Universidad Carlos III

Files and databases 2022-23

Course 2022-23

LabReport P3

Lecturers:	**Jorge Vico Pedrero and Francisco Javier Calle**		
Group:	**88**	**Lab User**	**FSDB228**
Student:	**József Iván Gafo**	**NIA:**	**100456709**
Student:	**Marcos González Vallejo**	**NIA:**	**100472206**

Index

1 Introduction

To start with, this lab consists of working with an initial physical design and acting as a database administrator to improve it so that the time and accesses to memory are lower down. We are given a package which allows us to check some statistics about 3 distinct processes which are 2 queries and an update. Moreover, the frequency of these 3 processes are 1% of the times query1 is run, another 1% of the times query2 is run and finally 98% of the times an update occurs. This tells us that it is of extreme importance to lower down the accesses and time spent by the updates as is the most probable process to occur. In terms of the initial physical design, we are given the same tables as in lab2. Those tables have the default PCTFREE and PCTUSED which is 10 and 60 respectively. Moreover, the default tablespace is 8k and the default structure is serial non consecutive. Our goal is to, by analyzing and identifying the drawbacks from the initial physical design, make use of the knowledge we acquired to improve that initial physical design. To improve the initial physical design, we can change the PCTFREE and PCTUSED, the tablespace, add indexes and a cluster. In this document, we will first analyze the initial physical design as a whole and then specifically for each process, query1, query2 and update. Then, we will implement our own physical design trying to fix those details that were preventing the initial design from being a great design.

2 Analysis

To start with, we looked for the average size of a record, total number of records and total number of blocks.

TABLE_NAME	AVG_ROW_LEN	NUM_ROWS	BLOCKS
ALBUMS	72	21561	244
ATTENDANCES	154	35621	748
CLIENTS	124	1500	28
CONCERTS	97	76348	1126
INVOLVEMENT	49	1224	13
LANGUAGES	10	3	5
MANAGERS	32	127	5
MUSICIANS	43	1649	13
NATIONALITIES	8	17	5
PERFORMANCES	56	856432	7300
PERFORMERS	29	695	5

TABLE_NAME	AVG_ROW_LEN	NUM_ROWS	BLOCKS
PUBLISHERS	20	30	5
SONGS	31	123698	622
STUDIOS	69	40	5
TOURS	39	5332	35
TRACKS	123	146275	2638

Default Values:

total number of records:

```
SQL> select sum(num_rows) from user_tables;

SUM(NUM_ROWS)
-------------
      1270552
```

total number of blocks:

```
SQL> select sum(blocks) from user_tables;

SUM(BLOCKS)
-----------
      12797
```

Now, some parameters of the original physical design were already selected by default: Bucket size: 8KB, PCTFREE = 10, PCTUSED = 60.

Moreover, the data files are serial and non consecutive with a primary index set on primary keys and the nature of the indexes are B trees.

By calculating the real density, we get 72,415%. Using the initial values.

This is the initial result of the run_test procedure with 10 iterations. As it can be seen, we get a lot of access: 281233.

```
SQL> begin pkg_costes.run_test(10); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 05/05/2023 11:05:54
TIME CONSUMPTION: 7384,2 milliseconds.
CONSISTENT GETS: 281233,4 blocks
```

To start with, we will analyze each process (update, query1 and query2) separately by using autotrace.

Query1:

For query1, we use the execution plan using autotrace traceonly and setting the time on.

```
Transcurrido: 00:00:04.43

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

-----------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                          | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                               |   175M|  8864M|       | 27218   (4)| 00:00:02 |
|   1 |  TEMP TABLE TRANSFORMATION               |                               |       |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9DB74F_334C22EB   |       |       |       |            |          |
|   3 |    HASH UNIQUE                           |                               |   307K|    29M|    33M| 11155   (1)| 00:00:01 |
|*  4 |     HASH JOIN                            |                               |   307K|    29M|       |  3994   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL                   | INVOLVEMENT                   |  1224 | 52632 |       |     5   (0)| 00:00:01 |
|   6 |      VIEW                                |                               |   288K|    16M|       |  3987   (1)| 00:00:01 |
|   7 |       SORT UNIQUE                        |                               |   288K|    28M|    17M|  3987   (1)| 00:00:01 |
|   8 |        UNION-ALL                         |                               |       |       |       |            |          |
|   9 |         INDEX FAST FULL SCAN             | PK_SONGS                      |   144K|  6061K|       |   170   (0)| 00:00:01 |
|  10 |         TABLE ACCESS FULL                | SONGS                         |   144K|  8317K|       |   172   (2)| 00:00:01 |
|  11 |   VIEW                                   | VW_FOJ_0                      |   175M|  8864M|       | 16063   (6)| 00:00:01 |
|* 12 |    HASH JOIN FULL OUTER                  |                               |   175M|    13G|  6856K| 16063   (6)| 00:00:01 |
|  13 |     VIEW                                 |                               |   134K|  5272K|       |  3899   (1)| 00:00:01 |
|  14 |      HASH GROUP BY                       |                               |   134K|    22M|       |  3899   (1)| 00:00:01 |
|* 15 |       HASH JOIN OUTER                    |                               |   134K|    22M|    14M|  3893   (1)| 00:00:01 |
|* 16 |        HASH JOIN                         |                               |   134K|    13M|       |   753   (1)| 00:00:01 |
|  17 |         TABLE ACCESS FULL                | ALBUMS                        | 22890 |   983K|       |    68   (0)| 00:00:01 |
|  18 |         TABLE ACCESS FULL                | TRACKS                        |   134K|  7908K|       |   684   (1)| 00:00:01 |
|  19 |        VIEW                              |                               |   307K|    21M|       |  1159   (1)| 00:00:01 |
|  20 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9DB74F_334C22EB   |   307K|    24M|       |  1159   (1)| 00:00:01 |
|  21 |     VIEW                                 |                               |   870K|    33M|       |  8839   (1)| 00:00:01 |
|  22 |      HASH GROUP BY                       |                               |   870K|   139M|       |  8839   (1)| 00:00:01 |
|* 23 |       HASH JOIN RIGHT OUTER              |                               |   870K|   139M|    24M|  8800   (1)| 00:00:01 |
|  24 |        VIEW                              |                               |   307K|    21M|       |  1159   (1)| 00:00:01 |
|  25 |         TABLE ACCESS FULL                | SYS_TEMP_0FD9DB74F_334C22EB   |   307K|    24M|       |  1159   (1)| 00:00:01 |
|  26 |        TABLE ACCESS FULL                 | PERFORMANCES                  |   870K|    78M|       |  1991   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------------------
```

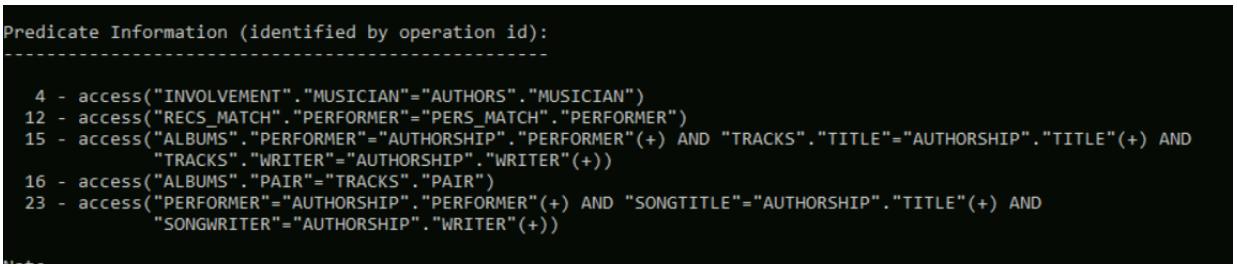
```
Estadisticas
----------------------------------------------------------
        104  recursive calls
         53  db block gets
      13338  consistent gets
       9116  physical reads
     138256  redo size
      24444  bytes sent via SQL*Net to client
       1917  bytes received via SQL*Net from client
         48  SQL*Net roundtrips to/from client
          4  sorts (memory)
          0  sorts (disk)
        695  rows processed
```

As we can see, it has 13338 accesses. The time required is not too big, as we expected for a query, around 4 seconds.

This amount of accesses is not good because we are only querying and by default, as is our case, tables have PCTFREE 10 and PCTUSED 60. Therefore, we are leaving 10% of the space of a bucket for future updates, but we can modify the PCTFREE of those tables that are queried to lower the accesses. However, we have to be careful not to decrease the PCTFREE of a table that is updated as is the case of tracks.

Moreover, in the execution plan we can see that a full scan is being performed in tables involvement, songs, albums, tracks and performances. The full scan of songs and involvement occurs because we are essentially joining those 2 tables. Therefore, each row of songs must be joined with the corresponding row of involvement, provoking that both tables must be read completely, causing a full scan using the combination key: musician. Moreover, the full scan of albums and tracks occurs because of a similar reason, we are joining both tables, therefore, we have to join each row of albums with the corresponding row of tracks, provoking a full scan using the combination key pair. What we essentially do, is taking one pair from albums for instance, and then full scan tracks looking for records with that same pair. Then, we choose the next pair in albums and do another full scan in tracks looking for records with that pair and so on. We keep doing this until we have full scanned albums. Finally, the full scan of performances occurs because of another join as well, this time is left join. However, when joining performances with authorship (composed of involvement and songs), we have to join each row of performances with the corresponding row of authorship, using the combination keys performer, title and writer.

In terms of costs, the cost of involvement is very low (5 cpu and 52632 bytes in memory) while for other tables such as songs we have 8317K Bytes used and a cost of cpu of 172. In albums, another full scan is performed where it uses 983K Bytes in memory and has a cost of 68. In tracks, another full scan is performed and uses 7908K Bytes in memory and has a cost of 684. Finally another full scan is performed in performances where it uses 68M Bytes and a cost of 1991 of cpu.



As it can be seen in this image with the predicate information, those tables that appear there, such as involvement, albums, tracks… are likely to be modified by adding an index, cluster or any other structure.

Once this has been analyzed, we think that the best solution is to add an index for songs, albums and performances as indexes are very good in selective processes and the where clauses, which appear in the shape of "using(a.pair = b.pair)" for example. We have decided not to do it in involvement nor tracks because in involvement it will not make much difference and in tracks it will be counterproductive in the future as that table will be used for an update (we will instead, use a cluster). Moreover, we will increase the tablespace for tables tracks and performances and change the PCTFREE and PCTUSED of the involved tables in the queries and update.

Moreover, as we can see, the costs of memory of the hash joins all over the execution plan are quite high, around 22 M although there is a hash join full outer that uses 13G. Additionally, the costs are very big as well, between 4k and 11k. We think that the best way to solve very big joins and group by is using clusters.

Summarizing, the initial physical design is not good for joins nor group by in general. Moreover, in selective processes, it is not that important to have PCTFREE, so for our design, we will lower the PCTFREE of those tables used in selective processes with the exception of tracks.

Query2:

For query2, we use the execution plan using autotrace traceonly and setting the time on.

```
Transcurrido: 00:00:03.15

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

-------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name         | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |              |     3 |   159 |       |  8228   (2)| 00:00:01 |
|*  1 |  COUNT STOPKEY           |              |       |       |       |            |          |
|   2 |   VIEW                   |              |     3 |   159 |       |  8228   (2)| 00:00:01 |
|*  3 |    SORT ORDER BY STOPKEY |              |     3 |   558 |       |  8228   (2)| 00:00:01 |
|   4 |     HASH GROUP BY        |              |     3 |   558 |       |  8228   (2)| 00:00:01 |
|*  5 |      HASH JOIN RIGHT OUTER|             |   870K|   154M|    12M|  8151   (1)| 00:00:01 |
|   6 |       VIEW               |              |   138K|    10M|       |   760   (2)| 00:00:01 |
|   7 |        HASH GROUP BY     |              |   138K|    14M|       |   760   (2)| 00:00:01 |
|*  8 |         HASH JOIN        |              |   138K|    14M|       |   754   (1)| 00:00:01 |
|   9 |          TABLE ACCESS FULL| ALBUMS       | 22890 |   983K|       |    68   (0)| 00:00:01 |
|  10 |          TABLE ACCESS FULL| TRACKS       |   138K|  9357K|       |   685   (1)| 00:00:01 |
|  11 |       TABLE ACCESS FULL  | PERFORMANCES |   870K|    86M|       |  1991   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

```
Estadisticas
----------------------------------------------------------
         89  recursive calls
         12  db block gets
      10615  consistent gets
       9814  physical reads
       2028  redo size
       1140  bytes sent via SQL*Net to client
       1247  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          3  sorts (memory)
          0  sorts (disk)
         10  rows processed

SQL>
```

As we can see, it has 10615 accesses. The time required is not too big as well, around 3 seconds.

This is not good because, as before, the PCTFREE and PCTUSED are not "optimized" for a selective process like a query because they are in the default value.

Additionally, in the execution plan, the tables where a full scan is performed are albums, tracks and performances. The full scan in tracks and albums occurs because we are joining 2 tables, therefore, we have to join each row of albums with the corresponding row of tracks, provoking that a full scan is performed in both tables using the combination key pair. Moreover, the second full scan between performances and the previously mentioned tables occurs because of a similar reason, which is that every row of performances has to be joined with the corresponding row of the other table (in this case subquery).

In terms of costs, in albums, 983K Bytes of memory are used and the cost of cpu is 68. For tracks, the memory used is 9357K Bytes and a cost of cpu of 685. Finally, for performances, memory used is 86M Bytes and a cost of 1991.

As before we will use an index for albums and performances and a cluster for tracks.

However, the most costly operations are the hash joins, order by and hash group by, which have a cost of cpu of around 8K and a high memory cost between 14M and 154M.

```
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")
```

For query2, the tables that are likely to be modified by adding structures such as an index or cluster are albums, tracks and other composed subqueries of the query2.

Summarizing, this query is cheaper than query1, mainly because fewer full scans and joins are performed.

As query2 is cheaper, we will give a bit more priority to query1.

Update:

```
SQL> UPDATE tracks set lyrics = dbms_random.string('a',dbms_random.value(900,1200))
  2        where searchk='B7303UX95490MX4//2';

1 fila actualizada.

Transcurrido: 00:00:00.04

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

-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT   |        |    28 | 56392 |   685   (1)| 00:00:01 |
|   1 |  UPDATE            | TRACKS |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| TRACKS |    28 | 56392 |   685   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

```
Estadisticas
-----------------------------------------------------------
          9  recursive calls
          9  db block gets
       2548  consistent gets
          2  physical reads
       2120  redo size
        494  bytes sent via SQL*Net to client
        933  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

First of all, we have to say that this result came from an update where seacrhk = fixed value ("B7303UX95490MX4//2") as in the workload, it did a for loop of 98 iterations. Moreover, the update is done 98 % of the time, so it makes a lot of sense to invest time in this process to lower the accesses.

However, we can calculate approximately the number of accesses that the update performs. We can do it by taking the total number of accesses when running the run_test and subtracting the sum of the queries. This way we get 281233-(13338+10615) = 257.280. However, below we do the analysis for the specific case stated above.

The consistent gets for a fixed value is 2548 and the time is 0,04 seconds (just for one iteration). To reduce the consistent gets we can increase the PCTFREE of tracks so that it has more space for the updates and maybe decrease the PCTUSED to lower the cost per update.

In terms of the execution plan, a full scan is performed in table tracks. This occurs because, although searchk is an identifier, because there is no searchk repeated, oracle does not know that, that is why, even if it finds the searchk we are looking for, it will do a full scan to check if there is one more.

To avoid doing the full scan we recommend implementing a cluster where its key is the pair, as commented above we use a cluster because it is more efficient in update operations than indexes.

Proposed improvements:

To increase performances and reduce the number access and time we propose to increase the PCTUSED to 90 and reduce the PCTFREE to 0 for the tables Performances, Albums, songs and Involvement and for the other tables are PCTUSED 80 and PCTFREE 5. But for tracks we

recommend to increase the PCTFREE to 20 and reduce the PCTUSED to 40. The reason for these changes is to increase the performance of the update since it is more frequent than the queries.

Moreover we will increase the tablespace to 16K for tables performances and tracks to decrease the number of buckets, as these 2 are very big tables and a lot of buckets are accessed in the fullscan.

Finally, we propose to use indexes for tracks, performances, songs and involvement and a cluster for tracks and albums (because it is more efficient in the join operation).

3 Physical Design

PCTUSED, PCTFREE:

For the physical design as analyzed previously, we decided for the table tracks to increase its PCTFREE to 20 and decrease its PCTUSED to 40. We increased the PCTFREE because we want to allow more space for future updates (since we are updating this table 98% of the time). We decrease the PCTUSED because we want to reduce the cost of update, in other words the reduction of PCTUSED means that a block is full at a lower space that results in an increase of the number of blocks used, however there is more space for future updates .

As for the other tables, since we only do the queries (2% of the time) , we decided to decrease (in the implicated tables) the PCTFREE to 0 and PCTUSED to 80. We decrease the PCTFREE because since we do not update those tables in our processes we don't need to allocate an space for the updates, however if this is for the application for a business model, we would reduce the PCTFREE to 5, in case we need to update some tables or insert new rows. We increased the PCTUSED because it would mean that there is more space for each block.

Now we will show screenshots of the execution plan for every process and then we will analyze the results.

QUERY1:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		174M	8798M		21135 (5)	00:00:01
1	TEMP TABLE TRANSFORMATION						
2	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9DEEE9_334C22EB					
3	HASH UNIQUE		186K	18M	20M	6909 (1)	00:00:01
* 4	HASH JOIN		186K	18M		2561 (1)	00:00:01
5	TABLE ACCESS FULL	INVOLVEMENT	1224	52632		5 (0)	00:00:01
6	VIEW		175K	9M		2555 (1)	00:00:01
7	SORT UNIQUE		175K	17M	10M	2555 (1)	00:00:01
8	UNION-ALL						
9	INDEX FAST FULL SCAN	PK_SONGS	87623	3679K		170 (0)	00:00:01
10	TABLE ACCESS FULL	SONGS	87623	5048K		171 (1)	00:00:01
11	VIEW	VW_FOJ_0	174M	8798M		14226 (7)	00:00:01
* 12	HASH JOIN FULL OUTER		174M	12G	6520K	14226 (7)	00:00:01
13	VIEW		128K	5013K		2921 (1)	00:00:01
14	HASH GROUP BY		128K	21M		2921 (1)	00:00:01
* 15	HASH JOIN OUTER		128K	21M	14M	2916 (1)	00:00:01
* 16	HASH JOIN		128K	12M		753 (1)	00:00:01
17	TABLE ACCESS FULL	ALBUMS	21639	929K		68 (0)	00:00:01
18	TABLE ACCESS FULL	TRACKS	128K	7520K		684 (1)	00:00:01
19	VIEW		186K	12M		704 (1)	00:00:01
20	TABLE ACCESS FULL	SYS_TEMP_0FD9DEEE9_334C22EB	186K	14M		704 (1)	00:00:01
21	VIEW		911K	34M		7901 (1)	00:00:01
22	HASH GROUP BY		911K	145M		7901 (1)	00:00:01
* 23	HASH JOIN RIGHT OUTER		911K	145M	15M	7861 (1)	00:00:01
24	VIEW		186K	12M		704 (1)	00:00:01
25	TABLE ACCESS FULL	SYS_TEMP_0FD9DEEE9_334C22EB	186K	14M		704 (1)	00:00:01
26	TABLE ACCESS FULL	PERFORMANCES	911K	82M		1786 (1)	00:00:01

Consistent gets =16195

query 2:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		3	159		8205 (2)	00:00:01
* 1	COUNT STOPKEY						
2	VIEW		3	159		8205 (2)	00:00:01
* 3	SORT ORDER BY STOPKEY		3	558		8205 (2)	00:00:01
4	HASH GROUP BY		3	558		8205 (2)	00:00:01
* 5	HASH JOIN RIGHT OUTER		911K	161M	11M	8124 (1)	00:00:01
6	VIEW		128K	10M		759 (2)	00:00:01
7	HASH GROUP BY		128K	13M		759 (2)	00:00:01
* 8	HASH JOIN		128K	13M		753 (1)	00:00:01
9	TABLE ACCESS FULL	ALBUMS	21639	929K		68 (0)	00:00:01
10	TABLE ACCESS FULL	TRACKS	128K	8648K		684 (1)	00:00:01
11	TABLE ACCESS FULL	PERFORMANCES	911K	90M		1786 (1)	00:00:01

consistent gets= 9836

UPDATE:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	UPDATE STATEMENT		1	123	685 (1)	00:00:01
1	UPDATE	TRACKS				
* 2	TABLE ACCESS FULL	TRACKS	1	123	685 (1)	00:00:01

consistent gets= 2520

Overall:

```
SQL> begin pkg_costes.run_test(10);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 08/05/2023 23:09:47
TIME CONSUMPTION: 5690,2 milliseconds.
CONSISTENT GETS: 312311,6 blocks
```

Analysis:

For the **Query 1 process** it is possible to observe that the cost of bytes remains the same (a small difference) but we can see that the cost on the CPU has decreased from 27218 to 21135. The decrease in CPU cost may be that when we increase the PCTUSED and decrease the PCTFREE we reduce the number of blocks needed, and we reduce the cost associated with reading the data. However, the consistent gets increases from 13338 to 16195. The increase of consistent gets may be the result of reducing the PCTUSED and increase of PCTFREE of the table tracks, because it is doing a full scan and joins with this table, in other words , it is accessing more buckets than previously (since a block is full with less rows). However with the combination of index and the PCTFREE and PCTUSED of the table tracks we can cancel the extra number of access.

For the **Query 2 process** the cost of bytes remains the same but now the cost of CPU remains almost the same, which would mean that the change of PCTFREE and PCTUSED doesn't impact, because the query is too small to notice the changes. However now the consistency gets improves from 10615 to 9836. The reason behind it, may be that, since we are doing less joins from the table tracks, the changes from albums benefits more than worsen from tracks.

For the **Update process**, we observe that the cost of bytes has been reduced and the cpu cost remains the same, however we see an improvement in the consistent gets from 2548 to 2520 (it may be seen small but when we update multiple times the

consistent gets will improve much more, because there is more available space for the updates and we don't need to use a new bucket).

Lastly we will analyze the **run_test procedure** with 10 iterations. As is possible to observe we see that the result has worsened because the consistent gets has increased from 281233 to 312311. The reason behind it may be that the consistent gets obtained from query 1 worsens more quickly than the improvement from query 2 and the update.

To improve this, we would recommend the increase of PCTUSED on tracks since is not doing much for the update process and prioritize more the query 1 because it has a heavier cost in consistent gets than the other processes or another reason would be to include an index on tracks to improve the time of the full scan.

Tablespace

We decided to increase the **tablespace** from 8Kb to 16Kb for the table tracks and performances. We change for these tables because these 2 tables are using a lot of buckets. By increasing the tablespace we would increase the performance when executing a full scan because when reading a bucket it would allow more data to be in the cache in memory and use less buckets in total. Also, by increasing the BS (tablespace) we would be increasing the real density.

These are the results when changing the PCTFREE, PCTUSED and with tablespaces added.

query1:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		178M	9004M		25415 (4)	00:00:01
1	TEMP TABLE TRANSFORMATION						
2	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9DEF56_334C22EB					
3	HASH UNIQUE		300K	29M	32M	10916 (1)	00:00:01
* 4	HASH JOIN		300K	29M		3912 (1)	00:00:01
5	TABLE ACCESS FULL	INVOLVEMENT	1224	52632		5 (0)	00:00:01
6	VIEW		282K	15M		3905 (1)	00:00:01
7	SORT UNIQUE		282K	27M	16M	3905 (1)	00:00:01
8	UNION-ALL						
9	INDEX FAST FULL SCAN	PK_SONGS	141K	5929K		170 (0)	00:00:01
10	TABLE ACCESS FULL	SONGS	141K	8135K		172 (2)	00:00:01
11	VIEW	VW_FOJ_0	178M	9004M		14500 (7)	00:00:01
* 12	HASH JOIN FULL OUTER		178M	13G	8152K	14500 (7)	00:00:01
13	VIEW		160K	6265K		3779 (1)	00:00:01
14	HASH GROUP BY		160K	27M		3779 (1)	00:00:01
* 15	HASH JOIN OUTER		160K	27M	17M	3772 (1)	00:00:01
* 16	HASH JOIN		160K	15M		545 (1)	00:00:01
17	TABLE ACCESS FULL	ALBUMS	22092	949K		70 (0)	00:00:01
18	TABLE ACCESS FULL	TRACKS	160K	9397K		474 (1)	00:00:01
19	VIEW		300K	20M		1133 (1)	00:00:01
20	TABLE ACCESS FULL	SYS_TEMP_0FD9DEF56_334C22EB	300K	23M		1133 (1)	00:00:01
21	VIEW		757K	28M		7597 (1)	00:00:01
22	HASH GROUP BY		757K	121M		7597 (1)	00:00:01
* 23	HASH JOIN RIGHT OUTER		757K	121M	24M	7564 (1)	00:00:01
24	VIEW		300K	20M		1133 (1)	00:00:01
25	TABLE ACCESS FULL	SYS_TEMP_0FD9DEF56_334C22EB	300K	23M		1133 (1)	00:00:01
26	TABLE ACCESS FULL	PERFORMANCES	757K	68M		1380 (1)	00:00:01

consistent gets= 7524

query 2:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		3	159		6875 (2)	00:00:01
* 1	COUNT STOPKEY						
2	VIEW		3	159		6875 (2)	00:00:01
* 3	SORT ORDER BY STOPKEY		3	558		6875 (2)	00:00:01
4	HASH GROUP BY		3	558		6875 (2)	00:00:01
* 5	HASH JOIN RIGHT OUTER		757K	134M	14M	6809 (1)	00:00:01
6	VIEW		160K	12M		553 (2)	00:00:01
7	HASH GROUP BY		160K	17M		553 (2)	00:00:01
* 8	HASH JOIN		160K	17M		546 (1)	00:00:01
9	TABLE ACCESS FULL	ALBUMS	22092	949K		70 (0)	00:00:01
10	TABLE ACCESS FULL	TRACKS	160K	10M		475 (1)	00:00:01
11	TABLE ACCESS FULL	PERFORMANCES	757K	75M		1380 (1)	00:00:01

consistent gets =5566

UPDATE

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	UPDATE STATEMENT		14	28196	475 (1)	00:00:01
1	UPDATE	TRACKS				
* 2	TABLE ACCESS FULL	TRACKS	14	28196	475 (1)	00:00:01

Consistent gets= 1345

Overall:

```
SQL> begin pkg_costes.run_test(10);end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 08/05/2023 23:55:18
TIME CONSUMPTION: 8280,7 milliseconds.
CONSISTENT GETS: 160580,2 blocks
```

For the query 1 we can observe that the cost of bytes increase slightly maybe is related on how the server performed when doing the trace only but on the other hand we observe that the cost of CPU has increased from a total of 21135 to 25415 we think that it could be related to, when increasing the tablespace we observe that the joins are having more difficulties (But it could also be related with the available resources when doing the query1). However the biggest change we observe is the consistent gets, because we went from 16195 to 7524. The reason behind this improvement is that when increasing the tablespace we are allowing more data to fit inside the bucket and we need less accesses to blocks to perform a full scan.

For the query 2, the cost of bytes and cpu is mostly the same. However we can see an improvement in the consistent gets that went from 9836 to 5566, the reasons are as stated on query 1 that when we increase the tablespace we require less access to buckets since there are less than before.

For the update process, we observe that the consistent gets has improved from 2520 to 1345 because we incremented the BS of tracks allowing to fit more data on a single bucket meaning we would need less access to do the full scan.

Lastly we will analyze the **run_test procedure** with 10 iterations. We had a very big improvement because it went from 312311 to 160580 access. In other words we would recommend implementing the change of tablespace, because it has a big impact on the number of accesses.

Also 160580 - (7524 + 5566) = 147.490 which is the number of accesses performed by the update process. In the screenshot of the update, we only show the accesses of a specific case.

Code of the tables

```
CREATE TABLE TRACKS (PAIR CHAR(15),...),
)
PCTFREE 20
PCTUSED 40
tablespace TAB_16k
--cluster album_tracks (PAIR)
;
```

```
CREATE TABLE ALBUMS(PAIR CHAR(15),...)pctfree 0
PCTUSED 90
--tablespace TAB_16k
--cluster album_tracks (PAIR)
;
```

```
CREATE TABLE SONGS (title      VARCHAR2(50),...)pctfree 0
PCTUSED 90
;
```

```
CREATE TABLE INVOLVEMENT (band  VARCHAR2(50),...)pctfree 0
PCTUSED 90;
```

```
CREATE TABLE PERFORMANCES (performer  VARCHAR2(50),...)pctfree 0
PCTUSED 90
tablespace TAB_16k;
```

Indexes and Hash Clusters:

We will create an index for table tracks using the column search key. This should decrease a lot of the accesses because, firstly, it is a process that occurs 98% of the time, therefore, an increase in the performance of this process will mean an increase in the whole performance of the processes as most accesses occur because of this update process.

```
SQL> begin pkg_costes.run_test(10); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 06/05/2023 12:04:51
TIME CONSUMPTION: 3907,8 milliseconds.
CONSISTENT GETS: 11631,7 blocks
```

```
create unique index ind1 on tracks(searchk) tablespace TAB_2k;
```

As the searchk is composed of pair and sequ, we know that, even if searchk is not a primary key, the index must be primary, as it is an identifying column. The reason why the accesses have decreased a lot is because by using the primary index, we use a B tree, in which we will look for the searchk that we want to use to update. Instead of doing a full scan of the whole table to look for the searchk, we just move from side to side of each subtree and once we are in the node that contains the desired value, we full scan it (much lower than the full table) until we find it.

We also tried adding some more indexes such as these ones:

```
create index ind3 on performances(performer,songtitle,songwriter) tablespace TAB_2k;

create index ind4 on songs(title,writer,cowriter) tablespace TAB_2k;

create index ind5 on involvement(musician) tablespace TAB_2k;
```

However, due to the small amount of access that these indexes may decrease, the improvement is almost non-existent, so we decided not to include them. There are very few accesses that these indexes may decrease because they are meant for the queries and each query is being performed 1% of the time.

Therefore, the only index that we use by now is the searchk index.

For the cluster, we have decided to do an indexed cluster for tables tracks and albums, as these 2 tables are being joined in both queries with the key pair. Therefore, this cluster will be composed of rows resulting from the joining of both tables using key pair. For this cluster, we had to change PCTFREE, PCTUSED and tablespace of table albums so that it matched with that of tracks because of the cluster.

The index is secondary so we will have a b+tree.

```
SQL>
SQL>
SQL> begin pkg_costes.run_test(1);
  2  end;
  3  /
Iteration 1
RESULTS AT 08/05/2023 20:50:28
TIME CONSUMPTION: 9963 milliseconds.
CONSISTENT GETS: 107046 blocks
```

This is the result after executing the test with the index and the cluster only one time. As it can be seen, we do more than 100.000 accesses therefore, we have decided not to use this cluster. The reason for this huge increase of accesses may be due to the index of the cluster which is by pair. When rows of tracks and albums are inserted, they are inserted into the cluster and in here, the rows are identified by pair and sequ.

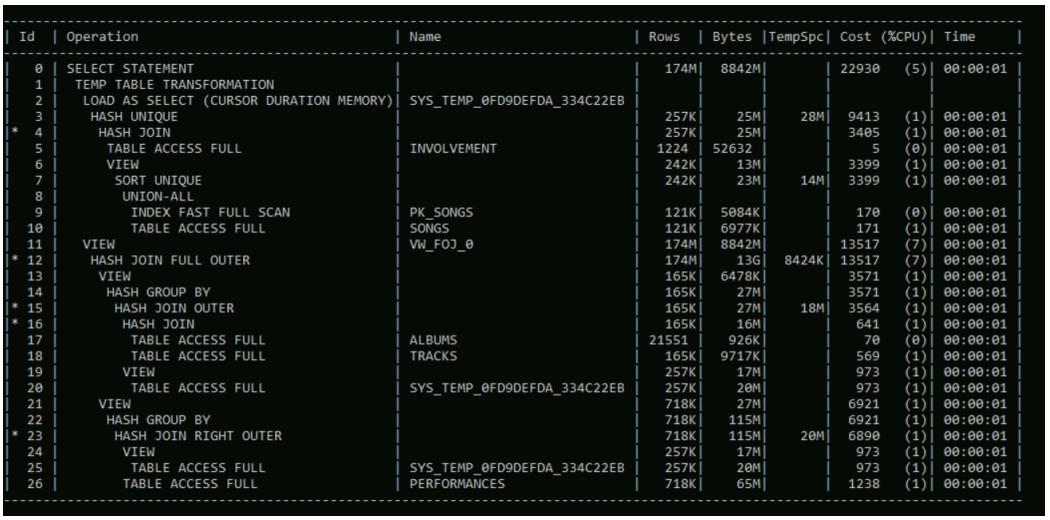
Therefore, when updating tracks, we are using the search key searchk, consequently, when updating, we have to enter into the cluster (because tracks is in the cluster) which has the shape of a b+tree but in this tree, the values of each entry are pair, and pair is not an identifier, therefore an index full scan must be done (because pair does not give us enough information to stop looking for more values) and as we are in a b+tree, we have to reach the bottom of the tree and in there, we follow each external pointer of each entry of each leave to a bucket and in there, search by attribute searchk and once we find it stop. However, to find that searchk we have traversed a very large amount of buckets, because each pointer of the leaves leads us to a bucket,

if searchk is not in that bucket, we go to the following pointer and repeat this until we find the searchk in the corresponding bucket.

Thanks to this analysis, we realized why the cluster that we have implemented is not good, which is basically because of a wrong selection of the indexing key (because pair is not identifiable) .

After creating the index, and removing the cluster, the processes stay like this (explication of each improvement was done before) :

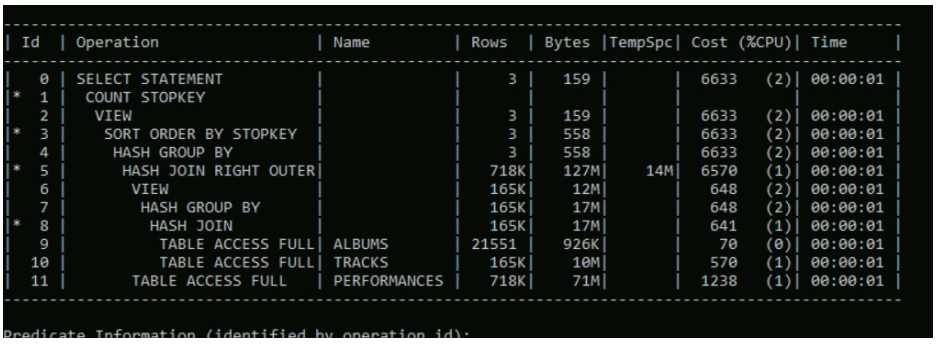
query1:



Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		174M	8842M		22930 (5)	00:00:01
1	TEMP TABLE TRANSFORMATION						
2	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9DEFDA_334C22EB					
3	HASH UNIQUE		257K	25M	28M	9413 (1)	00:00:01
* 4	HASH JOIN		257K	25M		3405 (1)	00:00:01
5	TABLE ACCESS FULL	INVOLVEMENT	1224	52632		5 (0)	00:00:01
6	VIEW		242K	13M		3399 (1)	00:00:01
7	SORT UNIQUE		242K	23M	14M	3399 (1)	00:00:01
8	UNION-ALL						
9	INDEX FAST FULL SCAN	PK_SONGS	121K	5084K		170 (0)	00:00:01
10	TABLE ACCESS FULL	SONGS	121K	6977K		171 (1)	00:00:01
11	VIEW	VW_FOJ_0	174M	8842M		13517 (7)	00:00:01
* 12	HASH JOIN FULL OUTER		174M	13G	8424K	13517 (7)	00:00:01
13	VIEW		165K	6478K		3571 (1)	00:00:01
14	HASH GROUP BY		165K	27M		3571 (1)	00:00:01
* 15	HASH JOIN OUTER		165K	27M	18M	3564 (1)	00:00:01
* 16	HASH JOIN		165K	16M		641 (1)	00:00:01
17	TABLE ACCESS FULL	ALBUMS	21551	926K		70 (0)	00:00:01
18	TABLE ACCESS FULL	TRACKS	165K	9717K		569 (1)	00:00:01
19	VIEW		257K	17M		973 (1)	00:00:01
20	TABLE ACCESS FULL	SYS_TEMP_0FD9DEFDA_334C22EB	257K	20M		973 (1)	00:00:01
21	VIEW		718K	27M		6921 (1)	00:00:01
22	HASH GROUP BY		718K	115M		6921 (1)	00:00:01
* 23	HASH JOIN RIGHT OUTER		718K	115M	20M	6890 (1)	00:00:01
24	VIEW		257K	17M		973 (1)	00:00:01
25	TABLE ACCESS FULL	SYS_TEMP_0FD9DEFDA_334C22EB	257K	20M		973 (1)	00:00:01
26	TABLE ACCESS FULL	PERFORMANCES	718K	65M		1238 (1)	00:00:01

consistent gets: 6290

query2:



Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		3	159		6633 (2)	00:00:01
* 1	COUNT STOPKEY						
2	VIEW		3	159		6633 (2)	00:00:01
* 3	SORT ORDER BY STOPKEY		3	558		6633 (2)	00:00:01
4	HASH GROUP BY		3	558		6633 (2)	00:00:01
* 5	HASH JOIN RIGHT OUTER		718K	127M	14M	6570 (1)	00:00:01
6	VIEW		165K	12M		648 (2)	00:00:01
7	HASH GROUP BY		165K	17M		648 (2)	00:00:01
* 8	HASH JOIN		165K	17M		641 (1)	00:00:01
9	TABLE ACCESS FULL	ALBUMS	21551	926K		70 (0)	00:00:01
10	TABLE ACCESS FULL	TRACKS	165K	10M		570 (1)	00:00:01
11	TABLE ACCESS FULL	PERFORMANCES	718K	71M		1238 (1)	00:00:01

Predicate Information (identified by operation id):

consistent gets:5025

Update:

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | UPDATE STATEMENT   |       |     1 |  2014 |     3   (0)| 00:00:01 |
|   1 |  UPDATE            | TRACKS|       |       |            |          |
|*  2 |   INDEX UNIQUE SCAN| IND1  |     1 |  2014 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

As it can be seen (individual update), the index that we created is being used and the cost is very low. Analysis of why the index is good has already been done before.

4 Evaluation

For the initial design, the result of the run_test procedure was this one:

```
SQL> begin pkg_costes.run_test(10); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 05/05/2023 11:05:54
TIME CONSUMPTION: 7384,2 milliseconds.
CONSISTENT GETS: 281233,4 blocks
```

As calculated before, the accesses done by the query1 are: 13338, by query2: 10615 and by the update, approximately 257280.

After introducing the new physical design, with the changes in PCTFREE, PCTUSED, tablespaces and the index (not the cluster because it worses the accesses), the result of the run_test is:

```
SQL> begin pkg_costes.run_test(10); end;
  2  /
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 06/05/2023 12:04:51
TIME CONSUMPTION: 3907,8 milliseconds.
CONSISTENT GETS: 11631,7 blocks
```

As it can be seen, the change is huge from the initial design to our implementation, decreasing from around 280000 accesses to just over 11000. Moreover, the cost of query 1 after implementing our physical design is 6290, for query2, 5025, therefore, for the update, 11631-(6290+5025) = 316.

This improvement occurs thanks to the following characteristics of our physical design:

PCTFREE and PCTUSED: We changed the PCTFREE and PCTUSED only of those tables that we used in the workload. For tracks we decided to increase PCTFREE from 10 to 20 so that there is more space for updates and there is no need to create new buckets because of lack of space. This however, produces a slight increase in the accesses of the queries, because what we are doing is decreasing the size of the part of the bucket where we store data, therefore, less information is inserted into a single bucket and more buckets must be used. Nevertheless, as updates are done 98% of the time, this increase in PCTFREE for tracks will mean a huge advantage in later steps. For the rest of the tables that are used in the queries, we decreased the PCTFREE to 0 to allow as much space occupied as possible to decrease the accesse. However, as stated before, the change in tracks affects but in a negative way to the queries because tracks is a very huge table, and if we need more buckets for it, even bigger.

Tablespace: After analyzing the blocks used by each table using select blocks from user_tables, we realized that table performances and tracks occupied a lot of buckets, and this was increasing the number of accesses in the queries a lot. Therefore, a way to decrease the number of accesses is to increase the size of a bucket, therefore, the data that was stored in 2 buckets, can now be stored in just one (not exactly like that, but it's the idea). From the first screenshot of the blocks and this one, we can see a huge improvement in tracks and performances, of about half of the initial buckets thanks to changing the tablespace.

TABLE_NAME	BLOCKS
ALBUMS	244
ATTENDANCES	622
CLIENTS	28
CONCERTS	1126
INVOLVEMENT	13
LANGUAGES	5
MANAGERS	5
MUSICIANS	13
NATIONALITIES	5
PERFORMANCES	3271
PERFORMERS	5

TABLE_NAME	BLOCKS
PUBLISHERS	5
SONGS	622
STUDIOS	5
TOURS	35
TRACKS	1507

Indexes and clusters: Finally, we implemented a cluster and several indexes, however, as stated in the previous section, some of the indexes and the cluster were

not improving our design so we decided to take them out. However we kept one index which is in table tracks attribute searchk. This attribute is very interesting because it is made up of the primary keys of tracks, which means that searchk is an identifier, therefore it must be a primary index. Therefore, we have a b tree which is actually pretty useful for our purposes because as seen in the execution plan, in tracks, a full scan is performed when updating, so with the b tree, instead of full scanning the whole table, we will just need to move from side to side of the tree until we find the node where the searchk is and then, full scan that small portion until we find the desired value. This was the biggest improvement we saw as we decreased the accesses from about 160000 (after PCTFREE, PCTUSED and tablespace implemented) to 11000.

5 Concluding Remarks

To conclude, in this report, we first analyzed the initial physical design that we were given, then we did our own physical design by mainly changing PCTFREE, PCTUSED, tablespaces, adding indexes and a cluster and finally we compared both designs. We first identified and analyzed the statistics of the initial design, and after the work, we again analyzed the statistics of the resulting design, after our implementation. The final results were very satisfying for us, not just because of good statistics, about 11.000 accesses, but also because of understanding what happened when we changed some aspects of the designs such as PCTFREE to increase or decrease the size for updates and data in a bucket, or the indexes, to change the structure of a given process.

In terms of achievement, as said before, we are very satisfied with how this lab has helped us to understand the theory given in class. By analyzing the results, we were able to identify and understand which were the changes that had to be done in order to increase the performance of the workload. However, we feel that we may have spent too much time trying to get the best possible design when the lab work was just meant to analyze.

As feedback for future editions, we would like to let you know that through the distinct sections of the template given, we felt like we were stating and explaining the same things as in previous sections but with a different scope (state, analyze, describe, settle). We are not sure if that was the idea, to explain the same thing but with different scope, but we just wanted to let you know.

Finally, we can say that we really liked the subject and the way it was explained in both theory and practice classes. The only thing that we found a little bit overwhelming was the large amount of theory given in the theory classes but apart from that, the rest of the course was excellent for us.