UpBit: Scalable In-Memory Updatable Bitmap Indexing

Manos Athanassoulis  Zheng Yan*  Stratos Idreos

* during an internship at DASlab
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 domains with large cardinality
- Addressed by bitvector encoding/compression

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

- Updating encoded bitvectors is very inefficient
Update?

encode

raw bitvector

13 zeros

encoded bitvector

decode

flip bit

re-encode

10 zeros

ending pattern

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

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

Efficient deletes by invalidation existence bitvector (EB)

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

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

Reads?
- bitwise AND with EB

Updates?
- delete-then-append
**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

A=10

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
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 5 to 10

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</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>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Updating UpBit ...

... row 5 to 10

1. find old value of row 5 (A=20)
Updating UpBit ...

... row 5 to 10

1. find old value of row 5 (A=20)
Updating UpBit ...

... row 5 to 10
1. find old value of row 5 (A=20)
2. flip bit of row 5 of UB of A=20
Updating UpBit ...

... row 5 to 10
1. find old value of row 5 (A=20)
2. flip bit of row 5 of UB of A=20
3. flip bit of row 5 of UB of A=10

A=10  A=20  A=30
UB   UB   UB

0  0  1
0  0  0
0  1  1
1  0  0
1  0  1
0  0  0
0  0  0
0  0  0

can we speed up step 1?
Design Element 2: fence pointers

efficient access of compressed bitvectors

fence pointers

we can find row 5 without decoding & scanning the whole bitvector
Updating UpBit ...

... row 5 to 10
1. find old value of row 5 (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>0</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>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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

UB
Updating UpBit (with fence pointers)...

... row 2 to 10

1. find old value of row 2 (A=20) using fence pointers
How dense should the fence pointers be?

fence pointers every $10^5$ entries
Querying
Querying UpBit ...

... A = 20

Return the XOR of A=20 and UB

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

... A = 20

Return the XOR of A=20 and UB

<table>
<thead>
<tr>
<th>A=20</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

maintain high compressibility of UB
query-driven merging
How frequently to merge UB back to VB?

Normalized Performance Benefit

merge back when 20 updates have been recorded
**UpBit**

update bitvectors

fence pointers

query-driven UB merging

**Goals**

Faster updates

Faster reads (than update-optimized)

Fast overall latency

**Experiments**

Synthetic data

- \( n \): # tuples
- \( d \): # domain values (cardinality)
- \( q \): # queries
- \( u \): % updates in the workload

TPC-H SF 100 data

Prototype C++ Implementation of UpBit using FastBit

Integrated in a prototype column-store system
UpBit supports very efficient updates

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

- 15-29x faster than UCB
- 51-115x faster than in-place
- 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

scalable read latency
UpBit as a general index: UpBit vs. Scan

\[ n = 1\, \text{B}, \quad d = 1000\, \text{distinct domain values (range)} \]
\[ n = 1\, \text{B}, \quad d \text{ varies for equality: 1000, 100, 10, 1} \]

- **Scan**
  - tight for-loop
  - SIMD
  - multi-core

- **Equality query**
  - all qualifying tuples have the same value
  - (always 1 bitvector)

- **Range query**
  - qualifying tuples may have many values
  - (bit-OR bitvectors)

**Latency (ms)**

**Selectivity**

- break-even: 10% for equality queries and 1% for range queries
UpBit as a general index: UpBit vs. Scan TPCH Q6

SELECT sum(l_extendedprice * l_discount) as revenue
FROM lineitem
WHERE l_shipdate >= date '[DATE]' 
AND l_shipdate < date '[DATE]' + interval '1' year

an update-aware bitmap index is a viable general-purpose index
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/