CS660: Intro to Database Systems

Class 11: Bitmap Indexes

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS660/

Motivation

Consider this relation:

```
CREATE TABLE Tweets (
        uniqueMsgID INTEGER, --unique message id
        tstamp TIMESTAMP,    --when was the tweet posted
        uid INTEGER,         --unique id of the user
        msg VARCHAR (140),   --the actual message
        zip INTEGER,         --zipcode when posted
        retweet BOOLEAN.     --retweeted?
);
```

What index should we build for …

Attribute	Index?
`uniqueMsg`	
`uid`	
`zip`	
`retweet`	

Example from Jignesh Patel

Motivation

Consider this relation:

```
CREATE TABLE Tweets (
        uniqueMsgID INTEGER,  --unique message id
        tstamp TIMESTAMP,     --when was the tweet posted
        uid INTEGER,          --unique id of the user
        msg VARCHAR (140),    --the actual message
        zip INTEGER,          --zipcode when posted
        retweet BOOLEAN.      --retweeted?
);
```

What index should we build for …

Attribute	Index?
`uniqueMsg`	B+ Tree
`uid`	B+ Tree
`zip`	B+ Tree / Hash
`retweet`	?

Example from Jignesh Patel

Motivation

Can we reduce the storage overhead?

Consider this relation:

```
CREATE TABLE Tweets (
        uniqueMsgID INTEGER, --unique message id
        tstamp TIMESTAMP,    --when was the tweet posted
        uid INTEGER,         --unique id of the user
        msg VARCHAR (140),   --the actual message
        zip INTEGER,         --zipcode when posted
        retweet BOOLEAN.     --retweeted?
);
```

What index should we build for …

Attribute	Index?	Size?
uniqueMsg	B+ Tree	$N_{msgs} \cdot (key_{size} + rowid_{size})$
uid	B+ Tree	$N_{users} \cdot (key_{size} + C_{users} \cdot rowid_{size})$
zip	B+ Tree / Hash	$N_{zips} \cdot \left(key_{size} + C_{zips} \cdot rowid_{size}\right)$
retweet	?	$2 \cdot (key_{size} + N/2 \cdot rowid_{size})$

Example from Jignesh Patel

Motivation

INTEGER = 8B

BOOLEAN = 1B (ideally 1 bit)

RID = 8B

10M tweets, 100K users, 43K zip codes

Consider this relation:

```
CREATE TABLE Tweets (
        uniqueMsgID INTEGER, --unique message id
        tstamp TIMESTAMP,    --when was the tweet posted
        uid INTEGER,         --unique id of the user
        msg VARCHAR (140),   --the actual message
        zip INTEGER,         --zipcode when posted
        retweet BOOLEAN.     --retweeted?
);
```

What index should we build for …

Attribute	Column Size	Index Nodes Size
`uniqueMsg`	76MB	152MB
`uid`	76MB	77MB
`zip`	76MB	76.6MB
`retweet`	9.5MB (ideally 1.2MB)	76MB

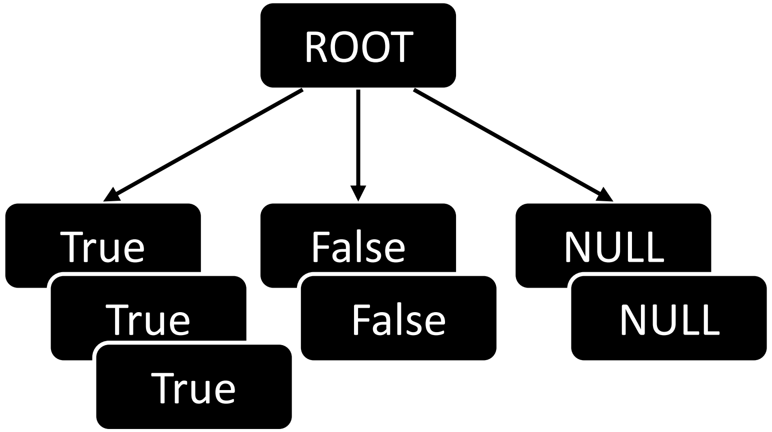
Crucial to reduce index size for low-cardinality columns!

Example from Jignesh Patel

Indexing for Low Cardinality Attributes

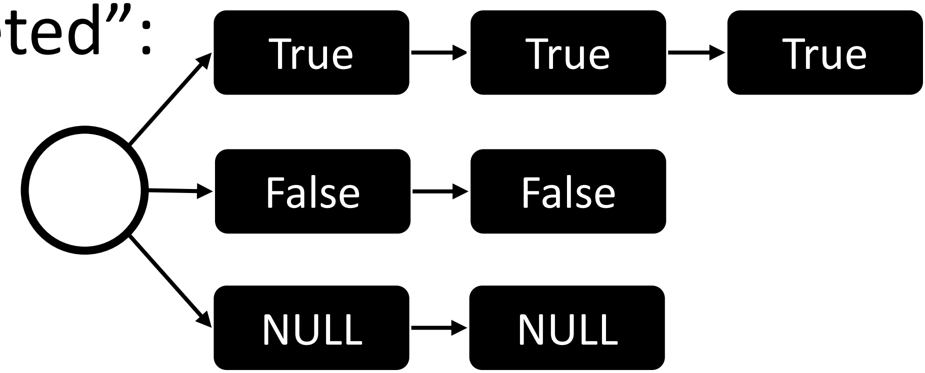
Consider building a **B+ tree** for "retweeted":

- Distinct values: True, False, NULL
- Lots of duplicates for each distinct value
- An awkward B+ tree with three long RID lists



Now consider building a **hash index** for "retweeted":

- All entries will be hashed in three buckets
- No redistribution or rehashing will solve this



Indexing for Low Cardinality Attributes

Now, consider only using **RID lists** for “retweeted”:

- Similar to B+ tree
- Traversing them for disjunctive/conjunctive queries is expensive

  OR AND

- Size problem: we keep a RID for every row >> column size

Attribute	Column Size	Index Nodes Size
`uniqueMsg`	76MB	152MB
`uid`	76MB	77MB
`zip`	76MB	76.6MB
`retweet`	9.5MB (ideally 1.2MB)	76MB

How to compress RID lists?

Patrick O'Neil
UMass Boston

Bitmap Index

First commercial system: Model 204 O'Neil

1987

Bitmap Index

Relation (heap file)

uniqueMsgID	…	zip	retweet
1		02135	Y
2		11243	Y
3		02215	N
4		90765	NULL
5		02134	N
…	…	…	…
10,000,000		53705	Y

Bitmap Index on retweet

r-Y	r-N
1	0
1	0
0	1
0	0
0	1
…	…
1	0

Query Processing

1. Scan “r-N” bitvector
2. For each bit set 1 return its position (RID)
3. Fetch the corresponding tuples

Critical Issue

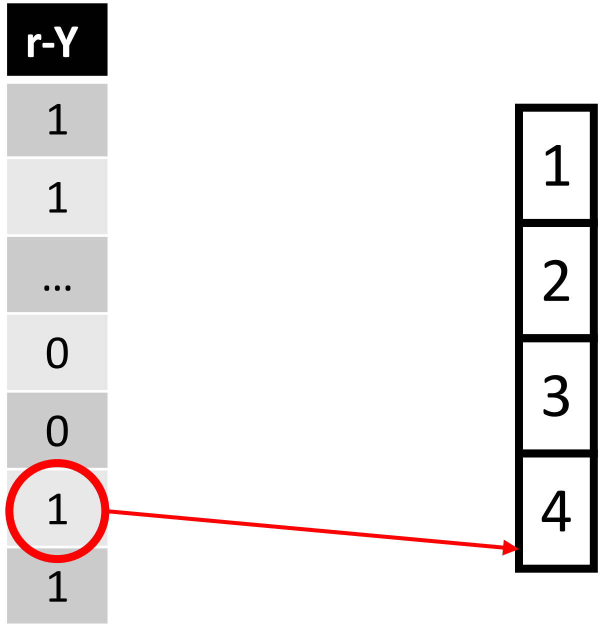
The order of bits in bitmap index **must follow** the order of rows in the file

```
SELECT * FROM Tweets WHERE retweet = ‘Y’
```

Mapping a bit position to a RID

Assume **fixed records per page** in the heapfile:

- Create the heapfile pages as **consecutive pages**
- Then construct **row id (page_id, slot#)** as follows:
    1. **page_id** = bit-position / #records_per_page
    2. **slot#** = bit-position % #records_per_page



Alternatively, rely on a **RID index** (most systems already have it).

Bitmap Index (other queries)

Relation (heap file)

uniqueMsgID	…	zip	retweet
1		02135	Y
2		11243	Y
3		02215	N
4		90765	NULL
5		02134	N
…	…	…	…
10,000,000		53705	Y

Bitmap Index on retweet

r-Y	r-N	r-NULL
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
…	…	…
1	0	0

```
SELECT COUNT(*)
FROM Tweets
WHERE retweet = 'Y'

SELECT *
FROM Tweets
WHERE retweet IS NOT NULL

SELECT COUNT(*)
FROM Tweets
WHERE retweet IS NULL
```

Attribute	Column Size	Index Nodes Size	Bitamp Index Size
`retweet`	9.5MB (ideally 1.2MB)	76MB	(3 bitvectors of 10M entries) 3.6MB

Index Sizes

INTEGER = 8B

BOOLEAN = 1B (ideally 1 bit)

RID = 8B

10M tweets, 100K users, 43K zip codes

Consider this relation:

```
CREATE TABLE Tweets (
        uniqueMsgID INTEGER, --unique message id
        tstamp TIMESTAMP,    --when was the tweet posted
        uid INTEGER,         --unique id of the user
        msg VARCHAR (140),   --the actual message
        zip INTEGER,         --zipcode when posted
        retweet BOOLEAN.     --retweeted?
);
```

Bitmap Index Size (bytes): #rows * (cardinality+1) / 8

What index should we build for …

Attribute	Column Size	Index Nodes Size	Bitmap Index Size
`uniqueMsg`	76MB	152MB	11TB
`uid`	76MB	77MB	116GB
`zip`	76MB	76.6MB	50GB
`retweet`	9.5MB (ideally 1.2MB)	76MB	3.6MB

Close to ideal!

Example from Jignesh Patel

Benefits of Bitmap Indexing

Column A	A=10	A=20	A=30
30	0	0	1
20	0	1	0
30	0	0	1
10	1	0	0
20	0	1	0
10	1	0	0
30	0	0	1
20	0	1	0

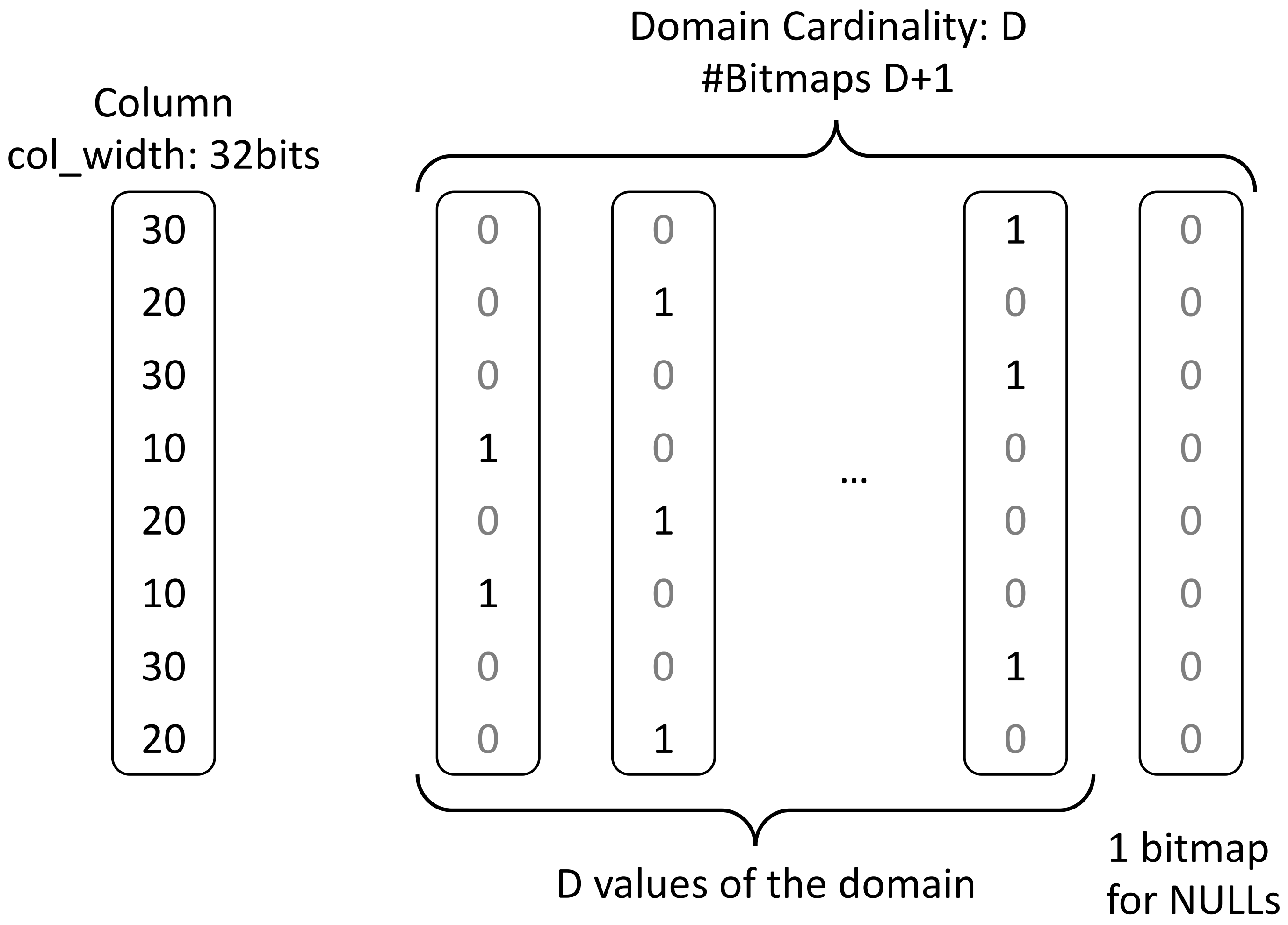
Specialized indexing

- 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

Storing a Bitmap Index



A bitmap **per domain value** and **1 for NULLs**

Each bitmap can be stored as a separate file

→Store **<u>one bit for every value for every row</u>**

Column size (bits) = #rows * col_width

Bitmap Idx size (bits) = #rows * (cardinality+1)

if ***cardinality+1 < col_width***, then

the bitmap index is smaller than the column!

If I use **32-bit integers** but only store **10 values**:

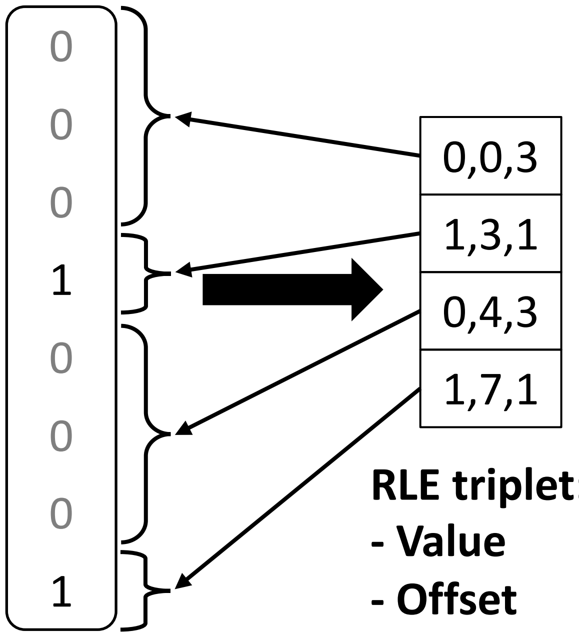
column size **32*N**, while bitmap idx size **11*N**

Even better:

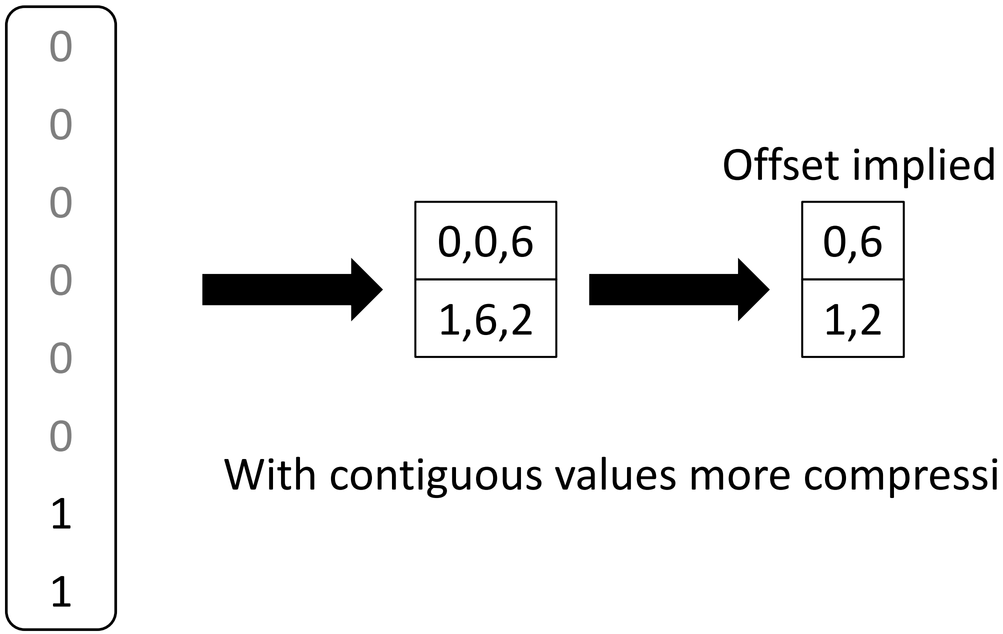
Bitmaps are **highly compressible!**

How to Compress a Bitmap?

Represent consecutive bit-values with their count "Run-Length Encoding"



RLE triplet:
- **Value**
- **Offset**
- **Length**

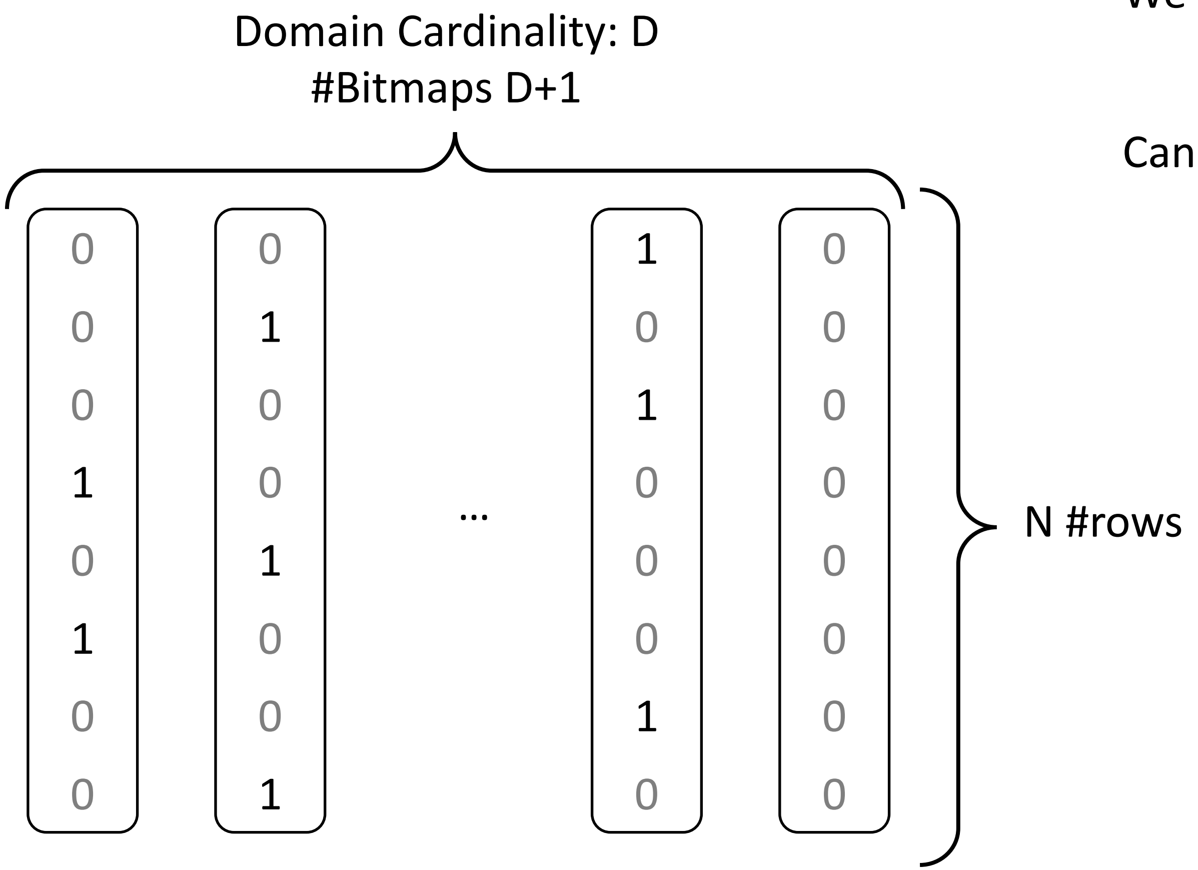


With contiguous values more compressible

Can we make the bitvector more compressible?

How to Compress a Bitmap?

Observation: Bitmaps are **sparse!**



We have N*(D+1) bits overall, and only N of them are set to 1

Can we do better than basic RLE?

Approach #1: General Purpose Compression
→ use standard compression algorithms (gzip, snappy, zstd)
→ must decompress entire bitmap index to process a query

Approach #2: Byte/Word-Aligned Bitmap Codes
→ structured variations of RLE

Approach #3: Roaring Bitmaps
→ a combination of RLE and RID lists

Oracle's Byte-Aligned Bitmap Codes (BBC)

Divide bitmap into chunks that contain different categories of bytes:

→ **Gap Byte**: All the bits are 0s

→ **Tail Byte**: Some bits are 1s

Encode each **chunk** that consists of **Gap Bytes** followed by **Tail Bytes**

→ **Gap Bytes** are compressed with **RLE**

→ **Tail Bytes** are stored **uncompressed** unless consisting of only 1-byte or has only one non-zero bit.

BBC discussion from Andy Pavlo

Oracle’s Byte-Aligned Bitmap Codes (BBC)

Bitmap

00000000	00000000	00000000	00100000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	01000000	00100010

Compressed Bitmap

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

Tail bytes

00000000	00000000	00000000	00100000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	01000000	00100010

Tail bytes

Compressed Bitmap

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap Gap bytes Tail bytes

#1	00000000	00000000	00000000	00100000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	01000000	00100010

Compressed Bitmap

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

#1	00000000	00000000	00000000	00100000
	00000000	00000000	00000000	00000000
#2	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	01000000	00100010

Compressed Bitmap

Oracle’s Byte-Aligned Bitmap Codes (BBC)

Bitmap

#1	00000000	00000000	00000000	00100000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	01000000	00100010

Compressed Bitmap

#1 (011) (1) (0011)

1-3 4 5-7

Position 3

3 Gap Bytes

Special tail

Chunk #1 (Bytes 1-4)

Header Byte:

→Number of Gap Bytes (bits 1-3)

→Is the tail special? (bit 4)

→bits 5-7:

- Number of verbatim bytes (if bit 4=0)
- Index of 1-bit in tail byte (if bit 4=1)

No extra gap length bytes since gap length < 7

No verbatim bytes since tail is special

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

	00000000	00000000	00000000	00100000
	00000000	00000000	00000000	00000000
#2	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	01000000	00100010

Compressed Bitmap

#1 (011) (1) (0011)
#2 (111) (0) (0010) 00001110
01000000 00100010

Chunk #2 (Bytes 5-20)

Header Byte:
→14 Gap Bytes, 2 Tail Bytes (not special)
→#gaps > 6, so we must use extra byte

One gap length byte
Two verbatim bytes for tail

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

00000000	00000000	00000000	00100000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	01000000	00100010

Compressed Bitmap

#1 (011) (1) (0011)

#2 $\boxed{(111)}$ (0) (0010) 00001110

01000000 00100010

for > 6 gap bytes, use 111

Chunk #2 (Bytes 5-20)

Header Byte:

→14 Gap Bytes, 2 Tail Bytes (not special)

→#gaps > 6, so we must use extra byte

One gap length byte

Two verbatim bytes for tail

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

00000000	00000000	00000000	00100000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	01000000	00100010

Compressed Bitmap

#1 (011) (1) (0011)
#2 (111) (0) (0010) $\boxed{00001110}$
01000000 00100010

Gap Length = 14

Chunk #2 (Bytes 5-20)

Header Byte:
→14 Gap Bytes, 2 Tail Bytes (not special)
→#gaps > 6, so we must use extra byte

One gap length byte
Two verbatim bytes for tail

Oracle's Byte-Aligned Bitmap Codes (BBC)

Bitmap

00000000	00000000	00000000	00100000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	01000000	00100010

Compressed Bitmap

2 verbatim bytes

#1 (011) (1) (0011)

#2 (111) (0) (0010) 00001110

01000000 00100010

Verbatim Tail Bytes

Not special tail

Chunk #2 (Bytes 5-20)

Header Byte:

→14 Gap Bytes, 2 Tail Bytes (not special)

→#gaps > 6, so we must use extra byte

One gap length byte

Two verbatim bytes for tail

Original Bitmap Size: 20 bytes

BBC Compressed: 5 bytes

Observation for BBC

Oracle's BBC is an **obsolete** format

→ Although it provides *good compression*, it is slower than recent alternatives due to **excessive branching** (too many if statements)

→ Word-Aligned Hybrid (WAH) encoding is a patented variation on BBC that provides better performance.

None of these support **random** access.

→ to check a specific bit, decompress the whole thing

Also: hard to update (flip a single bit)!

Word-Aligned Hybrid (WAH) Code

What is a “word”?

Word is the unit a processor reads: 32 bits or 64 bits (4 or 8 bytes)

BBC had a lot of CPU overhead

Idea: fetch words which is the natural size the CPU reads

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

$\boxed{1}$0001000 01001000 01000100 01000100

most significant bit:
0 for literal and 1 for fill

$\boxed{0}$0000000 00100100 00111000 00100101

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

1$\boxed{0}$001000 01001000 01000100 01000100

second most significant bit:
fill bit

00000000 00100100 00111000 00100101

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

10001000 01001000 01000100 01000100

remaining (w-2) bits:
fill length = 138953796 zeros

00000000 00100100 00111000 00100101

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

1$\boxed{1}$001000 01001000 01000100 01000100

second most significant bit:
fill bit

00000000 00100100 00111000 00100101

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

11001000 01001000 01000100 01000100

remaining (w-2) bits:
fill length = 138953796 ones

00000000 00100100 00111000 00100101

Word-Aligned Hybrid (WAH) Code

Two types of words

→ *fill* words (similar to gap)

→ *literal* words (similar to verbatim)

10001000 01001000 01000100 01000100

remaining (w-1) bits:
literal bits

0 $\boxed{\text{0000000 00100100 00111000 00100101}}$

Word-Aligned Hybrid (WAH) Code - example

```
128 bits
10000000 00000000 00000111 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111 11111111 11111111 11111111 11111111
```

Word-Aligned Hybrid (WAH) Code - example

```
128 bits
10000000 00000000 00000111 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111 11111111 11111111 11111111 11111111
```

```
31-bit groups
1000000000000000000000111 0000000 0000000000000000000000000000000
0000000000000000000000000000000 1111111111111111111111111111111 1111
```

```
WAH
01000000000000000000001110000000
10000000000000000000000000111110   62 zeros
11000000000000000000000000011111   31 ones
00000000000000000000000000001111
                                   last 4 bits
```

literal

fill

fill-bit

Encodes 2^30 (1 billion) consecutive bits with 1 word

Positive Implication:

A rare column-value is encoded with 1 word instead of a huge bitvector

BUT: if a few sparse bits are set query processing take time (de-encoding > expensive than accessing data)

Roaring Bitmaps

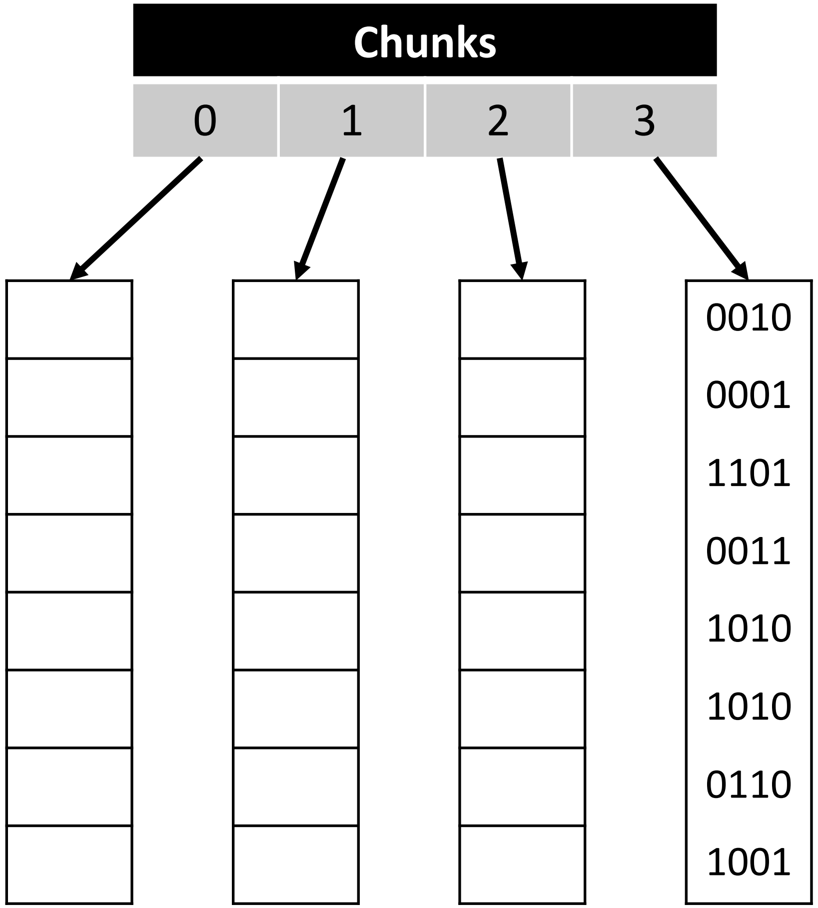
Store 32-bit integers (RIDS) in a compact **two-level** indexing data structure

→**Dense chunks** are stored using bitmaps

→**Sparse chunks** use packed arrays of 16-bit integers

Used in Lucene, Hive, Spark, Pinot.

Roaring Bitmaps



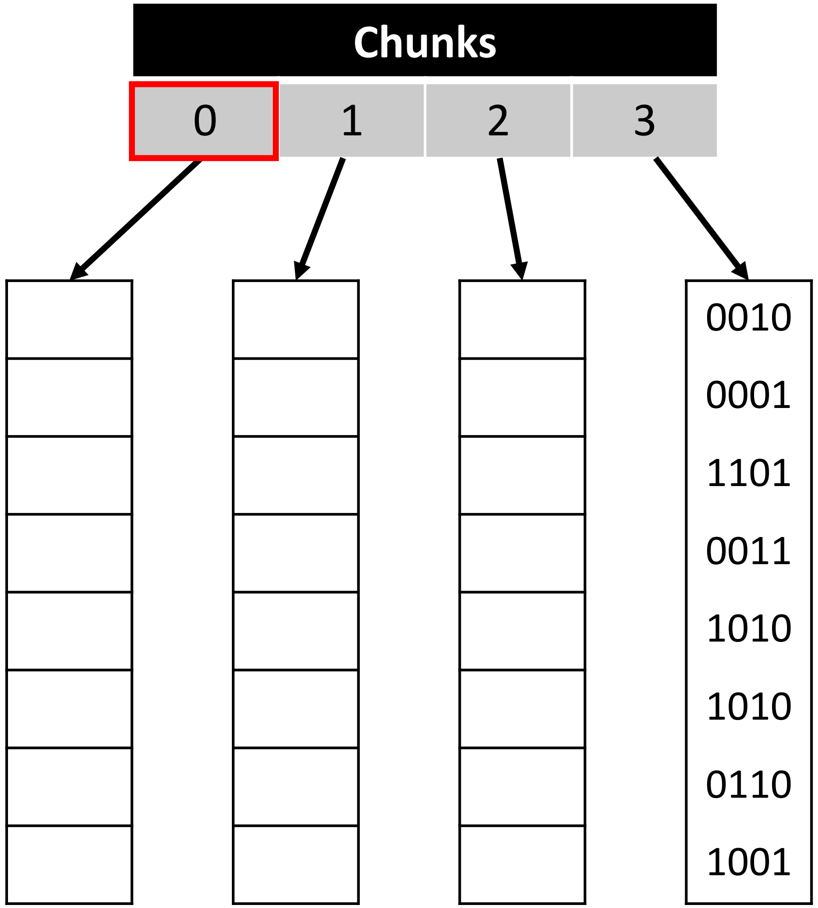
For each value k, assign it to a chunk based on k/2^16

→ Store k in the chunk's container

If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

k = 1200

Roaring Bitmaps



For each value k, assign it to a chunk based on k/2^16

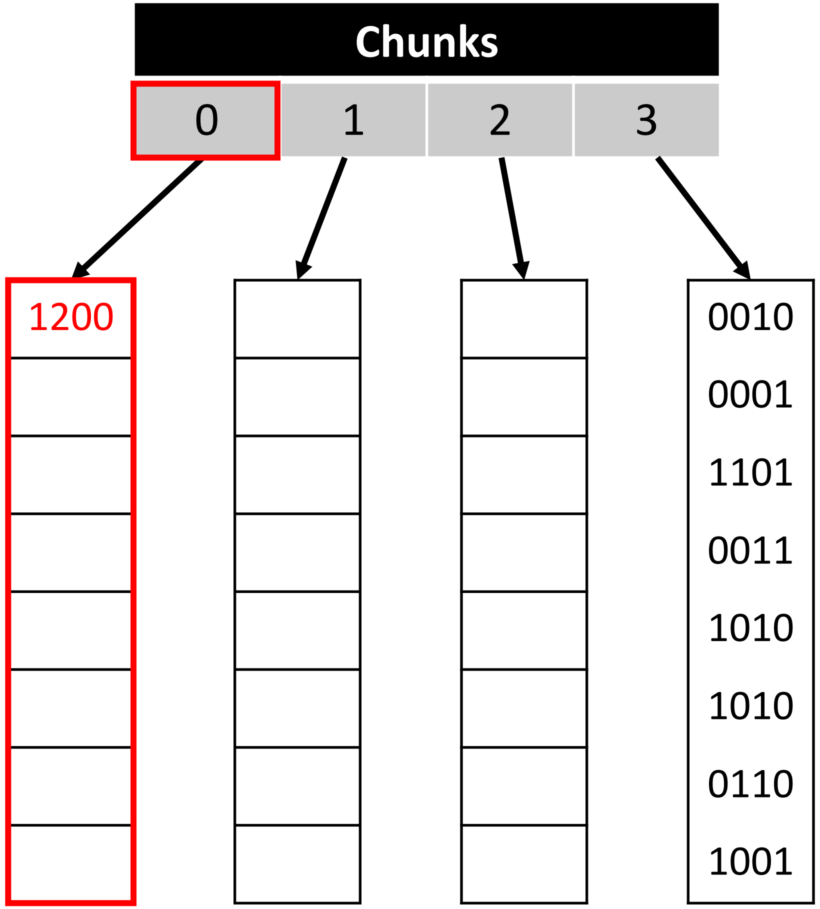
→ Store k in the chunk's container

If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

k = 1200

1200 / 2^16 = 0

Roaring Bitmaps



For each value k, assign it to a chunk based on k/2^16

→ Store k in the chunk's container

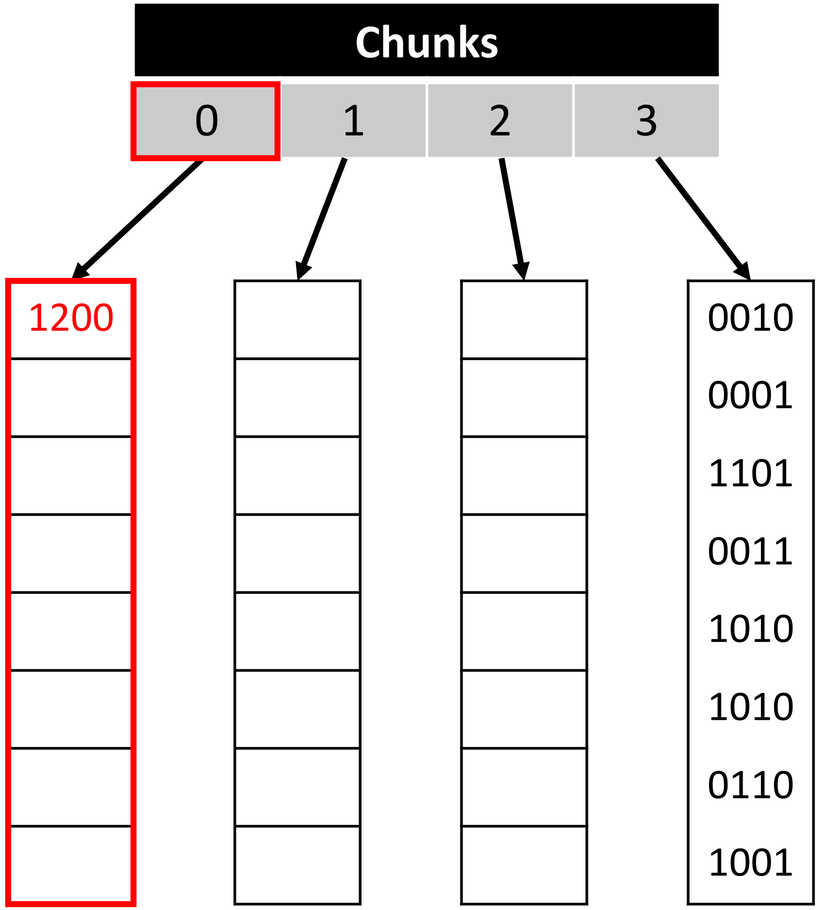
If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

k = 1200

1200 / 2^16 = 0

1200 % 2^16 = 1200

Roaring Bitmaps



For each value k, assign it to a chunk based on k/2^16

→ Store k in the chunk's container

If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

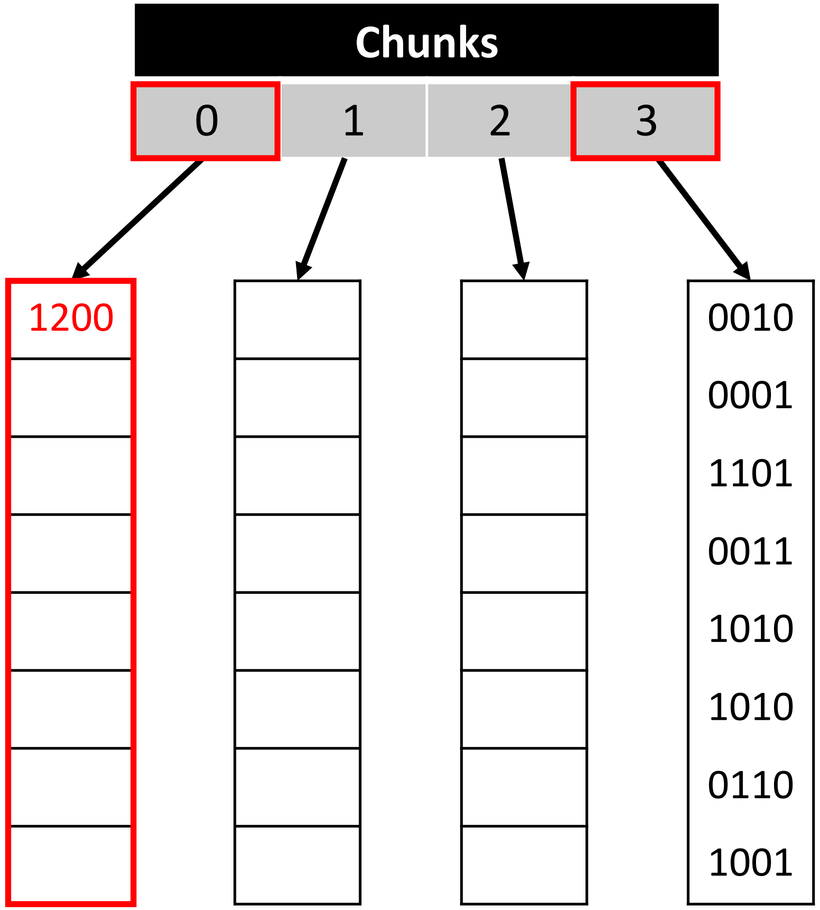
k = 1200

1200 / 2^16 = 0

1200 % 2^16 = 1200

k = 196626

Roaring Bitmaps



For each value k, assign it to a chunk based on k/2^16

→ Store k in the chunk's container

If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

k = 1200

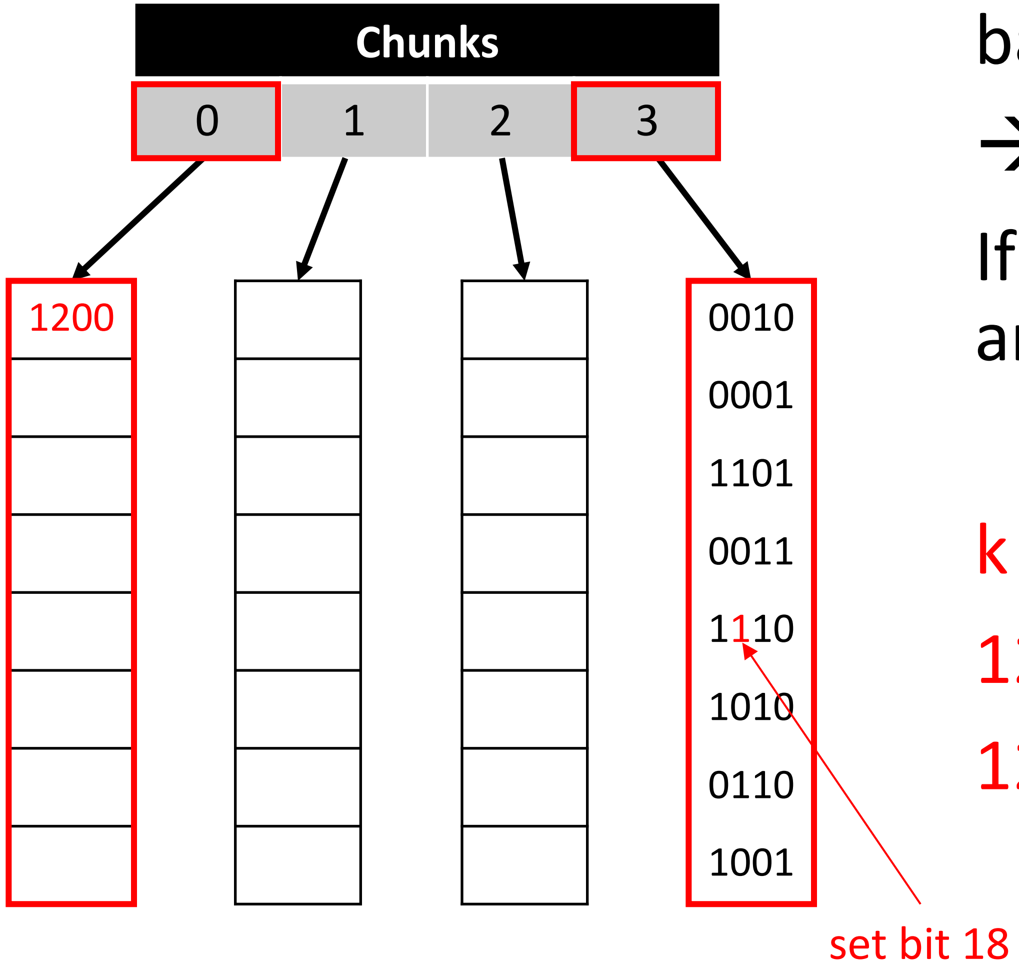
1200 / 2^16 = 0

1200 % 2^16 = 1200

k = 196626

196626 / 2^16 = 3

Roaring Bitmaps



For each value k, assign it to a chunk based on k/2^16

→ Store k in the chunk's container

If # of values in container < 4096, store as array. Otherwise, store as Bitmap.

k = 1200

1200 / 2^16 = 0

1200 % 2^16 = 1200

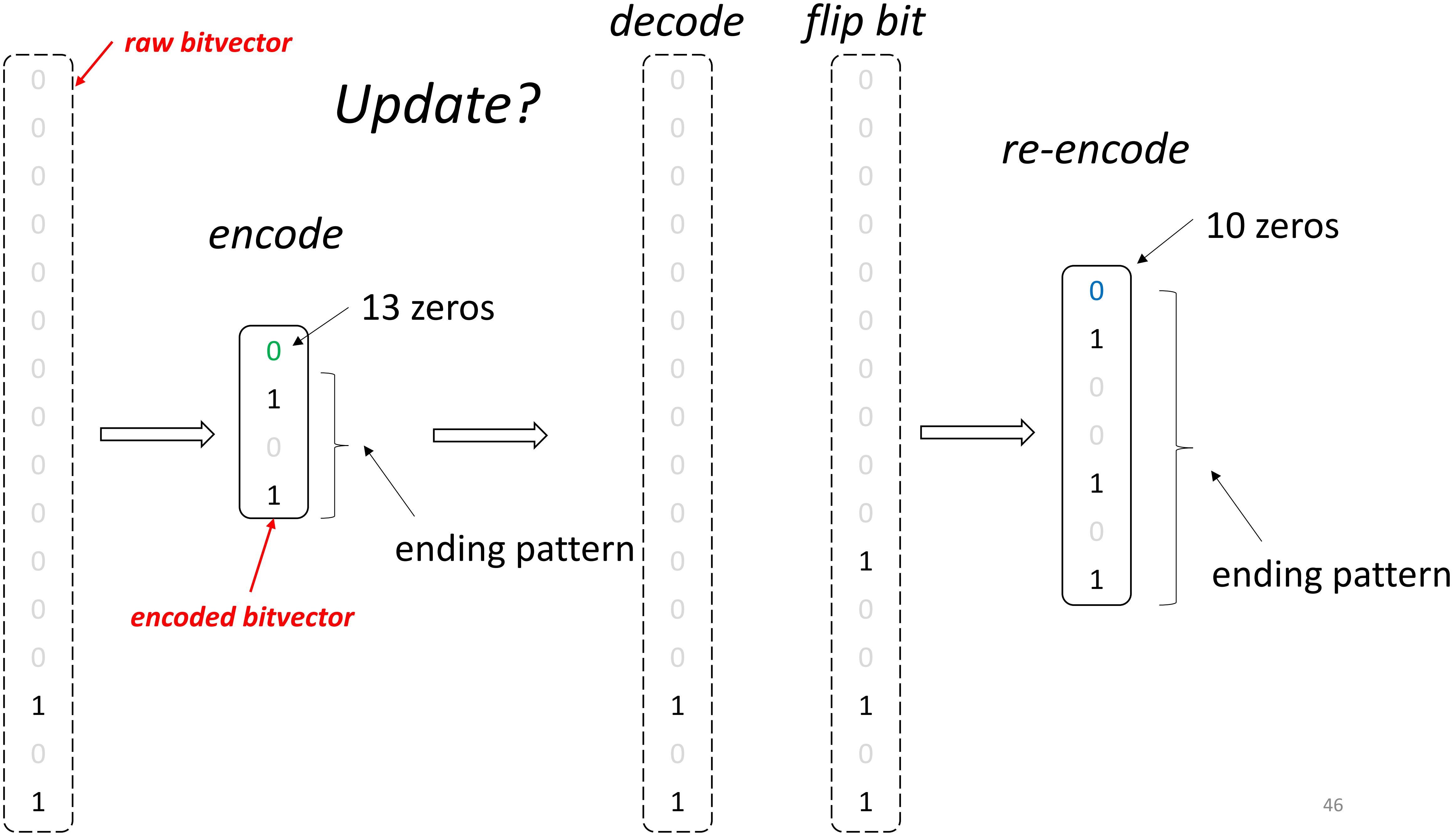
k = 196626

196626 / 2^16 = 3

196626 % 2^16 = 18

Updating Bitmap Indexes

Update?


raw bitvector
0
0
0
0
0
0
0
0
0
0
0
0
0
1
0
1
encode
13 zeros
0
1
0
1
ending pattern
encoded bitvector
decode
0
0
0
0
0
0
0
0
0
0
0
0
0
1
0
1
flip bit
0
0
0
0
0
0
0
0
0
0
1
0
0
1
0
1
re-encode
10 zeros
0
1
0
0
1
0
1
ending pattern

Goal

Bitmap Indexing with efficient ***Reads & Updates***

Bitmap Indexing and Deletes

Update Conscious Bitmaps (UCB), SSDBM 2007

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

efficient deletes by invalidation

existence bitvector (EB)

Prior Work: Bitmap Indexing and Deletes

Update Conscious Bitmaps (UCB), SSDBM 2007

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

efficient deletes by invalidation

existence bitvector (EB)

reads?

bitwise AND with EB

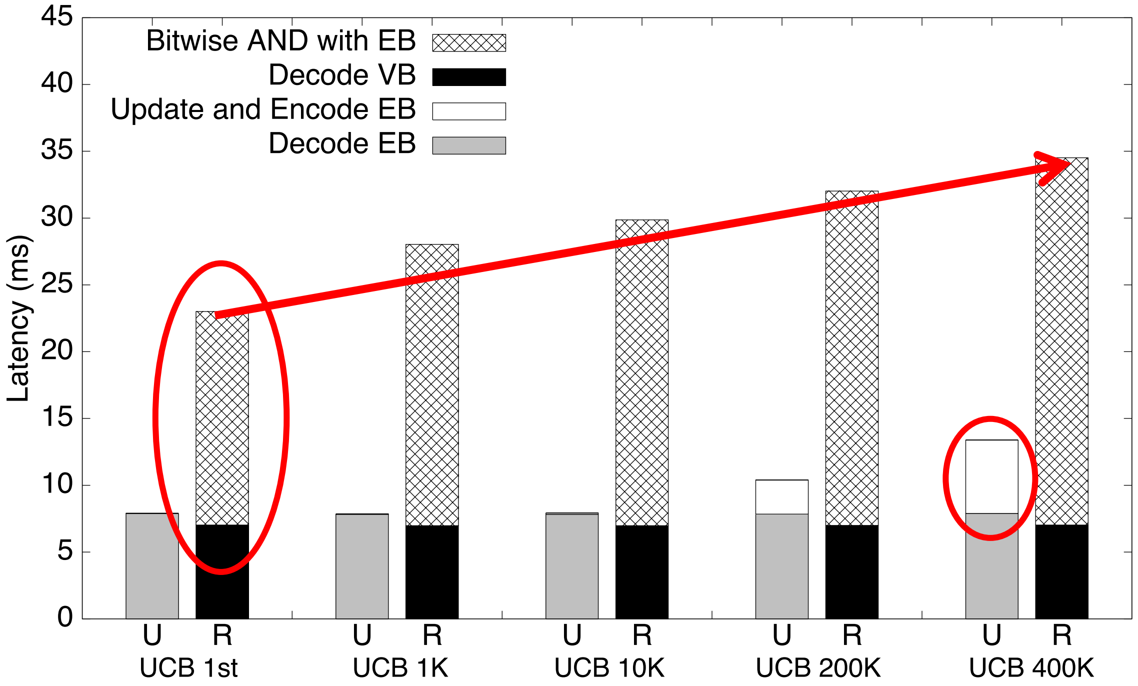
updates?

delete-then-append

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

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

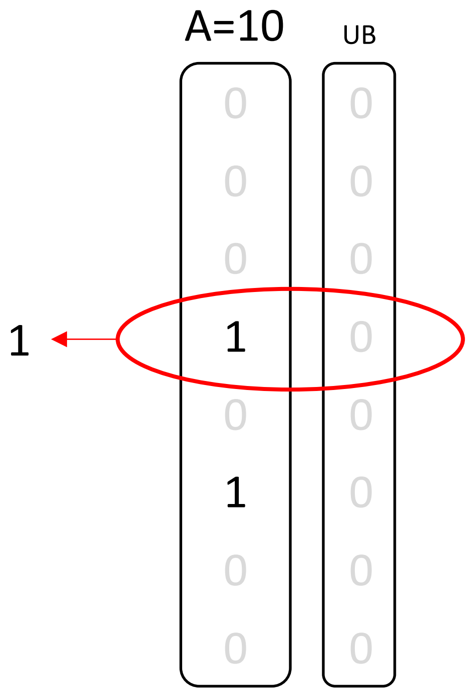
Bitmap Indexing for Reads & Updates

distribute update cost

efficient random accesses in compressed bitvectors

query-driven re-use results of bitwise operations

Design Element 1: update bitvectors

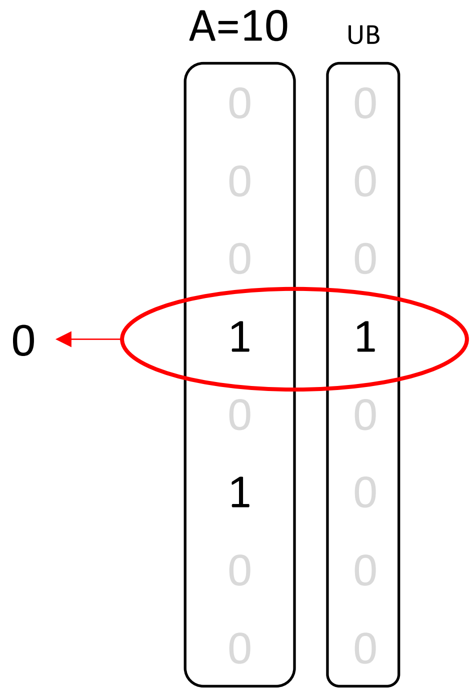


one per value of the domain
initialized to 0s

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 5 to 10

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

Updating UpBit …

… row 5 to 10

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

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

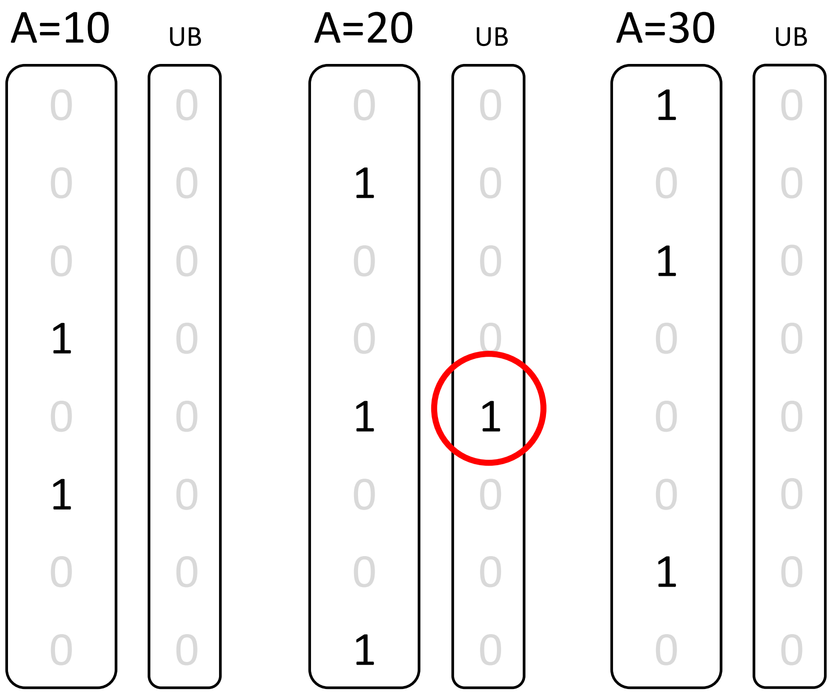
Updating UpBit …

… row 5 to 10

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

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

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 …

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

… row 5 to 10

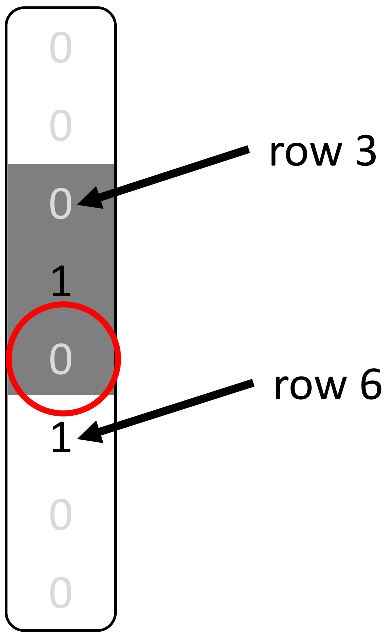
1. find old value of row 5 (A=20)
2. flip bit of row 5 of UB of A=20
3. flip bit of row 5 of UB of A=10

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

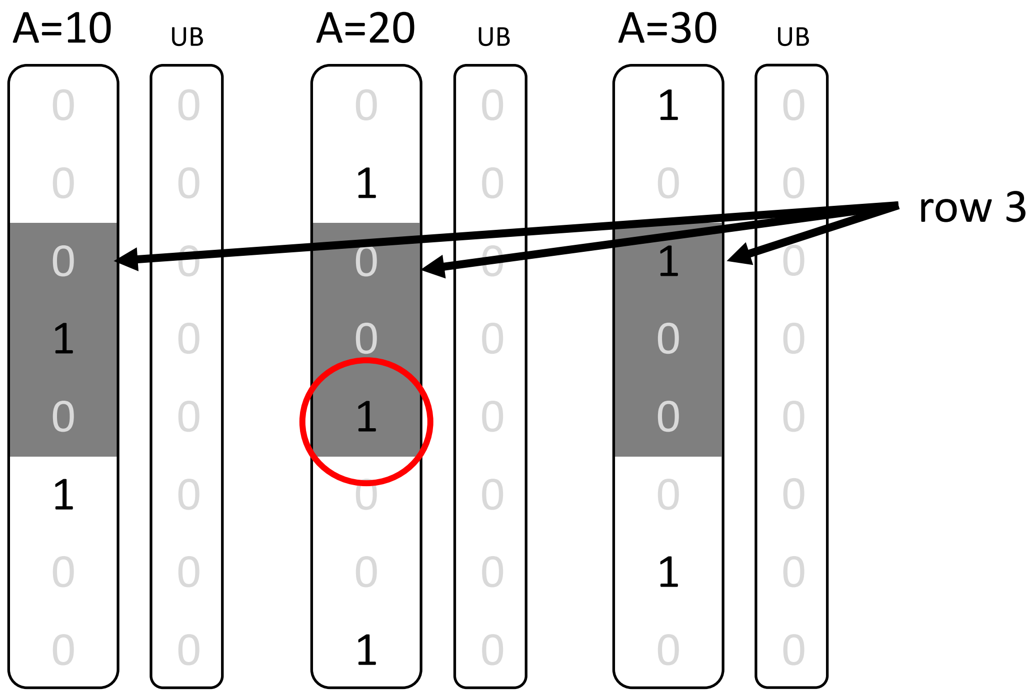
Updating UpBit …

… row 5 to 10

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

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

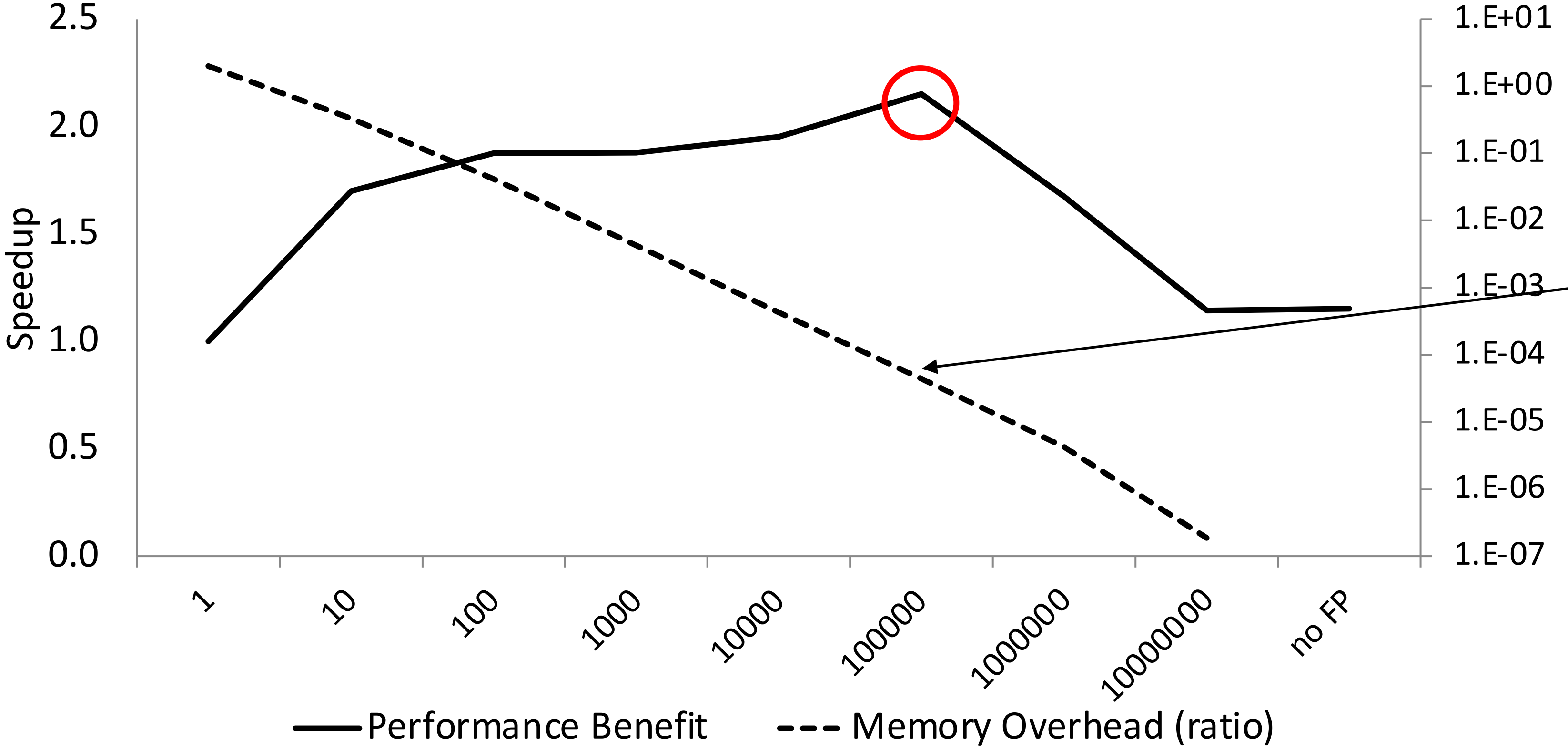
Updating UpBit (with fence pointers)…



… row 2 to 10

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

How dense should the *fence pointers* be?



0.004% space amplification

fence pointers every 10^5 entries

Querying

Querying UpBit …

… A = 20

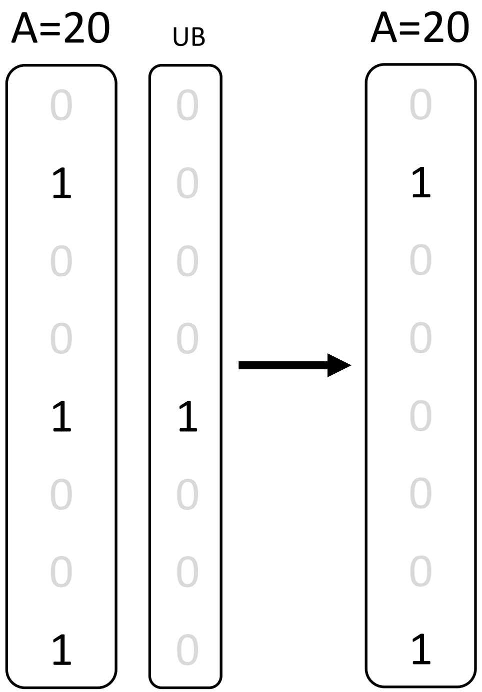
Return the XOR of A=20 and UB

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

Querying UpBit …

… A = 20

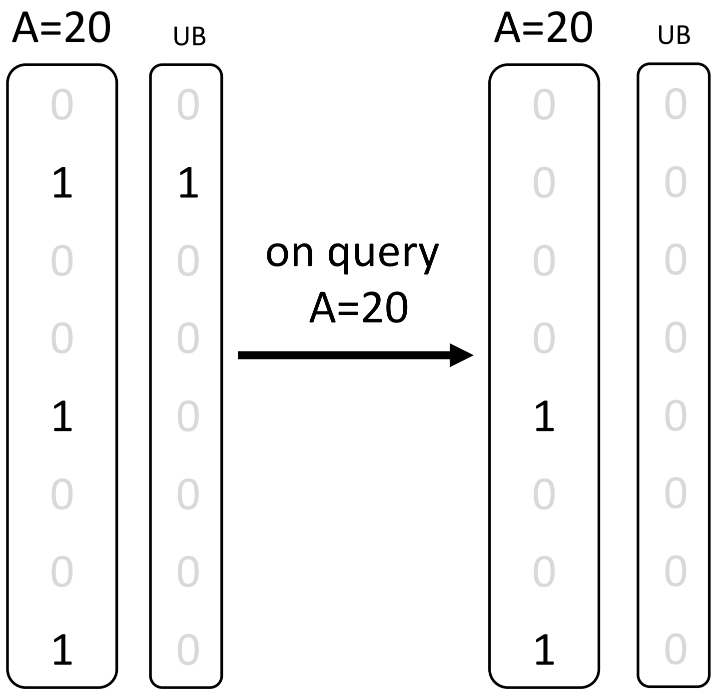
Return the XOR of A=20 and UB



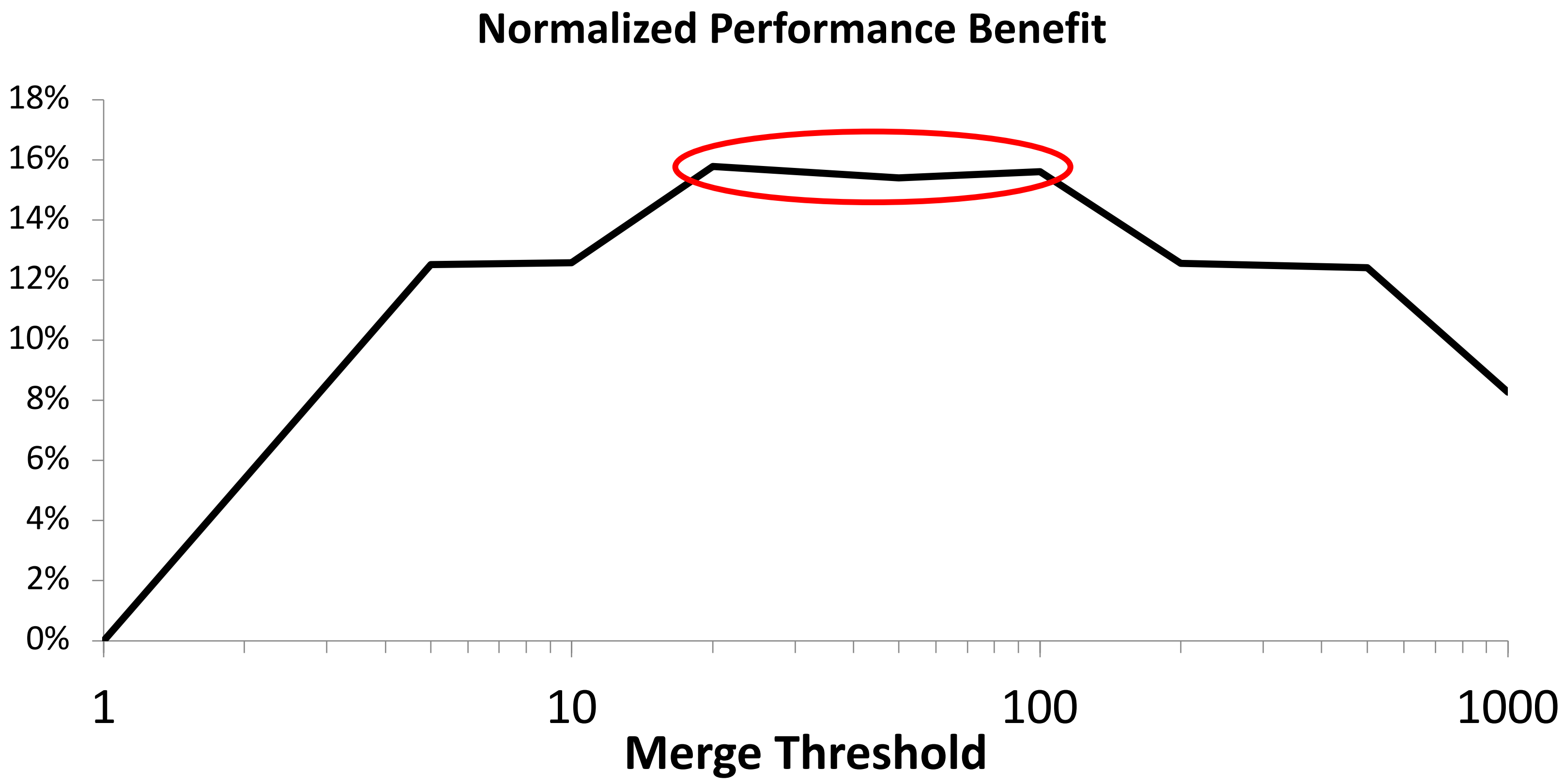
can we re-use the result?

Design Element 3: query-driven merging

maintain high compressibility of UB
query-driven merging



How frequently to merge UB back to VB?



merge back when 20-100 updates have been recorded

Deleting

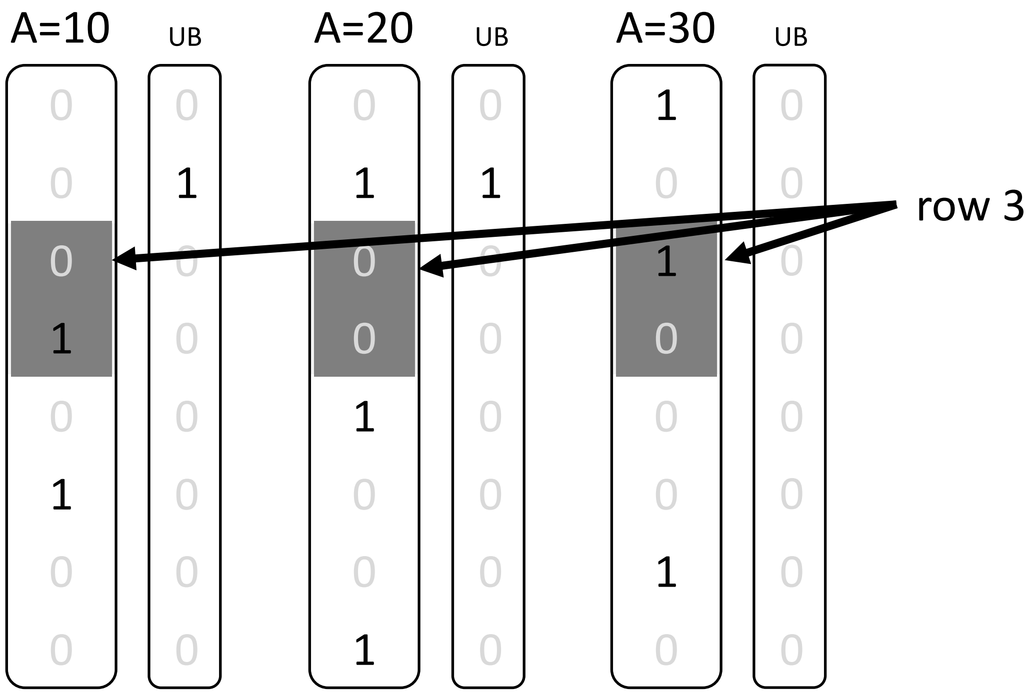
UpBit: **Up**datable **Bit**map Index

Deleting … … row 3

A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	1	1	1	0	0
0	0	0	0	1	0
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

UpBit: **Up**datable **Bit**map Index

Deleting …



… row 3

1. find value of row 3 (A=30)
   using fence pointers

UpBit: **Up**datable **Bit**map Index

Deleting …

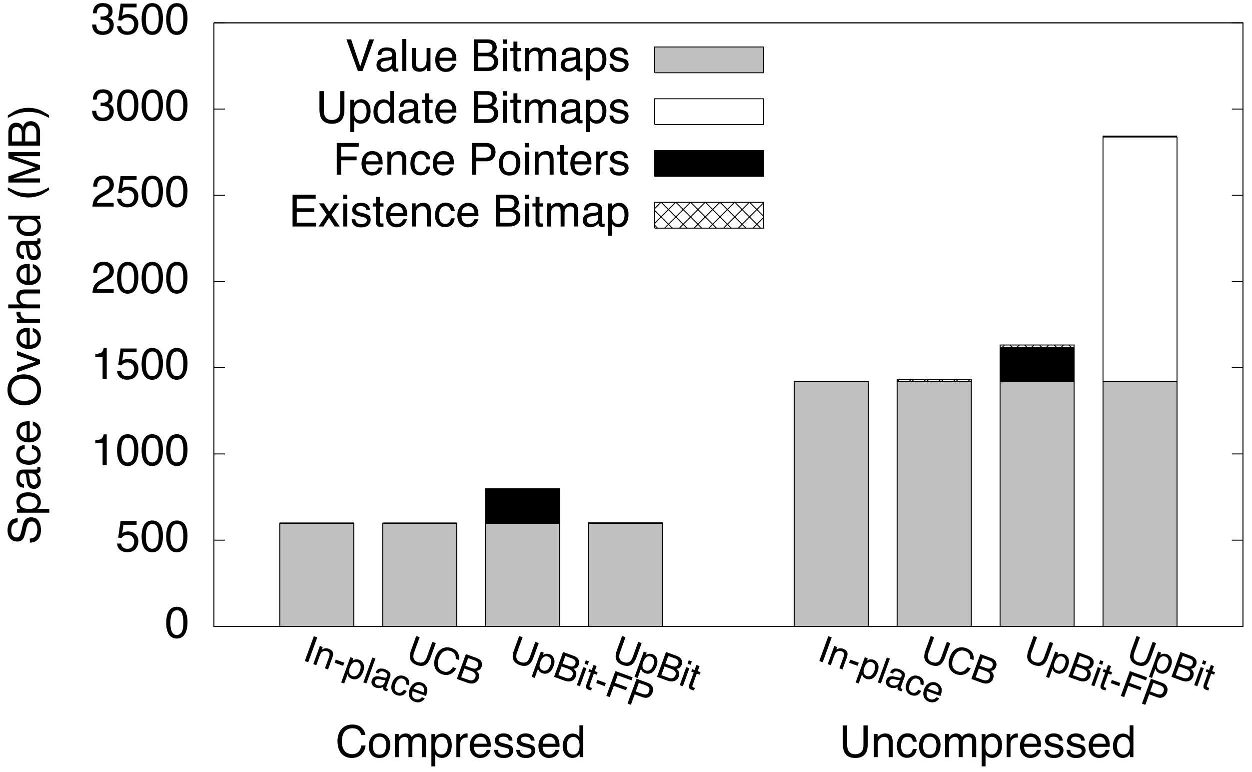
A=10	UB	A=20	UB	A=30	UB
0	0	0	0	1	0
0	1	1	1	0	0
0	0	0	0	1	1
1	0	0	0	0	0
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0

… row 3

1. find value of row 3 (A=30)
   using fence pointers
2. flip bit of row 3 of UB of A=30

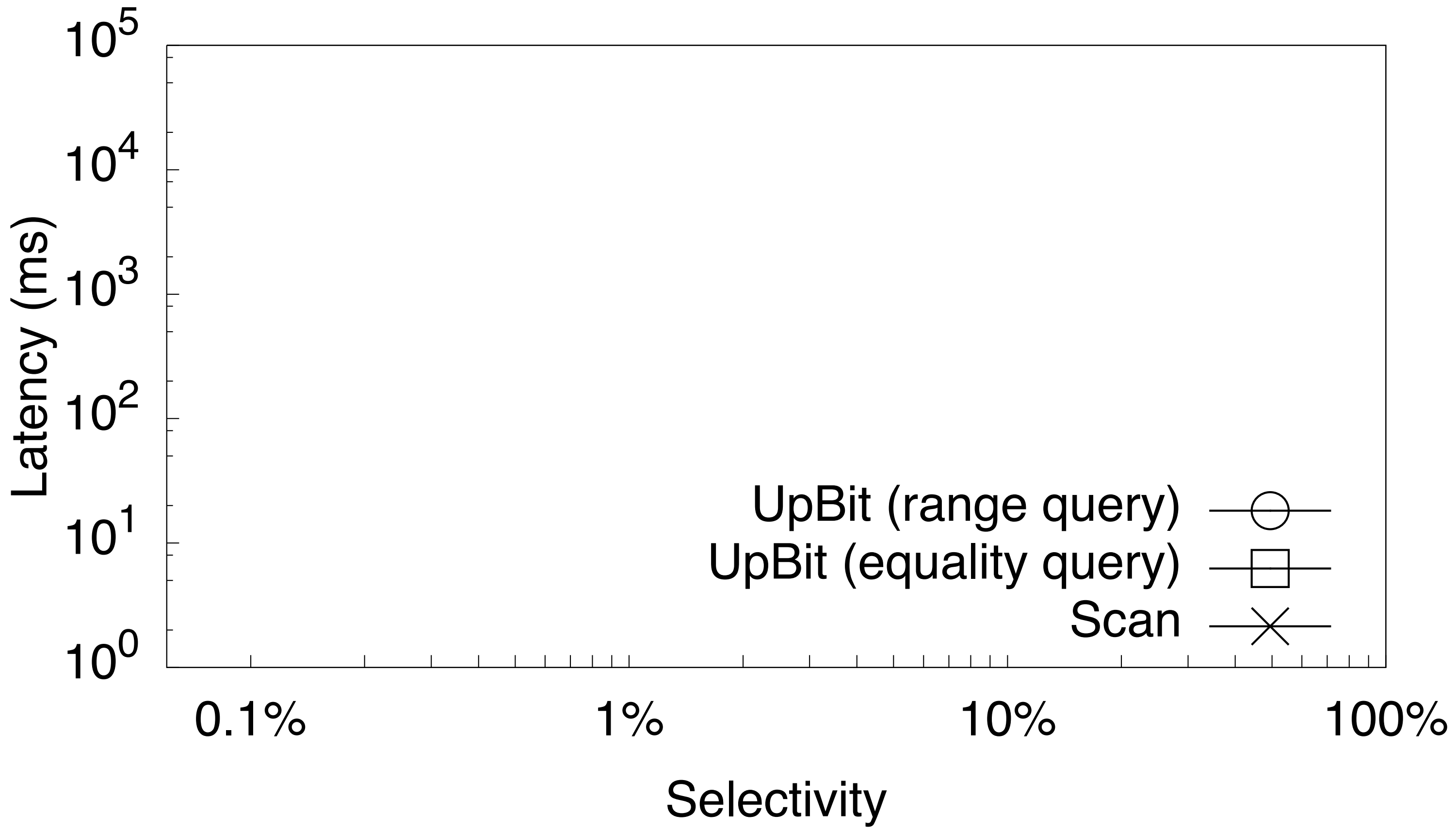
Memory Consumption

n = 100M, d = 100 distinct domain values



UpBit as a general index: UpBit vs. Scan

n = 1B, d = 1000 distinct domain values (range)
n = 1B, d varies for equality: 1000,100,10,1



Scan
tight for-loop
SIMD
multi-core

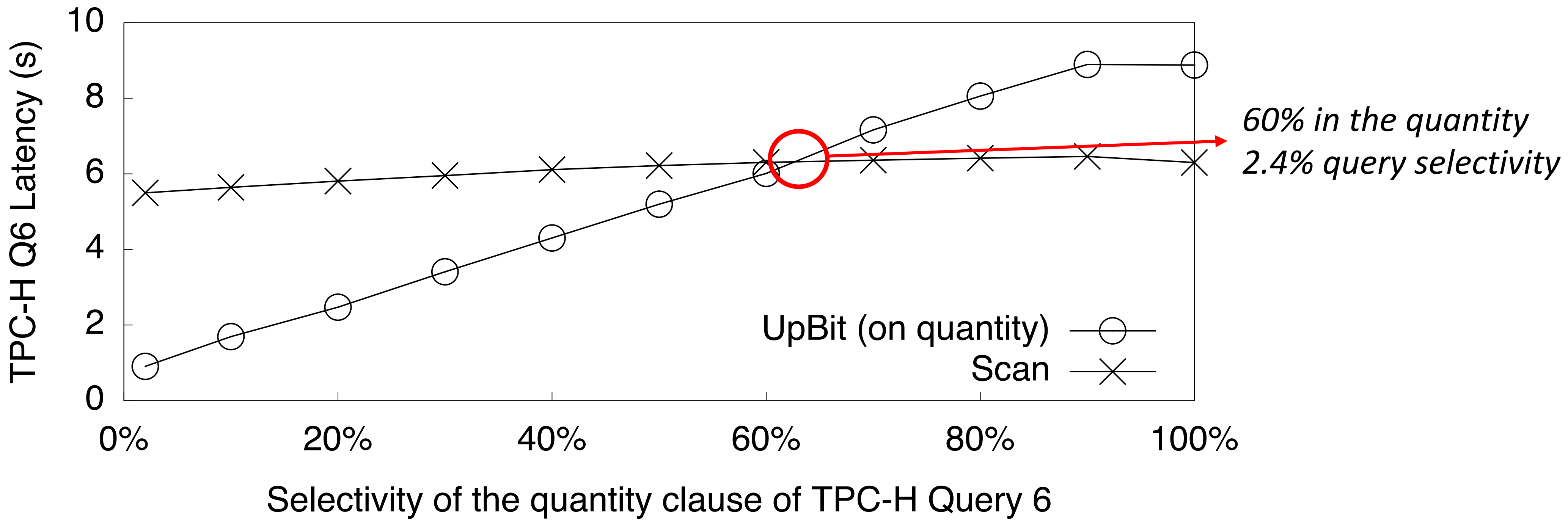
Equality query
all qualifying tuples have the same value (always 1 bitvector)

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

break-even: 10% for equality queries and 1% for range queries

UpBit as a general index: UpBit vs. Scan TPCH Q6

TPCH SF 100 (overall query selectivity 0.08% - 3.9%)



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***