HICAMP Bitmap

A Space-Efficient Updatable Bitmap Index for In-Memory Databases

Bo Wang, Heiner Litz, David R. Cheriton
Stanford University
DAMON'14

Database Indexing

- Databases use precomputed indexes to speed up processing
  + avoid full scan
  – compete for space with data buffering
  – maintenance cost at update

- hash table
  + fast access
  – no range query
  – inefficient for non-unique index

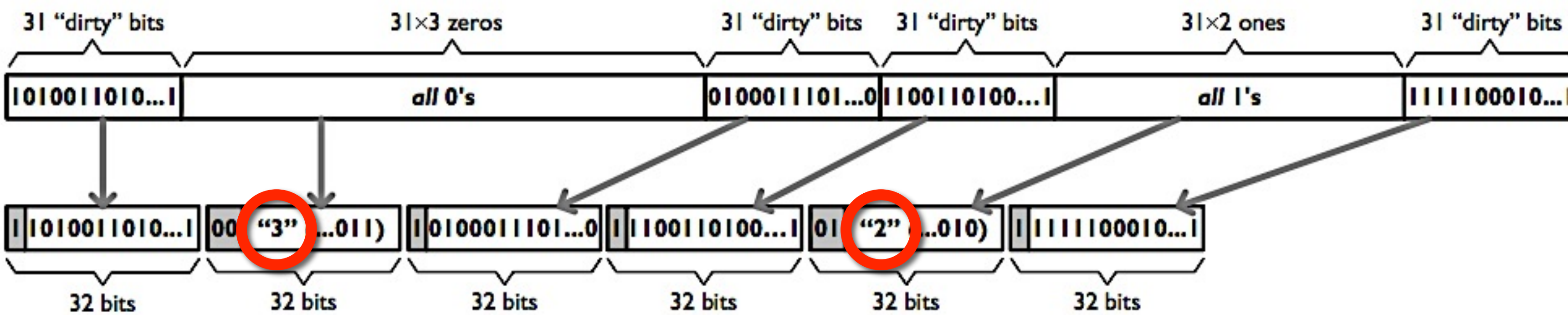
- b-tree
  + range query
  + efficient update
  – complex concurrent structural modification
  – large size (node structure, fill factor)

- bitmap
  + small size (bit-wise compression)
  + efficient for non-unique index
  – high cardinality
  – inefficient update compressed bitmap

Conflict between space cost and data manipulability

Bitmap Compression

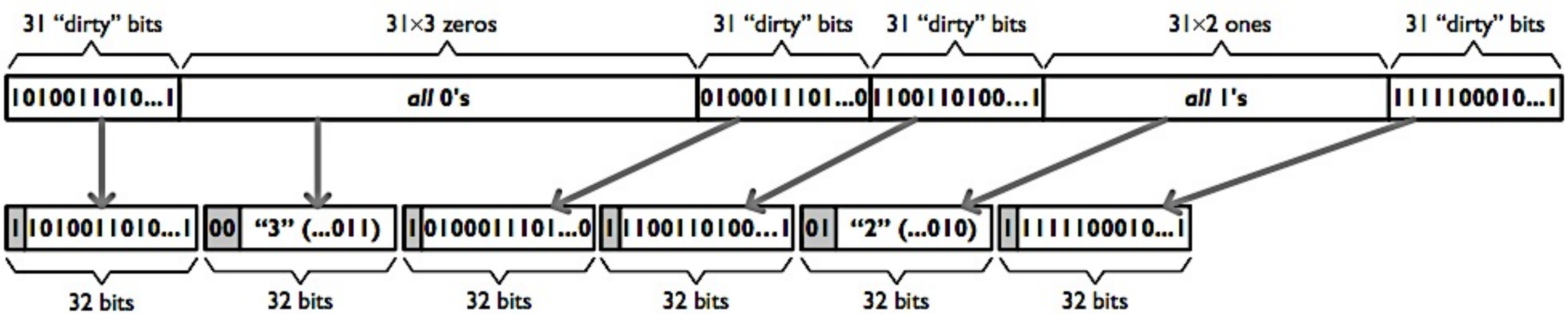
- Raw bitmap index is huge: *#rows x cardinality*
- Bitmap index is sparse: *only one non-zero per row*
  - long streams of zeros
- Run-length encoding (RLE)
  - Byte-aligned bitmap code (BBC, 1995)
  - Word-aligned hybrid code (WAH, 2003)



Source: Colantonio, Alessandro, and Roberto Di Pietro. "Concise: Compressed 'n' composable integer set." Information Processing Letters 110.16 (2010): 644-650.

Update Compressed Bitmap

- Naïve approach
  - Sequentially locate the bit to change
  - Decompress / flip / recompress
  - Possible change in memory size
- Delta structure
  - Keep changes to bitmap index in a delta structure
  - Merge by rebuilding bitmap regularly
  - Space and runtime overhead



Source: Colantonio, Alessandro, and Roberto Di Pietro. "Concise: Compressed 'n' composable integer set." Information Processing Letters 110.16 (2010): 644-650.

Updating a compressed bitmap index is inefficient

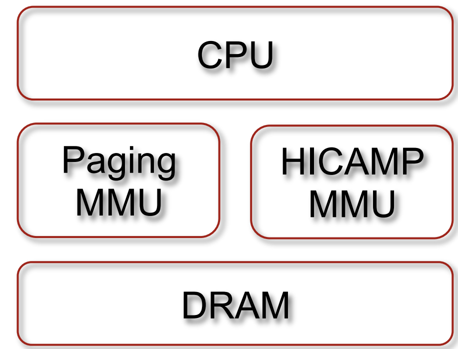
Can a *compressed* bitmap index be *updated* efficiently?

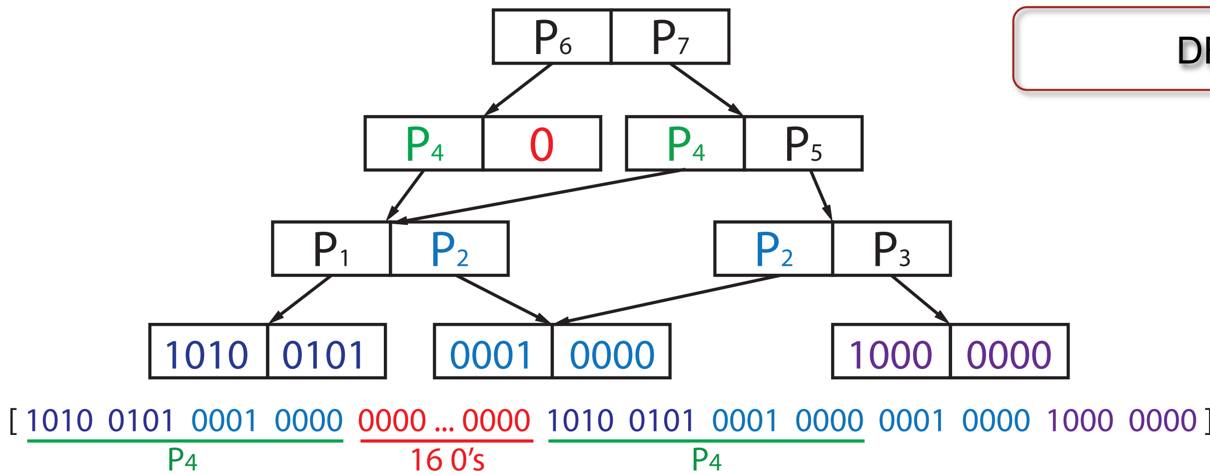
Yes, with HICAMP bitmap index

HICAMP Memory

HICAMP[1, 2] is a new *memory management unit* (MMU) which manages data as a *directed acyclic graph* (DAG) of fixed-width lines (e.g. 64B)

- Same content is stored only once
- Deduplicate with pointer references
- Zero lines are referred by zero pointers
- Hierarchical deduplication



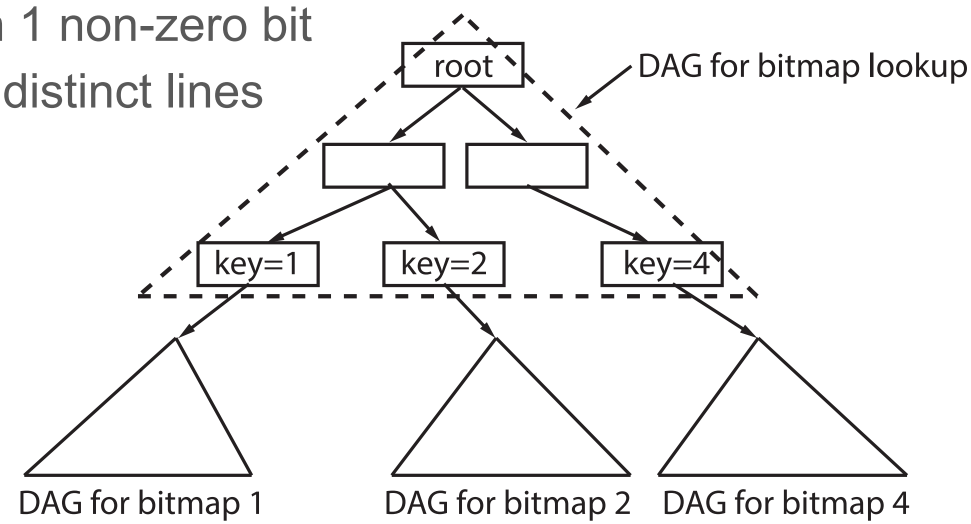


Deduplicate rather than compress data in hardware

[1] David Cheriton, et. al. HICAMP: architectural support for efficient concurrency-safe shared structured data access. ASPLOS'12
[2] HICAMP Systems, Inc. www.hicampsystems.com

HICAMP Bitmap Index

- Two-level structure
  - each bitmap is stored as a separate HICAMP DAG
  - a DAG (indexed by key) to lookup bitmaps
- Deduplication
  - a 64B line indexes 512 records
  - a pointer reference takes 4B, i.e. 16 references per line
    - it takes only 4B to dedup a 64B line
  - bitmap index is sparse. #unique lines is small
    - only 512 distinct lines with 1 non-zero bit
    - less than 8MB to store all distinct lines with 2 non-zero bits



Lookup / Update on HICAMP Bitmap

- Lookup operation
  - to lookup *i*-th bit in the bitmap
  - calculate *leaf id* and *offset in leaf*
  - traverse DAG using *leaf id* as the key in hardware
  - locate the *i*-th bit with *offset in leaf* in software

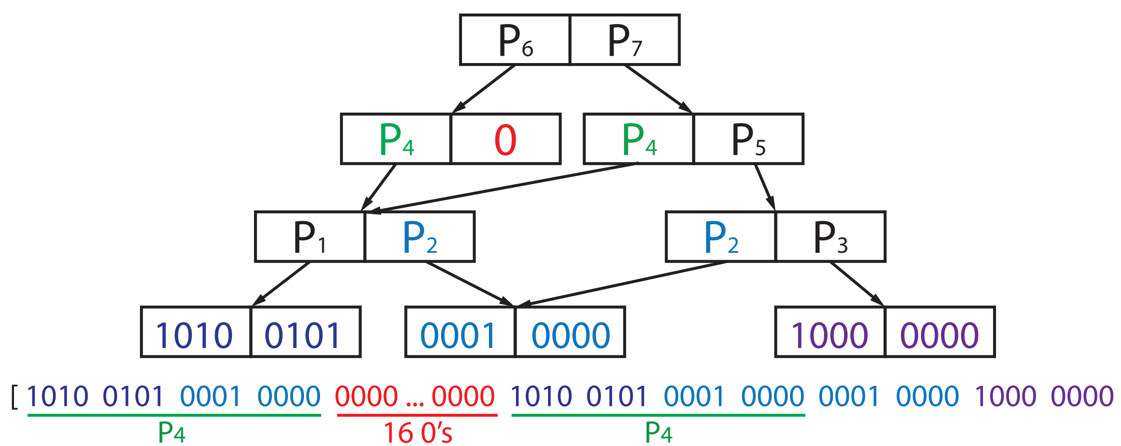
- Lookup complexity
  - O(log *n*), *n* is the size of bitmap

- Update operation
  - lookup the corresponding bit and flip it
  - deduplication is handled by HICAMP MMU (lookup by content)

Compact bitmap format preserves regular layout for efficient update

Scan on HICAMP Bitmap

- Scan operation
  - skip zero lines with DAG structure
  - find next non-zero leaf in hardware
  - find next non-zero bits in a leaf in software
  - DAG-aware prefetch in HICAMP MMU



- Complexity
  - O(m log n), m is #non-zero lines, n is size of bitmap

Efficient scan operation with SW / HW collaboration

How to deal with *curse of dimensionality*?

- Space overhead of a large number of bitmaps
- Runtime overhead on scanning many bitmaps for a range query
- Common approach
  - binning + candidate check
  - but, candidate check is not cheap (branch + cache miss)

Multi-bit Bitmap Index

- Encode a record with *n* bits (signature) rather than one
  - bin_width = $2^n - 1$
  - bin_id = value / bin_width
  - signature = value % bin_width
- Merge $2^n - 1$ bins into one (similar to bitmap binning)
- Use signatures to reduce candidate checking

- Example: 4-bit bitmap index
  - bin_width = $2^4 - 1 = 15$
  - value 50
    - bin_id = 50/15 = 3
    - sigature = 50%15 = 5

data array

	bin range	50	10	15	20	35	31	4	46
bin[0]	1 ~ 15	0000	1010	1111	0000	0000	0000	0100	0000
bin[1]	16 ~ 30	0000	0000	0000	0101	0000	0000	0000	0000
bin[2]	31 ~ 45	0000	0000	0000	0000	0101	0001	0000	0000
bin[3]	46 ~ 60	0101	0000	0000	0000	0000	0000	0000	0001

Make binning favorable to both equality and range queries

Compaction Results on TPC-H

- Experiment Setup
  - Simulate HICAMP memory on top of ZSim, an instruction-driven architectural simulator
  - Evaluate on selected columns from TPC-H, 50 million rows per column
- 2 ~ 250x smaller than B+tree
- 3 ~ 650x smaller than other commonly used structures (RB-tree etc.)
- Similar memory consumption as software compressed bitmap

Cardinality	Column name	B+Tree (d=128)	B+Tree (d=1024)	AVL Tree	Red-Black Tree	Skip List	WAH	HICAMP Bitmap
7	line number	25	24	64	64	53	0.9	1.7
50	quantity	25	24	64	64	53	4.4	1.2
2526	ship date	25	24	64	64	53	1.7e-3	0.09
100000	supplier key	23	19	64	64	53	6.8	12.7‡

† unit: bytes/record ‡ indexed with 8-bit bitmaps

Conclusions

- Demonstrated how hardware innovation breaks the conflict between space cost and data manipulation plagued by compression
- With HICAMP memory, bitmap index can be both space-efficient and update-friendly
  - A good fit for OLTP and OLAP at same time
- Multibit bitmap alleviates the high cardinality problem and the need for candidate checking

Thanks to

Michael Chan

Amin Firoozshahian

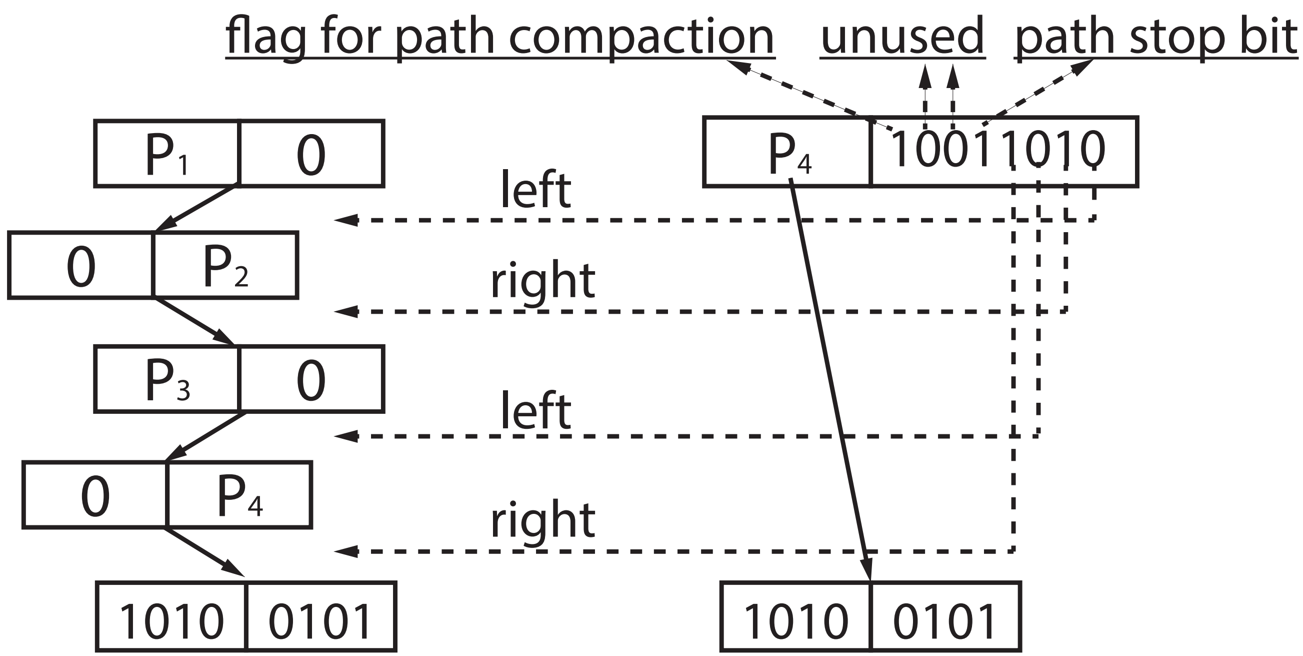
Christopher Ré

Alex Solomatnikov

Questions?

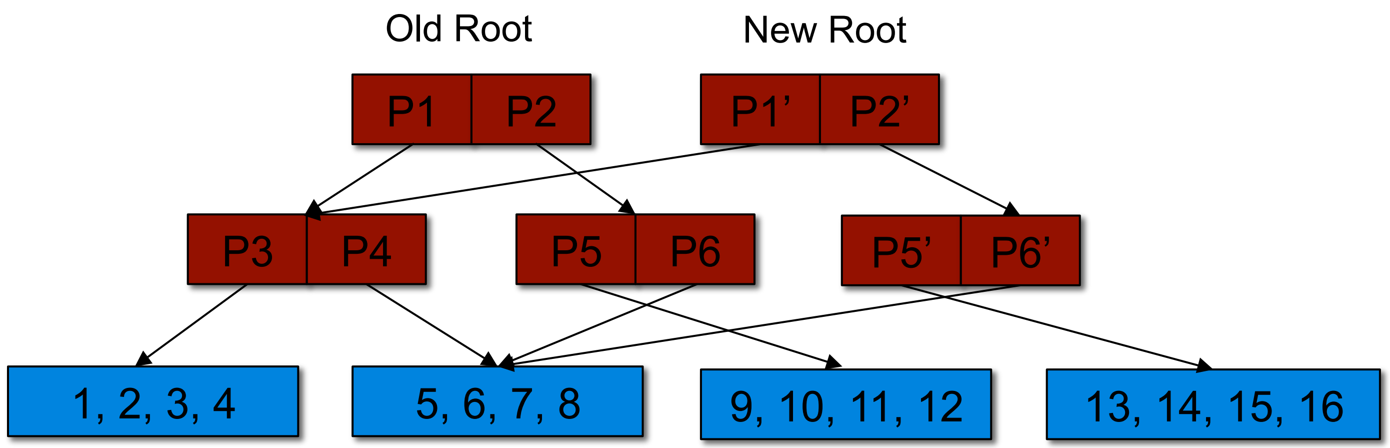
Backup Slides

Path Compaction



Copy-on-Write

- HICAMP copy-on-write
  - Writes are not executed in-place
  - Instead, a new copy is created
- Each transaction generates a new snapshot at low cost
- Old versions are automatically released once the reference counts reach zero



Change array {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 5, 6, 7, 8}
to {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16} in HICAMP

Compaction Results on Uniform/Zipf Distribution

- Evaluate multibit bitmap on uniform and zipf distributions with different cardinalities
  - 3 ~ 12x smaller than B+tree
  - 8 ~ 30x smaller than AVL tree, RB tree and skiplist
  - higher compaction ratio under zipf distribution due to concentration of non-zero appearances
  - sizes of tree-based indexing structures almost don't change

Cardinality	B+Tree	AVL/RB	Skiplist	WAH	Multibit
unif 10	25	64	53	1.2	2.0
unif 100	25	64	53	5.7	7.0
unif 1000	25	64	53	7.4	8.0
zipf 10	25	64	53	0.9	1.9
zipf 100	25	64	53	1.2	3.0
zipf 1000	25	64	53	1.3	2.4

Table 2: Memory consumption on uniform/zipf dist.