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:

```
<table>
<thead>
<tr>
<th>sex</th>
<th>custid</th>
<th>name</th>
<th>sex</th>
<th>rating</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>112</td>
<td>Joe</td>
<td>M</td>
<td>3</td>
<td>00100</td>
</tr>
<tr>
<td>M</td>
<td>115</td>
<td>Ram</td>
<td>M</td>
<td>5</td>
<td>00001</td>
</tr>
<tr>
<td>F</td>
<td>119</td>
<td>Sue</td>
<td>F</td>
<td>5</td>
<td>00001</td>
</tr>
<tr>
<td>M</td>
<td>112</td>
<td>Woo</td>
<td>M</td>
<td>4</td>
<td>0010</td>
</tr>
</tbody>
</table>
```

Many queries can be answered using bit-vector ops!

**Bitmap Indexes**

• 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
• 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

• Oracle: create bitmap index
  `sexx` on `custs` (sex);
• Bitmap indexes can be used with AND and OR predicates
• Example
  ```sql
  SELECT name FROM sailors s
  where s.rating = 10 and sex = 'M' or sex = 'F'
  ```
  ```
  ResultBV = BV1 & BV2 | BV3
  ```

• Loop through on-bits, finding rows and output name

**Oracle Bitmap index plan**

```sql
EXPLAIN PLAN FOR SELECT * FROM t WHERE c1 = 2 AND c2 <> 6 OR c3 BETWEEN 10 AND 20;
```
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.

<table>
<thead>
<tr>
<th>TIME_ID</th>
<th>C</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>JAN</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>2291</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>553</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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('1Q95', '2Q95', '3Q95') 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.

• Or, use a bitmap on store.sales_district and another on time.fiscal_period and join on the 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

• OK, so get one b-vector for matching times, BVT
• Similarly, get another b-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/o, about 1 sec.
• If result has 10,000 bits on, time <= 10 sec, still tolerable
• If result has many, 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 INDEX RANGE SCAN | SALES_CHANNEL_BIX |
- |  11 | BITMAP INDEX RANGE SCAN | TIMES |
- |  12 | BITMAP INDEX RANGE SCAN | CUSTOMERS |
- |  13 | BITMAP INDEX RANGE SCAN | SALES_CUST_BIX |
- |  14 | BITMAP INDEX RANGE SCAN | SALES_TIME_BIX |
- |  15 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  16 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  17 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  18 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  19 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  20 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  21 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  22 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  23 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  24 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  25 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |
- |  26 | BITMAP INDEX RANGE SCAN | SALES_STORE_BIX |

• 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
• 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.