Key Compression

Macneil Fernandes©2005

Key Compression

- Key compression *enables compression* of portions of the key column values in an *index*.
- This reduces the storage overhead of repeated values.
- Keys in an index have two pieces
  - **a grouping piece***(Repeating part of the key)*
  - **a unique piece.(***If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece*.**)**
- Key compression is a method of breaking off the *grouping piece* and storing it so it can be *shared by multiple unique pieces*.

Macneil Fernandes©2005

Prefix and Suffix Entries

- Key compression breaks the index key into a *prefix entry* (the grouping piece) and a *suffix entry* (the unique piece).
- Compression is achieved by sharing the *prefix entries* among the suffix entries in an index block.
- For example,
  - **In a key made up of three columns (column1, column2, column3) the default prefix is (column1, column2). For a list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of (1,2), (1,3) in the prefix are compressed.**
- Also one can specify the length of the prefix, *(i.e. number of columns to be included in the prefix.)*
- For example,
  - **If you specify prefix length 1, then the prefix is column1 and the suffix is (column2, column3). For the list of values (1,2,3), (1,2,4), (1,2,7),(1,3,5), (1,3,4), (1,4,4) the repeated occurrences of 1 in the prefix are compressed.**

Macneil Fernandes©2005

Performance and Storage Considerations

- Key compression leads to
  - **A huge saving in space (stores more keys in each index block)**
  - **Less I/O and**
  - **Better performance.**
  - **Increase in the CPU time required to reconstruct the key column values during an index scan.**
  - **It also incurs some additional storage overhead, because every prefix entry has an overhead of 4 bytes associated with it.**

Macneil Fernandes©2005

Implementing Key Compression

- Key compression can be useful in the following situations:
  - **For a non-unique index to which `ROWID` is appended to make the key unique. The duplicate key is stored as a prefix entry on the index block without the `ROWID`. The remaining rows become suffix entries consisting of only the `ROWID`.**
  - **For a unique multi-column index.**
- Enable key compression using the *`COMPRESS`* clause. The prefix length (as the number of key columns) can also be specified to identify how the key columns are broken into a prefix and suffix entry.

```
CREATE INDEX emp_ename ON emp(ename)
TABLESPACE users
COMPRESS 1;
```

- The `COMPRESS` clause can also be specified during rebuild. For example, during rebuild you can disable compression as follows:

```
ALTER INDEX emp_ename REBUILD NOCOMPRESS;
```

Macneil Fernandes©2005

Bitmap Indexes

Macneil Fernandes©2005

Bitmap Indexes

- In a ***bitmap index***, a *bitmap for each key value* is used *instead of a list of rowids*.
- Each bit in the bitmap corresponds to a possible rowid.
- If the bit is set, then it means that the row with the corresponding rowid contains the key value.
- A *mapping function* converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a regular index even though it uses a different representation internally.

Macneil Fernandes©2005

Bitmap Indexes

- The advantages of using bitmap indexes are greatest for low ***cardinality columns***: *(i.e. columns in which the number of distinct values is small compared to the number of rows in the table.)*
- If the number of distinct values of a column is less than 1% of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index.
- For example,
  - **On a table with 1 million rows, a column with 10,000 distinct values is a candidate for a bitmap index.**
- Even columns with a lower number of repetitions and thus higher cardinality can be candidates if they tend to be involved in complex conditions in the `WHERE` clauses of queries.

Macneil Fernandes©2005

Bitmap Index Example

- Consider the Table Given Below:
- From the Table Data we find the low cardinality columns to be:
  - **MARITAL_STATUS (three possible values),**
  - **REGION(three possible values),**
  - **GENDER(two possible values),**
- Therefore, it is appropriate to create bitmap indexes on these columns.

CUSTOMER #	MARITAL_STATUS	REGION	GENDER
101	single	east	male
102	married	central	female
103	married	west	female
104	divorced	west	male
105	single	central	female
106	married	central	female

Macneil Fernandes©2005

Bitmap Index Example

- The Figure below illustrates the *Bitmap index* for the `REGION` column in this example.
- It consists of three separate bitmaps, one for each region.
- Each entry or bit in the bitmap corresponds to a single row of the `CUSTOMER` table.
- The value of each bit depends upon the values of the corresponding row in the table.

101	single	east	male

The bitmap `REGION='east'` contains a one as its first bit. This is because the region is `east` in the first row of the `CUSTOMER` table.

REGION= 'east'	REGION= 'central'	REGION= 'west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

The bitmap REGION='east' has a zero for its other bits because none of the other rows of the table contain east as their value for REGION.

Macneil Fernandes©2005

Bitmap Index Example

- Similarly we can represent the *Bitmap index* for the GENDER column.

CUSTOMER #	GENDER
101	male
102	female
103	female
104	male
105	female
106	female

GENDER = ‘male’	GENDER = ‘female’
1	0
0	1
0	1
1	0
0	1
0	1

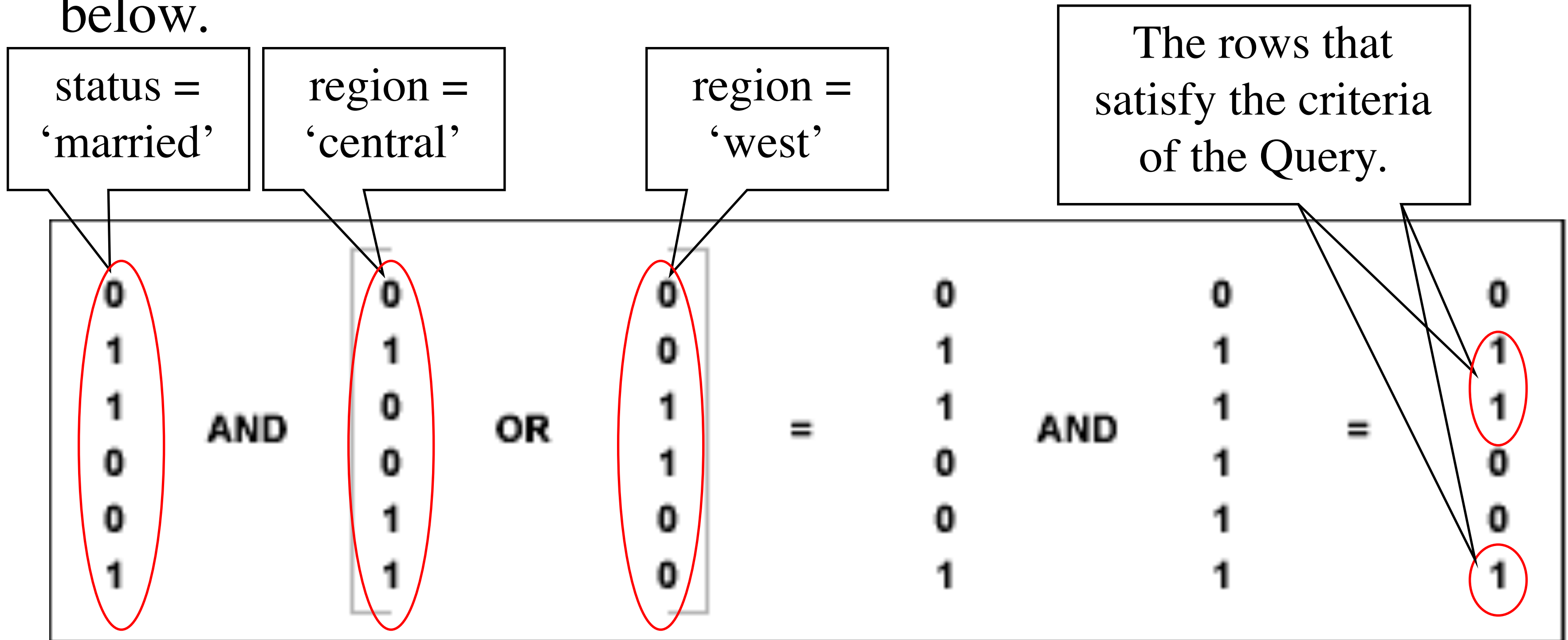
- Consider the SQL Query:

```
SELECT COUNT(*) FROM CUSTOMER
 WHERE marital_status = 'married'
  AND region IN ('central','west');
```

Macneil Fernandes©2005

Bitmap Index Example *(Executing a Query Using Bitmap Indexes)*

- Bitmap indexes can process this query with great efficiency by counting the number of ones in the resulting bitmap, as illustrated below.



- Result of the Query:

COUNT(*)

3