When should an index be used?

Christian Antognini
Trivadis AG
Zürich, Switzerland

Introduction

One of the biggest problems tuning a SQL statement or judging if its execution plan is optimal, is to decide which index(es) should be used. Also remember that not all join methods can use the same index(es), therefore using an index sometimes can also force a specific join method to be used.

In my opinion there are only three possible approaches to solve this problem:

- blindly believe that the optimizer has generated the best or at least a good execution plan
- test each combination to determine which is the best
- know how the optimizer works and understand object statistics, thus only a small subset of all possible execution plans should be tested

Practical experience excludes the first approach (except for "simple" SQL statements), because in many cases the optimizer is not able to propose an optimal execution plan. This problem can lead to very poor performance with today's typical databases.

The second approach it is not very practical, so I will not recommend it.

The third is therefore the best one. The problem with this approach is to have some good rules that can drastically reduce the number of tests to be executed.

The aim of this article is to propose a set of rules that can be used to decide if a B*Tree index should be used or not. To reduce complexity only the SELECT statement is discussed, but the same principles are also valid for DML statements. In addition, the following topics are not part of this article:

- partitioned objects
- bitmap indexes
- index organized tables
- object tables

The reader should know how the optimizer works and how to get object statistics. A complete coverage of these subjects and many others is given in our course *SQL-Optimizer and Performance Workshop*.

Terminology

Here are some terms that need an explanation or that have a different meaning from that found in the Oracle documentation.

equality search: is the operation used to get the row(s) of a single key via index, e.g.:

```
SELECT ename
FROM emp
WHERE EMPNO = 1002
```

block gets: the same meaning as *query* in TKPROF and *consistent gets* in SQL*Plus with AUTOTRACE. It is therefore different from the *db block gets* in SQL*Plus with AUTOTRACE.

physical reads: the number of physical disk accesses and not the number of blocks read from disk, e.g. on UNIX is the number of calls to the system call *read*. It is therefore different from *disk* in TKPROF and *physical reads* in SQL*Plus with AUTOTRACE.

Index Statistics

Here is a description of the most important index statistics contained in the data dictionary. If possible, I also give an SQL statement or PL/SQL block to compute the statistics using the table's data. For the statistics derived from other statistics I also give a formula used for the computation. The examples are based on the EMP table and a hypothetical index on the attribute COMM (remember it can be NULL).

ALL_INDEXES.NUM_ROWS: this is the number of rows in the index or NOT NULL values in the referenced table.

```
SELECT count(comm)
FROM emp
```

ALL_INDEXES.DISTINCT_KEYS and **INDEX_STATS.DISTINCT_KEYS**: these are the number of distinct keys in the index, for the primary keys is the same as ALL_INDEXES.NUM_ROWS.

```
SELECT count(unique comm)
FROM emp
```

ALL_INDEXES.BLEVEL and **INDEX_STATS.HEIGHT**: the BLEVEL is the depth of the index (root block not included). The HEIGHT is the height of the index (root block included). Therefore:

$$HEIGHT = BLEVEL + 1$$

HEIGHT is the minimum number of index blocks to be accessed to have a ROWID.

ALL_INDEXES.LEAF_BLOCKS and **INDEX_STATS.LF_BLKS**: these are the number of index leaf blocks. The leaf blocks are the lowest level index blocks, they contain the keys and the ROWIDs.

ALL_INDEXES.CLUSTERING_FACTOR: this value indicates how many adjacent index entries don't refer to the same data block in the table. It is used to compute the cost of an index range scan:

- if the value is near the number of table's blocks, then the table has the same order as the index
  ⇒ good performance because the number of block gets is low
- if the value is near the number of table's rows, then the table and the index are ordered in a completely different way
  ⇒ poor performance because the number of block gets is high

```
DECLARE
  l_clustering_factor  BINARY_INTEGER := 0;
  l_previous_bnr       BINARY_INTEGER := 0;
  l_previous_fnr       BINARY_INTEGER := 0;
BEGIN
  FOR r IN (SELECT dbms_rowid.rowid_block_number(rowid) bnr,
                   dbms_rowid.rowid_to_absolute_fno(rowid, user, 'EMP') fnr
            FROM emp
            ORDER BY comm)
  LOOP
    IF (l_previous_bnr <> r.bnr OR l_previous_fnr <> r.fnr)
    THEN
      l_clustering_factor := l_clustering_factor + 1;
    END IF;
    l_previous_bnr := r.bnr;
    l_previous_fnr := r.fnr;
  END LOOP;
  dbms_output.put_line('clustering_factor = ' || l_clustering_factor);
END;
```

For further information about the clustering factor refer to the article *Performance dank SQL Fine-Tuning* of Urs Meier (see the SOUG Newsletter 1/2000 or www.trivadis.com).

ALL_INDEXES.AVG_DATA_BLOCKS_PER_KEY: this is the average number of data blocks in the table referenced by a single key.

```
SELECT trunc(avg(count(comm)))
FROM (
  SELECT unique comm,
         dbms_rowid.rowid_block_number(rowid) bnr,
         dbms_rowid.rowid_to_absolute_fno(rowid, user, 'EMP') fnr
  FROM emp
  WHERE comm IS NOT NULL
)
GROUP BY comm
```

This value is derived from the other statistics with the following formula:

$$AVG\_DATA\_BLOCKS\_PER\_KEY = GREATEST\left(1, TRUNC\left(\frac{CLUSTERING\_FACTOR}{DISTINCT\_KEYS}\right)\right)$$

ALL_INDEXES.AVG_LEAF_BLOCKS_PER_KEY: this is the average number of leaf blocks that store a single key. This value is derived from the other statistics with the following formula:

$$AVG\_LEAF\_BLOCKS\_PER\_KEY = GREATEST\left(1, TRUNC\left(\frac{LEAF\_BLOCKS}{DISTINCT\_KEYS}\right)\right)$$

INDEX_STATS.BLKS_GETS_PER_ACCESS: reading the description in the Oracle manuals this value seems very interesting. Unfortunately it is quite useless because it is computed with the following formula:

$$BLKS\_GETS\_PER\_ACCESS = \frac{1}{2} + HEIGHT + \frac{ROWS\_PER\_KEY}{2}$$

The biggest problem is that the CLUSTERING_FACTOR is not used for the computation. In the next section I will propose a better calculation for such a value.

ALL_TABLES.BLOCKS: this is the number of blocks below the table high water mark. This is the number of block gets during a full table scan, regardless of whether they contain rows or not. It is not an index statistic but will be used in the next section.

How many Block Gets?

In our course *SQL-Optimizer and Performance Workshop* and in some books you can read about the 95/5 rule:

> **if more than 5% of the table's rows have to be retrieved, it is better to use a full table scan**

Practical experience shows that such a rule is not really useful, because the performance is affected too much by the physical storage of the data and from the hardware. Therefore my proposal is to use the statistics shown in the previous section to compute how many block gets should be done for a specific operation on a specific index and compare it with a full table scan. In the following formulas, except for the full table scan, an index and the referenced table are accessed, e.g. the execution plan is similar to the following one.

```
SELECT STATEMENT
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
    INDEX (RANGE SCAN) OF 'EMP_COMM_I' (NON-UNIQUE)
```

Equality Search: how many block gets are necessary to retrieve all rows via an equality search?

To answer this question let's take a look at the operations that must be executed:

❶ descend the index starting at the root block looking for the leaf block(s) containing the key

❷ scan all leaf block(s) containing the key and get the ROWIDs (same as AVG_LEAF_BLOCKS_PER_KEY)

❸ access the table via the ROWIDs (same as AVG_DATA_BLOCKS_PER_KEY)

Therefore the following formula can be used to compute the number of **B**lock **G**ets per **E**quality **S**earch:

$$BGES = ROUND\left(\overbrace{BLEVEL}^{❶} + \overbrace{GREATEST\left(1, \frac{LEAF\_BLOCKS}{DISTINCT\_KEYS}\right)}^{❷} + \overbrace{GREATEST\left(1, \frac{CLUSTERING\_FACTOR}{DISTINCT\_KEYS}\right)}^{❸}\right)$$

If the blocks are not in the buffer cache, the number of BGES is the same as the number of physical reads.

Full Range Scan: how many block gets are necessary to retrieve all rows of a table via a range scan?

To answer this question let's take a look at the operations that must be executed:

❶ descend the index starting at the root block looking for the leaf block(s) containing the key

❷ scan all leaf blocks and get the ROWIDs

❸ access the table via the ROWIDs

Therefore the following formula can be used to compute the number of **B**lock **G**ets per **F**ull **R**ange **S**can:

$$BGFRS = ROUND\left(\overbrace{BLEVEL}^{❶} + \overbrace{LEAF\_BLOCKS}^{❷} + \overbrace{CLUSTERING\_FACTOR}^{❸}\right)$$

If the blocks are not in the buffer cache, the number of BGFRS is the same as the number of physical reads.

Full Table Scan: how many physical reads are necessary to retrieve all rows via a full table scan?

In the previous section I wrote that the number of block gets for a full table scan is the number of blocks below the high water mark. To perform this operation the multi-block read is used, therefore the following formula can be used to compute the **P**hysical **R**eads per **F**ull **T**able **S**can:

$$PRFTS = CEIL\left(\frac{BLOCKS}{DB\_FILE\_MULTIBLOCK\_READ\_COUNT}\right)$$

The DB_FILE_MULTIBLOCK_READ_COUNT is the INIT.ORA parameter that defines how many blocks are read by each physical read.

Usually during a full table scan the blocks are read from the disk. Exceptions to this are:

- the parameter CACHE has been specified at table level (since Oracle7)
- the hint CACHE has been specified at statement level (since Oracle7)
- the table is smaller than the INIT.ORA parameter CACHE_SIZE_THRESHOLD, usually the default value is 1/10 of the INIT.ORA parameter DB_BLOCK_BUFFERS (only Oracle8)
- the table is smaller than the undocumented INIT.ORA parameter _SMALL_TABLE_THRESHOLD, usually the default value is 1/50 of the INIT.ORA parameter DB_BLOCK_BUFFERS (only Oracle8i)

If at least one of these conditions is true, the blocks can already be in the buffer cache, thus no physical reads are necessary.

For older Oracle versions setting the parameter DB_FILE_MULTIBLOCK_READ_COUNT to a value larger than the default (usually 8)

increased the performance of full table scans. This however is not always the case. Sometimes the performance is almost the same, and sometimes the performance can also decrease.

My suggestion is to use the default value at instance level and only set higher values for specific jobs. If you really want to modify this parameter at instance level, make sure that the performance is really better and also remember that:

- modifying this parameter also modifies the cost of each full table scan
- the maximum I/O-size used by Oracle is usually 1MB
- the maximum I/O-size of the OS can be smaller than 1MB (for example the default on Solaris is 128KB and on Windows 2000 is 256KB)
- if you use RAID-0 or RAID-5 the stripe size must be "compatible" with the Oracle I/O-size

Range Scan Limit: what is the maximum number of rows that should be retrieved via index?

To have good performance the number of physical reads must be minimized. Based on the previous formulas (BGFRS and PRFTS), it is possible to compute the maximum number of rows to retrieve via index, i.e. the **R**ange **S**can **L**imit:

$$RSL = NUM\_ROWS \frac{PRFTS}{BGFRS}$$

This formula returns the RSL when no data is in the buffer cache, i.e. if a maximum of RSL rows are selected the index scan is always better than a full table scan.

Two reasons can lead to a "modification" of the RSL:

- the index and data blocks (except for a full table scan without "cache option" as written in the previous section) tend to be kept in the buffer cache, thus it is possible to also have good performance even when more rows than RSL are selected
- the performance of the disk sub-system, because selecting, for example, via a full table scan at 10MB/s, 50MB/s or 100MB/s require varying times to perform the task

The optimizer itself has the same problem to compute the cost of an execution plan. To solve it the optimizer uses the INIT.ORA parameter OPTIMIZER_INDEX_COST_ADJ to adjust the cost of an index access. The optimizer itself has the same problem to compute the cost of an execution plan. To solve it the optimizer uses the INIT.ORA parameter OPTIMIZER_INDEX_COST_ADJ to adjust the cost of an index access. The default value is 100 percent, lower values decrease the cost proportionally (e.g. setting it to 50 percent makes the index access half as expensive as the default).

Therefore, it is good to use the same parameter for the computation of the RSL.

$$RSL = NUM\_ROWS \frac{PRFTS}{BGFRS \frac{\text{OPTIMIZER\_INDEX\_COST\_ADJ}}{100}}$$

Rules

❶ An index should be used for an equality search when BGES is smaller than PRFTS.

❷ An index should be used for a range scan if the number of rows retrieved is smaller than RSL.

Warning:

- these rules are only valid if the INIT.ORA parameter DB_FILE_MULTIBLOCK_READ_COUNT and OPTIMIZER_INDEX_COST_ADJ are correctly configured (i.e. they must correspond to the real performance of the system)
- these rules are not valid for highly skewed data distribution

Script: spindsta.sql

The script spindsta.sql can be used to display the most important index statistics explained in this article. It can be downloaded from www.trivadis.com. Here an excerpt of a sample output:

```
SQL> @spindsta

Please enter table name, wildcards allowed

e.g.: EMP, emp, E%, e% or %

Table Name <%>: rec%
Hit <RETURN>...

                      Index Statistics for Table RECIPIENT

IndexName          Uni  %NN   IndexKeys   DistKeys      RSL    BGES   PRFTS
------------------ --- ---- ----------- ---------- -------- ------- -------
REC_ADR_ID$MLG_ID YES  100     493,835    493,835      760       4     629
REC_PK             YES  100     493,835    493,835   32,642       4     629
REC_MLG_ID         NO   100     493,835         10   33,697     924     629


PRFTS = Physical Reads per Full Table Scan
RSL   = Range Scan Limit (max. rows# to be retrieved by this index)
BGES  = Blocks Gets per Equality Search (BGES > PRFTS = bad)
```

The table RECIPIENT has 2 interesting indexes. As you can see REC_PK and REC_ADR_ID$MLG_ID have the same number of distinct keys, but the RSL is very different. This is caused by the clustering factor.

```
SQL> SELECT index_name, clustering_factor
  2  FROM user_indexes
  3  WHERE table_name = 'RECIPIENT'

INDEX_NAME                     CLUSTERING_FACTOR
------------------------------ -----------------
REC_ADR_ID$MLG_ID                         403380
REC_PK                                      5030
REC_MLG_ID                                  5030
```

This is a typical case where a rule like the 95/5 cannot be used. In fact the number of block gets generated by an index range scan is very different for these two indexes.

To confirm such a difference you can compare the elapsed time, the physical reads and the consistent gets of the following example:

```
SQL> SET TIMING ON
SQL> SET AUTOTRACE TRACE EXP STAT

SQL> SHOW PARAMETER OPTIMIZER_INDEX_COST_ADJ

NAME                          TYPE    VALUE
----------------------------- ------- ------------------------------------------
optimizer_index_cost_adj      integer 100

SQL> SELECT /*+ index(recipient rec_pk) */ count(unique mlg_id)
  2  FROM recipient;

Elapsed: 00:00:29.02

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=9516 Card=1)
  SORT (GROUP BY)
    TABLE ACCESS (BY INDEX ROWID) OF 'RECIPIENT' (Cost=9516 Card=493835)
      INDEX (FULL SCAN) OF 'REC_PK' (UNIQUE) (Cost=4486 Card=493835)

Statistics
----------------------------------------------------------
          0  db block gets
       9580  consistent gets
       8671  physical reads

SQL> SELECT /*+ index(recipient rec_adr_id$mlg_id) */ count(unique id)
  2  FROM recipient;

Elapsed: 00:01:38.42

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=408780 Card=1)
  SORT (GROUP BY)
    TABLE ACCESS (BY INDEX ROWID) OF 'RECIPIENT' (Cost=408780 Card=493835)
      INDEX (FULL SCAN) OF 'REC_ADR_ID$MLG_ID' (UNIQUE) (Cost=5400 Card=493835)

Statistics
----------------------------------------------------------
        208  db block gets
     408869  consistent gets
      18235  physical reads
```

The optimizer also uses the same formulas to compute the cost. In fact the cost of the two queries is exactly the BGFRS multiplied by OPTIMIZER_INDEX_COST_ADJ of the index (in this case 100/100 = 1). Also notice that the number of consistent gets is close to the BGFRS, i.e. the statistics are good ☺.

```
SQL> SELECT index_name, clustering_factor+blevel+leaf_blocks bgfrs
  2  FROM user_indexes
  3  WHERE table_name = 'RECIPIENT';

INDEX_NAME                       BGFRS
------------------------------ ----------
REC_ADR_ID$MLG_ID                408780
REC_PK                             9516
REC_MLG_ID                         9218
```

Conclusion

A good understanding of object statistics is, with the knowledge of the optimizer, a core requirement to tune SQL statements. The script spindsta.sql can make easier or at least help the tuning process, but remember nothing is as good as a good test…