Bitmaps for the Data Warehouse

Indexes related to data warehousing

- New indexing techniques: Bitmap indexes, Join indexes, array representations, compression, precomputation of aggregations, etc.
- E.g., Bitmap index:

Bit-vector: **1 bit for each possible value.** *Many queries can be answered using bit-vector ops!*

sex	custid	name	sex	rating	rating
10	112	Joe	M	3	00100
10	115	Ram	M	5	00001
01	119	Sue	F	5	00001
10	112	Woo	M	4	00010

Bitmap Indexes

- A bitmap index uses one bit vector (BV) for each distinct keyval
- The number of bits = #rows
- Example of last slide, 4 rows, 2 columns with bitmap indexes
  - Sex = 'M': BV = 1101
  - Sex = 'F': BV = 0010
  - Rating = 3, BV = 1000
  - Rating = 4, BV = 0001
  - Rating = 5, BV = 0110

  Bitmap index for sex column

  Bitmap index for rating column
- Underlying idea: it's not hard to convert between a table's row numbers and the row RIDs
- RIDs have file#, page#, row# within page, where file# is fixed for one heap table, and page# ranges from 0 up to some limit.
- For the kind of read-mostly data that bitmap indexes are used, the pages are full, so the RIDs (page#, row# in a certain file) look like (0,0), (0,1), (0,2), (1,0), (1,1), … easily converted to row indexes 0, 1, 2, 3, 4, 5, … and back again

Bitmap Indexes

- Implementation: B+-tree of key values, bitmap for each key
- Size = #values*#rows/8 if not compressed
- Bitmaps can be compressed, done by Oracle and others
- Main restriction: slow row insert/delete, so NG for OLTP
  - But great for data warehouses:
  - Data warehouses are updated only periodically, traditionally
- Low cardinality (#values in column) a clear fit
  - Example: rating, with 10 values
- But in fact, cardinality can be fairly high with compression
- Oracle example: bitmap index on unique column!

Bitmap Indexes

- Oracle: create bitmap index sexx on custs(sex);
- Bitmap indexes can be used with AND and OR predicates
- Example

  Select name from sailors s

  where s.rating = 10 and sex = 'M' or sex = 'F'

  BV1 BV2 BV3

  ResultBV = BV1 & BV2 | BV3
- Each bit on in ResultBV shows a row that satisfies the predicate
- Loop through on-bits, finding rows and output name

Oracle Bitmap index plan

- EXPLAIN PLAN FOR SELECT * FROM t WHERE c1 = 2 AND c2 <> 6 OR c3 BETWEEN 10 AND 20;
-
- EXPLAIN PLAN FOR
- SELECT * FROM t WHERE c1 = 2 AND c2 <> 6 OR c3 BETWEEN 10 AND 20;
- SELECT STATEMENT
- TABLE ACCESS T BY INDEX ROWID
- BITMAP CONVERSION TO ROWID -- get ROWIDs for each on-bit
- BITMAP OR --top level OR
- BITMAP MINUS --to remove null values of c2
- BITMAP MINUS -- to calc c1 = 2 AND c2 <> 6
- BITMAP INDEX C1_IND SINGLE VALUE --c1 = 2 BV
- BITMAP INDEX C2_IND SINGLE VALUE --c2 = 6 BV
- BITMAP INDEX C2_IND SINGLE VALUE --c2 = null BV (no not null on col)
- BITMAP MERGE --merge BV's over C3 range
- BITMAP INDEX C3_IND RANGE SCAN

Oracle Bitmap join index

```
CREATE BITMAP INDEX sales_cust_gender_bjix ON sales(customers.cust_gender) FROM sales, customers
     WHERE sales.cust_id = customers.cust_id LOCAL;
```

The following query shows a case using this bitmap join index:

```
SELECT sales.time_id, customers.cust_gender, sales.amount
    FROM sales, customers
    WHERE sales.cust_id = customers.cust_id;
```

This Join index has two bitmaps, themselves in the leaves of a little B+-tree:

```
M: 10110001111...   one bit for each row of sales table
F: 01001110000...
```

Here the join is replaced by f_rid to row# to gender lookup using the join index.

```
TIME_ID   C AMOUNT
--------- - ----------
01-JAN-98 M  2291
01-JAN-98 F  114
01-JAN-98 M  553
...
```

Oracle bitmap join indexes for star q's

```
SELECT store.sales_district, time.fiscal_period, SUM(sales.dollar_sales)
 FROM sales, store, time
WHERE sales.store_key = store.store_key AND sales.time_key = time.time_key
 AND store.sales_district IN ('San Francisco', 'Los Angeles') AND
 time.fiscal_period IN ('3Q95', '4Q95', '1Q96')
GROUP BY store.sales_district,time.fiscal_period;
```

- Here, could use a bitmap join index on store.sales_district and another on time.fiscal_period.
- Then Oracle could OR the SF and LA bitmaps, and OR the three fiscal_period bitmaps, then AND the two bit vectors together to obtain a foundset on the fact table.

Bitmaps for star schemas

- Bitmaps can be AND'd and OR'd
- So bitmaps on dimension tables are helpful
- But often not so crucial since dimension tables are often small
- Real problem is dealing with the huge the fact table: that's where the bitmap join indexes come to the rescue.
- Or, alternatively, bitmap indexes on the FK columns.

Bitmaps for star schemas

- The dimension tables are not large, maybe 100 rows
- Thus the FK columns in the fact table have only 100 values
- Bitmap indexes can pinpoint rows once determined.
- Bitmaps can be AND'd and OR'd
- Example: `calendar_quarter_desc IN('1999-01','1999-02')`
- matches say 180 days in time table, so 180 FK values in fact's time_key column
- OR together the 180 bitmaps, get a bit-vector locating all fact rows that satisfy this predicate

Bitmaps for Star Schemas

- OK, so get one bit-vector for matching times, BVT
- Similarly, get another bit-vector for matching stores, BVS
- Another for matching products, BVP

  Result = BVT&BVS&BVP
  - If result has 100 bits on or less, it's a "Needle-in-the-haystack" query, answer in <= 100 i/os, about 1 sec.
  - If result has 10,000 bits on, time <= 100 sec, still tolerable
  - If result has more, this simple approach isn't so great
- Note we can quickly determine the number of results, so count(*) doable even when select … is too costly.

Bitmap steps of star query plan

```
| 9 |       BITMAP CONVERSION TO ROWIDS|
| 10 |       BITMAP AND                 |
| 11 |        BITMAP MERGE              |
| 12 |         BITMAP KEY ITERATION     |
| 13 |          BUFFER SORT             |
|* 14 |           TABLE ACCESS FULL     | CHANNELS
|* 15 |          BITMAP INDEX RANGE SCAN| SALES_CHANNEL_BIX
| 16 |        BITMAP MERGE              |
| 17 |         BITMAP KEY ITERATION     |
| 18 |          BUFFER SORT             |
|* 19 |           TABLE ACCESS FULL     | TIMES
|* 20 |          BITMAP INDEX RANGE SCAN| SALES_TIME_BIX
| 21 |        BITMAP MERGE              |
| 22 |         BITMAP KEY ITERATION     |
| 23 |          BUFFER SORT             |
|* 24 |           TABLE ACCESS FULL     | CUSTOMERS
|* 25 |          BITMAP INDEX RANGE SCAN| SALES_CUST_BIX
| 26 |      TABLE ACCESS BY USER ROWID  | SALES
```