Working Of Indexes Oracle 10g

Navpreet singh
MINT, India
Paras Mehra
Assistant Professor, BBK DAV College Amritsar
Ankur Sharma
Assistant Professor, BBK DAV College Amritsar

Abstract:
Oracle 10g is one of the best Relational Database Management System used for managing the database in an effective way. One of the important tasks in database management is to improve the query performance. Oracle includes number of data structures to fast or improve the queries. One of them is indexes. Indexes are provide a faster access of tables. Oracle 10g uses indexes to avoid the full table scans. Oracle provides several indexes those improve the performance of the queries.

Keywords: Index, B-tree Index, Bitmap Index, Database, RDBMS.

General Terms:

1. Introduction:

Database is commonly defined as collection of files or table. DBMS stands for Database Management System which is collection of unified programs used to manage overall activities of the database. Oracle is one of the most commonly used commercial relational database systems. In Oracle provides several indexes which improve the query performance. Indexes are optional structures which are associated with tables and clusters also. In oracle, Database index provides a faster access of data retrieving from the tables. There is no difference in result if you do not use the index, but performance time is less with indexes.

In Oracle indexes are

1. Domain index: specifically for an application or cartridge.
2. Global and local index: related to partitioned tables.
3. Reverse key index: mostly used for RAC (Real Application Clusters).
4. Bitmap index: this is compact index, specially for columns with a small set of values
5. Function-based indexes: contain the pre-computed value of a function/expression
6. Hash cluster index: specifically for a hash cluster
7. B-tree index: the most common and default index
8. B-tree cluster index: specifically used for clusters.

Indexes are physically independent from the table to which they are associated. Thats the region they require some storage space. User can create or drop an index on the eligible columns of the table without affecting the table data. Oracle maintains indexes automatically when user insert, update, and delete rows in the table. If user create or drop an index, the work is continued without any effect but the difference is data might be slower without index.

2. Comparing the Indexes

In oracle we do not prefer to use a bitmap index on a unique column. The size of the bitmap index depends on the cardinality of the columns on which it is created. A bitmap index on the MARITAL_STATUS column will be shorter than a B-tree index on the same column. A bitmap index on ENO (primary key) will be much larger than a B-tree index on this column. But because fewer users access decision-support systems (DSS) systems than would access transaction-processing (OLTP) ones, resources are not a problem for these applications.

To prove this point we will create two tables DEMO_NORMAL AND DEMO_RANDOM.
The DEMO_NORMAL is organized table and that the DEMO_RANDOM is randomly created table. In both tables, column ENO has 100-percent distinct values and is eligible for primary key constraint. If you define ENO column as a primary key then it will a B-tree index. Because oracle does not support bitmap index as a primary key index.

To analyze this, we will perform the following steps:
1. On DEMO_NORMAL:
   1.1. First you create a bitmap index on the ENO column and then execute queries.
   1.2. Secondly you create a B-tree index on the ENO column and execute queries, and then compare queries to fetch the results for different sets of values.
2. On DEMO_RANDOM:
   2.1. Same as Step 1.1.
   2.2. Same as Step 1.2.
3. On DEMO_NORMAL:
   3.1. Same as Step 1.1, the queries are executed with in some range.
   3.2. Same as Step 1B, the queries are executed within some range. And then compare the results.
4. On DEMO_RANDOM:
   4.1. Same as Step 3.1.
   4.2. Same as Step 3.2.

Step 1.1 (on DEMO_NORMAL)
In this section, we will create a bitmap index on the DEMO_NORMAL table and then check for the size of the table and the size of the index with clustering factor. We will execute some queries and note the result of these queries in which bitmap index is used.

```sql
1 create bitmap index demo_normal_end on DEMO_normal(eno)
   size /
   index created.
Elapsed: 01:00:19.00
SQL> end
SQL> get db
1 analyze table demo_normal compute statistics for table for all indexes for all indexed columns
   size /
   table analyzed.
Elapsed: 01:00:19.00
1 select substr(segment_name,1,20) segment_name, bytes/1024/1024 "Size in MB"
   where segment_name in ('DEMO_NORMAL','demo_normal.end')
   size /
   SEGMENT_NAME ----------------- Size in MB
   DEMO_NORMAL ----------------- 44
   demo_normal.end --------------- 44
Elapsed: 01:00:59.00
```

see In the above query show us the size of the bitmap index is 44MB.

```
SQL> select * from demo_normal where eno=500;
```

**Execution Plan**

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
<td>3 (0)</td>
<td>0.00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEMO_NORMAL</td>
<td>1</td>
<td>34</td>
<td>3 (0)</td>
<td>0.00:00:01</td>
</tr>
<tr>
<td>* 3</td>
<td>BITMAP INDEX SINGLE VALUE</td>
<td>DEMO_NORMAL_END</td>
<td>1</td>
<td>34</td>
<td>3 (0)</td>
<td>0.00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

```
  3 - access("ENO"=500)
```

**Statistics**

- 0 recursive calls
- 0 db block gets
- 0 consistent gets
- 0 physical reads
- 0 redo size
- 558 bytes sent via SQL*Net to client
- 558 bytes received via SQL*Net from client
- 2 SQL*Net roundtrips to/from client
- 0 sorts (memory)
- 0 sorts (disk)
- 1 rows processed
Step 1.2 (on DEMO_NORMAL)

In this section we will drop this bitmap index and create a B-tree index on the ENO column and then check for the size of the bitmap index.

```
SQL> create index demo_normal_btree on demo_normal(eno);
Index created.
SQL> analyze table demo_normal compute statistics for table for all indexes for all indexed columns;
Table analyzed.
```

```
SQL> select substr(segment_name,1,30) segment_name, bytes/1024/1024 "Size in MB"
where segment_name in ('DEMO_NORMAL','DEMO_NORMAL_BTREE');
```

```
SEGMENT_NAME | Size in MB
--------------|-------------
DEMO_NORMAL   | 44          
DEMO_NORMAL_BTREE | 18          
```

```
SQL> select index_name, clustering_factor from user_indexes;
```

```
INDEX_NAME | CLUSTERING_FACTOR
-------------|-------------------
DEMO_NORMAL_BTREE | 5524          
```

```
SQL>
```

Form above query the size of bitmap index is 18 and bitmap index is 44. So b-tree index is smaller than the bitmap index. Now we’ll run the same queries, using our B-tree index.

```
SQL> select * from DEMO_normal where eno=1000;
Enter value for eno: 1000
old 1: select * from DEMO_normal where eno=1000
new 1: select * from DEMO_normal where eno=1000
```

```
Execution Plan
Stamp: 2012-12-17-17:30:45
 predictors: 10000000000000000
 1  SELECT STATEMENT
    1  TABLE ACCESS BY INDEX ROWID DEMO_NORMAL
6  TABLE ACCESS BY INDEX ROWID DEMO_NORMAL_BTREE
```

```
Predicate Information (identified by operation id):

2  - access("ENO"=1000)
```

```
Statistics
14  recursive calls
 5  db block gets
 5  consistent gets
 0  physical reads
 0  redo size
550 bytes sent via SQL*Net to client
381 bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
1  rows processed
```

From the above query you can see, when the queries are executed for different set of values, the number of consistent gets and physical reads are same for bitmap and B-tree indexes on a unique column ENO.
Step 2.1 (on DEMO_RANDOM)

In this section we’ll perform same queries on DEMO_RANDOM:

1. SQL create bitmap index random_eno_bit on demo_random(eno);
   Index created.

2. SQL analyze table demo_random compute statistics for table for all indexes for all indexed columns;
   Table analyzed.

3. SQL select substring(segment_name,1,30) segment_name, bytes/1024/1024 'Size in MB'
   from user_segments
   where segment_name in ('DEMO_RANDOM','RANDOM_ENO_BIT');

4. SQL select index_name, clustering_factor from user_indexes;

<table>
<thead>
<tr>
<th>SEGMENT_NAME</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMO_RANDOM</td>
<td>45</td>
</tr>
<tr>
<td>RANDOM_ENO_BIT</td>
<td>20</td>
</tr>
</tbody>
</table>

5. SQL select index_name, clustering_factor from user_indexes;

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>CLUSTERING_FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_ENO_BIT</td>
<td>10000000</td>
</tr>
<tr>
<td>DEMO_NORMAL_BTREE</td>
<td>5524</td>
</tr>
</tbody>
</table>

Step 2.2 (on DEMO_RANDOM)

In this section, we will repeat step 1.1, we will drop the bitmap index from ENO column and after that we create a B-tree index on the ENO column.

1. SQL create index random_eno_bit on demo_random(eno);
   Index created.

2. SQL analyze table demo_random compute statistics for table for all indexes for all indexed columns;
   Table analyzed.

3. SQL select substring(segment_name,1,30) segment_name, bytes/1024/1024 'Size in MB'
   from user_segments
   where segment_name in ('DEMORANDOM', 'RANDOM_ENO_BIT');

<table>
<thead>
<tr>
<th>SEGMENT_NAME</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMO_RANDOM</td>
<td>45</td>
</tr>
<tr>
<td>RANDOM_ENO_BIT</td>
<td>18</td>
</tr>
</tbody>
</table>

4. SQL select index_name, clustering_factor from user_indexes;

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>CLUSTERING_FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_ENO_BIT</td>
<td>99909909</td>
</tr>
<tr>
<td>DEMO_NORMAL_BTREE</td>
<td>5524</td>
</tr>
</tbody>
</table>
Form above query it shows that the size of the index is nearly equal to the size of the index on DEMO_NORMAL table.

Above query show as, the results are almost identical to which we perform in section 1.1 and 1.2. The data distribution did not affect the amount of consistent gets and physical reads for a ENO column(unique column).
Step 3.1(on DEMO_NORMAL)

In this section, we will create the bitmap index on Eno column and run some queries within range.

```
SQL> select * from demo_normal where eno between Range1 and Range2;
Enter value for range1: 1
Enter value for range2: 2000
old 1: select * from demo_normal where eno between Range1 and Range2
new 1: select * from demo_normal where eno between 1 and 2000
2000 rows selected.
Elapsed: 00:00:00.06
```

```
Execution Plan

Plan hash value: 608122099

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT STATEMENT</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>370 (   0)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>2</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>370 (   0)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>370 (   0)</td>
<td>00:00:06</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

- access("ENO")=1 and "ENO"<2000

Step 3.2(on DEMO_NORMAL)

In this section, we will create the B-Tree index on Eno column and run some queries within range.

```
SQL> select * from demo_normal where eno between Range1 and Range2;
Enter value for range1: 1
Enter value for range2: 2000
old 1: select * from demo_normal where eno between Range1 and Range2
new 1: select * from demo_normal where eno between 1 and 2000
2000 rows selected.
Elapsed: 00:00:00.06
```

```
Execution Plan

Plan hash value: 3951899871

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>SELECT STATEMENT</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>19 (   0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>9</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>19 (   0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>10</td>
<td>INDEX RANGE SCAN</td>
<td>DEMO_NORMAL</td>
<td>1999</td>
<td>67966</td>
<td>19 (   0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

- access("ENO")=1 and "ENO"<2000

Statistics

- recursive calls
  - db block gets
  - consistent gets
  - physical reads
  - redo size
  - bytes sent via SQL*Net to client
  - bytes received via SQL*Net from client
  - SQL*Net roundtrips to/from client
  - sorts (memory)
  - sorts (disk)
  - rows processed
<table>
<thead>
<tr>
<th>BITMAP</th>
<th>ENO(Range)</th>
<th>B-TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent read</td>
<td>Physical read</td>
<td>Consistent read</td>
</tr>
<tr>
<td>287</td>
<td>20</td>
<td>286</td>
</tr>
</tbody>
</table>

From the above table, the number of consistent gets and physical reads with both indexes is again nearly identical.

**Step 4.1(on DEMO_RANDOM)**

In this section, we will create the Bitmap index on DEMO_RANDOM table and run some queries within range. This also shows the impact of clustering factor.

```sql
SQL> select * from demo_random where eno between &range1 and &range2;
Enter value for range1: 1
Enter value for range2: 2000
old 1: select * from demo_random where eno between &range1 and &range2
new 1: select * from demo_random where eno between 1 and 2000

2000 rows selected.

Elapsed: 00:00:00.11

Execution Plan

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEMO_RANDOM</td>
<td>1999</td>
<td>67966</td>
<td>372 (1)</td>
<td>00:00:05</td>
</tr>
<tr>
<td>2</td>
<td>BITMAP INDEX RANGE SCAN</td>
<td>RANDOM_ENO_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

3 - access("ENO")=1 AND "ENO"<2000

Statistics

1 recursive calls
0 db block gets
2142 consistent gets
20 physical reads
0 redo size
95702 bytes sent via SQL*Net to client
1844 bytes received via SQL*Net from client
135 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
2000 rows processed
```

**Step 4.2(on DEMO_RANDOM)**

In this section, we will create the B-Tress index on DEMO_RANDOM table and run some queries within range.
SQL> select * from demo_random where eno between range1 and range2;
Enter value for range1: 1
Enter value for range2: 2000
old 1: select * from demo_random where eno between &range1 and &range2
new 1: select * from demo_random where eno between 1 and 2000

2000 rows selected.

Elapsed: 00:00:00.06

Execution Plan

Plan hash value: 3926125765

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1999</td>
<td>67966</td>
<td>1275 (4)</td>
<td>00:00:16</td>
</tr>
<tr>
<td>*</td>
<td>TABLE ACCESS FULL</td>
<td>DEMO_RANDOM</td>
<td>1999</td>
<td>67966</td>
<td>1275 (4)</td>
<td>00:00:16</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

1 - filter("ENO"<=2000 AND "ENO">1)

Statistics

1 recursive calls
0 db block gets
5678 consistent gets
0 physical reads
0 redo size
87759 bytes sent via SQL*Net to client
1844 bytes received via SQL*Net from client
135 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
2000 rows processed

<table>
<thead>
<tr>
<th>BITMAP</th>
<th>ENO(Range)</th>
<th>B-TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent read</td>
<td>Physical read</td>
<td>Consistent read</td>
</tr>
<tr>
<td>2142</td>
<td>20</td>
<td>1-2000</td>
</tr>
</tbody>
</table>

Conclusion

For the above table ,

1. The bitmap index take 00:00:05 time and B-Tree index take 00:00:16
2. The bitmap index COST(%CPU) is 372 in B-Tree It is 1275
3. The optimizer opted for a full table scan rarely for the bitmap index,
4. The optimizer opted for a full table scan for the B-tree index.

This is due to the clustering factor: The optimizer does consider the value of the clustering factor when generating execution plans using a B-Tree index, But in Bitmap index it does not consider. In this section the bitmap index performs more efficiently than the B-tree index.

References: