UpBit: Scalable In-Memory Updatable Bitmap Indexing

Guanting Chen, Yuan Zhang
2/21/2019
BACKGROUND
Some words to know

Bitmap: a bitmap is a mapping from some domain to bits

Bitmap Index: A bitmap index is a special kind of database index that uses bitmaps
### Bitmap Index

<table>
<thead>
<tr>
<th>Column A</th>
<th>A=10</th>
<th>A=20</th>
<th>A=30</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Typically a single bit per row
- One bitvector per value
- Advantage: Fast read for equality and range queries.
- Need to be compressed for space-efficient
Keep Bitvectors Small

- Bitvectors contain redundancy
- Reduce redundancy
- And improve read performance
- So we have compression and encoding!
How to compress/encode?

One way:

- **raw bitvector**
  - `0`
  - `0`
  - `0`
  - `0`
  - `1`
  - `0`

- **encode**
  - `0`
  - `1`
  - `0`
  - `0`
  - `1`
  - `1`

  **Update?**
  - `0`
  - `0`
  - `0`
  - `0`
  - `1`
  - `0`

- **decode**
  - `0`
  - `1`
  - `0`
  - `0`
  - `1`

- **flip bit**
  - `0`
  - `1`
  - `0`
  - `0`
  - `1`
  - `1`

- **re-encode**
  - `0`
  - `1`
  - `1`
  - `1`
  - `1`

  **encoded bitvector**
  - `0`
  - `1`
  - `0`
  - `0`
  - `1`
  - `1`

  **13 zeros**
  - `0`
  - `0`
  - `0`
  - `0`

  **ending pattern**
  - `0`

  **10 zeros**
  - `1`
  - `1`
  - `1`
  - `1`

  **ending pattern**
THE PROBLEM
### Traditional Bitmap Index

- **Read-optimized**
- **Bitvectors are encoded**

<table>
<thead>
<tr>
<th>Base Data</th>
<th>Bitmap Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>rid</td>
<td>Column A</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
</tbody>
</table>
Traditional Bitmap Index

What if we want to update?

- Costly decoding whole bitvectors
- Re-encoding updated bitvectors

Do not offer efficient updates!!!
To Solve The Problem

Bitmap indexing should deliver both:

● Good READ performance

● Efficient UPDATE!!!
POSSIBLE SOLUTION
UCB: Update Conscious Bitmap

- State-of-the-art update-optimized
- Using existence bitvector (EB): indicate bits are valid or not
## UCB Advantages

- Efficient deletes by invalidation

<table>
<thead>
<tr>
<th>A=10</th>
<th>A=20</th>
<th>A=30</th>
<th>EB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
UCB Advantages

- Faster updates by deleting then appending

```
<table>
<thead>
<tr>
<th>rid</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>EB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Pad</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>rid</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>EB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Inv</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
### UCB Read

<table>
<thead>
<tr>
<th>rid</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>EB</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Inv</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- **probe A=20**

- Read: bitwise AND with EB

(b) Probe for value B using UCB.
UCB Limitations

- More updates, read becomes more expensive
Why?

- Repetitive bitwise operations
- Single auxiliary bitvector
WE STILL HAVE THE PROBLEM!
The Bitmap Index Should:

- Distribute update cost
- Efficiently access compressed bitvectors
- Re-use results of bitwise operations
A BETTER SOLUTION
UpBit: Updatable Bitmap Indexing
Offer efficient updates without hurting read performance!!!
Maintain update bitvectors (UB):

- One per value
- Initialized to 0s
- Every update flips on a bit on UB
- Double the amount of uncompressed data
- Sparse, compressed size is small (only small ones)
UpBit: Update

Three steps to update:

1. Find old value of row 2 (20)
2. Flip bit of row 2 of UB of 20
3. Flip bit of row 2 of UB of 10
UpBit VS UCB: Update

- No single bitvector (EB) receives all updates
- Distribute the update burden to multiple UB
Will be faster if we speed up step 1!

How?
UpBit: Retrieve Value Of A Row

Using fence pointer:
- Avoid decoding entire bitvector
- Decode only a small part of the bitvector
- Efficiently retrieve a value
(c) Search for a single value with UpBit.

Return the XOR of VB and UB
Can we re-use the result?

How?
UpBit: Merge

Why merge?

- Accumulated operations lead to less compressible UB
- More expensive bitwise operations and decoding
- Need to maintain high compressibility of UB

How and When to merge?
UpBit: Merge

Merge periodically:
- Maintain a threshold based on # updates

Query-driven merge:
- “query then merge”
- Use the result of XOR
- Update VB using the result
- Set UB to zeros
<table>
<thead>
<tr>
<th>UpBit VS UCB: Read</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UCB:</strong></td>
</tr>
<tr>
<td>● Bitwise AND between VB and EB</td>
</tr>
<tr>
<td>● Decode and encode entire bitvector</td>
</tr>
<tr>
<td>● No merge (merge EB means merge with all VB)</td>
</tr>
<tr>
<td>● Read performance does not scale with updates</td>
</tr>
</tbody>
</table>
BENCHMARKING
Scalability

- 100M values of real life data set
- 100 unique values of domain
- 100k operations of query mix
Update & Read Performance

- Fence pointer enables fast read & update
- UpBit has 8% read overhead at most due to XOR operations
UpBit vs Scan

- Compared with a fast scan, UpBit is faster for range queries with up to 1% selectivity.
Design element - Fence Pointer

UpBit-FP: Using fence pointer and only **ONE** update bitvector (like UCB’s EB)

- Fence pointer alone - **NOT** maintaining **COMPRESSIBILITY**
- Updates bitvectors needs fence pointer to amortize their cost
Design element - Merging Threshold

- Threshold of 10 updates (bits set to 1) leads to fastest workload execution

20% updates workload

50% updates workload

overall comparison
Further improvement - Parallelism

- Each pair of VB & UB are actually decoupled and domain-isolated
- Which means they can be queried in parallel

<table>
<thead>
<tr>
<th>Base Data</th>
<th>UpBit Index</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>A in [10, 20]</th>
</tr>
</thead>
<tbody>
<tr>
<td>rid Column A</td>
<td>rid 10 VB UB 20 VB UB 30 VB UB</td>
<td>rid 10 UB</td>
<td>20 VB UB</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 30</td>
<td>1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 20</td>
<td>2 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 30</td>
<td>3 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 10</td>
<td>4 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 20</td>
<td>5 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 10</td>
<td>6 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 30</td>
<td>7 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 20</td>
<td>8 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Probe (A in [10, 20])
Further improvement - Parallelism

- Performance UpBit(Update) does not scale with the number of threads after 8
CONCLUSION
Design goals

- Multiple Update Vectors → distribute update cost → maintain more compressible UBs
- Fence pointer → partial decoding → efficient retrieval of a value at arbitrary position
- Query-driven UB merging → keep maintaining high compressibility
Pros

- **On the surface:**
  - Straightforward design idea, clear illustration (easy to understand)

- **Underneath:**
  - Interesting details and effects (the way it distributes update cost and maintains compressibility is really cool!)
  - Concrete pseudocode (better understanding of logical implementation underneath the design)
Cons

- Does not cover the materialization of BitMap index
- What if Read-Optimized BitMap index also employs fence pointer?
- What about different cardinality?
THANK YOU