

# **BITMAP INDEXES**

---

*CS 564- Fall 2018*

---

*ACKs: Jignesh Patel, Theo Rekatsinas*

---

# WHAT IS THIS LECTURE ABOUT?

---

- Bitmap Indexes
- Bitslice Indexes

# 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?  
);
```

How can we speed up the following query?

```
SELECT * FROM Tweets  
WHERE zip BETWEEN 53000 AND 54999 ;
```

**B+ tree on attribute zip**

# 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?  
);
```

How many bytes does a B+ tree use for each record?

- at least key + rid, so: **key-size + rid-size**

Can we do better than that (in terms of storage overhead)?

- **yes!** especially when the attribute domain is small

---

# **BITMAP INDEX**

---

# THE BITMAP INDEX

- Consider building an index to answer *equality* queries on the **retweet** attribute
- Issues with building a B+ tree:
  - three distinct values: yes, no, NULL
  - many duplicates for each distinct value
  - a weird B+ tree with three long rid lists
- **Bitmap Index**: build three *bitmap arrays* (stored on disk), one for each value
  - the  $i^{\text{th}}$  bit in each bitmap corresponds to the  $i^{\text{th}}$  tuple  
(we need to map the  $i^{\text{th}}$  position to a rid!)

# BITMAP: EXAMPLE

table (stored in heapfile)

uniqueMsgID	...	zip	retweet
1	...	11324	yes
2	...	53705	yes
3	...	53706	no
4	...	53705	NULL
5	...	90210	no
...	...	...	...
1,000,000,000	...	53705	yes

bitmap index (on retweet)

yes	no	null
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
...	...	...
1	0	0

SELECT \* FROM Tweets WHERE retweet = "no" ;

- scan the "no" bitmap file
- for each bit set to 1, compute the tuple rid
- fetch the tuple

# A CRITICAL ISSUE

- We need an efficient way to compute a bit position:
  - layout the bitmap in page-id order
- We need an efficient way to map a bit position to a rid:
  - fix the # records per page in the heapfile
  - lay the pages out so that page-ids are sequential and increasing
  - then construct **rid (page-id, slot#)**
    - **page-id** = bit-position / #records-per-page
    - **slot#** = bit-position % #records-per-page

With variable length records, we have to set the limit based on the size of the largest record, which may result in under-filled pages!



# BITMAP: OTHER QUERIES

table (stored in heapfile)

uniqueMsgID	...	zip	retweet
1	...	11324	yes
2	...	53705	yes
3	...	53706	no
4	...	53705	NULL
5	...	90210	no
...	...	...	...
1,000,000,000	...	53705	yes

bitmap index (on retweet)

yes	no	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 = "no" ;
```

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

# STORING A BITMAP INDEX

- One bitmap for each value, and one for NULL
- to store each bitmap, use one file for each
- Bitmaps can be compressed!

**index size** = #tuples \* (domain size + 1) *bits*

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

#distinct values < data entry size in the B+-tree

---

# **BITSLICE INDEX**

---

# MOTIVATION

Reconsider 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?  
);
```

with the following query:

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

**Building a bitmap index on zip is not a good idea!**

# BITSLICE INDEX



table (stored in heapfile)

uniqueMsgID	...	zip	retweet
1	...	11324	yes
2	...	53705	yes
3	...	53706	no
4	...	53705	NULL
5	...	90210	no
...	...	...	...
1,000,000,000	...	53705	yes

1 slice per bit  
+ one more slice for the NULL values!

bitslice index

00010110000111100	
01101000111001001	
01101000111001010	
01101000111001001	
10110000001100010	
...	
01101000111001001	

slice 16

slice 0

# BITSLICE INDEX: QUERIES

uniqueMsgID	...	zip	retweet
1	...	11324	yes
2	...	53705	yes
3	...	53706	no
4	...	53705	NULL
5	...	90210	no
...	...	...	...
1,000,000,000	...	53705	yes

00010110000111100	1
01101000111001001	0
01101000111001010	0
01101000111001001	0
10110000001100010	0
...	...
01101000111001001	0

slice 16

slice 0

SELECT \* FROM Tweets WHERE zip <= 11324 ;

- walk through each slice constructing a **result bitmap**
- skip entries that have 1 in the first three slices (16, 15, 14)

---

# OTHER QUERIES

---

- We can also do **aggregates** with Bitslice indices:
  - e.g. SUM(attr): add bit-slice by bit-slice
    - count the number of 1s in **slice 17**, and multiply the count by  $2^{17}$
    - count the number of 1s in **slice 16**, and multiply the count by ...
- We can store each slice using methods like what you have for a bitmap (we can compress again!)

# BITMAP VS BITSlice INDEX

---

- Bitmaps are better for low cardinality domains
- Bitslices are better for high cardinality domains
- It is generally easier to “do the math” with bitmap indices