**Anton Slizh’s**

**U1M4.LW. Access and Join Methods Part 1**

*GitHub: https://github.com/drapejny/DataCamp2022*

**Task 1**

Step 1:

Creating table t2

CREATE TABLE t2 AS

SELECT TRUNC(rownum / 100) id, RPAD(rownum, 100) t\_pad

FROM dual

CONNECT BY rownum < 100000;

Step 2:

Creating index on t2(id)

CREATE INDEX t2\_idx1 ON t2 (id);

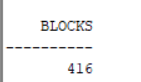
Step 3:

Selecting block count

SELECT blocks

FROM user\_segments

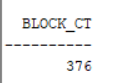
WHERE segment\_name = 'T2';



Selecting used blocks count

SELECT COUNT(DISTINCT (dbms\_rowid.rowid\_block\_number(rowid))) block\_ct

FROM t2;



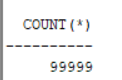
Enable autotrace

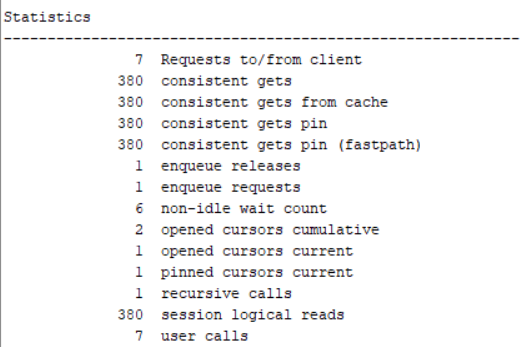
SET AUTOTRACE ON;

Selecting all rows count and show statistics

SELECT COUNT(\*)

FROM t2;





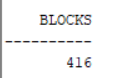
Step 4:

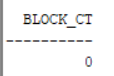
Deleting all rows from t2

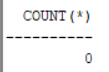
DELETE FROM t2;

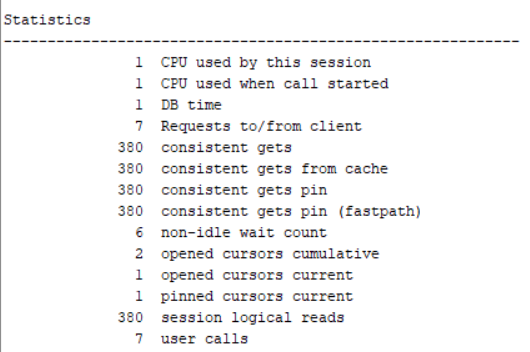
Step 5:

Repeating step 3









Step 6:

Inserting one row into t2 table

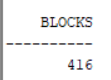
INSERT INTO t2 (ID, T\_PAD)

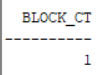
VALUES (1, '1');

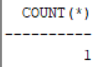
COMMIT;

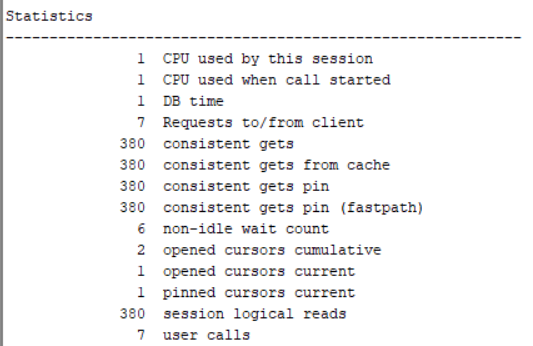
Step 7:

Repeating step 3









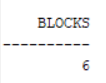
Step 8:

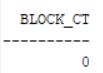
Truncating t2 table

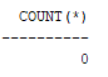
TRUNCATE TABLE t2;

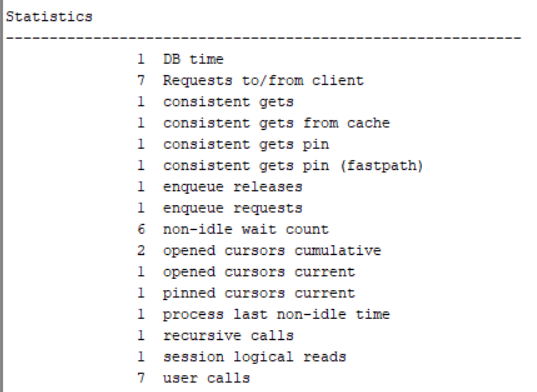
Step 9:

Repeating step 3









Result table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Count of Blocks | Count of Used Blocks | Count of Rows | Consistent gets | Description |
| 1 | 416 | 376 | 99999 | 380 | Selecting on filled table |
| 2 | 416 | 0 | 0 | 380 | Selecting on empty table after *DELETE* operation |
| 3 | 416 | 1 | 1 | 380 | Selecting on table with 1 row |
| 4 | 6 | 0 | 0 | 1 | Selecting on empty table after TRUNCATE operation |

Take a closer look at 4th line in the table. The number of blocks differ from other selections. The main reason is that we used *TRUNCATE* operator to delete all rows from the table. Oracle Database deallocate all space used by removed rows except that specified by the MINEXTENTS storage parameter. MINEXTENTS specify the min number of extents to allocate when the object is created. By default MINEXTENTS parameter equals to 1. As I understand, 6 is the number of blocks in the remaining extent. Obviously, also decreased the number of consistent gets metric.

**Task 2**

Step 1:

Creating table t2 and index t2\_idx1

CREATE TABLE t2 AS

SELECT TRUNC(rownum / 100) id, RPAD(rownum, 100) t\_pad

FROM dual

CONNECT BY rownum < 100000;

CREATE INDEX t2\_idx1 ON t2 (id);

Step 2:

Creating table t1

CREATE TABLE t1 AS

SELECT MOD(rownum, 100) id, rpad(rownum, 100) t\_pad

FROM dual

CONNECT BY rownum < 100000;

Step 3:

Creating index t1\_idx1

CREATE INDEX t1\_idx1 ON t1(id);

Step 4:

Calculating statistics for both tables

EXEC dbms\_stats.gather\_table\_stats( USER, 't1',method\_opt => 'FOR ALL COLUMNS SIZE 1', CASCADE => TRUE);

EXEC dbms\_stats.gather\_table\_stats( USER, 't1',method\_opt => 'FOR ALL COLUMNS SIZE 1', CASCADE => TRUE);

Step 5:

Selecting clustering factor

SELECT RPAD(t.table\_name || '.' || i.index\_name, 10) idx\_name,

i.clustering\_factor,

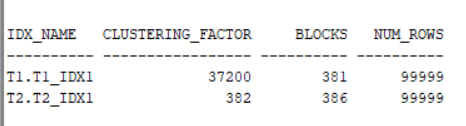
t.blocks,

t.num\_rows

FROM user\_indexes i, user\_tables t

WHERE i.table\_name = t.table\_name

AND t.table\_name IN( 'T1','T2' );



The clustering factor tells you the number of blocks that will be read when scanning the index. In other words, how many I/O operations would be performed if every row in the table would be read by using the index, following the index order.

For the *T2* table the *CLUSTERING\_FACTOR (382)* is near the number of blocks (386) in the table. Which means that rows in the table are well ordered by index column (id). In other words, when we scanning all table by index, we don’t need to load each block for each row. We see that next rows are located at the same block and we don’t need load this block again.

For the *T1* table the *CLUSTERING\_FACTOR (37200)* is near the number of rows (99999) in the table. Which means that every time we access the row, we need load another block, because rows are not ordered in the disk space (according to index column (id)) and, if we scan all table with index, we need to load new block almost every time.

As I understand, best selective performance in executing *SELECT* clause filtered by *IN(…)* will be at index with better *CLUSTERING\_FACTOR* value. For some values we can read whole block with proper rows at one time, rather than loading one block per one row.

**Task 3**

Step 1:

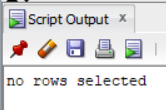
Creating unique index on t1(t\_pad)

CREATE UNIQUE INDEX udx\_t1 ON t1( t\_pad );

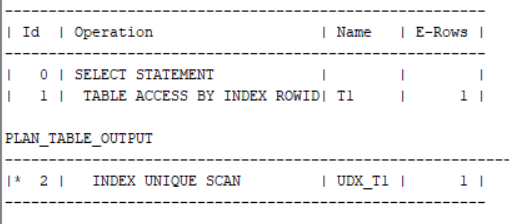
Step 2:

Selecting table t1 using unique index

SELECT t1.\* FROM t1 where t1.t\_pad = '1';

****

Execution plan



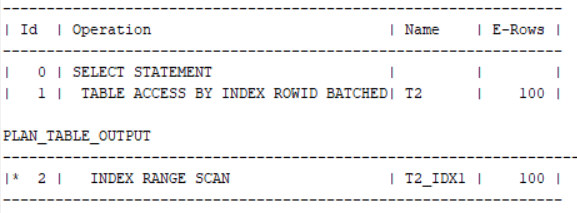
First, we created *UNIQUE INDEX* on *t\_pad* column, which ensures that no two rows have duplicate values in the *t\_pad* column. Oracle use index represented as B-Tree to fast find specified value (t\_pad = 1). Scanning starts from the root element of the tree and searching down to the leaves. At the execution plan this process represented as *INDEX UNIQUE SCAN.* After value was found or not, Oracle extracts rowId and loads block with specified row. At the execution plan this process represented as *TABLE ACCESS BY INDEX ROWID.*

**Task 4**

Step 1:

Selecting on t2 tabel using index T2\_IDX1

SELECT t2.\* FROM t2 where t2.id = '1';



As we use not unique index on *id* column, two or more rows can have duplicate id values. Because all leaves of the index tree are linked with next and previous leaf, Oracle can use INDEX RANGE SCAN to get the range of rows, suitable for the condition. Then Oracle retrieve blocks where rows are located. This process is represented as *TABLE ACCESS BY INDEX ROWID BATCHED.*

**Task 5**

Step 1:

Creating table employees

CREATE TABLE employees AS

SELECT \*

FROM scott.emp;

Step 2:

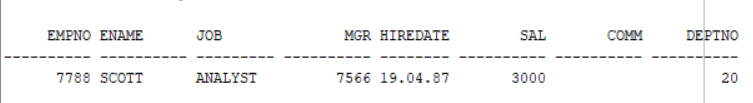
Creating index

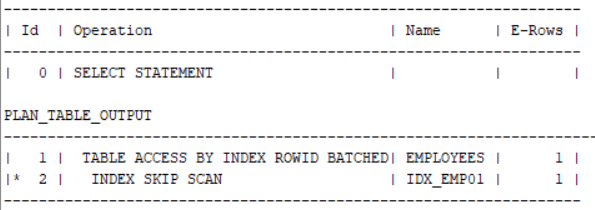
CREATE INDEX idx\_emp01 ON employees(empno, ename, job);

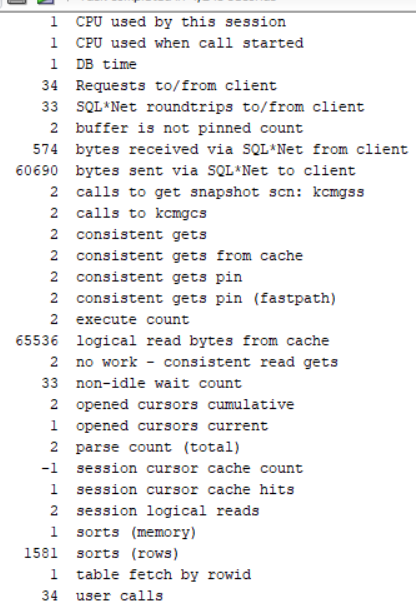
Step 3:

Selecting table using index skip scan

SELECT /\*+INDEX\_SS(emp idx\_emp01)\*/ emp.\* FROM employees emp where ename = 'SCOTT';

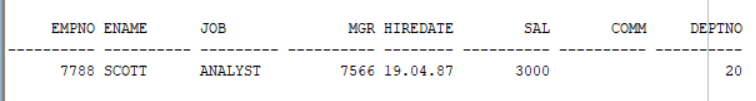


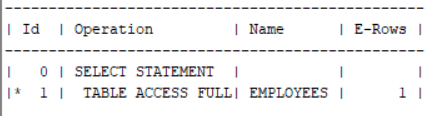


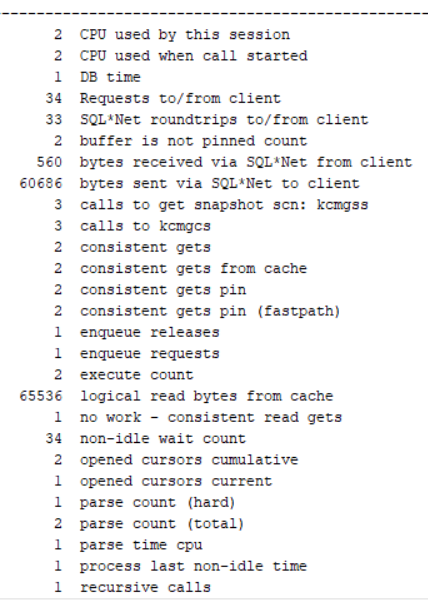


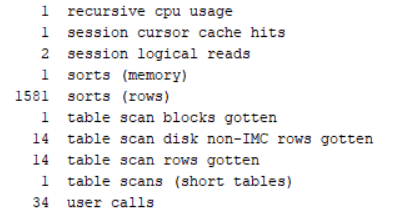
Selecting table using full table access

SELECT /\*+FULL(emp)\*/ emp.\* FROM employees emp WHERE ename = 'SCOTT';









Skip scanning lets a composite index be split logically into smaller subindexes. In skip scanning, the initial column of the composite index is not specified in the query. According to statistics of executing queries, using skip scanning slightly improved performance, especially CPU using.

**Conclusion**

At this lab we have considered with different scanning methods, their pros and cons. I have already discussed about job I made and different types of scanning at the end of tasks.