BS DEGREE IN INFORMATICS ENGINEERING
Academic year: 2022/2023 - 2nd year, 2nd term
Subject: File Structures and Databases
Third Assignment's Report: Physical Design on Oracle DB

uc3m | Universidad Carlos III de Madrid

Group:	**89**	**Lab User**	**fsdb253**
Student:	**ÁLVARO CABRERA NIETO**	**NIA:**	**100472152**
Student:	**GONZALO CARRETERO HERNÁNDEZ**	**NIA:**	**100472147**
Student:	**JIAHAO CHEN**	**NIA:**	**100472232**

1 Introduction

As a starting point and following the instructions received, we begin by restoring the original state of the DB, which can be achieved by creating and populating the tables using the original scripts. This also implies deleting all the triggers, views, packages, procedures and functions created in Assignment 2. Finally, we need to run the new script, which updates the tracks table with two new columns.

Our main objective is to optimise three processes (one 'UPDATE' and two QUERIES) once we have analysed them and planned possible improvements. The performance test of these processes will be performed using a procedure: run_test(x), where x is the number of iterations analysed. In our case, we decided to use x = 5. For each change done, we evaluate it separately based on the original design.

In addition to this starting point, we have various information about the work. We may modify the physical parameters of the objects, such as 'tablespace' and 'PCTFREE', using indexes or clusters, etc. Also, we know the following rules or indications:

- We cannot create new tablespaces as we can only use the 2kB, 8kB and 16kB existing ones (which is 8kB by default).
- We cannot use 'keep buffer pool'.
- We cannot create bitmaps or materialised views.
- We must create one cluster and at least one index.

Lastly, we analysed the costs at the beginning of the practice: 281228 blocks and 7757,8 seconds:

```
SQL> begin pkg_costes.run_test(5);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 04/05/2023 13:01:23
TIME CONSUMPTION: 7757,8 milliseconds.
CONSISTENT GETS: 281228 blocks

Procedimiento PL/SQL terminado correctamente.
```

2 Analysis

After an analysis of the default configuration of ORACLE SQL, we know that its structure is serial non-consecutive, with buckets of 8kB (4 blocks per bucket - blocks of 2kB), its distributed free space is 10%, and its minimum occupation density is 60%. Since we are in a serial structure, insertions and storage are optimal, while searches by non-privileged keys require full-scan.

It is worth mentioning that the workload consists of an 'UPDATE' operation and two Queries (QUERY 1, QUERY 2), with a relative frequency of (0.98, 0.01, 0.01), respectively. Therefore, optimal optimization modifications will mainly affect the 'UPDATE' operation, as it is performed a much larger number of times.

We conduct the analysis primarily based on the **mean case scenario (taking into account operations after its 2º iteration and not the 1º time).**

Throughout the whole analysis, the PCTfree is kept by default at 60%, since we do not have any deletions, we consider PCTfree irrelevant to increase the performance.

PCTFREE is also kept by default as 10%, although the only 'UPDATE' operation is on 'tracks' and it does not vary the size of the record, we consider it prudent to keep 10% of PCTfree just in case in the future there are 'INSERT' or 'UPDATE' operations that vary the size of the data. If we consider that there will be NO operations ('UPDATE' or 'INSERT') performed on the rest EVER, we can also set PCTFREE to 0.

Original plan of the three operations ('UPDATE', QUERY1, QUERY2):

1. Update

Firstly, we evaluate the data at the beginning by executing the corresponding command:

UPDATE tracks set lyrics = dbms_random.string('a',dbms_random.value(900,1200)) where searchk = 'u7022RO14453W47//1';

Mean case (2º operation forward):

```
SQL> UPDATE tracks set lyrics = dbms_random.string('a',dbms_random.value(900,1200)) where searchk = 'u7022RO14453W47//1';

0 filas actualizadas.

Transcurrido: 00:00:00.03

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 973582657

-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT   |        |    28 | 56392 |   686   (1)| 00:00:01 |
|   1 |  UPDATE            | TRACKS |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| TRACKS |    28 | 56392 |   686   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   2 - filter("SEARCHK"='u7022RO14453W47//1')

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       2474  consistent gets
          0  physical reads
          0  redo size
        494  bytes sent via SQL*Net to client
        925  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
```

It should be considered that in this case **we are taking one specific value for Searchk**, **and its cost is not representative**, since we are performing a full scan (the cost of locating a particular Serachk depends on the location, on average the cost is (N+1)/2).

As we can observe from the above execution plan, there are two operations involved in this 'UPDATE' process:

2. UPDATE

This is the 'UPDATE' operation of the corresponding record. Once it has been located by the search algorithm. It is deleted and the updated version is inserted. Depends on the data being inserted:

- constant-sized records: content is altered. **selection+ k acc**
- serial non-consecutive: if gap is big enough, record is updated there. **selection+ k acc**
- any other case: record is deleted and reinserted modified. **(selection + k + k) accesses**
- **Cluster - storage, size, tablespace?**

1. TABLE ACCESS FULL

Consists of a retrieval of all rows from the table TRACKS, in order to find the one that corresponds to the value of searchk indicated.

3. Query 1

As we can see, in this query there are several tables and operations involved. There are five related tables: 'songs', 'involvement', 'albums', 'tracks' (which was used in the previous 'UPDATE' process) and 'performances'.

Regarding the executed operations in this query, four are involved:

1. **WITH**
   Using a 'WITH' clause to define multiple subqueries: 'authors', 'authorship', 'recordings', 'recs_match' and 'pers_match'.

2. **JOIN**
   Joining the 'authorship' table with the 'recordings' table to calculate the percentage of recordings that each performer has contributed to.

3. **JOIN**
   Joining the 'authorship' table with the 'performances' table to calculate the percentage of live performances that each performer has contributed to.

4. **FULL JOIN**
   Finally, the query uses a 'FULL JOIN' to combine the results from the two subqueries pers_match and recs_match. It returns the performer along with the percentage of recordings and live performances that they have contributed to.

We run the query, and we get the following result from autotrace:

```
Transcurrido: 00:00:03.36

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 2759212254

-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                               |   172M|  8737M|       | 22358   (5)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                               |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9E2E7E_334C22EB   |       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                               |   228K|    22M|    24M|  8384   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                               |   228K|    22M|       |  3058   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                   |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                               |   214K|    12M|       |  3052   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                               |   214K|    20M|    12M|  3052   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                               |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                      |   107K|  4507K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                         |   107K|  6184K|       |   171   (1)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                      |   172M|  8737M|       | 13973   (7)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                               |   172M|    12G|  7904K| 13973   (7)| 00:00:01 |
|  13 |     VIEW                                 |                               |   155K|  6076K|       |  3331   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                               |   155K|    26M|       |  3331   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                               |   155K|    26M|    17M|  3324   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                               |   155K|    15M|       |   686   (1)| 00:00:01 |
|  17 |         TABLE ACCESS FULL                | ALBUMS                        | 20182 |   867K|       |    68   (0)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | TRACKS                        |   155K|  9113K|       |   616   (1)| 00:00:01 |
|  19 |        VIEW                              |                               |   228K|    15M|       |   862   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9E2E7E_334C22EB   |   228K|    18M|       |   862   (1)| 00:00:01 |
|  21 |     VIEW                                 |                               |   745K|    28M|       |  7585   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                               |   745K|   119M|       |  7585   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                               |   745K|   119M|    18M|  7553   (1)| 00:00:01 |
|  24 |        VIEW                              |                               |   228K|    15M|       |   862   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9E2E7E_334C22EB   |   228K|    18M|       |   862   (1)| 00:00:01 |
|  26 |        TABLE ACCESS FULL                 | PERFORMANCES                  |   745K|    67M|       |  1990   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------------
```

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

   4 - access("INVOLVEMENT"."MUSICIAN"="AUTHORS"."MUSICIAN")
  12 - access("RECS_MATCH"."PERFORMER"="PERS_MATCH"."PERFORMER")
  15 - access("ALBUMS"."PERFORMER"="AUTHORSHIP"."PERFORMER"(+) AND "TRACKS"."TITLE"="AUTHORSHIP"."TITLE"(+) AND
              "TRACKS"."WRITER"="AUTHORSHIP"."WRITER"(+))
  16 - access("ALBUMS"."PAIR"="TRACKS"."PAIR")
  23 - access("PERFORMER"="AUTHORSHIP"."PERFORMER"(+) AND "SONGTITLE"="AUTHORSHIP"."TITLE"(+) AND
              "SONGWRITER"="AUTHORSHIP"."WRITER"(+))

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - this is an adaptive plan


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      11095  consistent gets
       6244  physical reads
          0  redo size
      24444  bytes sent via SQL*Net to client
       1918  bytes received via SQL*Net from client
         48  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
        695  rows processed
```

Regarding the first subquery:

> WITH authors as (select title,writer, writer musician from songs UNION select title,writer,cowriter musician from songs)

We observe that the operations done are:

- INDEX FAST FULL SCAN to read all the rows from the primary key index (in the first select) - id 9
- FULL SCAN (the second select) - id 10
- UNION the two queries - id 8
- SORT UNIQUE - Sorting that remove duplicate from the resulted set - id 7

Regarding the second subquery:

> … authorship as (select distinct band performer, title, writer, 1 flag FROM involvement join authors using(musician))

We observe that the operations done are:

- TABLE ACCESS FULL - Involvement table - id 5
- HASH JOIN with Authors. The database engine creates an in-memory hash table from the smaller table involved in the join (Authors). Then, it scans the larger table and checks each row against the hash table to find the matching rows. - id 4
- HASH UNIQUE - removes duplicate rows from the result set using hashing - id 3

Regarding the third subquery:

> … recordings as (select performer,tracks.title,writer from albums join tracks using(pair))

We observe that the operations done are:

- HASH JOIN - id 16
- TABLE ACCESS FULL - Albums - id 17

- TABLE ACCESS FULL - Tracks - id 18

Regarding the fourth subquery:

```
… recs_match as (select performer, round(sum(flag)*100/count('c'),2) pct_recs
                 from recordings left join authorship
                     using(performer,title,writer)
                 group by performer),
```

We observe that the operations done are:

- TABLE ACCESS FULL- authorship - id 20
- HASH JOIN OUTER id 15
- HASH JOIN GROUP BY id 14

Regarding the fifth subquery:

```
… pers_match as (select performer, round(sum(flag)*100/count('c'),2) pct_pers
                 from (select performer, songtitle title, songwriter writer
                       from performances) P
                     left join authorship using(performer,title,writer)
                 group by performer)
```

We observe that the operations done are:

- TABLE ACCESS FULL- authorship - id 20
- HASH JOIN OUTER id 15
- HASH JOIN GROUP BY id 14

Regarding the main query:

```
SELECT performer, pct_recs, pct_pers
  from recs_match full join pers_match using(performer))
```

We observe that the operations done are:

- HASH JOIN FULL OUTER id 12 full join pers_match and recs_match

In order to consider improvements, we should focus on how the tables are accessed, as well as the repercussions in the subsequent operations (mainly 'JOIN's):

Songs:

- INDEX FAST FULL SCAN: select title,writer, writer musician from songs
- Participate in HASH JOIN, UNION and SORT

Involvement:

- TABLE FULL ACCESS: select distinct band performer, title, writer, 1 flag FROM involvement

- Participate in HASH JOIN

Albums:
- TABLE FULL ACCESS: select performer,tracks.title,writer from albums join tracks using(pair)
- Participate in HASH JOIN OUTER and HASH GROUP BY

Tracks:
- TABLE FULL ACCESS: select performer,tracks.title,writer from albums join tracks using(pair)
- Participate in HASH JOIN OUTER and HASH GROUP BY

Performances:
- TABLE FULL ACCESS: select performer, songtitle title, songwriter writer from performances
- Participate in HASH JOIN OUTER and HASH GROUP BY

4. Query 2

As we can see, this second query involves several tables and operations. There are three related tables: 'albums', 'tracks' (also used in the 'UPDATE' process) and 'performances', where all of them are also used in the previous query.

The list of involved operations is the following:

1. **WITH**
   Using a 'WITH' clause to define two subqueries, 'recordings' and 'playbacks'.

2. **JOIN**
   In the first subquery ('recordings'), it queries all the performers with the songs they have recorded (taking into account the first time they recorded that song). This is achieved by the 'JOIN' of Albums and Tracks using Pair and grouping by performer, tracks. title, and writer.

3. **LEFT JOIN**
   In the second subquery ('playbacks'), the percentage of performances done by a performer whose song the performer has recorded is calculated, by summing the total of each performer from the previous subquery and dividing it by the total number of performances doing a 'LEFT JOIN', to take into account all the performances even if the songs done weren't recorded. Also, the average age of the songs performed relative to when they were recorded, if it is the case, is calculated. It is all grouped by performer and ordered by the mentioned percentage for the main query to select afterwards.

4. **WHERE**

The main query queries the first 10 performers from the previous subquery as well as the percentage mentioned and the average age of the performed songs expressed in years, months and days.

We run the query, and we get the following result from autotrace:

```
Transcurrido: 00:00:02.66

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1502392682

-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name         | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |              |    10 |   530 |       |  8658   (2)| 00:00:01 |
|*  1 |  COUNT STOPKEY           |              |       |       |       |            |          |
|   2 |   VIEW                   |              |   542 | 28726 |       |  8658   (2)| 00:00:01 |
|*  3 |    SORT ORDER BY STOPKEY |              |   542 | 56910 |       |  8658   (2)| 00:00:01 |
|   4 |     HASH GROUP BY        |              |   542 | 56910 |       |  8658   (2)| 00:00:01 |
|*  5 |      HASH JOIN RIGHT OUTER|              |   856K|    85M|  9560K|  8582   (1)| 00:00:01 |
|   6 |       VIEW               |              |   146K|  7843K|       |  3608   (1)| 00:00:01 |
|   7 |        HASH GROUP BY     |              |   146K|    11M|    13M|  3608   (1)| 00:00:01 |
|*  8 |         HASH JOIN        |              |   146K|    11M|       |   788   (1)| 00:00:01 |
|   9 |          TABLE ACCESS FULL| ALBUMS       | 21561 |   631K|       |    68   (0)| 00:00:01 |
|  10 |          TABLE ACCESS FULL| TRACKS       |   146K|  7570K|       |   719   (1)| 00:00:01 |
|  11 |       TABLE ACCESS FULL  | PERFORMANCES |   856K|    40M|       |  1991   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

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

   1 - filter(ROWNUM<=10)
   3 - filter(ROWNUM<=10)
   5 - access("R"."WRITER"(+)="P"."SONGWRITER" AND "R"."TITLE"(+)="P"."SONGTITLE" AND
              "P"."PERFORMER"="R"."PERFORMER"(+))
       filter("P"."WHEN">"R"."REC"(+))
   8 - access("ALBUMS"."PAIR"="TRACKS"."PAIR")


Estadisticas
----------------------------------------------------------
         13  recursive calls
          0  db block gets
      10219  consistent gets
      10185  physical reads
          0  redo size
       1140  bytes sent via SQL*Net to client
       1243  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
         10  rows processed
```

Regarding the first subquery:

```
… recordings as (select performer,tracks.title, writer,
                        min(rec_date) rec, 1 token
                 from albums join tracks using(pair)
                 group by performer,tracks.title,writer)
```

We observe that the operations done are:

- HASH JOIN - id 8
- TABLE ACCESS FULL - Performances - id 11
- TABLE ACCESS FULL - Tracks - id 10
- HASH GROUP BY - id 7

Regarding the second subquery:

```
… playbacks as (select P.performer, sum(token)*100/count('x') percentage,
                avg(nvl2(rec,when-rec,rec)) as age
               FROM performances P left join recordings R
                 on(P.performer=R.performer AND R.title=P.songtitle
                   AND R.writer=P.songwriter AND P.when>R.rec)
               GROUP BY P.performer
               ORDER BY percentage desc)
```

We observe that the operations done are:

- HASH JOIN RIGHT OUTER - id 5
- TABLE ACCESS FULL - Performances - id 11
- HASH GROUP BY - id 4
- SORT ORDER BY STOPKEY - id 3

Regarding the main query:

```
SELECT performer, percentage, floor(age/365.2422) years,
    floor(mod(age,365.2422)/30.43685) months,
    floor(mod(age,365.2422)-(floor(mod(age,365.2422)/30.43685)*30.43685)) days
  FROM playbacks WHERE rownum<=10
```

We observe that the operations done are:

- COUNT STOPKEY - id 1
- SELECT STATEMENT - id 0

As we observe in the above execution plan, the following operations are performed on each table:

Albums:

- TABLE ACCESS FULL: select performer from Albums
- Participate in HASH JOIN, HASH GROUP BY

Tracks:

- TABLE ACCESS FULL: select title, writer, rec_date from Tracks
- Participate in HASH JOIN, HASH GROUP BY

Performances:
- TABLE ACCESS FULL: select performer, songtitle, songwriter, when from Performances
- Participate in HASH JOIN, HASH GROUP BY, SORT ORDER BY STOPKEY

We see that all tree tables participate in:

HASH JOIN:
Example:
On albums.pair = tracks.pair:
- The database engine creates an in-memory hash table from the smaller table involved in the join (Albums). Then, it scans the larger table (Tracks) and checks each row against the hash table to find the matching rows. This type of join is efficient among other reasons because the resulting table is kept in RAM for the next HASH GROUP BY done on it.

HASH JOIN RIGHT OUTER:
Example:
On P.performer = R.performer AND R.title = P.songtitle AND R.writer = P.songwriter AND P.when > R.rec:
- In order to do this, the table performances is previously full scanned and joined with the view that results of the previous section, keeping all the rows from the table performances in case of not finding a matching row.

SORT ORDER BY STOP KEY:
Example:
order subquery playbacks by percentage desc:
- The resulting table after a hash join is not necessarily sorted by any value, and therefore this operation may be more costly than having some type of index.

Possible Improvements of the three operations ('UPDATE', QUERY1, QUERY2):

1. **Bigger Bucket Size and PCTfree for Tracks -> NOT SELECTED**

- **Expected benefits**

Considering that we are working in a serial non-consecutive organization, it may be convenient to have a larger bucket size or larger percentage of PCTfree, in case the stored data changes frequently. It could be more efficient to increase the bucket size so that the possibility of having insufficient space in the bucket for updating decreases, as this would mean finding extra space in another bucket, leading to extra memory accesses.

- **Possible drawbacks**

One possible disadvantage of bigger buckets is for example longer search times for the record in the bucket for certain operations and higher PCTfree may reduce the data density.

However, this is not the case because as we can see from the operation done (dbms_random.string('a',dbms_random.value(900,1200))) the data is always a IEEE_754 double precision float of 64 bytes before and after the operation. We will confirm this in the following phases.

Code:

CREATE TABLE TRACKS (...) TABLESPACE TAB_16k;

UPDATE tracks set lyrics = dbms_random.string('a',dbms_random.value(900,1200))
where searchk = '**I424182F6231LB2//7'**

16k- Table Tracks

```
SQL> begin pkg_costes.run_test(5);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 04/05/2023 13:11:02
TIME CONSUMPTION: 8321,6 milliseconds.
CONSISTENT GETS: 149194,6 blocks

Procedimiento PL/SQL terminado correctamente.
```

As we can see, the time consumption did not reduce (+565 ms). Also, the number of consistent gets is 149194,6 blocks for a 16K size bucket (298389.2 actual blocks), which has to be multiplied by 2 to compare it with the consistent gets for an 8K bucket (+17162 blocks). However, due to the fact there are other processes done in the test (query 1 and query 2), we consider to be more significant the time factor over consistent gets to make our decision.

Also, by increasing the bucket size and leaving the PCTFREE by 10%, the distributed free space also increased and it is shown indirectly also that the size of distributed free space in this case is not relevant to reduce the nº of accesses.

2. **Unique Index on Searchk of bucket size 2k -> SELECTED**

- **Expected benefits**

An index (of type B-tree since it is Unique) should allow the DB to quickly locate rows in a table based on the value of 'searchk' (alternative key) using 'WHERE', rather than scanning the entire table. The smaller size of the bucket should allow the database to reduce the number of accesses to locate the record. On the other hand, since indexes do not take up so much space and are frequently accessed, with the adequate buffer policy, the hit ratio should increase significantly.

- **Possible drawbacks**

This index may not be convenient in some cases, for example, if there were frequent operations regarding 'INSERT', 'DELETE' AND 'UPDATE' that affected the index key 'searchk', which is not our case (as we perform 'UPDATE' operations on lyrics). A bigger bucket size for the nodes of the tree may be better if the query requires us to full scan the bucket. Also, it is costly if the Indexing key is frequently updated, which is not the case. We need additional storage for the index and we may need a maintenance process, like sorts in memory.

CREATE UNIQUE INDEX ind_tracks ON Tracks(searchk) TABLESPACE TAB_2K;

If we use the procedure 'run_test(x)', we can evaluate the 'consistent gets' and 'time consumption' required:

```
SQL> CREATE UNIQUE INDEX ind_tracks ON Tracks(searchk) TABLESPACE TAB_2K;

Indice creado.

SQL> begin pkg_costes.run_test(5);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 04/05/2023 13:32:56
TIME CONSUMPTION: 7332,2 milliseconds.
CONSISTENT GETS: 19673,4 blocks

Procedimiento PL/SQL terminado correctamente.
```

As expected, we observe a significant reduction in time and gets (-400 ms, -261554 blocks), which indicated to us that it was the right decision.

In addition to this improvement, we can see below the result by using a bucket size of 16k for the index:

```
SQL> CREATE UNIQUE INDEX ind_tracks ON Tracks(searchk) TABLESPACE TAB_16K;

Indice creado.

SQL> begin pkg_costes.run_test(5);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 04/05/2023 13:24:53
TIME CONSUMPTION: 7628,4 milliseconds.
CONSISTENT GETS: 19492,6 blocks

Procedimiento PL/SQL terminado correctamente.
```

As we observe, the time is slightly higher and the number of consistent gets is the same (-128 ms, -261735 blocks = +272 ms, +181 blocks in comparison to the previous one) (taking into account that we are using buckets of 16k, so we would have to multiply it by 4 to compare it to buckets of 2k). Therefore, **2k buckets are much better as we said**.

3. **Creating a single table hashed cluster for the table TRACKS -> NOT SELECTED**

- **Expected benefits:**

A cluster should allow the DB to reduce the number of accesses by using the appropriate hash key and hashing function that allow Oracle to quickly locate the bucket where it is stored.

- **Possible drawbacks:**

If the hashing function is not chosen wisely, the density might be very low. And therefore, increases storage space. Also, when performing a full scan to join tables in queries 1 and 2, due to the low density, the number of accesses and time consumption may increase.

select avg_row_len, num_rows, blocks from user_tables where table_name = 'TRACKS';

```
Conectado.
SQL> select avg_row_len, num_rows, blocks from user_tables where table_name = 'TRACKS';

AVG_ROW_LEN   NUM_ROWS     BLOCKS
----------- ---------- ----------
        104     146278       2260
```

-> Avg: 65 records per block
-> 6760 bytes per block
-> for density 70% -> Size of Cluster = 6760 / 0.70 = 9657 B
-> 146278 / (9657/104) = 1575 blocks
-> HASHFUNCTION = 4 digits (from position 2 to 6) of Searchk
-> HASHKEYS: 10000 possible results from HASHFUNCTION
-> TABLESPACE TAB_16k: we perform full-scan in the table in QUERY 1 and QUERY 2

CREATE CLUSTER identity (SearchKey varchar2(20)) SIZE 9657 TABLESPACE TAB_16k SINGLE TABLE HASHKEYS 10000 HASH IS TO_NUMBER(SUBSTR(SearchKey, 2,4));

CREATE TABLE (...
searchk varchar2(20),
lyrics VARCHAR2(4000),
…) CLUSTER identity (searchk);

```
SQL> begin pkg_costes.run_test(5);end;
  2  /
begin pkg_costes.run_test(5);end;
*
ERROR en linea 1:
ORA-03292: La tabla a truncar es parte de un cluster
ORA-06512: en "FSDB253.PKG_COSTES", linea 39
ORA-06512: en "FSDB253.PKG_COSTES", linea 71
ORA-06512: en linea 1


Transcurrido: 00:00:00.02
```

When running the test procedure we get this error, so we try to truncate the cluster:

```
SQL> truncate cluster identity;
truncate cluster identity
                *
ERROR en linea 1:
ORA-03293: El cluster a truncar es un HASH CLUSTER
```

Then, we realize that there are references like chained records in the overflow area:

delete tracks; --we delete all rows
COMMIT; --save changes

ALTER CLUSTER IDENTITY DEALLOCATE UNUSED; --we deallocate unused space for the cluster

By running the test command, we get:

```
Transcurrido: 00:02:03.78
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 10/05/2023 22:05:13
TIME CONSUMPTION: 7612,2 milliseconds.
CONSISTENT GETS: 39587,4 blocks

Procedimiento PL/SQL terminado correctamente.

Transcurrido: 00:02:01.27
```

We see there is a significant decrease in consistent gets compared to the original design (-145 ms, -241640 blocks) (which is mainly due to the accesses saved in the 'UPDATE' operation). However, the time consumption is only slightly less.

As mentioned before, this is due to a lower density (when executing a full scan of the table tracks in the join performed in query 1 and query 2) as now it is more costly, so although the selection in the 'UPDATE' operation is much more efficient now, we do not get as many improvements as we would like.

4. **Hinting index (involvement pk_Songs) - Query 1 -> NOT SELECTED**

- **Possible benefits**

By hinting Oracle to use Index pk_Songs(), it should use INDEX FULL SCAN instead. Since we are doing a Full Scan of the B-tree, the order is preserved and it should be more efficient to perform the SORT UNIQUE operation in id 7.

- **Expected drawbacks**

However, since we select the column "cowriter" also, this means that when visiting the nodes of the B-tree, we have to go to the place where the record is physically located since the indexing keys are only (musician, writer).

We add a hint to tell Oracle to use it:

We hint Oracle in the query:
/*+ INDEX(Songs pk_songs) */

The result of the Automatic Plan indeed changed:

```
Transcurrido: 00:00:05.06

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1015796000

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                            |   221M|    10G|       | 30186   (4)| 00:00:02 |
|   1 |  TEMP TABLE TRANSFORMATION               |                            |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9EA485_334C22EB|       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                            |   248K|    24M|    26M| 14513   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                            |   248K|    24M|       |  8723   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                            |   233K|    13M|       |  8717   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                            |   233K|    22M|    13M|  8717   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                            |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                   |   116K|  4900K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS BY INDEX ROWID BATCHED| SONGS                    |   116K|  6723K|       |  5601   (1)| 00:00:01 |
|  11 |          INDEX FULL SCAN                 | PK_SONGS                   |   116K|       |       |   623   (0)| 00:00:01 |
|  12 |   VIEW                                   | VW_FOJ_0                   |   221M|    10G|       | 15672   (8)| 00:00:01 |
|* 13 |    HASH JOIN FULL OUTER                  |                            |   221M|    16G|  8552K| 15672   (8)| 00:00:01 |
|  14 |     VIEW                                 |                            |   168K|  6576K|       |  3556   (1)| 00:00:01 |
|  15 |      HASH GROUP BY                       |                            |   168K|    28M|       |  3556   (1)| 00:00:01 |
|* 16 |       HASH JOIN OUTER                    |                            |   168K|    28M|    18M|  3549   (1)| 00:00:01 |
|* 17 |        HASH JOIN                         |                            |   168K|    16M|       |   686   (1)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | ALBUMS                     | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  19 |         TABLE ACCESS FULL                | TRACKS                     |   168K|  9864K|       |   616   (1)| 00:00:01 |
|  20 |        VIEW                              |                            |   248K|    17M|       |   937   (1)| 00:00:01 |
|  21 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA485_334C22EB|   248K|    19M|       |   937   (1)| 00:00:01 |
|  22 |     VIEW                                 |                            |   883K|    33M|       |  8449   (1)| 00:00:01 |
|  23 |      HASH GROUP BY                       |                            |   883K|   141M|       |  8449   (1)| 00:00:01 |
|* 24 |       HASH JOIN RIGHT OUTER              |                            |   883K|   141M|    20M|  8410   (1)| 00:00:01 |
|  25 |        VIEW                              |                            |   248K|    17M|       |   937   (1)| 00:00:01 |
|  26 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA485_334C22EB|   248K|    19M|       |   937   (1)| 00:00:01 |
|  27 |        TABLE ACCESS FULL                 | PERFORMANCES               |   883K|    80M|       |  1991   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------------
```

We use the 'run_test(x)' procedure:

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 07/05/2023 13:51:20
TIME CONSUMPTION: 7835,2 milliseconds.
CONSISTENT GETS: 281292,8 blocks

Procedimiento PL/SQL terminado correctamente.
```

We see that the results are worse now (+79 ms, +65 blocks). By checking the new Auto Plan, we see that indeed accessing the Files where the record are physically stored to retrieve the

value of "cowriter" is very costly (id 10). Another thing to note is the hardware technology, since we are performing random accesses with the Index, it is generally more costly than performing serial accesses in disks (except in SSD).

5. **Hinting Index (involvement pk_involvement) - Query 1 -> NOT SELECTED**

CREATE INDEX ind_involvement ON involvement(musician);

Since Involvement(musician) is already in the pk_involvement, we do not have to create another key but give the system a hint to use it.

We are thinking of performing an INDEX RANGE SCAN using the column "musician" instead of 'TABLE FULL SCAN'. To do this, we need to change the way that the primary key of the table Involvement is implemented so musician is a prefix:

CREATE TABLE INVOLVEMENT (.. CONSTRAINT PK_INVOLVEMENT PRIMARY KEY(musician, band, role),..)

- **Expected benefits**

Primary keys of Involvement are: Musician, band, role. In this case, we see that we are performing a 'JOIN' with the column "musician". It may be better to do an index range scan rather than doing a full scan to find the rows with a particular value of "musician".

- **Possible drawbacks**

The selectivity of column musician: there might be lots of resultant rows after selecting by a particular value of 'musician'. Remember that the pk_involvement consists of band, musician and role. And we are just selecting by musician.
The size of table 'Involvement' is not big as we see, so the result may not be very different than full-scanning
The technology of the hardware favours doing the serial full scan and random accesses may be worse.

We hint Oracle in the query:
/*+ INDEX_ASC(INVOLVEMENT PK_INVOLVEMENT) */

Execution plan:

```
Transcurrido: 00:00:07.21

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1568859433

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                            |   221M|    10G|       | 24765   (5)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                            |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9EA526_334C22EB|       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                            |   248K|    24M|    26M|  9093   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                            |   248K|    24M|       |  3303   (1)| 00:00:01 |
|   5 |      INDEX FULL SCAN                     | PK_INVOLVEMENT             |  1224 | 52632 |       |    14   (0)| 00:00:01 |
|   6 |      VIEW                                |                            |   233K|    13M|       |  3287   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                            |   233K|    22M|    13M|  3287   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                            |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                   |   116K|  4900K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                      |   116K|  6723K|       |   171   (1)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                   |   221M|    10G|       | 15672   (8)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                            |   221M|    16G|  8552K| 15672   (8)| 00:00:01 |
|  13 |     VIEW                                 |                            |   168K|  6576K|       |  3556   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                            |   168K|    28M|       |  3556   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                            |   168K|    28M|    18M|  3549   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                            |   168K|    16M|       |   686   (1)| 00:00:01 |
|  17 |         TABLE ACCESS FULL                | ALBUMS                     | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | TRACKS                     |   168K|  9864K|       |   616   (1)| 00:00:01 |
|  19 |        VIEW                              |                            |   248K|    17M|       |   937   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA526_334C22EB|   248K|    19M|       |   937   (1)| 00:00:01 |
|  21 |     VIEW                                 |                            |   883K|    33M|       |  8449   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                            |   883K|   141M|       |  8449   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                            |   883K|   141M|    20M|  8410   (1)| 00:00:01 |
|  24 |        VIEW                              |                            |   248K|    17M|       |   937   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA526_334C22EB|   248K|    19M|       |   937   (1)| 00:00:01 |
|  26 |        TABLE ACCESS FULL                 | PERFORMANCES               |   883K|    80M|       |  1991   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------------
```

By running the test command, we get:

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 10/05/2023 22:43:53
TIME CONSUMPTION: 7791,4 milliseconds.
CONSISTENT GETS: 281226 blocks

Procedimiento PL/SQL terminado correctamente.

Transcurrido: 00:01:12.62
SQL>
```

We see in the execution plan that Oracle is performing an 'INDEX FULL SCAN' and not an 'INDEX RANGE SCAN' as we indicated. The reason for this is that there is a 'HASH JOIN' performed in id 4, so it would not make sense to perform an 'INDEX FULL SCAN' since we are joining two tables using a 'HASH' function. Therefore, we saw that ORACLE performed an 'INDEX FULL SCAN', which results in more or less the result as 'FULL SCAN'. So it is not worth implementing this change.

6. **Unique Index (PAIR, performer) for Albums -> SELECTED**

- **Expected benefits**

By creating an 'INDEX' with the columns (PAIR, performer), we can hint ORACLE to use 'INDEX FAST FULL SCAN' on this 'INDEX' to keep the advantages of serial accesses and not have to access the files where the records are stored since all the information are in the 'INDEX' needed, resulting in better performance because we are only accessing the columns we need and there will be more nodes in one bucket of same size if not using index. Note that we can use the INDEX for both queries.

- **Possible drawbacks**

We are required extra memory for the 'INDEX'. And there will be maintenance cost if there are changes (update, insert, delete) in the indexing keys.

We use buckets of 16K since we are performing 'FULL SCAN'.

```
CREATE UNIQUE INDEX ind_albums ON Albums(PAIR, performer) TABLESPACE TAB_16K;
```

query 1:

We hint Oracle in the query to be sure it is not doing anything else:

```
/*+INDEX_FFS(Albums ind_albums)*/
```

Execution plan:

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1658623763

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                             |   221M|    10G|       | 24715   (5)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                             |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9EA62F_334C22EB |       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                             |   248K|    24M|    26M|  9084   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                             |   248K|    24M|       |  3294   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                 |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                             |   233K|    13M|       |  3287   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                             |   233K|    22M|    13M|  3287   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                             |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                    |   116K|  4900K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                       |   116K|  6723K|       |   171   (1)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                    |   221M|    10G|       | 15631   (8)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                             |   221M|    16G|  8552K| 15631   (8)| 00:00:01 |
|  13 |     VIEW                                 |                             |   168K|  6576K|       |  3515   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                             |   168K|    28M|       |  3515   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                             |   168K|    28M|    18M|  3508   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                             |   168K|    16M|       |   645   (1)| 00:00:01 |
|  17 |         INDEX FAST FULL SCAN             | IND_ALBUMS                  | 20399 |   876K|       |    27   (0)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | TRACKS                      |   168K|  9864K|       |   616   (1)| 00:00:01 |
|  19 |        VIEW                              |                             |   248K|    17M|       |   937   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA62F_334C22EB |   248K|    19M|       |   937   (1)| 00:00:01 |
|  21 |     VIEW                                 |                             |   883K|    33M|       |  8449   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                             |   883K|   141M|       |  8449   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                             |   883K|   141M|    20M|  8410   (1)| 00:00:01 |
|  24 |        VIEW                              |                             |   248K|    17M|       |   937   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA62F_334C22EB |   248K|    19M|       |   937   (1)| 00:00:01 |
|  26 |        TABLE ACCESS FULL                 | PERFORMANCES                |   883K|    80M|       |  1991   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------------
```

We see that the Cost in id 17 increased from 68 to 27.

query 2:

Hint Oracle to be sure it is not doing anything else:

/*+INDEX_FFS(Albums ind_albums)*/

Execution plan:

```
Transcurrido: 00:00:02.72

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1557371625

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |    10 |   530 |       |  8326   (2)| 00:00:01 |
|*  1 |  COUNT STOPKEY               |              |       |       |       |            |          |
|   2 |   VIEW                       |              |   490 | 25970 |       |  8326   (2)| 00:00:01 |
|*  3 |    SORT ORDER BY STOPKEY     |              |   490 | 91140 |       |  8326   (2)| 00:00:01 |
|   4 |     HASH GROUP BY            |              |   490 | 91140 |       |  8326   (2)| 00:00:01 |
|*  5 |      HASH JOIN RIGHT OUTER   |              |   883K|   156M|    13M|  8248   (1)| 00:00:01 |
|   6 |       VIEW                   |              |   145K|    11M|       |   753   (2)| 00:00:01 |
|   7 |        HASH GROUP BY         |              |   145K|    15M|       |   753   (2)| 00:00:01 |
|*  8 |         HASH JOIN            |              |   145K|    15M|       |   747   (1)| 00:00:01 |
|   9 |          INDEX FAST FULL SCAN| IND_ALBUMS   | 20399 |   876K|       |    27   (0)| 00:00:01 |
|  10 |          TABLE ACCESS FULL   | TRACKS       |   145K|  9773K|       |   719   (1)| 00:00:01 |
|  11 |       TABLE ACCESS FULL      | PERFORMANCES |   883K|    87M|       |  1991   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

We see after changing TABLE FULL ACCESS BY INDEX FAST FULL SCAN, the performance in the plan (id 9) increased to 27 from 68 (before).

By running the test, we get:

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 12/05/2023 17:51:09
TIME CONSUMPTION: 7609,6 milliseconds.
CONSISTENT GETS: 281000,2 blocks

Procedimiento PL/SQL terminado correctamente.

Transcurrido: 00:01:13.44
```

It is observed that the change increases the performance just very slightly (-147 ms, -227 blocks), so it is not worth implementing the change since we need extra space for the INDEX.

7. **Index (PAIR, title, writer, rec_date) for Tracks -> SELECTED**

We can try using a hint to use the primary index of tracks - pk_tracks.

- **Expected benefits**

The advantages are the same as for the ind_albums. The only difference is that we are using in this case a B+ tree since the index keys are not privileged. However, this should not affect the performance compared to a B-tree, since we are performing FULL SCAN on the tree, in this case, we would simply full scan the leaves nodes. Note that we can use the INDEX for both queries.

- **Possible drawbacks**

The disadvantages are the same ones as for the previous 'INDEX'.

TABLESPACE is 16k because big-sized buckets are convenient for doing full scans.

```
CREATE INDEX ind_tracks_2 ON Tracks(PAIR, title, writer, rec_date) TABLESPACE TAB_16K;
```

Query 1:

Hint Oracle to be sure it is not doing anything else:
/*+INDEX_FFS(Tracks ind_tracks_2)*/

Execution plan:

```
Transcurrido: 00:00:04.55

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3067493649

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                               |   190M|  9649M|       | 24021   (5)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                               |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9EA7C9_334C22EB   |       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                               |   248K|    24M|    26M|  9084   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                               |   248K|    24M|       |  3294   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                   |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                               |   233K|    13M|       |  3287   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                               |   233K|    22M|    13M|  3287   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                               |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                      |   116K|  4900K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                         |   116K|  6723K|       |   171   (1)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                      |   190M|  9649M|       | 14937   (7)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                               |   190M|    14G|  7360K| 14937   (7)| 00:00:01 |
|  13 |     VIEW                                 |                               |   144K|  5659K|       |  3029   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                               |   144K|    24M|       |  3029   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                               |   144K|    24M|    16M|  3022   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                               |   144K|    14M|       |   287   (1)| 00:00:01 |
|  17 |         TABLE ACCESS FULL                | ALBUMS                        | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  18 |         INDEX FAST FULL SCAN             | IND_TRACKS                    |   144K|  8489K|       |   218   (1)| 00:00:01 |
|  19 |        VIEW                              |                               |   248K|    17M|       |   937   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA7C9_334C22EB   |   248K|    19M|       |   937   (1)| 00:00:01 |
|  21 |     VIEW                                 |                               |   883K|    33M|       |  8449   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                               |   883K|   141M|       |  8449   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                               |   883K|   141M|    20M|  8410   (1)| 00:00:01 |
|  24 |        VIEW                              |                               |   248K|    17M|       |   937   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EA7C9_334C22EB   |   248K|    19M|       |   937   (1)| 00:00:01 |
|  26 |        TABLE ACCESS FULL                 | PERFORMANCES                  |   883K|    80M|       |  1991   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------------
```

For query 1, we see in the plan that we are now using INDEX FAST FULL SCAN (id 18) instead of TABLE ACCESS FULL. The cost is also better (218) versus (616) before.

Query 2:

Hint Oracle to be sure it is not doing anything else:
/*+INDEX_FFS(Tracks ind_tracks_2)*/

Execution plan:

```
Transcurrido: 00:00:02.30

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1786938751

-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |    10 |   530 |       |  7897   (2)| 00:00:01 |
|*  1 |  COUNT STOPKEY               |              |       |       |       |            |          |
|   2 |   VIEW                       |              |   490 | 25970 |       |  7897   (2)| 00:00:01 |
|*  3 |    SORT ORDER BY STOPKEY     |              |   490 | 91140 |       |  7897   (2)| 00:00:01 |
|   4 |     HASH GROUP BY            |              |   490 | 91140 |       |  7897   (2)| 00:00:01 |
|*  5 |      HASH JOIN RIGHT OUTER   |              |   883K|   156M|    13M|  7818   (1)| 00:00:01 |
|   6 |       VIEW                   |              |   145K|    11M|       |   324   (3)| 00:00:01 |
|   7 |        HASH GROUP BY         |              |   145K|    15M|       |   324   (3)| 00:00:01 |
|*  8 |         HASH JOIN            |              |   145K|    15M|       |   317   (1)| 00:00:01 |
|   9 |          TABLE ACCESS FULL   | ALBUMS       | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  10 |          INDEX FAST FULL SCAN| IND_TRACKS   |   145K|  9773K|       |   248   (1)| 00:00:01 |
|  11 |       TABLE ACCESS FULL      | PERFORMANCES |   883K|    87M|       |  1991   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

For query 2, we see in the plan that we are now using INDEX FAST FULL SCAN (id 10) instead of TABLE ACCESS FULL. The cost is also better (248) versus (719).

We can run the test procedure to get the overall results:

```
Transcurrido: 00:00:00.00
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 12/05/2023 18:19:52
TIME CONSUMPTION: 7295,2 milliseconds.
CONSISTENT GETS: 278775,4 blocks

Procedimiento PL/SQL terminado correctamente.

Transcurrido: 00:01:21.92
```

The run_test also confirms it, as both time and consistent gets are lower now (-461 ms, -2452 blocks).

8. **Index on (performer, songtitle, songwriter)-> SELECTED**

- **Expected benefits**

The advantages are the same as for the ind_tracks_2. Another advantage of using this index is that both queries (QUERY 1, QUERY 2), the attribute tuple (performer, songtitle, songwriter) of TABLE PERFORMANCES is accessed several times.

- **Possible drawbacks**

The index may not be as efficient as planned if many 'INSERT', 'UPDATE' or 'DELETE' operations were executed, as the index would grow in size (which can already be considered to be big as it includes three attributes) and maintenance cost also.

We use TAB_16K since it is convenient to use big buckets to full scan.

CREATE INDEX ind_performances ON Performances(performer, songtitle, songwriter, when) TABLESPACE TAB_16K;

Hint Oracle to be sure it is not doing anything else: /*+ INDEX_FFS(Performances ind_performances) */

Execution plan:

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 234724055

-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                               |   190M|  9649M|       | 23900   (5)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                               |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9EAB22_334C22EB   |       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                               |   248K|    24M|   26M|  9084   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                               |   248K|    24M|       |  3294   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                   |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                               |   233K|    13M|       |  3287   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                               |   233K|    22M|   13M|  3287   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                               |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                      |   116K|  4900K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                         |   116K|  6723K|       |   171   (1)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                      |   190M|  9649M|       | 14817   (7)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                               |   190M|    14G|  7360K| 14817   (7)| 00:00:01 |
|  13 |     VIEW                                 |                               |   144K|  5659K|       |  3529   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                               |   144K|    24M|       |  3529   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                               |   144K|    24M|   16M|  3523   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                               |   144K|    14M|       |   788   (1)| 00:00:01 |
|  17 |         TABLE ACCESS FULL                | ALBUMS                        | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | TRACKS                        |   144K|  8489K|       |   718   (1)| 00:00:01 |
|  19 |        VIEW                              |                               |   248K|    17M|       |   937   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EAB22_334C22EB   |   248K|    19M|       |   937   (1)| 00:00:01 |
|  21 |     VIEW                                 |                               |   883K|    33M|       |  7829   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                               |   883K|   141M|       |  7829   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                               |   883K|   141M|   20M|  7790   (1)| 00:00:01 |
|  24 |        VIEW                              |                               |   248K|    17M|       |   937   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9EAB22_334C22EB   |   248K|    19M|       |   937   (1)| 00:00:01 |
|  26 |        INDEX FAST FULL SCAN              | IND_PERFORMANCES              |   883K|    80M|       |  1371   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------
```

It can be seen that the cost of the index fast full scan on the table performances is better than the previous cost of the table access full (1371 vs 1990).

For query 2, we also use the index:

Again, we hint Oracle in the query to be sure it is not doing anything else:
/*+ INDEX(Performances ind_performances) */

Execution plan:

```
Transcurrido: 00:00:02.83

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1740871749

-------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name            | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                 |    10 |   530 |       |  7746   (2)| 00:00:01 |
|*  1 |  COUNT STOPKEY           |                 |       |       |       |            |          |
|   2 |   VIEW                   |                 |   490 | 25970 |       |  7746   (2)| 00:00:01 |
|*  3 |    SORT ORDER BY STOPKEY |                 |   490 | 91140 |       |  7746   (2)| 00:00:01 |
|   4 |     HASH GROUP BY        |                 |   490 | 91140 |       |  7746   (2)| 00:00:01 |
|*  5 |      HASH JOIN RIGHT OUTER|                 |   883K|   156M|    13M|  7668   (1)| 00:00:01 |
|   6 |       VIEW               |                 |   145K|    11M|       |   794   (2)| 00:00:01 |
|   7 |        HASH GROUP BY     |                 |   145K|    15M|       |   794   (2)| 00:00:01 |
|*  8 |         HASH JOIN        |                 |   145K|    15M|       |   788   (1)| 00:00:01 |
|   9 |          TABLE ACCESS FULL| ALBUMS          | 20399 |   876K|       |    68   (0)| 00:00:01 |
|  10 |          TABLE ACCESS FULL| TRACKS          |   145K|  9773K|       |   719   (1)| 00:00:01 |
|  11 |       INDEX FAST FULL SCAN| IND_PERFORMANCES|   883K|    87M|       |  1371   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

As observed, the time consumption has also decreased slightly - id 11 (1371 vs 1991).

We use run_test(5):

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 12/05/2023 18:52:23
TIME CONSUMPTION: 7336,4 milliseconds.
CONSISTENT GETS: 273790,2 blocks

Procedimiento PL/SQL terminado correctamente.

Transcurrido: 00:01:11.37
```

After applying the index on both query 1 and query 2, it can be seen that both the time consumption and the consistent gets have indeed improved (-420 ms, -7437 blocks).

9. **Tab-16k for Involvement and Songs -> SELECTED**

- **Expected benefits**

Since we did not use a index for TABLE INVOLVEMENT and SONGS and they are full scanned, it might be good to have a larger bucket size to improve the performance of full scanning (more records in the same bucket and reduce the number of times of changes between buckets).

Also, if the database is expected to grow rapidly (not in this case if we are just considering the 3 operations), using larger buckets can reduce the frequency of bucket splits, which can improve performance.

- **Possible drawbacks**

Not suitable for Selective Queries: if the query requires retrieving a small subset of data given some condition. Nonetheless, since there is no selective process, we believe that we should not be worried. One possible disadvantage is wasted space, but since we are using serial base organization, we are not worried.

The results of running the test are the following:

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 12/05/2023 20:19:22
TIME CONSUMPTION: 6420,6 milliseconds.
CONSISTENT GETS: 271265,6 blocks

Procedimiento PL/SQL terminado correctamente.
```

Indeed, time consumption is reduced (-916ms, -2525 blocks).

10. **Cluster for performances -> THEORETICAL ANALYSIS**

Another possible change could be implementing a single table hash cluster of the table performances, with the hashing key being the attributes performers, songwriter and songtitle.

- **Possible benefits**

It is quite convenient when having operations like GROUP BY performer, since all records with the same performer are in the same bucket. Also, if at any point a query is done using a WHERE statement on 'performer', we could improve a lot the accesses to this table (which is not the case in both queries).

- **Drawbacks**

However, in the current two queries, this table is fully scanned and participates in a hash join before GROUP BY. Therefore, creating a hashed cluster would not add overall any benefits.

Since we already have one cluster, following the instructions, we are not implementing this cluster. Anyways, we expect it to be detrimental for the queries' performance.

3 Physical Design

As we have explained in the previous section, there are some ways to improve the performance of the used processes ('UPDATE', QUERY1, QUERY2). Now, we gather all the selected improvements together:

1. **Unique Index on Searchk of bucket size 2k**

The unique index (B-tree) on Searchk is able to drastically reduce the number of accesses and time consumption of 'UPDATE', which accounts for 98% of the total executed commands. This implies that the whole database will be benefited from this index. By using this approach, the DB will be able to quickly locate rows by 'searchk' (alternative key), rather than scanning the entire table. In addition to this, the smaller size of the bucket (2k) reduces the number of accesses to locate the record.

The code implementation is the following:

```
CREATE UNIQUE INDEX ind_tracks ON Tracks(searchk) TABLESPACE TAB_2K;
```

2. **Index ind_performances**

The index is convenient to 'FAST FULL SCAN' the table and store only the necessary columns. We use 16k as our bucket size as we expect to have more rows recorded (with only the necessary columns) than the original table with the same bucket size, so it is more efficient.

The code implementation is the following:

```
CREATE INDEX ind_performances ON Performances(performer, songtitle, songwriter, when)
TABLESPACE TAB_16K;
```

By hinting the index (performances ind_performances), we make sure that Oracle is using the index in FAST FULL SCAN mode.

```
/*+ INDEX_FFS(Performances ind_performances ) */
```

3. **Unique Index (PAIR, performer) for Albums**

The reason for this Index is the same as ind_performances.

The code implementation is the following:

```
CREATE UNIQUE INDEX ind_albums ON Albums(PAIR, performer) TABLESPACE
TAB_16K;
```

By hinting the index (performances ind_performances), we make sure that Oracle is using the index in FAST FULL SCAN mode.

/*+INDEX_FFS(Albums ind_albums)*/

4. **Index (PAIR, title, writer, rec_date) for Tracks**

The reason for this Index is the same as ind_performances.

The code implementation is the following:

CREATE INDEX ind_tracks_2 ON Tracks(PAIR, title, writer, rec_date) TABLESPACE TAB_16K;

By hinting the index (performances ind_performances), we make sure that Oracle is using the index in FAST FULL SCAN mode.

/*+INDEX_FFS(Tracks ind_tracks_2)*/

5. **Tab-16k for Involvement and Songs**

By using tablespaces of size 16kB in all the tables Involvement and Songs (which are the tables that do not use any INDEX) we can take advantage of the larger bucket size to improve the performance when full scanning.

The code implementation is the following:

```
CREATE TABLE involvement (...) TABLESPACE TAB_16K;
CREATE TABLE songs (...) TABLESPACE TAB_16K;
```

Apart from the above-mentioned changes, the rest of the physical design of the original state is kept:

- PCTFREE: 10%. We believe that we should keep 10% in case there are any future inserts or updates that modify the record size. However, we can also suppose that there will not be any other operations besides the 3 proposed ones. In that case, we can set PCTFREE to 0% to take most of the space of a bucket.
- PCTUSED: 60%. Since there are no deletes, we consider it irrelevant to change it.
- BUCKET SIZE: 8KB. For those that were not mentioned in the changes, since they do not participate in any of the 3 operations. It is irrelevant to change them.
- Serial non-consecutive base organization.

4 Evaluation

After our physical design has been implemented, we can now evaluate globally the impact of the changes done with respect to the initial physical design.

The performance of the initial physical design is the following:

```
SQL> begin pkg_costes.run_test(5);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 04/05/2023 13:01:23
TIME CONSUMPTION: 7757,8 milliseconds.
CONSISTENT GETS: 281228 blocks

Procedimiento PL/SQL terminado correctamente.
```

With our new physical design, explained in the previous section, the performance is the following:

```
SQL> begin pkg_costes.run_test(5); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
RESULTS AT 12/05/2023 20:24:18
TIME CONSUMPTION: 2590 milliseconds.
CONSISTENT GETS: 11502,4 blocks

Procedimiento PL/SQL terminado correctamente.
```

As can be seen, consistent gets and the time has been significantly improved (-5167ms, -269726 blocks). The reduction in the number of blocks is mainly due to an important optimization of the 'UPDATE' (which has a much bigger frequency than both queries) by the creation of an index for SearchK. And when it comes to the improvement of time consumption, it is due to the general optimization (change in bucket sizes and creation of indexes to FAST FULL SCAN) of all the tables involved in the processes.

5 Concluding Remarks

Firstly, we want to say that we are satisfied with the overall work and the obtained results. We believe we have done a good job and have a good understanding of the Assignment, its implementation and its objectives. We have established our knowledge about indexes, clusters and access costs.

As mentioned in every section, we have improved the performance of the DB in many ways and by using several implementations. This means that both the number of accesses and the time consumption have improved drastically, which could be achieved mainly by using several ORACLE possibilities (index, hint, cluster, etc.). Nonetheless, there were some parts where we were a bit confused about what we could, should and must do at every moment. As the teacher said: "this optimization part can be as long as a two-year master's degree - we are asking you to show your knowledge, not create the best optimization DB plan".

In addition, we would like to say that we liked the practice, we found it highly interesting to better understand how DB optimisation works, even though we think that sometimes it was too complicated or confusing.

On the three complete assignments, we liked how we progressively went through the different phases of database design, in this case for a music database management system. Furthermore, we believe that the experience will be useful for us in the future when dealing with clients (especially the first assignment), improving the DB and implementing new features (especially the second assignment) and understanding costs and optimising the DB (especially the third assignment).

About the whole course, we think it has been very interesting and informative and has given us new and useful perspectives, not only about the world of files and databases but also about everything related to computer science. However, we believe that the course content is too extensive and the uploaded material is insufficient or could be improved. We think that the videos are very useful and, although we know that they take a lot of time and effort, we think that for this last assignment, it would have been convenient to have a similar video regarding the interpretation of the autotrace plan, since we were not able to do any examples in the lab class.

Having said all this, we appreciate the teacher's approachability and passion for teaching at all times.