UpBit: Scalable In-Memory Updatable Bitmap Indexing

Contents

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

- Point query
- Range query
- Update
- Append
- Delete

WHERE A=2

A=2
0
0
1
0
1
1
0
0
1

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

- Point query
- Range query
- Update
- Append
- Delete

WHERE A<2

A=1	∨	A=2	=	
1		0		1
0		0		0
0		1		1
1		0		1
0		1		1
0		1		1
0		0		0
0		0		0
0		1		1

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

- Point query
- Range query
- Update
- Append
- Delete

A[1] = 2

A=2	A=3
0	0
1	0
1	0
0	0
1	0
1	0
0	1
0	1
1	0

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

- Point query
- Range query
- Update
- Append
- Delete

Append A=1

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0
1	1	0	0

Basic Bitmap Index

A	A=1	A=2	A=3
1	1	0	0
3	0	0	1
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

- Point query
- Range query
- Update
- Append
- Delete

Delete A[1]

A	A=1	A=2	A=3
1	1	0	0
3	0	0	0
2	0	1	0
1	1	0	0
2	0	1	0
2	0	1	0
3	0	0	1
3	0	0	1
2	0	1	0

FastBit and WAH

- WAH: word-aligned hybrid
  - Space overhead
    - Data size ↑
    - Cardinality (number of distinct values) ↑

WAH

- If we have a long range of consecutive bits with the same value, we can compress them

WAH

Raw bit vector (155 bits)

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

WAH

Raw bit vector (155 bits)

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

31 bits

WAH

- Encoded bit vector are partitioned into 32-bit words
  - Literal word
    - | 0 | 31-bit raw bitmap |
  - Fill word
    - | 1 | 1-bit fill bit | # of 31-bit raw bitmap filled with the fill bit |

WAH

Raw bit vector (155 bits)

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

32 bits

WAH

- Frequent encoding and decoding are inefficient
- Getting the i^{th} position will require to decode all bit vectors from start to the i^{th} position
- Query/update will need to encode the entire bit vector

Update-Conscious Bitmap (UCB)

index	A	A=1	A=2	A=3	EB
1	1	1	0	0	1
2	3	0	0	1	1
3	2	0	1	0	1
4	1	1	0	0	1
5	2	0	1	0	1
6	2	0	1	0	1
7	3	0	0	1	1
8	3	0	0	1	1
9	2	0	1	0	1

UCB (query)

index	A
1	1
2	3
3	2
4	1
5	2
6	2
7	3
8	3
9	2

A=1	A=2	A=3	EB
1	0	0	1
0	0	1	1
0	1	0	1
1	0	0	1
0	1	0	1
0	1	0	1
0	0	1	1
0	0	1	1
0	1	0	1

WHERE A<2

A=1 ∨ A=2 ∧ EB =

A=1		A=2		EB		
1		0		1		1
0		0		1		0
0		1		1		1
1		0		1		1
0	∨	1	∧	1	=	1
0		1		1		1
0		0		1		0
0		0		1		0
0		1		1		1

UCB (efficient deletion)

index	A
1	1
2	3
3	2
4	1
5	2
6	2
7	3
8	3
9	2

A=1	A=2	A=3	EB
1	0	0	1
0	0	1	1
0	1	0	1
1	0	0	1
0	1	0	1
0	1	0	1
0	0	1	1
0	0	1	1
0	1	0	1

index	A
1	1
3	2
4	1
5	2
6	2
7	3
8	3
9	2

A=1	A=2	A=3	EB
1	0	0	1
0	0	1	0
0	1	0	1
1	0	0	1
0	1	0	1
0	1	0	1
0	0	1	1
0	0	1	1
0	1	0	1

UCB (update)

index	A	A=1	A=2	A=3	EB
1	1	1	0	0	1
2	3	0	0	1	1
3	2	0	1	0	1
4	1	1	0	0	1
5	2	0	1	0	1
6	2	0	1	0	1
7	3	0	0	1	1
8	3	0	0	1	1
9	2	0	1	0	1

index	A	A=1	A=2	A=3	EB
1	1	1	0	0	1
		0	0	1	0
3	2	0	1	0	1
4	1	1	0	0	1
5	2	0	1	0	1
6	2	0	1	0	1
7	3	0	0	1	1
8	3	0	0	1	1
9	2	0	1	0	1
2	2	0	1	0	1

UCB

- When the number of updates goes up
  - The compressibility of the EB goes down
  - When reading, the time needed to decoding and re-encoding goes up
- Therefore, It is not very scalable

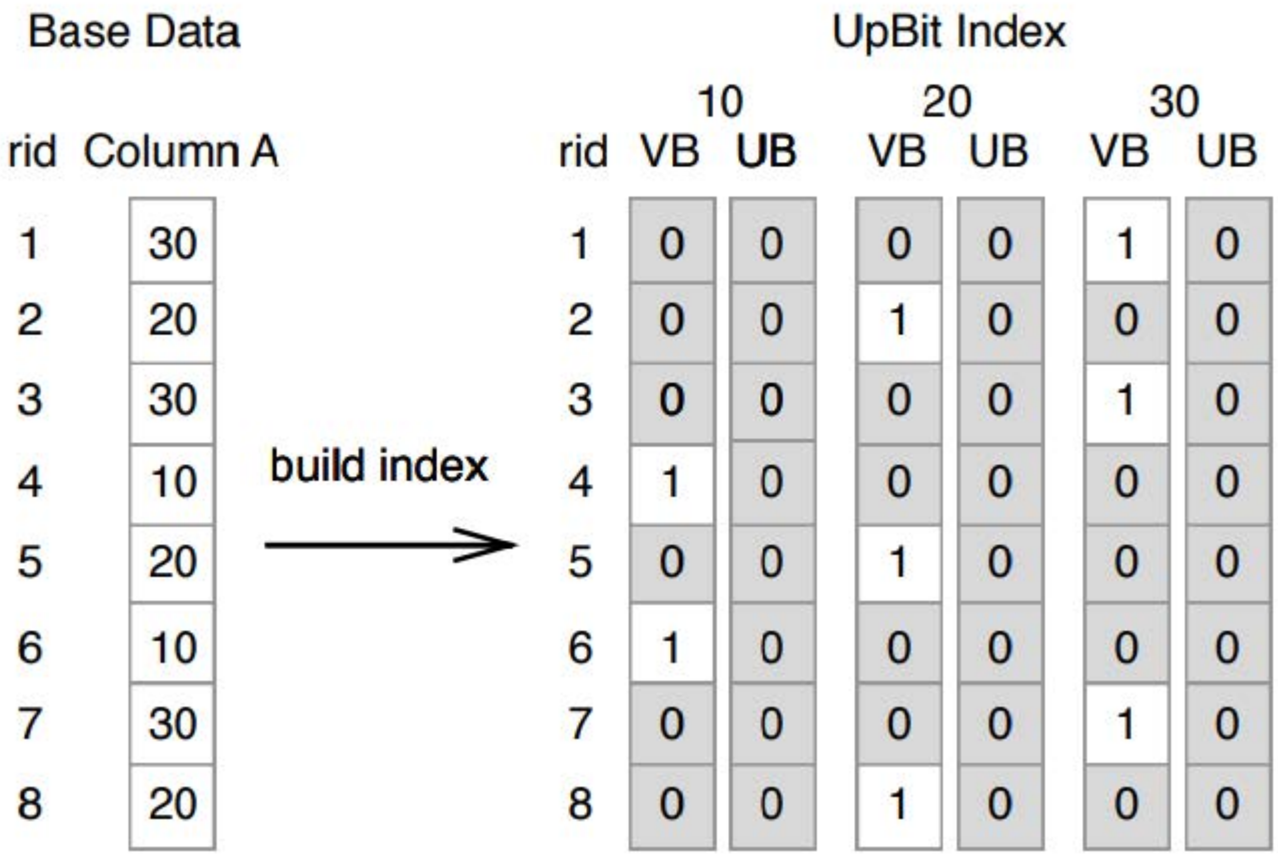
Updatable Bitmap (UpBit)

- The compressibility of the EB goes down when the number of updates goes up
  - UpBit introduces one update bit vector (UB) for each value bit vector (VB), and UB is merged to VB periodically to decrease the compressibility of UB
- Decoding and re-encoding is inefficient when updating and reading
  - UpBit adds fence pointers to compressed VB to enable partial decoding

Notations

- An attribute A
- d unique values
- $VB = \{V_i \mid \forall i \in \{1, \dots, d\}\}$
- $UB = \{U_i \mid \forall i \in \{1, \dots, d\}\}$

The internals of UpBit



The internals of UpBit

- Value-Bitvector Mapping
- Update Bitvectors

UpBit Operations (Retrieving the value of a row)

get_value (index: *UpBit*, **row:** *k***)**

---

for each i $\in \{1, 2, ..., d\}$ **do**
    $temp\_bit = V_i.get\_bit(k) \oplus U_i.get\_bit(k)$
    if $temp\_bit$ **then**
        Return val_i
    end if
end for

Algorithm 3: Get value of row k using UpBit.

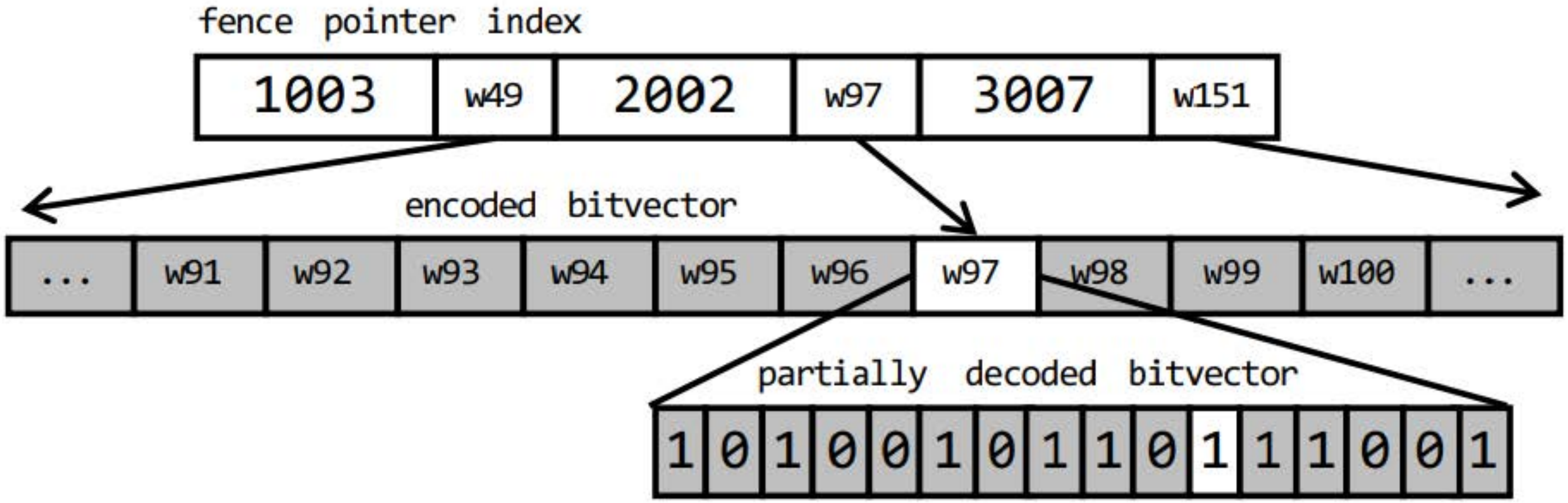
UpBit Operations (Get a bit in a particular row)

get_bit (bitvector: B, row: k)

```
pos = fence_pointer.nearest(k)
while pos < k do
    if isFill(B[pos]) then
        value, length = decode(B[pos])
        if (pos + length) * 31 < k then
            pos+ = length
        else
            Return value
        end if
    else
        if pos * 31 - k < 31 then
            Return B[pos]&(1 << (k%31))
        else
            pos++
        end if
    end if
end while
```

Algorithm 4: Get k_{th} bit of a bitvector using UpBit.

UpBit Operation (Make use of the fence pointers)



UpBit Operations (Searching)

search (index: *UpBit*, **value:** *val***)**

Find the i bitvector that *val* corresponds to

if U_i contains only zero **then**

    return V_i

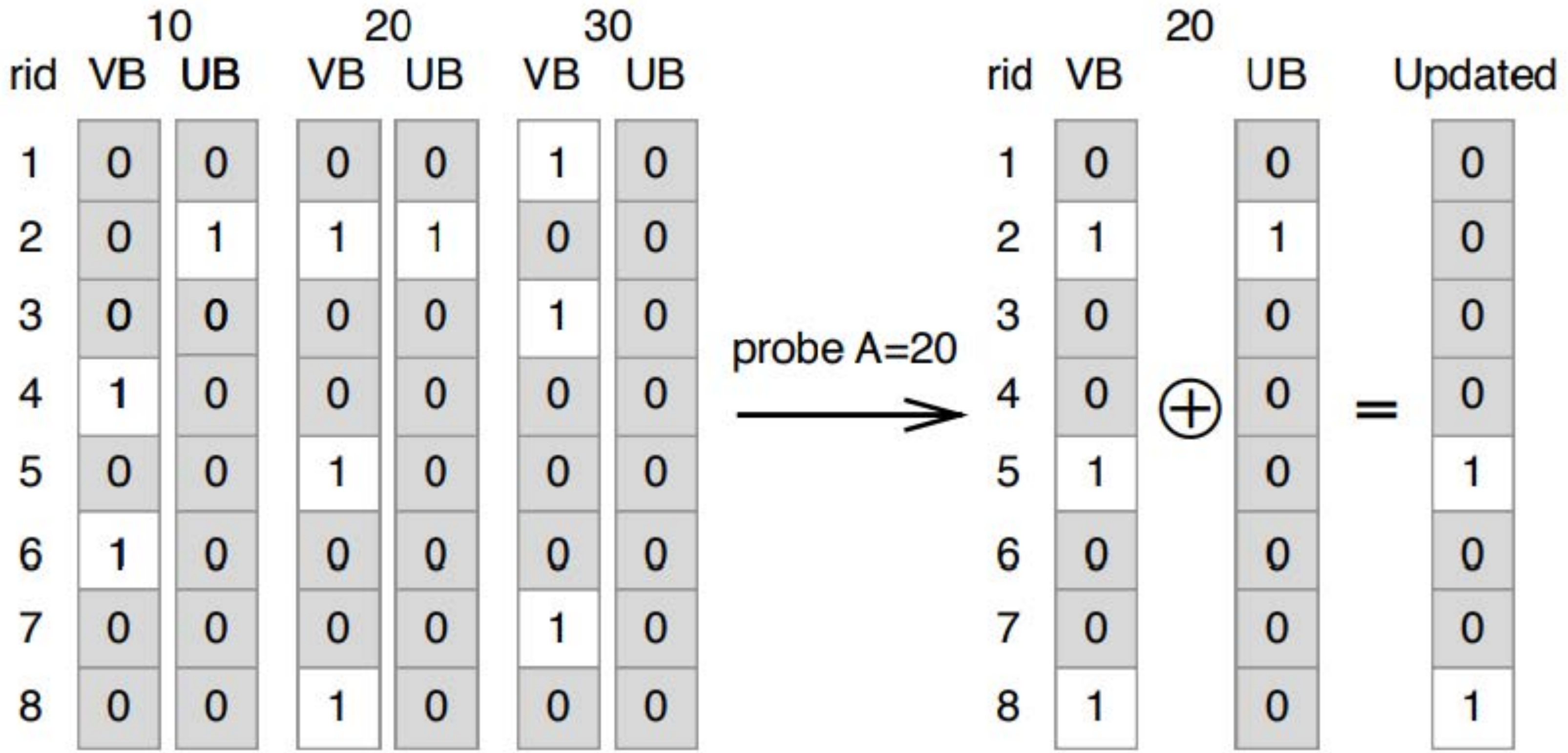
else

    return $V_i \oplus U_i$

end if

Algorithm 1: Searching UpBit for value *val*.

UpBit Operations (Searching)



UpBit Operations (Deleting)

delete_row (index: *UpBit*, **row:** k**)**

1: Find the *val* of row k
2: Find the i bitvector that *val* corresponds to
3: $U_i[k] = \neg U_i[k]$

Algorithm 2: Deleting row k with UpBit.

UpBit Operations (Deleting)

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

Delete row 2 →

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

UpBit Operations (Updating)

update_row (index: $UpBit$**, row:** k**, value:** val**)**

1: Find the i bitvector that val corresponds to
2: Find the old value old_val of row k
3: Find the j bitvector that old_val corresponds to
4: $U_i[k] = \neg U_i[k]$
5: $U_j[k] = \neg U_j[k]$

UpBit Operations (Updating)

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

Update row 2 from 20 to 10

→

	10		20		30	
rid	VB	UB	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	1	1	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

UpBit Operations (Inserting)

insert_row (index: *UpBit*, **value:** *val***)**

Find the i bitvector that *val* corresponds
if U_i does not have enough empty padding space **then**
    Extend U_i padding space
end if
$U_i.\#elements{+}{+}$
$U_i[\#elements] = 1$

Algorithm 6: Insert new value, *val*.

UpBit Operations (Inserting)

	10		20		30	
rid	VB	**UB**	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
Pad	0	0	0	0	0	0

Update row 2 from 20 to 10

→

	10		20		30	
rid	VB	**UB**	VB	UB	VB	UB
1	0	0	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	1	0	0	0	0	0
7	0	0	0	0	1	0
8	0	0	1	0	0	0
9	0	0	1	0	0	0

UpBit Operations (Merging)

merge (index: *UpBit*, **bitvector:** *i*)

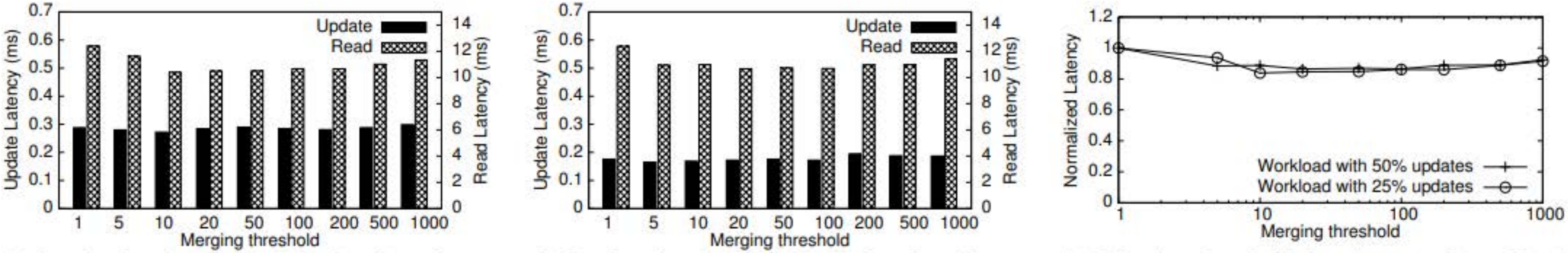
$V_i = V_i \oplus U_i$
$comp\_pos = 0$
$uncomp\_pos = 0$
$last\_uncomp\_pos = 0$
for each i $\in \{1, 2, ..., length(V_i)\}$ **do**
    if $isFill(V_i[pos])$ **then**
        $value, length += decode(V_i[pos])$
        $uncomp\_pos += length$
    else
        $uncomp\_pos{+}{+}$
    end if
    if $uncomp\_pos - last\_uncomp\_pos > THRESHOLD$ **then**
        $FP.append(comp\_pos, uncomp\_pos)$
        $last\_uncomp\_pos = uncomp\_pos$
    end if
    $comp\_pos{+}{+}$
end for
$U_i \leftarrow 0s$

Algorithm 7: Merge UB of bitvector *i*.

Tuning UpBit

- The UB-VB merging threshold
- The fence pointer granularity
- The level of parallelism used

Tuning UpBit (merging threshold)



(a) Read and update latency as a function of merging threshold for a workload with 20% updates.

(b) Read and update latency as a function of merging threshold for a workload with 50% updates.

(c) Merging threshold for the overall workload combining reads and updates.

Tuning UpBit (fence pointers granularity)

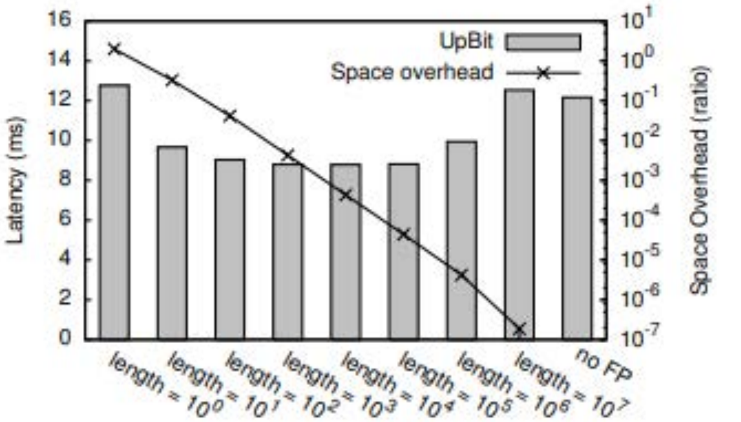


Figure 20: UpBit's optimal behavior needs fence pointers every 10^3-10^5 values having less than 0.5% space overhead.

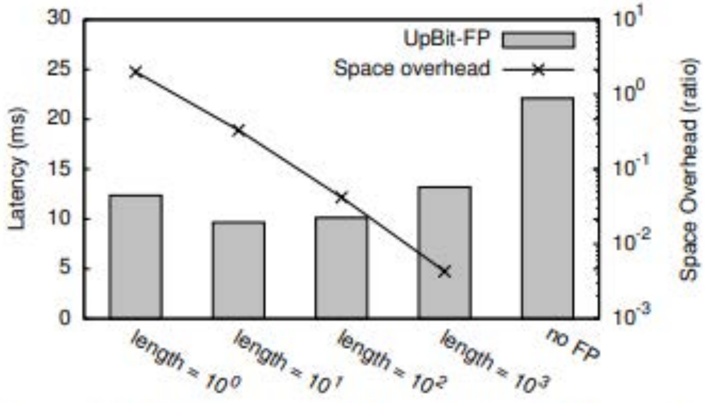


Figure 23: Fence pointers alone offer more than 2× better performance, having less than 10% space overhead.

Tuning UpBit (# of parallelism)

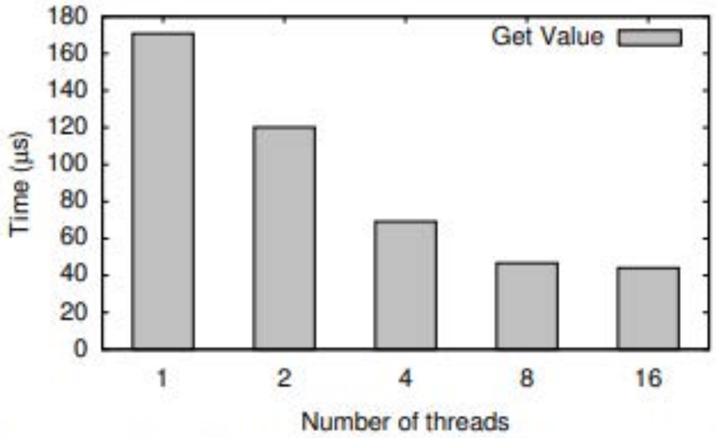


Figure 21: Bitvectors parallel scans scale with number of threads, leading to 3.9× improvement in *get_value*.

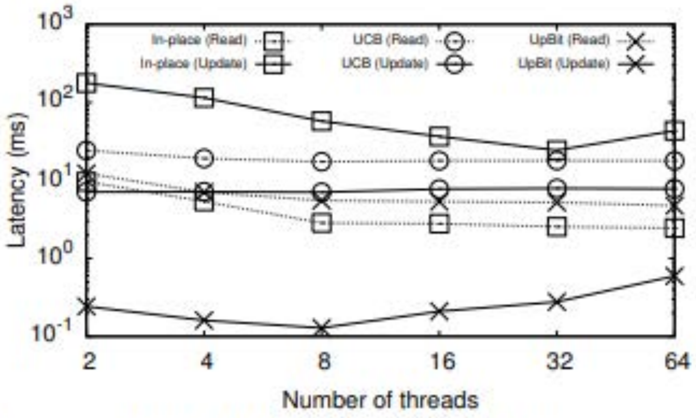


Figure 22: Updates with UpBit are two orders of magnitude faster than other approaches and scale for up to 8 threads.

Performance and conclusion

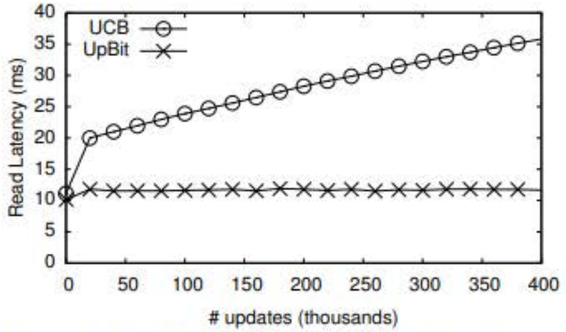


Figure 9: When stressing UpBit with updates, it delivers scalable read performance, addressing the most important limitation observed for UCB.

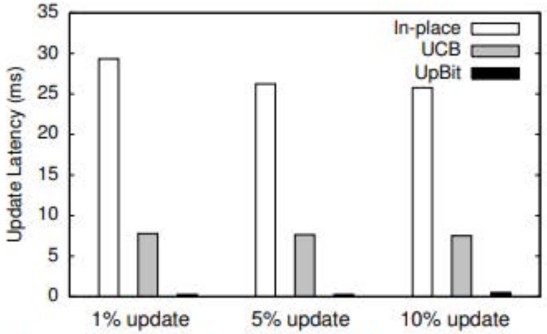


Figure 10: UpBit delivers $51 - 115\times$ faster updates than in-place updates and $15 - 29\times$ faster updates than state-of-the-art update-optimized bitmap index UCB.

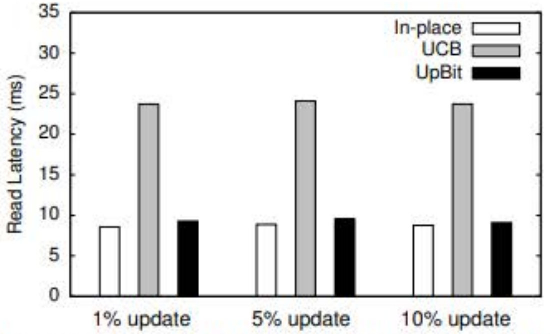
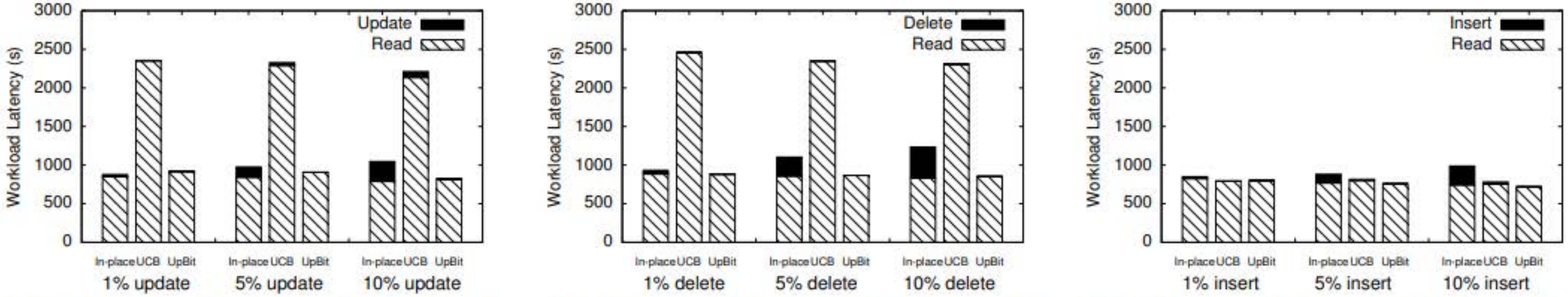


Figure 11: UpBit outperforms update-optimized indexes by nearly $3\times$ in terms of read performance while it loses only 8% compared to read-optimized indexes.



(a) UpBit vs. UCB vs. in-place for updates. (b) UpBit vs. UCB vs. in-place for deletes. (c) UpBit vs. UCB vs. in-place for inserts.

Figure 12: As we vary the percentage of updates, deletes or inserts from 1% to 10%, UpBit has the lowest overall workload latency when compared with any other setup. UpBit achieves similar read performance to a read-optimized bitmap index and drastically better updates (a) and deletes (b) than both read-optimized and update-optimized indexes. When inserting new values (c) all approaches have a similar low overhead on read performance. In-place updates cannot gradually absorb the new values, hence, inserting cost does not scale.

Performance and conclusion

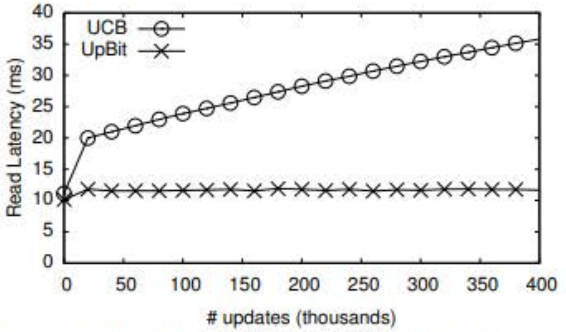


Figure 9: When stressing UpBit with updates, it delivers scalable read performance, addressing the most important limitation observed for UCB.

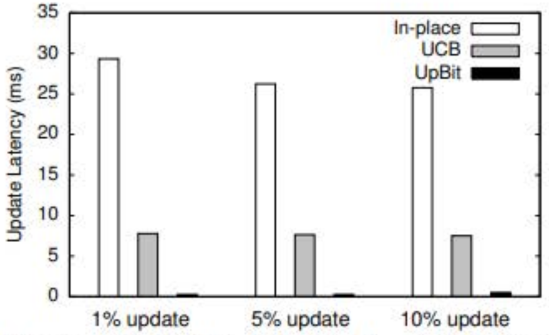


Figure 10: UpBit delivers $51 - 115\times$ faster updates than in-place updates and $15 - 29\times$ faster updates than state-of-the-art update-optimized bitmap index UCB.

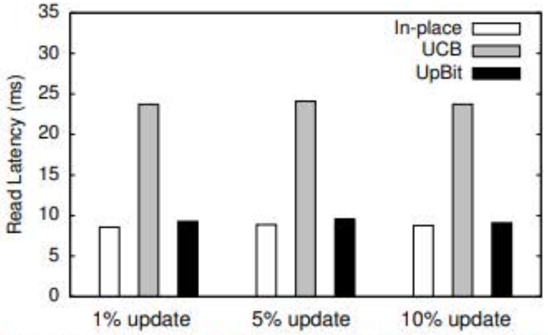
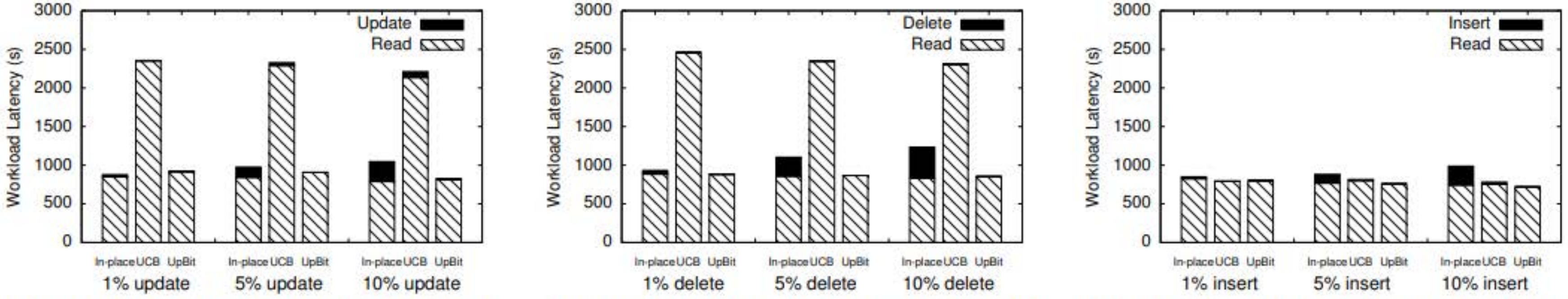


Figure 11: UpBit outperforms update-optimized indexes by nearly $3\times$ in terms of read performance while it loses only 8% compared to read-optimized indexes.



(a) UpBit vs. UCB vs. in-place for updates. (b) UpBit vs. UCB vs. in-place for deletes. (c) UpBit vs. UCB vs. in-place for inserts.

Figure 12: As we vary the percentage of updates, deletes or inserts from 1% to 10%, UpBit has the lowest overall workload latency when compared with any other setup. UpBit achieves similar read performance to a read-optimized bitmap index and drastically better updates (a) and deletes (b) than both read-optimized and update-optimized indexes. When inserting new values (c) all approaches have a similar low overhead on read performance. In-place updates cannot gradually absorb the new values, hence, inserting cost does not scale.

Performance and conclusion

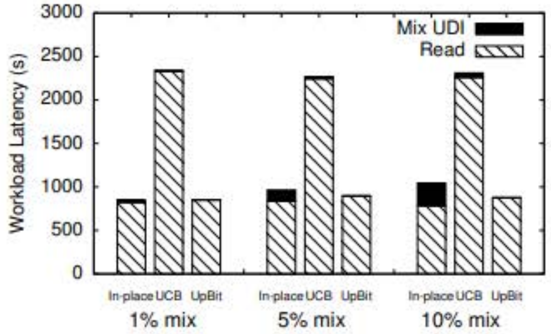


Figure 13: For general UDI workload, the overhead of maintaining a gradually less compressible EB overwhelms UCB, while UpBit offers faster workload execution than both approaches.

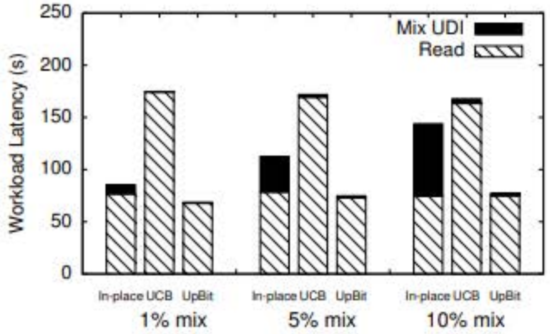


Figure 14: For a data set with larger domain cardinality ($d = 1000$) the update cost is relatively higher, and UpBit has a bigger benefit over in-place updates for the same number of updates.

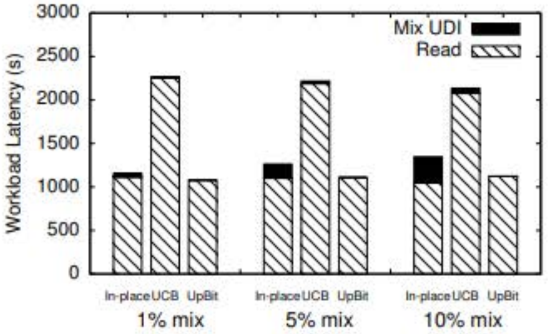


Figure 15: When increasing the data set size ($n = 1B$, $d = 100$), the qualitative behavior of all approaches remain the same. The average latency increases linearly with the data set size.

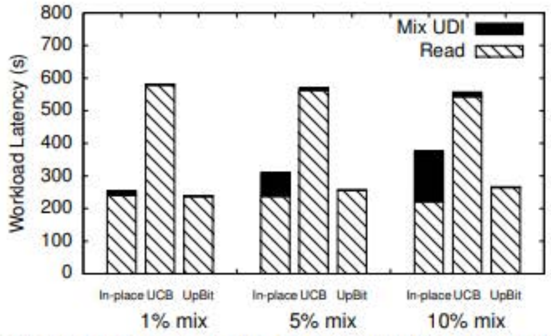


Figure 16: For skewed data (zipfian with $S = 1.5$), the latency decreases as most bitvectors are nearly empty. UpBit faces a small overhead because it has the same distribution of FPs in all VBs.

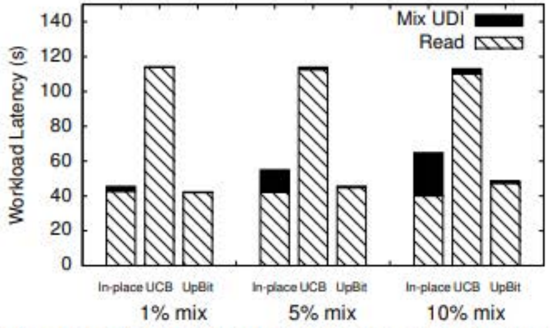


Figure 17: UpBit outperforms all other approaches with real data as well (Berkeley Earth data set with $n = 31$M values, and domain cardinality $d = 114$) for a workload with 1%, 5% or 10% updates.

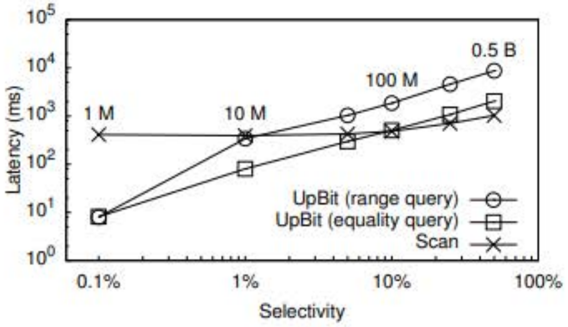


Figure 18: Compared with a fast scan, UpBit is faster for range queries with up to 1% selectivity. Equality queries with similar selectivity are much more efficient because we avoid the bitwise OR between VBs.