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

Stanford University
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)

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

Updating a compressed bitmap index is inefficient
Can a *compressed* bitmap index be *updated* efficiently?

Yes, with HICAMP bitmap index
HICAMP Memory

HICAMP\cite{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

\[ \text{P} \]
\[ \text{1010 0101 0001 0000 1000 0000 ... 0000} \]

\[ \text{P} \]
\[ \text{0 0001 0000} \]

\[ \text{P} \]
\[ \text{P4 P5 P6 P7} \]

\section*{Deduplicate rather than compress data in hardware}

\cite{1} David Cheriton, et. al. HICAMP: architectural support for efficient concurrency-safe shared structured data access. ASPLOS’12
\cite{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 \textit{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
  - $\text{bin\_width} = 2^n - 1$
  - $\text{bin\_id} = \text{value} / \text{bin\_width}$
  - $\text{signature} = \text{value} \% \text{bin\_width}$
- Merge $2^n - 1$ bins into one (similar to bitmap binning)
- Use signatures to reduce candidate checking

Example: 4-bit bitmap index
- $\text{bin\_width} = 2^4 - 1 = 15$
- value 50
  - $\text{bin\_id} = 50/15 = 3$
  - $\text{signature} = 50\% 15 = 5$

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

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

† 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

flag for path compaction  unused  path stop bit

P_1  0
0  P_2
P_3  0
0  P_4
1010  0101

1010  0101
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

<table>
<thead>
<tr>
<th>Cardinality</th>
<th>B+Tree</th>
<th>AVL/RB</th>
<th>Skiplist</th>
<th>WAH</th>
<th>Multibit</th>
</tr>
</thead>
<tbody>
<tr>
<td>unif 10</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>1.2</td>
<td>2.0</td>
</tr>
<tr>
<td>unif 100</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>5.7</td>
<td>7.0</td>
</tr>
<tr>
<td>unif 1000</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>7.4</td>
<td>8.0</td>
</tr>
<tr>
<td>zipf 10</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>0.9</td>
<td>1.9</td>
</tr>
<tr>
<td>zipf 100</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>1.2</td>
<td>3.0</td>
</tr>
<tr>
<td>zipf 1000</td>
<td>25</td>
<td>64</td>
<td>53</td>
<td>1.3</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Table 2: Memory consumption on uniform/zipf dist.