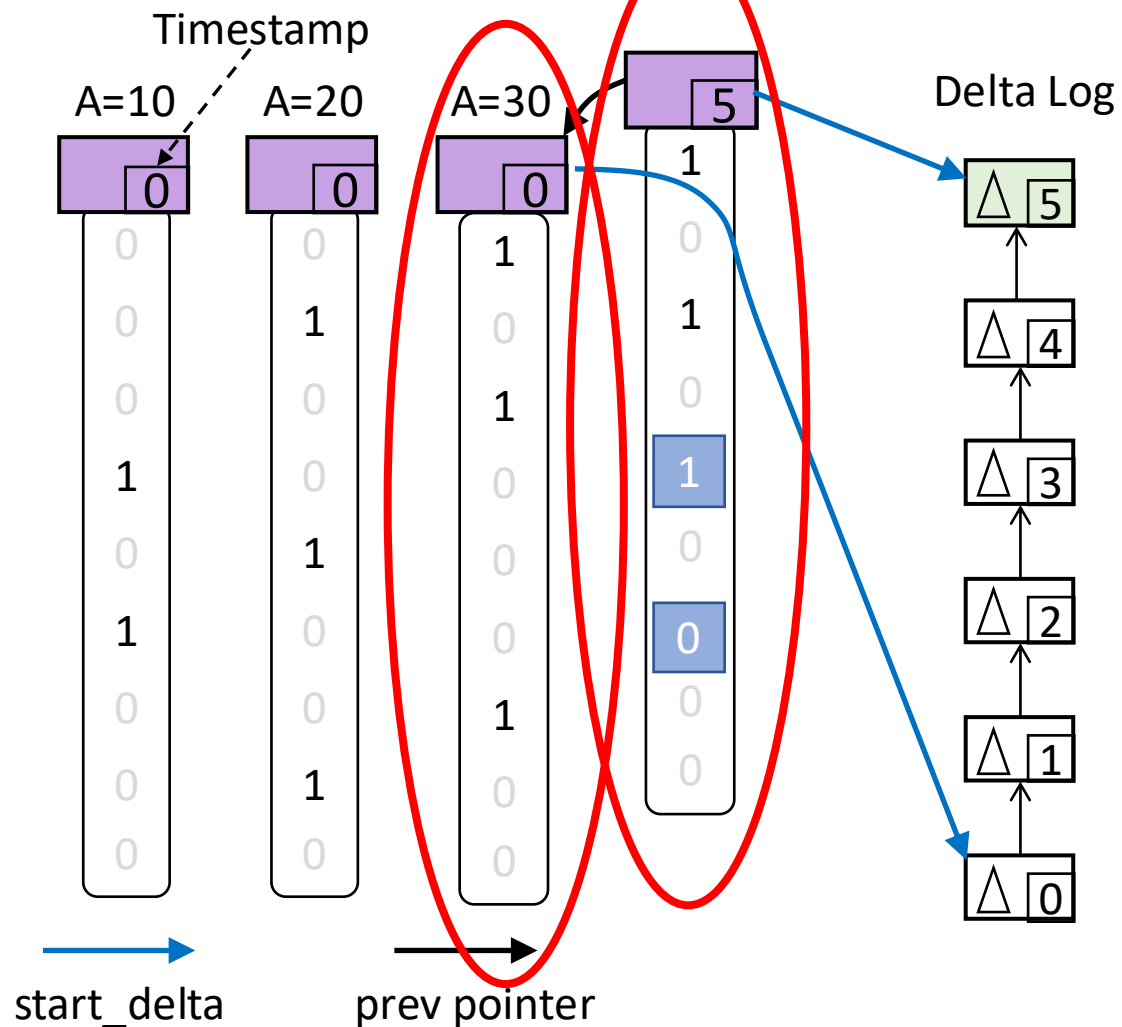
- Retrieve start-ts when started (e.g., 4)
- Fetch HUD sets from Delta Log
- Merge HUD sets into bitvectors on demand

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** → **version chain**

Each version:

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

Example: merging bitvectors on value 30

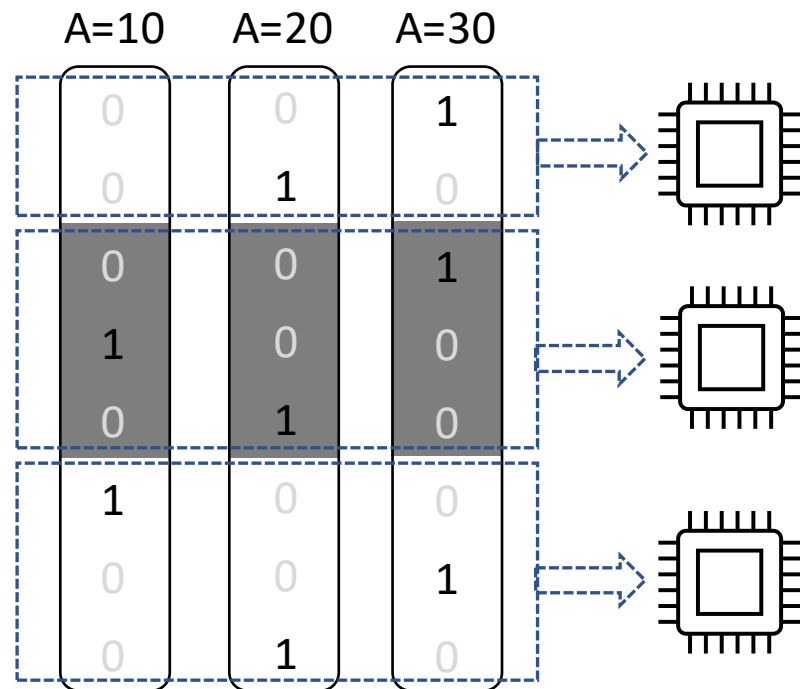
- Merge update information in HUDs into a private copy (with timestamp = 5)
- Point the new copy to a synthetic ULE, bypassing merged HUDs

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,

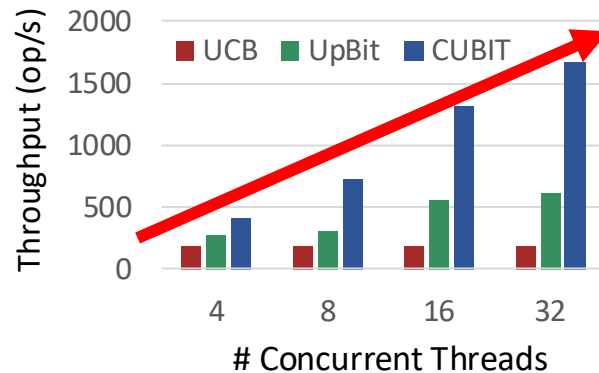
CUBIT's segmentation for parallel processing

Each CPU core merges a group of segments across bitvectors

Intermediate results reside in cache

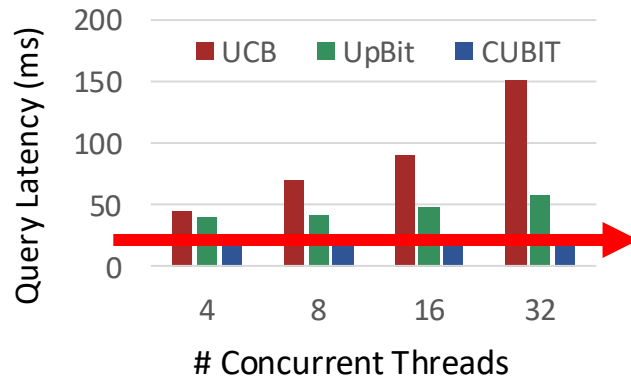
CUBIT Scales on Multicores

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



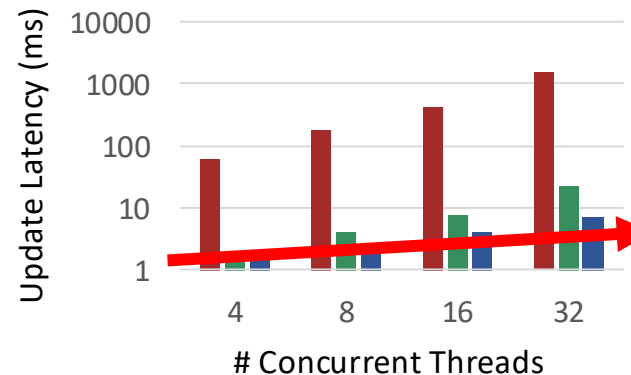
Throughput scales with #cores

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



Queries latency stays flat

CUBIT provides wait-free queries

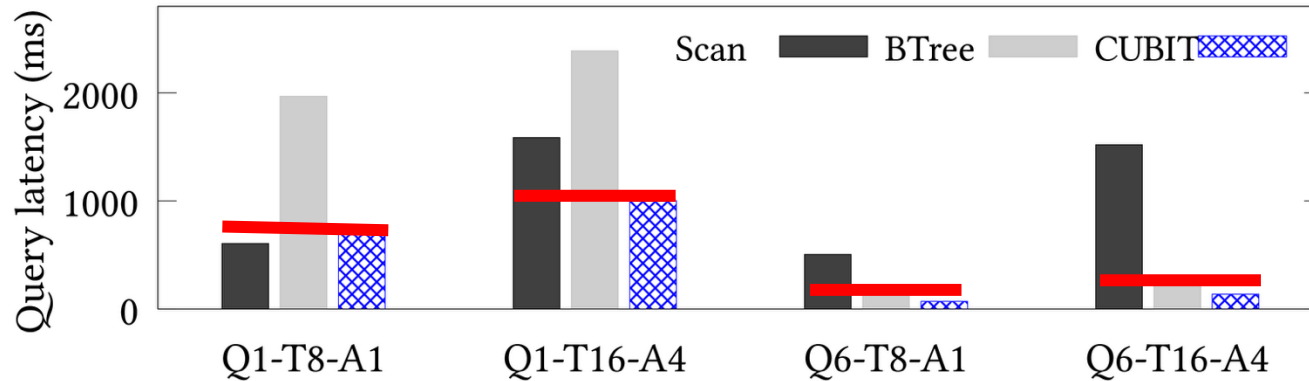


UDIs latency mostly flat

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

CUBIT benefits HTAP



DBx1000-based in house DBMS

CH-benCHmark mixing

- TPC-H analytical queries over 30M tuples
- TPC-C transactions inserting 10M new tuples during each trial

- For Q1 (high selectivity, 36%)
 - CUBIT outperforms BTree and is comparable to Scan with 1 analytical thread (Q1-T8-A1)
 - CUBIT outperforms Scan with 4 analytical threads because of few cache misses (Q1-T16-A4)
- For Q6 (low selectivity, 1.8%)
 - CUBIT outperforms both Scan and BTree at various concurrency levels

CUBIT outperforms Scan and BTree by 2x – 11x at high concurrency level

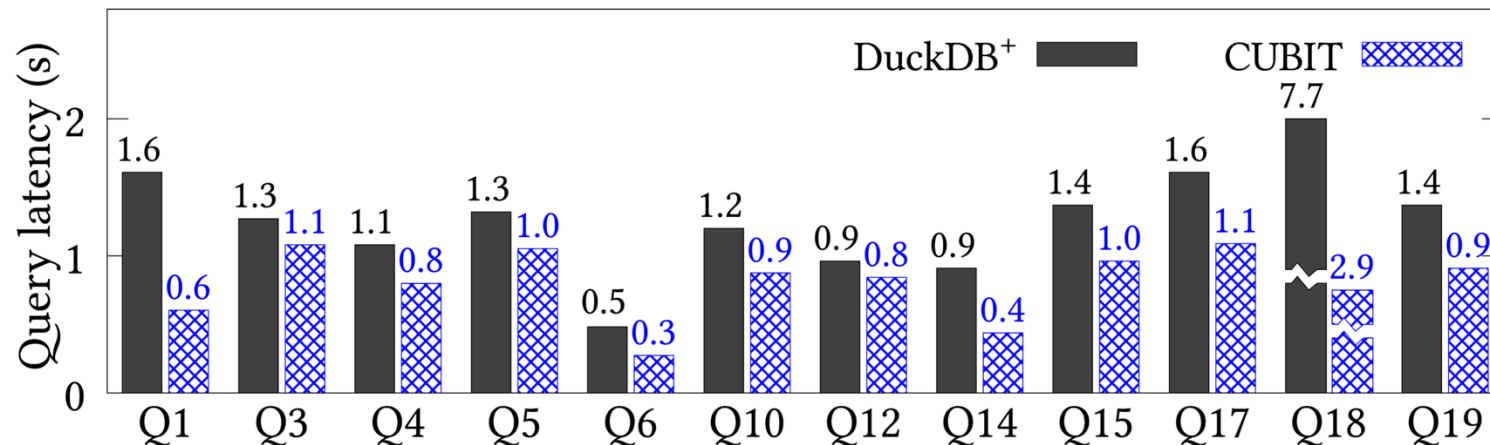
CUBIT benefits OLAP

TPC-H SF 10 dataset

Single-core execution to focus on executor logic

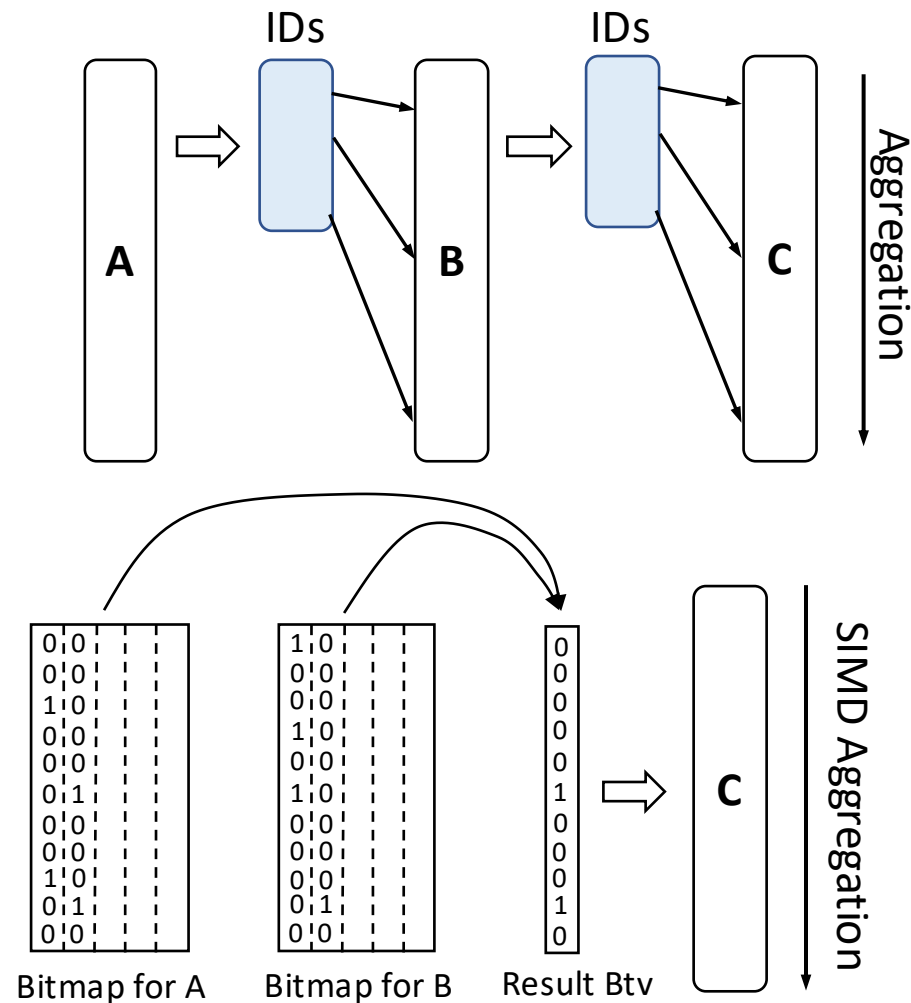
Maintain CUBIT instances on attributes in WHERE/GROUP BY clauses, including

- 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*

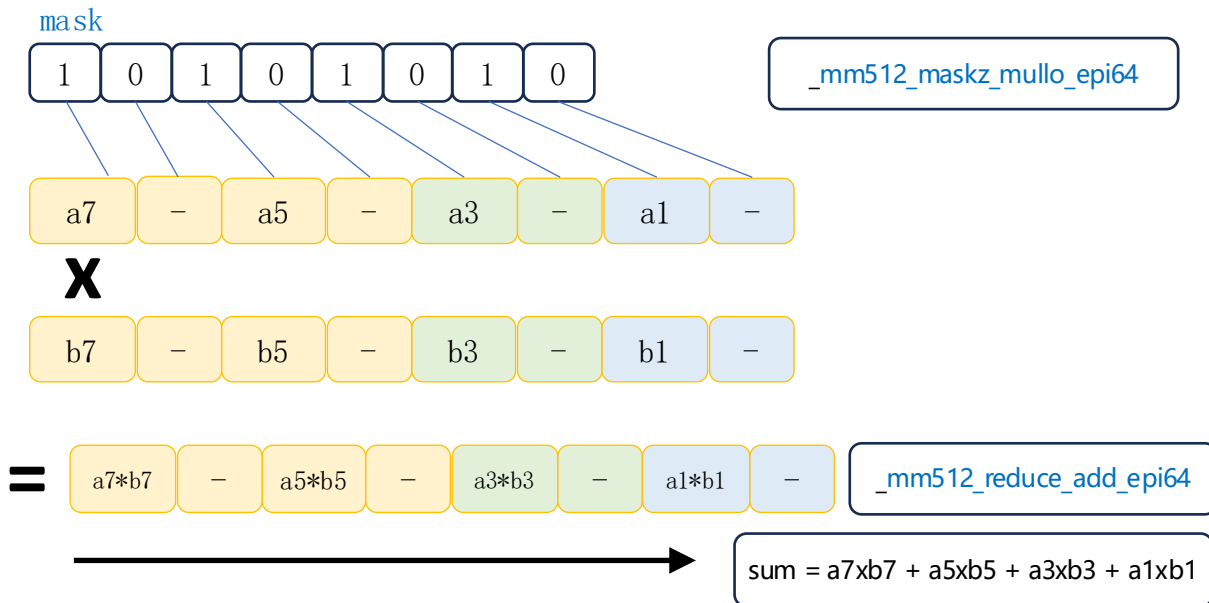
Benefit: reducing data read from storage

→ 1.6x query speedup

CUBIT-powered Scan probes less data

How to leverage SIMD?

Using Result Bitvectors - AVX-512 for Aggregation



Recent wider (512-bit) vector registers

- Available on Intel Xeon and AMD EPYC processors

Map compressed bitvector to AVX-512 masks

- 0-run: skip the corresponding rows
- 1-run: process every lane unmasked
- Literal word: split it into mask-width chunks (8 bits for 64-bit lanes) and load each chunk into a mask register

Vectorized aggregation

- Masked arithmetic per batch
- Reduce active lanes per batch

AVX-512 mask registers bridge bitmap indexing and vectorized aggregation

What about Range Queries?

What about range queries?

CUBIT focuses on point queries:

- it adopts **equality encoding** scheme
- assumes each query **reads only one bitvector**

However,

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

- **equality encoding** → many bitvector ORs for high-cardinality attributes
- **range encoding** → one bitvector

BUT: storage and update challenges

(a)

1	3
2	7
3	1
4	6
5	2
6	5
7	9
8	0
9	7
10	4
11	8
12	5

(b) Equality Encoding (EE)

	E ₀	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	E ₇	E ₈	E ₉
1	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0
3	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0	0
10	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	1	0	0	0	0

(c) Range Encoding (RE)

	R ₀	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈
1	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	0	0	1	1
3	0	1	1	1	1	1	1	1	1
4	0	0	0	0	0	0	1	1	1
5	0	0	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	1	1
7	0	0	0	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	1	1
10	0	0	0	0	1	1	1	1	1
11	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	1	1	1	1

(1) Barely compressible
(2) High update cost

Limitations with Range Encoding (RE)

is 3 <= 0? {v | s.t. v <= 5} ↓ Modified by most Inserts/Deletes

	R ₀	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈
1	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	0	0	1	1
3	0	1	1	1	1	1	1	1	1
4	0	0	0	0	0	0	1	1	1
5	0	0	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	1	1
7	0	0	0	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	1	1
10	0	0	0	0	1	1	1	1	1
11	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	1	1	1	1

$P(v, b): v \leq b$

For a tuple value v , if the domain value b of a bitvector satisfies a given predicate $P(v, b): v \leq b$, the corresponding bit of the vector is set to 1, otherwise 0.

Answer a range query by reading a bitvector

- E.g., query $A \leq 5$ is answered by reading R_5



Good for range queries

but ...

~50% 1s



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)

Efficient range queries like RE

As *space efficient* as EE

Match *update efficiency* of EE (CUBIT)

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):

- A distinct point bitvector to each value
- Same as EE bitvectors (highly compressible)

Grouping:

- Partition the domain into groups
- One cumulative bitvector for each group

Cumulative bitvectors (A_i):

- ORs 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	1	1	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

{0-3} 4 {4-7} {8-9}

(a) (b) Group Encoding (GE)

With EE: Merge a group of bitvectors

- E.g., for “X ≤ 4”, OR 5 bitvectors

With GE: Use cumulative bitvectors

- and a few point bitvectors in boundary groups
- E.g., for “X ≤ 4”, OR 2 bitvectors

$$X \leq 7 \Rightarrow A_0 \vee A_1$$

$$X \leq 8 \Rightarrow A_0 \vee A_1 \vee P_2^0$$

Significantly fewer than point bitvectors merged using EE

Leveraging cumulative bitvectors, GE provides fast range queries

Efficient updates with GE

		0				{0-3}				9 {8-9}				
		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	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 from 9 to 0 updates bits in P_0^0, A_0, P_2^1, A_2

Update cost matches EE

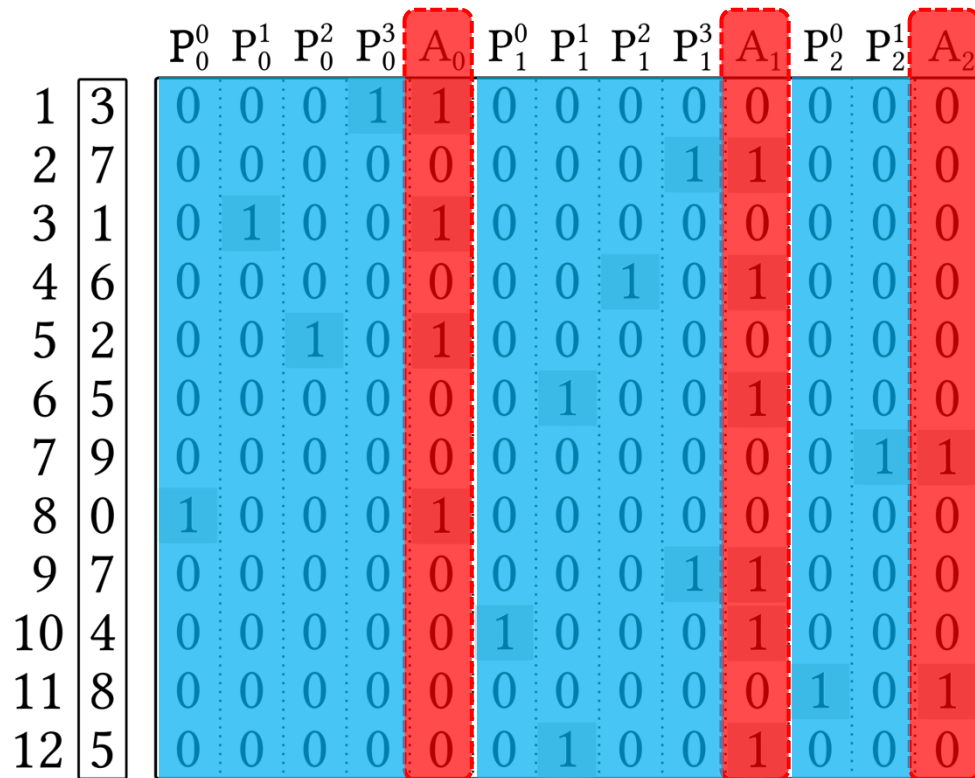
- EE updates two bits (in old and new bitvectors)

Orders of magnitude lower than RE

- RE updates on average $Cardinality/2$ bits

By avoiding RE, GE enables efficient updates

GE is Space Efficient

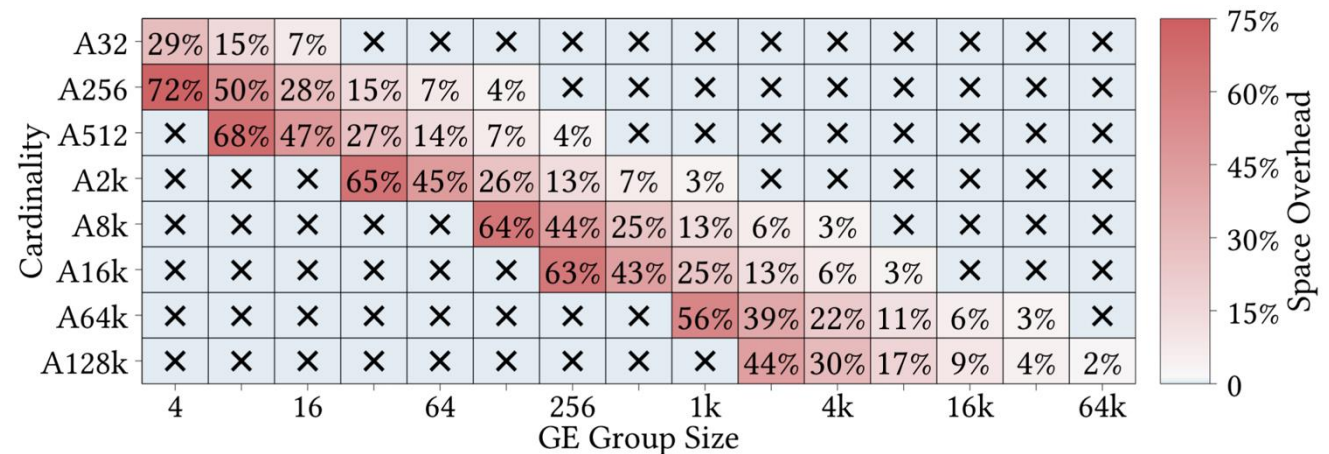


GE's space overhead over EE:

- The blue part is exactly the EE space
- # of Cumulative Bitvectors = Cardinality/Group size
- E.g., when cardinality = 16K and group size = 1K
 - Extra 25% overhead

For group size used in evaluation (C/G = 16):

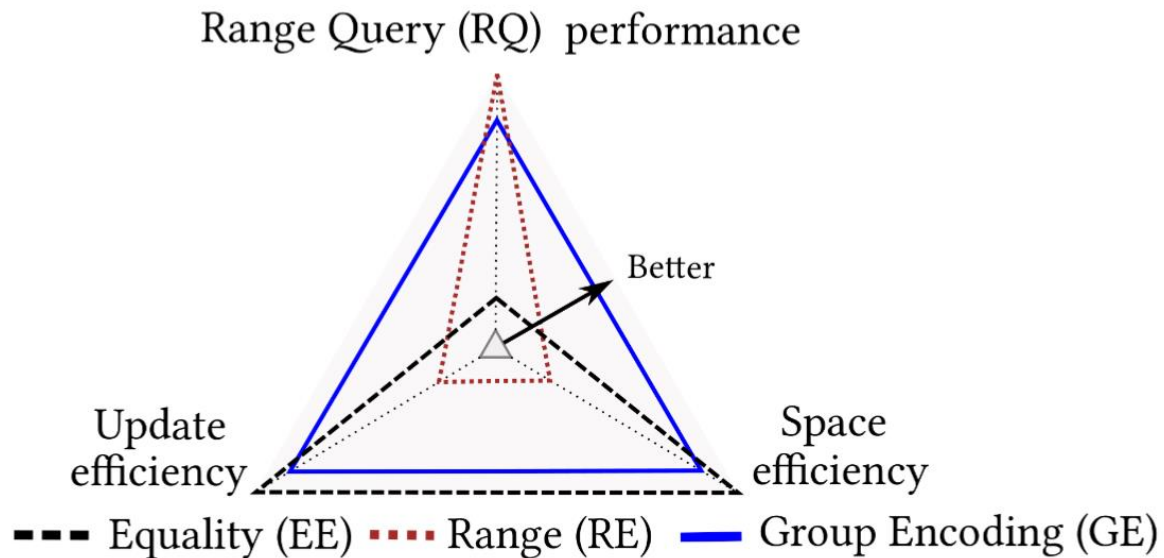
- The space overhead is ~26%



GE has similar space efficiency as EE

Goals of GE

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



Experiments

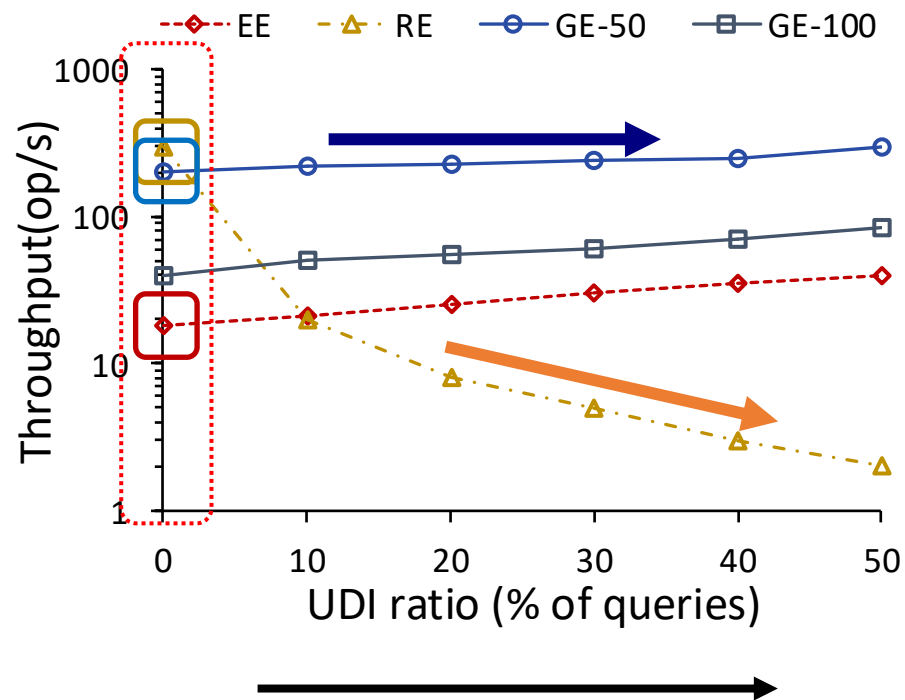
Prototype C++ implementation of UpBit (RE), CUBIT (EE), and RABIT (GE)

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)
Bitvectors are compressed using WAH
15% query range



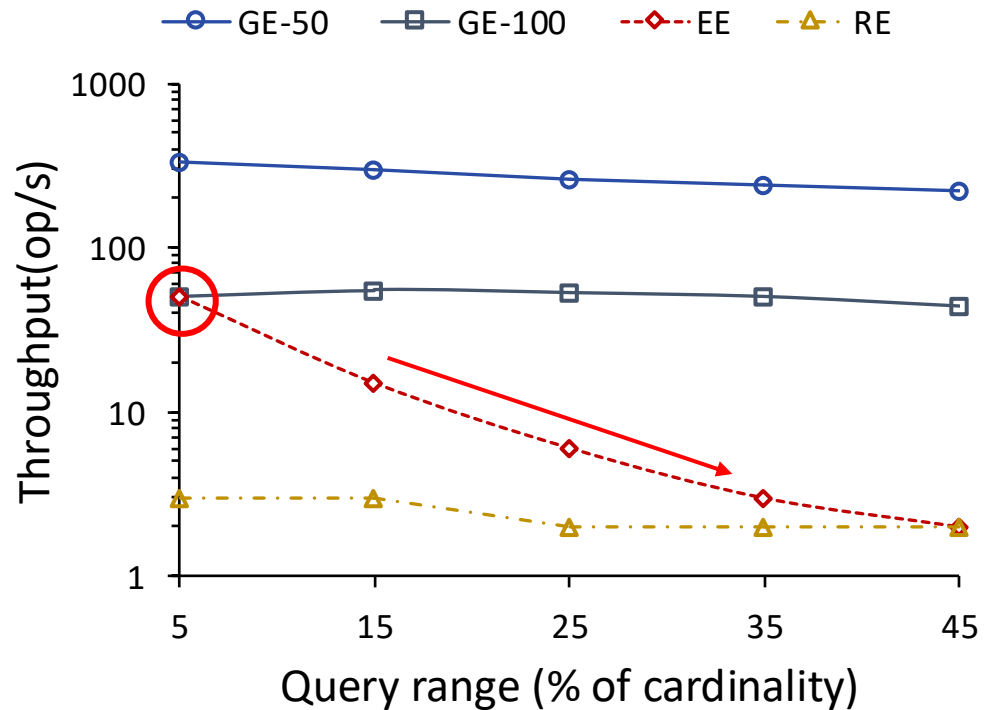
Varying UDI ratio

- For read-only workloads (0 UDIs), **RE** provides the highest performance
 - A proper Group size (50) makes **GE** (almost) as efficient as **RE**
 - **EE** is the slowest as it must OR many bitvectors
- With more UDIs
 - **GE** is the fastest & robust
 - **RE** performance drops significantly

GE is order of magnitudes faster than EE and RE for range queries in the presence of UDIs

GE is Optimal for Range Queries

100M tuples with 1000 cardinality
GE in three group size (50, 100, 200)
Bitvectors are compressed using WAH
20% UDI ratio



Varying query range

- GE is stable due to cumulative bitvectors
- For 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 has the lowest throughput due to the conflicts between UDIs and queries

GE benefits (of 1-2 OOM) remain for any query range

Proper GE tuning is important (small groups cannot benefit short range queries)

RABIT (with GE) benefits OLAP

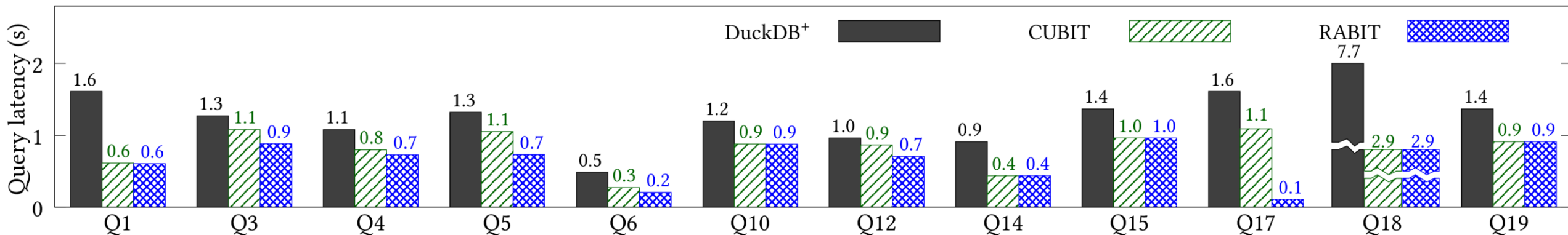
DuckDB (TPC-H SF 10 dataset)

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)



Bitmap Indexes as General-Purpose Indexes!

UPBIT enables **efficient updates** in bitmap indexes 

CUBIT enables **efficient concurrent updates** in bitmap indexes 

RABIT enables efficient bitmap indexing for **high-cardinality attributes** 

Scan ***less data*** from storage

Leverage evolving **SIMD in query processing** to boost joins and aggregation

Next open questions:

Can we build a ***bitvector-only query engine***?

Is prior **bitvector encoding** optimized for query processing?

Bitmap index processing with **GPUs**!

Thanks!

