

Fall 2013

BITMAP INDEXING

Motivation

Consider the following table:

```
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?  
);
```

In the past, we have used a B+-tree for the uid and the zip values.

In a B+-tree, how many bytes do we use for each record?

Can we do better, i.e. an index with lower storage overhead?
Especially for attributes with small domain cardinalities?

Bit-based indices: Two flavors

a) Bitmap indices and

b) Bitslice indices

Bitmap Indices

- Consider building an index to answer equality queries on the **retweet** attribute
- Issues with building a B-tree:
 - Three distinct values: True, False, NULL
 - Lots of duplicates for each distinct value
 - Sort of an odd B-tree with three long rid lists
- Bitmap Index: Build three bitmap arrays (stored on disk), one for each value.
 - The i^{th} bit in each bitmap correspond to the i^{th} tuple (need to map i^{th} position to a rid)

Bitmap Example

Table (stored in a heapfile)

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,0000,000,000		53705	Y

Bitmap index on "retweet"

R-Yes	R-No
1	0
1	0
0	1
0	0
0	1
...	...
1	0

SELECT * FROM Tweets WHERE retweet = 'N'

1. Scan the R-No Bitmap file
2. For each bit set to 1, compute the tuple #
3. Fetch the tuple # (s)

Critical Issue

- Need an efficient way to compute a bit position
 - Layout the bitmap in page id order.
- Need an efficient way to map a bit position to a record id.

How?

1. If you fix the # records per page in the heapfile
2. And lay the pages out so that page #s are sequential and increasing
3. Then can construct **rid (page-id, slot#)**
 - **page-id** = Bit-position / #records-per-page
 - **slot#** = Bit-position % #records-per-page

Implications
of #1?



Other Queries

Table (stored in a heapfile)

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,0000,000,000		53705	Y

Bitmap index on "retweet"

R-Yes	R-No	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 = 'N'
```

```
SELECT * FROM Tweets WHERE retweet IS NOT NULL
```

Storing the Bitmap index

- One bitmap for each value, and one for Nulls
- Need to store each bitmap
- Simple method: 1 file for each bitmap
- Can compress the bitmap!

Index size?

When is a bitmap index more space efficient than a B+-tree?

Bit-sliced Index: Motivation

(Re)consider the following table:

```
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?  
);
```

```
SELECT * FROM Tweets WHERE zip = 53706
```

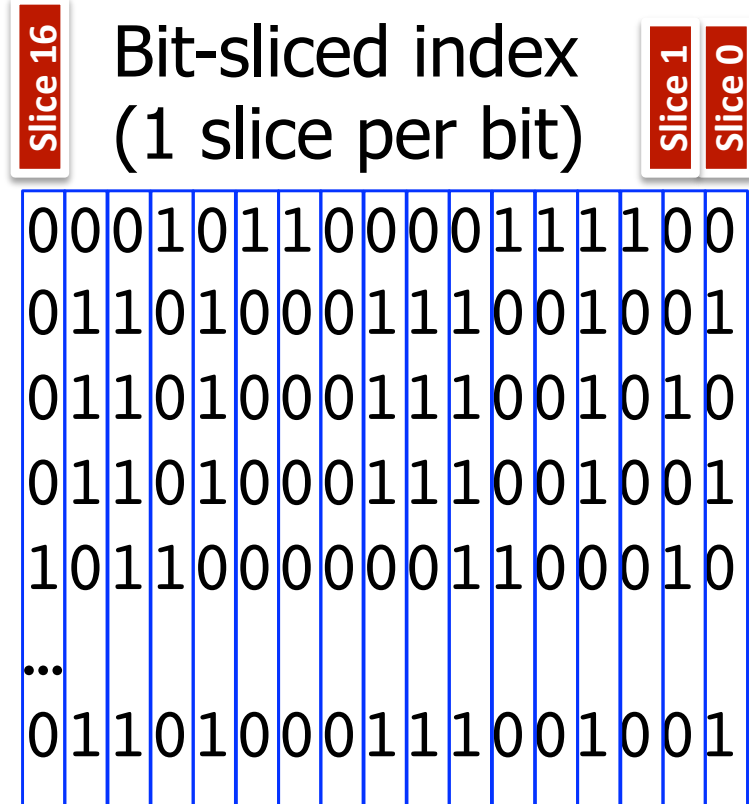
Would we build a bitmap index on zipcode?

Bit-sliced index

Table

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,0000,000,000		53705	Y

Why do we have 17 bits for zipcode?



Query evaluation: Walk through each slice constructing a **result bitmap**

e.g. $\text{zip} \leq 11324$, skip entries that have 1 in the first three slices (16, 15, 14)

Are we missing anything in the bit-sliced index above?

Bitslice Indices

- Can also do aggregates with Bitslice indices
 - E.g. SUM(attr): Add bit-slice by bit-slice.
First, count the number of 1s in the **slice17**, and multiply the count by 2^{17}
Then, count the number of 1s in the **slice16**, and multiply the count by ...
- Store each slice using methods like what you have for a bitmap.
 - Note once again can use compression

Bitmap v/s Bitslice

- Bitmaps better for low cardinality domains
- Bitslice better for high cardinality domains
- Generally easier to “do the math” with bitmap indices