UpBit: Scalable In-Memory Updatable Bitmap Indexing

Manos Athanassoulis\textsuperscript{1} \quad Zheng Yan\textsuperscript{2} \quad Stratos Idreos\textsuperscript{1}

\textsuperscript{1}Harvard University \quad \textsuperscript{2}University of Maryland
## Indexing for Analytical Workloads

<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>

### Specialized indexing
- Compact representation of query result
- Query result is readily available

### Bitvectors
- Can leverage fast Boolean operators
- Bitwise AND/OR/NOT faster than looping over meta data
Bitmap Indexing Limitations

Index Size

- **Space-inefficient for large domains**
- **Addressed by bitvector encoding/compression**

**core idea:** run-length encoding in prior work

**but ...**

- **Updating encoded bitvectors is very inefficient**
Update?

encode

13 zeros

ending pattern

decode

flip bit

re-encode

10 zeros

ending pattern
Goal

Bitmap Indexing with efficient Reads & Updates
**Prior Work:** Bitmap Indexing and Deletes

Update Conscious Bitmaps (UCB), SSDBM 2007

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

Efficient deletes by invalidation existence bitvector (EB)
Prior Work: Bitmap Indexing and Deletes

**Update Conscious Bitmaps (UCB), SSDBM 2007**

<table>
<thead>
<tr>
<th></th>
<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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Efficient deletes by invalidation existence bitvector (EB)

**Reads?**
- Bitwise AND with EB

**Updates?**
- Delete-then-append

```plaintext
A=10    A=20    A=30    EB  
0       0       1       1   
0       0       1       1   
1       0       0       1   
0       1       0       1   
0       0       1       1   
0       1       0       1   
0       0       1       1   
0       1       0       1   

A=20    EB   
0       1   
1       0   
0       1   
1       1   
0       1   
0       1   
1       1   
1       1   
```

7
Prior Work: Limitations

n=100M tuples, d=100 domain values, 50% updates / 50% reads

- read cost increases with #updates
  - why?
  - bitwise AND with EB is the bottleneck
  - update EB is costly for >> #updates

UCB performance does not scale with #updates

- single auxiliary bitvector
- repetitive bitwise operations
Bitmap Indexing for Reads & Updates

distribute update cost

efficient random accesses in compressed bitvectors

query-driven re-use results of bitwise operations
Design Element 1: update bitvectors

one per value of the domain initialized to 0s

the current value is the XOR

every update flips a bit on UB
Design Element 1: update bitvectors

one per value of the domain initialized to 0s

the current value is the XOR

every update flips a bit on UB

... distribute the update burden
Updating UpBit ...

... row 2 to 10

<table>
<thead>
<tr>
<th></th>
<th>A=10</th>
<th>UB</th>
<th>A=20</th>
<th>UB</th>
<th>A=30</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Updating UpBit...

... row 2 to 10

1. find old value of row 2 (A=20)
Updating UpBit ...

... row 2 to 10

1. find old value of row 2 (A=20)

<table>
<thead>
<tr>
<th>A=10</th>
<th>UB</th>
<th>A=20</th>
<th>UB</th>
<th>A=30</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Updating UpBit ...

... row 2 to 10

1. find old value of row 2 (A=20)
2. flip bit of row 2 of UB of A=20
Updating UpBit ...

... row 2 to 10
1. find old value of row 2 (A=20)
2. flip bit of row 2 of UB of A=20
3. flip bit of row 2 of UB of A=10

can we speed up step 1?

<table>
<thead>
<tr>
<th>A=10</th>
<th>UB</th>
<th>A=20</th>
<th>UB</th>
<th>A=30</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Design Element 2: fence pointers

efficient access of compressed bitvectors
fence pointers
Updating UpBit ...

... row 2 to 10

1. find old value of row 2 (A=20)

<table>
<thead>
<tr>
<th>A=10</th>
<th>UB</th>
<th>A=20</th>
<th>UB</th>
<th>A=30</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Updating UpBit (with fence pointers)...

... row 2 to 10

1. find old value of row 2 (A=20) using fence pointers
Querying
Querying UpBit ...

... A = 20

Return the XOR of A=20 and UB
Querying UpBit ...

... A = 20

Return the XOR of A=20 and UB

can we re-use the result?
Design Element 3: query-driven merging

maintain high compressibility of UB query-driven merging
UpBit supports very efficient updates

- n=100M tuples, d=100 domain values
- 100k queries (varying % of updates)

**Updates:**
- 15-29x faster than UCB
- 51-115x faster than in-place

**Reads:**
- Only 8% read overhead over optimal
- 3x faster reads than UCB
UpBit offers robust reads

n=100M tuples, d=100 domain values
50%/50% update/read queries

updates: 15-29x faster than UCB — only 8% read overhead over optimal
51-115x faster than in-place
3x faster reads than UCB
More in the paper ...

**Tuning:** how frequent to merge UB to the index?

**Tuning:** what is the optimal granularity of fence pointers?

**Optimizations:** multi-threaded reads and updates

**Performance:** full query analysis (scientific data and TPCH)
UpBit: achieving scalable updates

- distribute the update burden
- update bitvectors
- efficient bitvector accesses
- fence pointers
- avoid redundant bitwise operations
- query-driven merging of UB

Thanks!

http://daslab.seas.harvard.edu/rum/