# Table access

## Task 1: Full Scan, High-Water Mark and Consistent Gets

The autotrace provides instantaneous feedback including the returned rows, execution plan, and statistics. The user doesn't need to be concerned about trace file locations and formatting since the output is displayed instantly on the screen. This is very important data that can be used to tune the SQL statement.

**N.B. It also rollbacks all uncommitted changes.**

The consistent gets Oracle metric is the number of times a consistent read was requested to get data from a data block.

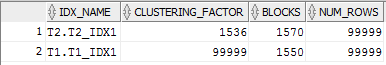
– Consistent Gets was reads from memory (sga).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Operation | Count of Blocks | Count of Used Blocks | Count of Rows | Consistent gets |
| 1 | 99 999 rows | 1664 | 1536 | 99999 | 1539 |
| 2 | Delete | 1664 | 0 | 0 | 1541 |
| 3 | Insert 1 row | 1664 | 1 | 1 | 1541 |
| 4 | Truncate | 8 | 0 | 0 | 3 |

|  |  |
| --- | --- |
| № | Description |
| 1 | We created a table with 99 999 rows, they use 1536 blocks, overall number of blocks is 1664, bcs other 128 datablocks are responsible for storing metadata.  Consistent gets is 1539: 1536 is direct(physical) read, 3 is cashed read. |
| 2 | After DELETE we have 0 rows and they don’t use any blocks, but still count of blocks is 1664, because HWM isn’t reset and is placed as if there are 99 999 rows. So Oracle scan all blocks under HWM, even though there are no data. |
| 3 | After INSERT 1 datablock is used, but overall number of blocks is still 1664 (the reason is higher) |
| 4 | After truncation HWM was reset and the associated index was reset as well.  So count of blocks is 8, because 1 segment is 8 datablocks, so an empty table takes 8 datablocks (bcs 1 table = 1 segment) |

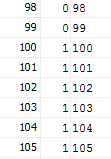
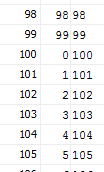
# Index Scan types

## Task 2: Index Clustering Factor



**The index clustering factor** is a measure of how many I/Os the database would perform if it were to read every row in that table via the index in index orderor how many time the database should go to the datablocks to retrieve all needed data.

**Why do we have different factors for t1\_idx1 and t2\_idx:**



**T1) T2)**

t1\_idx1 just goes uteratively from 0 to 99. When it reaches 99 it starts again from 0.

t2\_idx2 groups values by hundreds ([0-99]-0, [100-199]-1 etc)

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, so the database perfoms a minimum number of I/Os on the table to read the entire table via the index, because the next row needed from an index key would likely be the next row in the table block as well. The query would not be skipping all over the table to find row after row—they are naturally next to each other on the block. And on the contrary , if the rows in the table are not in the same order on disk as the index keys—if the data is scattered—the query will tend to perform the maximum number of I/Os on the table, as many as one I/O for every row in the table. That is because as the database scans through the index, the next row needed will probably *not* be on the same block as the last row. The database will have to discard that block and get another block from the buffer cache. The query will end up reading every block from the buffer as many times as it has rows on it.

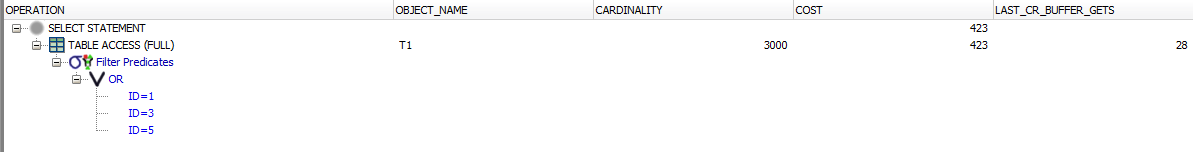
Here we see, that in the first case (t2\_idx2) the clustering factor is near **the number of blocks** in the table (bcs rows are physically in the same place).

In the the second case (t1\_idx1) the clustering factor is near **the number of rows** in the table (bcs the data is randomly scattered).

select \*

from T1

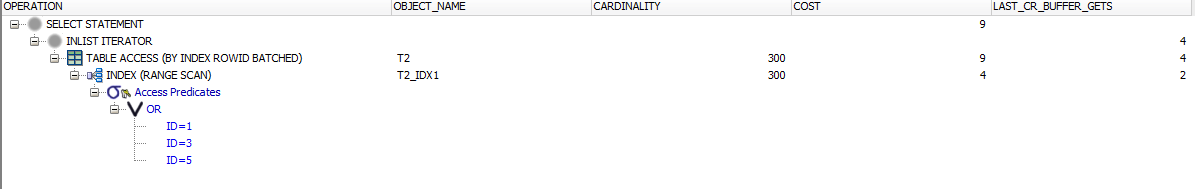
where id IN (1,3,5)



select \*

from T2

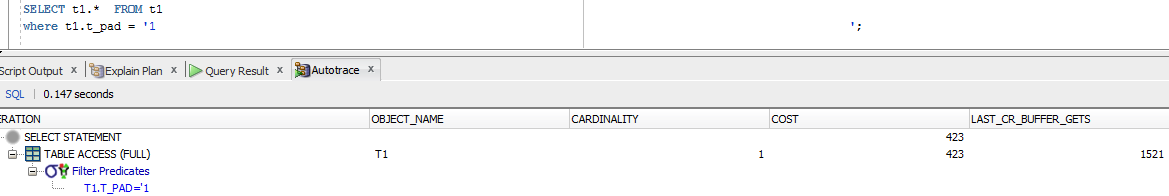
where id IN (1,3,5)



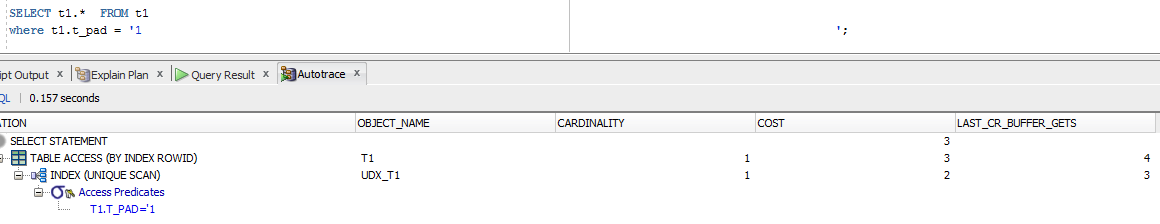
The better performance has clustering index.

## Task 3: Index Unique Scan

Without unique index:



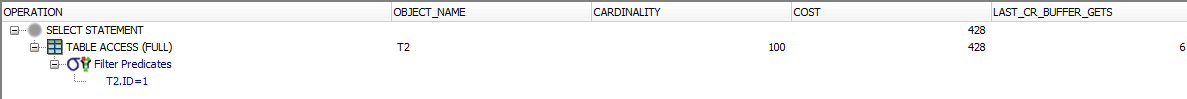
With unique index:



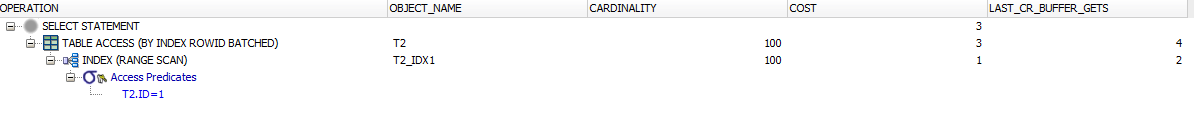
Costs in the second case are considerably smaller because of unique scan: unique indexes guarantee that only one row will ever be returned for a specified value, so when the database finds the needed value in the tree it stops searching.

## Task 4: Index Range Scan

Without index:



With index:



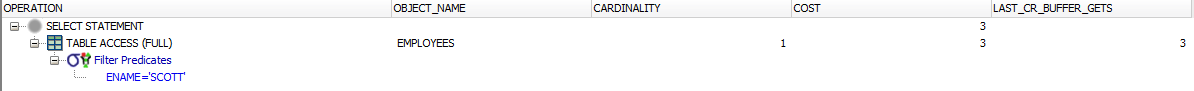
Here we use index range scan, it means that a predicate contains a condition that will return a range of data

(where t2.id ='1 ';)

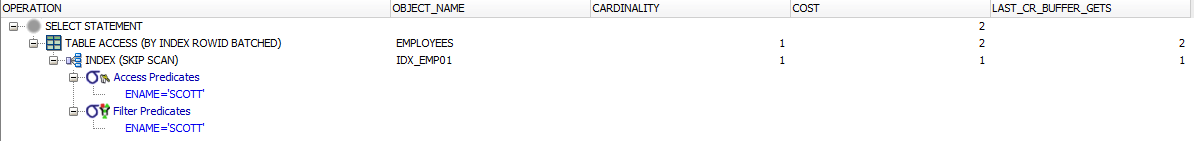
It doesn’t need to search through all table, it starts from the root block to the first leaf block containing an entry matching the specified condition. Here he retrieves a rowid and then retrieves a table data block. After that the next rowed will be retrieved.This back-and-forth between the index leaf blocks and the data blocks will continue until all the matching index entries have been read. If the range of entries matching the condition is large enough, it is likely that more than one leaf block will have to be accessed, so the next leaf block needed can be read using a pointer stored in the current leaf block that leads to the next leaf block (there’s also a pointer to the previous leaf block). Since these pointers exist, there is no need to go back up to the branch block to determine where to go next.

## Task 5: Index Skip Scan

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



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



Here we used hints to tell the database which scan to perform (skip scan or full scan);

A skip scan is relevant here because the predicate contains a condition on a non-leading column in an index and the leading columns are fairly distinct (therefore is splitting a multi-column index into smaller subindexes which are determined by the number of distinct values in the leading columns of the index). So it goes in each subindex, finds all matches, as they end it moves to the next leave with subindexes.