6
Indexes

This chapter tells how to use indexes in a data warehousing environment, and discusses the following types of index:

- Bitmap Indexes
- B-tree Indexes
- Local Versus Global

Bitmap Indexes

Bitmap indexes are widely used in data warehousing applications, which have large amounts of data and ad hoc queries but a low level of concurrent transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries
- A substantial reduction of space usage compared to other indexing techniques
- Dramatic performance gains even on hardware with a relatively small number of CPUs or small amount of memory
- Very efficient maintenance during parallel DML and loads

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Attention:

Bitmap indexes are available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for more information about the features available in Oracle8*i* and the Oracle8*i* Enterprise Edition.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. In a regular index, this is achieved by storing a list of rowids for each key corresponding to the rows with that key value. 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, and if the bit is set, 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. If the number of different key values is small, bitmap indexes are very space efficient.

Bitmap indexes are most effective for queries that contain multiple conditions in the WHERE clause. Rows that satisfy some, but not all conditions are filtered out before the table itself is accessed. This improves response time, often dramatically.

Benefits for Data Warehousing Applications

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data. These indexes are primarily intended for decision support (DSS) in data warehousing applications where users typically query the data rather than update it.

Parallel query and parallel DML work with bitmap indexes as with traditional indexes. (Bitmap indexes on partitioned tables must be local indexes; see "Index Partitioning" for more information.) Parallel create index and concatenated indexes are also supported.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. A gender column, which only has two distinct values (male and female), is ideal for a bitmap index. However, data warehouse administrators will also choose to build bitmap indexes on columns with much higher cardinalities.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B-tree index, particularly when this column is often queried in conjunction with other columns.

B-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as CUSTOMER_NAME or PHONE_NUMBER. In a data warehouse, B-tree indexes should only be used for unique columns or other columns with very high cardinalities (that is, columns that are almost unique). The majority of indexes in a data warehouse should be bitmap indexes.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be quickly resolved by performing the corresponding boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example

Table 6-1 shows a portion of a company's customer data.

Table 6-1 Bitmap Index Example

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

Because MARITAL_STATUS, REGION, GENDER, and INCOME_LEVEL are all low-cardinality columns (there are only three possible values for marital status and region, two possible values for gender, and four for income level), bitmap indexes are ideal for these columns. A bitmap index should not be created on CUSTOMER# because this is a unique column. Instead, a unique B-tree index on this column in order would provide the most efficient representation and retrieval.

Table 6-2 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

Table 6-2 Sample Bitmap

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

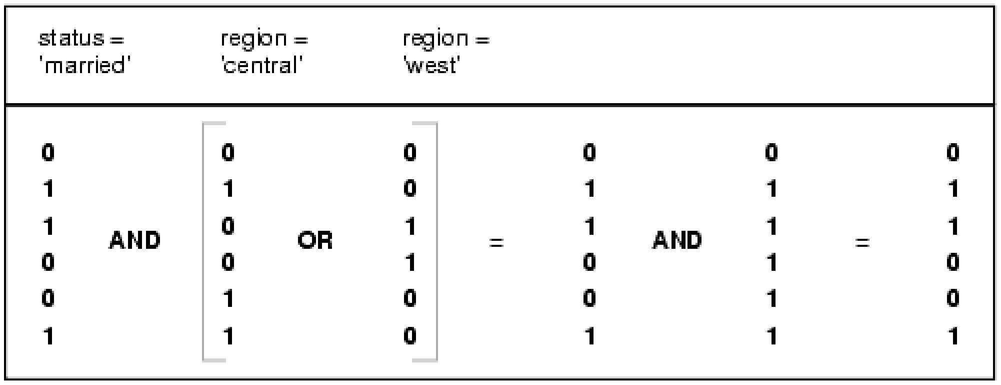
Each entry (or *bit*) in the bitmap corresponds to a single row of the CUSTOMER table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap REGION='east' contains a one as its first bit: this is because the region is "east" in the first row of the CUSTOMER table. 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.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers live in the central or west regions?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM customer
    WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

Bitmap indexes can process this query with great efficiency by merely counting the number of ones in the resulting bitmap, as illustrated in Figure 6-1. To identify the specific customers who satisfy the criteria, the resulting bitmap would be used to access the table.

Figure 6-1 Executing a Query Using Bitmap Indexes



Bitmap Indexes and Nulls

Bitmap indexes include rows that have NULL values, unlike most other types of indexes. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function COUNT.

Example

```
SELECT COUNT(*) FROM emp;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have NULL data. If nulls were not indexed, the optimizer would only be able to use indexes on columns with NOT NULL constraints.

Bitmap Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables. The only restriction is that bitmap indexes must be local to the partitioned table--they cannot be global indexes. (Global bitmap indexes are supported only on nonpartitioned tables).

B-tree Indexes

A B-tree index is organized like an upside-down tree. The bottommost level of the index holds the actual data values and pointers to the corresponding rows, much like the index in a book has a page number associated with each index entry. See *Oracle8i Concepts* for an explanation of B-tree structures.

In general, you use B-tree indexes when you know that your typical query refers to the indexed column and retrieves a few rows. In these queries, it is faster to find the rows by looking at the index. However, there is a tricky issue here--return to the analogy of a book index to understand it. If you plan to look at every single topic in a book, you might not want to look in the index for the topic and then look up the page. It might be faster to read through every chapter in the book. Similarly, if you are retrieving most of the rows in a table, it might not make sense to look up the index to find the table rows. Instead, you might want to read or scan the table.

B-tree indexes are most commonly used in a data warehouse to index keys which are unique or near-unique. In many cases, it may not be necessary to index these columns in a data warehouse, because unique constraints can be maintained without an index, and since typical data warehouse queries may not yield better performance with such indexes. Bitmap indexes should be more common that B-tree indexes in most data warehouse environments.

Local Versus Global

A B-tree index on a partitioned table can be local or global. Global indexes must be fully rebuilt after a direct load, which can be very costly when loading a relatively small number of rows into a large table. For this reason, it is strongly recommended that indexes on partitioned tables should be defined as local indexes unless there is a well-justified performance requirement for a global index. Bitmap indexes on partitioned tables are always local. See "Partitioning Data" for further details.

Library Product Contents Index