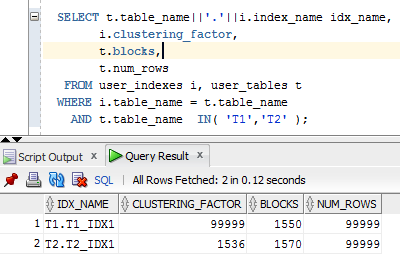
# Task 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Count of Blocks | Count of Used Blocks | Count of Rows | Consistent gets | Description |
| 1 | 1664 | 1536 | 99999 | 1539 | I created table with 99999 rows. They use 1536 blocks There are unused blocks. Consistent gets is 1539: 1536 is direct (physical) read, 3 is cashed read. |
| 2 | 1664 | 0 | 0 | 1541 | I deleted all rows from the table  The count of rows is 0 and they don’t use any blocks, but still count of blocks is 1664.  That happens because HWM isn’t reset and is placed as if there are 99 999 rows. |
| 3 | 1664 | 1 | 1 | 1541 | After INSERT one datablock is used, but overall number of blocks is still 1664.  That happens because HWM isn’t reset and is placed as if there are 99 999 rows. |
| 4 | 8 | 0 | 0 | 3 | I truncated the table. Count of blocks is 8, because 1 segment is 8 datablocks.  So an empty table takes 8 datablocks. |

# Index Scan types

## Task 2: Index Clustering Factor

Clustering factor:



The clustering factor is a measure of the ordered-ness of an index in comparison to the table that it is based on. It means how perfect are the data ordered in comparison with the native table with data.

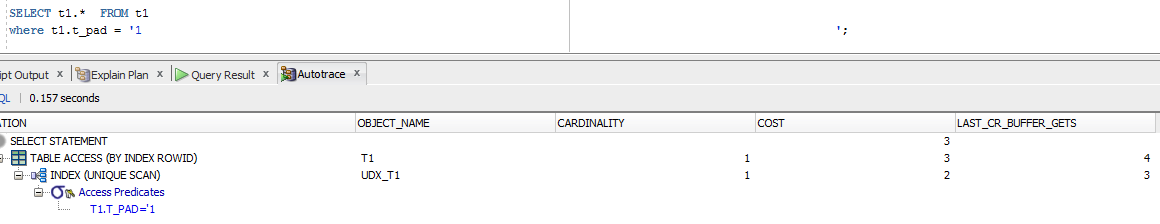
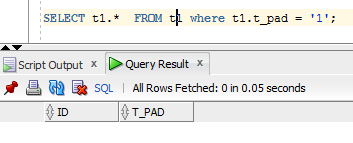
If the value is close to the total number of blocks, then the table is very well ordered. In this case, index entries in one leaf block usually indicate the rows that are in the same data blocks.

If the value is close to the total number of rows, then the table is very unordered. In this case, it is unlikely that the index entries in one leaf block point to the same data blocks.

Index t1\_idx1 just goes uteratively from 0 to 99. When it reaches 99 it starts again from 0. Index t2\_idx2 groups values by hundreds ([0-99]-0, [100-199]-1 etc) Index t2\_idx2 has a better perfomance, because the rows of a table on disk are sorted in about the same order as the index keys.

The clustering index will have better performance while executing **select** clause filtered by **IN (list of values).**

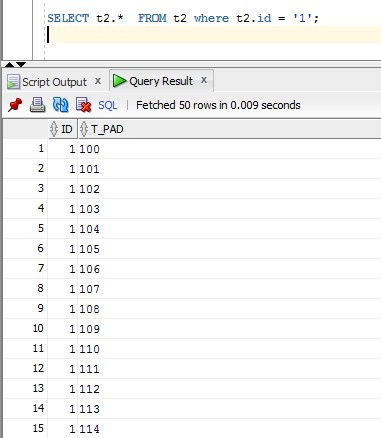
## Task 3: Index Unique Scan



Cost is small because unique indexes guarantee that only one row will be returned for a specified value.

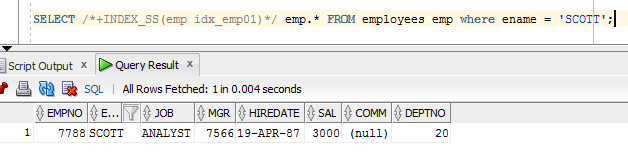
So when the database finds the needed value it stops searching.

## Task 4: Index Range Scan

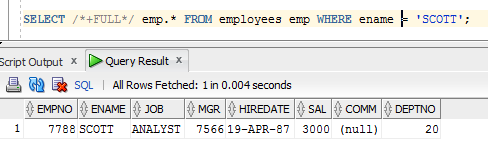


I used index range scan. The index starts working from the root block to the first leaf containing an entry matching the specified condition. Here he retrieves a rowid and then retrieves a table data block. Then the next rowed will be retrieved. This back-and-forth between the index leaf blocks and the data blocks will continue until all the matching index entries have been read.

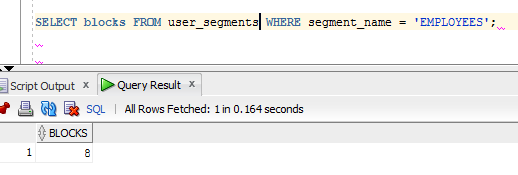
## Task 5: Index Skip Scan

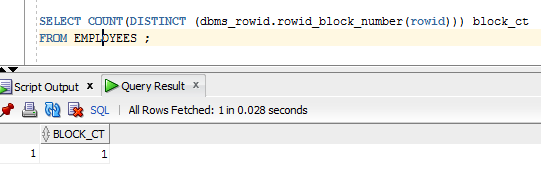


|  |  |
| --- | --- |
| consistent gets | 2 |
| consistent gets from cache | 2 |
| consistent gets pin | 2 |
| consistent gets pin (fastpath) | 2 |
| CPU used by this session | 2 |
| CPU used when call started | 2 |
| DB time | 3 |
| enqueue releases | 1 |
| enqueue requests | 1 |
| execute count | 2 |
| logical read bytes from cache | 16384 |
| no work - consistent read gets | 2 |
| non-idle wait count | 25 |
| opened cursors cumulative | 2 |
| parse count (hard) | 1 |
| parse count (total) | 2 |
| recursive calls | 1 |
| Requests to/from client | 25 |
| session cursor cache hits | 1 |
| session logical reads | 2 |
| sorts (memory) | 2 |
| sorts (rows) | 2356 |
| SQL\*Net roundtrips to/from client | 25 |
| table fetch by rowid | 1 |
| user calls | 27 |
| workarea executions - optimal | 5 |



|  |  |
| --- | --- |
| consistent gets | 3 |
| consistent gets from cache | 3 |
| consistent gets pin | 3 |
| consistent gets pin (fastpath) | 3 |
| CPU used by this session | 3 |
| CPU used when call started | 3 |
| DB time | 4 |
| enqueue releases | 1 |
| enqueue requests | 1 |
| execute count | 2 |
| logical read bytes from cache | 24576 |
| no work - consistent read gets | 1 |
| non-idle wait count | 25 |
| opened cursors cumulative | 2 |
| parse count (hard) | 1 |
| parse count (total) | 2 |
| recursive calls | 1 |
| Requests to/from client | 25 |
| session cursor cache hits | 1 |
| session logical reads | 3 |
| sorts (memory) | 2 |
| sorts (rows) | 2356 |
| SQL\*Net roundtrips to/from client | 25 |
| table scan blocks gotten | 1 |
| table scan disk non-IMC rows gotten | 14 |
| table scan rows gotten | 14 |
| table scans (short tables) | 1 |
| user calls | 27 |
| workarea executions - optimal | 5 |





Instead of restricting the search path using a predicate from the statement, Skip Scans are initiated by probing the index for distinct values of the prefix column. Each of these distinct values is then used as a starting point for a regular index search. The result is several separate searches of a single index that, when combined, eliminate the affect of the prefix column. Essentially, the index has been searched from the second level down.

The optimizer uses statistics to decide if a skip scan would be more efficient than a full table scan.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Count of Blocks | Count of Used Blocks | Count of Rows | Consistent gets | Description |
| 1 | 8 | 1 | 1 | 2 | Skip Scan. It return the range of data.  Consistent gets is 2: 1 is direct (physical) read, 1 is cashed read. |
| 2 | 8 | 1 | 1 | 3 | Full scan. It scans full table. Consistent gets is 3: 1 is direct (physical) read, 2 is cashed read. |