| **Group:** | **89** | **Lab User** | **286** |
| --- | --- | --- | --- |
| **Student:** | **Alberto Pascau Sáez** | **NIA:** | **100495775** |
| **Student:** | **Silvia Bayo Martínez** | **NIA:** | **100495811** |
| **Student:** | **Carmen Serrano Pérez** | **NIA:** | **100495711** |

# Introduction

This assignment focuses on the analysis and improvement of the physical design of a relational database using Oracle. The objective is to enhance performance by reducing access cost and response time for the most representative queries. Our goal is to improve performance through physical design strategies including the use of indexes, clustering, and hash-based storage.

To achieve this, we begin by restoring the original database, measuring the cost of the initial workload, and analyzing each query's execution plan and access behavior. We then propose, implement, and test multiple physical design alternatives for each query measuring their effect on performance. Finally, we select the most effective combination of physical structures and validate the improvement through a final workload execution, comparing the results with the initial state.

# Analysis

## Output of the RUN\_TEST PROCEDURE

The run test procedure was run with 10 as the input for the number of iterations. The result obtained was:

begin

2 pKG\_COSTES.RUN\_TEST(10);

3 end;

4 /

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

Iteration 6

Iteration 7

Iteration 8

Iteration 9

Iteration 10

RESULTS AT 23/04/2025 17:41:45

TIME CONSUMPTION (run): 1069.1 milliseconds.

CONSISTENT GETS (workload):60493 acc

CONSISTENT GETS (weighted average):6049.3 acc

PL/SQL procedure successfully completed.



From this output we can analyse two important pieces of information.

1. The time consumption for the tests was of: 1069.1 milliseconds
2. The consistent gets (weighted average) had: 6049.3 acc

After developing all optimizations the tests will be run again to check if the development made any change and analyse the quantity of that change.

## QUERY 1: Analysis

Given the first query:

select \* from editions where pub\_place='…'; -- values like 'Madrid', 'Segovia', or 'Barataria', to name a few examples



We’ll perform an initial analysis by setting the autotrace and executing queries for Madrid and Segovia. This is because Madrid is a popular value (many rows will be returned) while Segovia isn’t. And this difference will help us find a balanced approach to the optimization of the query.

SQL> select \* from editions where pub\_place='Madrid';

82450 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 1741989577

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 207 | 44712 | 2054 (1)| 00:00:01 |

|\* 1 | TABLE ACCESS FULL| EDITIONS | 207 | 44712 | 2054 (1)| 00:00:01 |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("PUB\_PLACE"='Madrid')

Statistics

----------------------------------------------------------

41 recursive calls

86 db block gets

12927 consistent gets

958 physical reads

17436 redo size

17922970 bytes sent via SQL\*Net to client

60833 bytes received via SQL\*Net from client

5498 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

82450 rows processed

SQL> select \* from editions where pub\_place='Segovia';

71 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 1741989577

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 207 | 44712 | 2054 (1)| 00:00:01 |

|\* 1 | TABLE ACCESS FULL| EDITIONS | 207 | 44712 | 2054 (1)| 00:00:01 |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("PUB\_PLACE"='Segovia')

Statistics

----------------------------------------------------------

1 recursive calls

0 db block gets

7566 consistent gets

4494 physical reads

0 redo size

16790 bytes sent via SQL\*Net to client

418 bytes received via SQL\*Net from client

6 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

71 rows processed



We can see that the autotrace shows us a lot of data, from this data, the important output we focus on is the number of reads and consistent gets for each query:

* MADRID: 12927 consistent gets ; 958 physical reads
* SEGOVIA: 7566 consistent gets ; 4494 physical reads

Interesting also to notice the difference between the number of rows returned for Madrid and for Segovia. Madrid has, as hypothesized, many more than segovia.

We can also use the following query to see the data of all tables to see the editions data, this analysis will also be useful for other queries as we focus only on table editions.

SQL> select table\_name, avg\_row\_len, num\_rows, blocks from user\_tables;

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

ASSIGN\_BUS

ASSIGN\_DRV

BIBUS 0 0 5

BIBUSERO 0 0 5

BIBUSERO\_STATE 0 0 5

BIBUSES

BIBUS\_STATE 0 0 5

BOOKS

BOOK\_EDITIONS 0 0 0

BOOK\_ENTRIES 0 0 5

BOOK\_LOANS 0 0 0

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

BOOK\_RESERVATIONS 0 0 5

COPIES

COURSES 0 0 5

DRIVERS

EDITIONS 221 240632 7552

ENROLLMENTS 0 0 5

ID\_ROUTES 6 150 5

LOANS

MORE\_AUTHORS

MUNICIPALITIES

MUNICIPALITY 0 0 0

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

M\_LIBRARY 0 0 0

POSTS

PROFESSORS 0 0 5

ROUTES

SERVICES

SROUTES 0 0 5

STOPS

STUDENTS 0 0 5

USERS

31 rows selected.



From this output, we are only interested in the non-zero values, specifically those from the ediciones table, as it is the only one that has been analyzed.

We can observe that the table contains a total of 240,632 entries and 7,552 blocks. These values will be used for future comparisons after performing the optimizations.

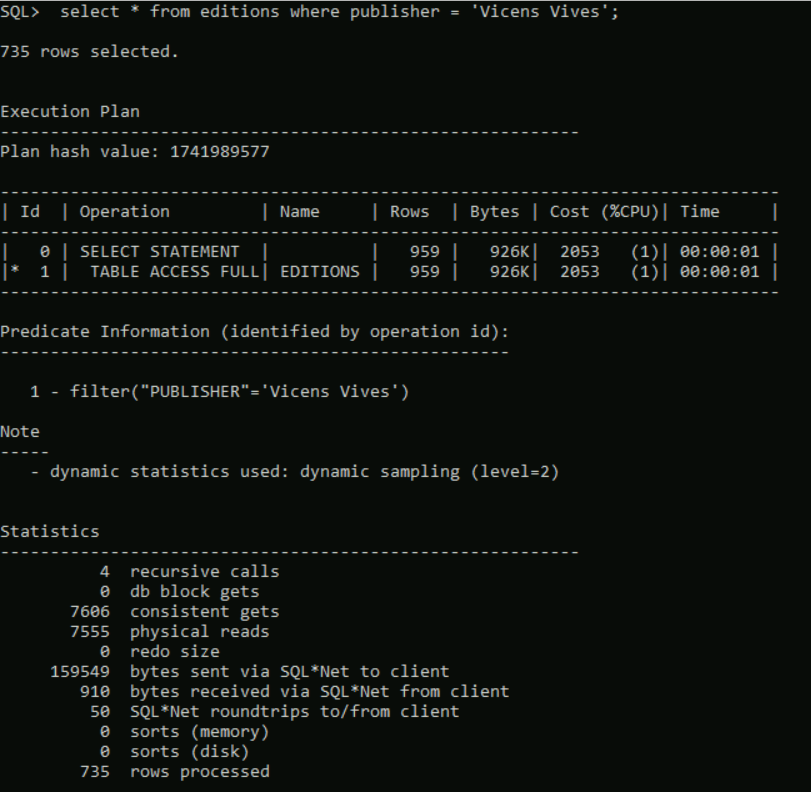
## QUERY 2: Analysis

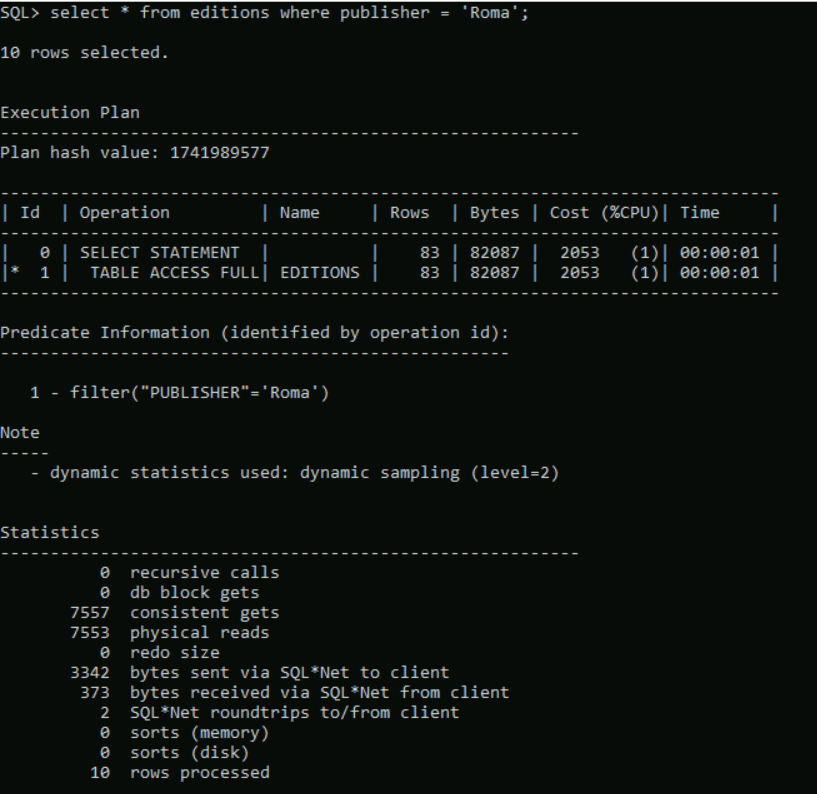
Given the second query:

select \* from editions where publisher = '...';



We performed an initial analysis by executing the queriy with two representative values: 'Vicens Vives' (a frequent publisher) and 'Roma' (a less frequent one).





From the execution of the second query using two different values, we observed the following:

* For 'Vicens Vives' (735 rows returned), the system required 7606 consistent gets and 7555 physical reads
* For 'Roma' (10 rows returned), the system still required 7557 consistent gets and 7553 physical reads

Despite the significant difference in the number of rows returned (735 vs. 10), both queries triggered a full table scan, with nearly identical access costs. This demonstrates that the current physical design does not take advantage of the column's selectivity.

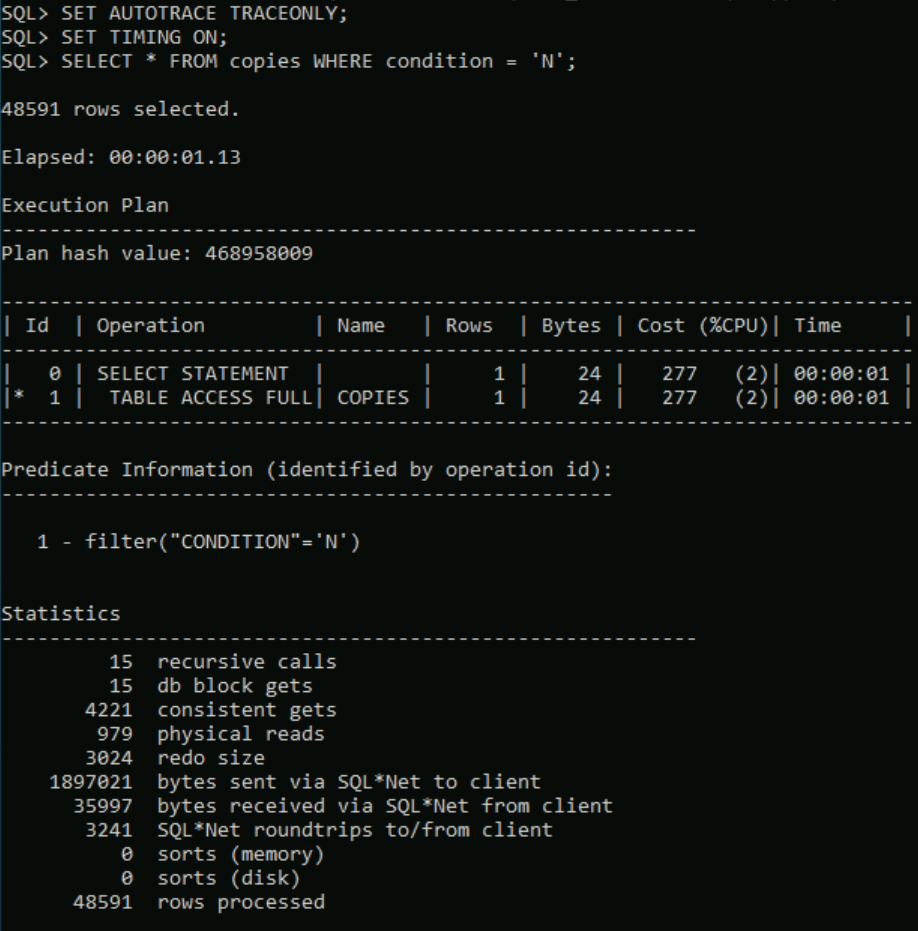
## QUERY 3: Analysis

Given the third query, we have performed an initial analysis

SELECT \* FROM copies WHERE condition = 'N';



This query searches all the copies with a specific conservation condition value which can take the following values: ‘N’, ‘G’, ‘W’, ‘V’, ‘D’ .



We executed the query with the condition value set to ‘N’ and enabling autotrace and timing. As we can see in the screenshot, we have gotten the following results, the execution plan of this query is the table full access. The cost is of 278, the consistent gets are relatively high, meaning that a considerable amount of logical reads are needed in order to retrieve the matching rows. And the physical reads is also high, meaning that there is significant disk access.

## QUERY 4:Analysis

Given the fourth query:

select \* from editions;

We’ll perform the analisis with the autotrace. Besides, we’ll also use the output of last part of the analysis performed for the first query

The output after performing the query with the autotrace on is:

SQL> select \* from editions;

240632 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 1741989577

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 240K| 49M| 2055 (1)| 00:00:01 |

| 1 | TABLE ACCESS FULL| EDITIONS | 240K| 49M| 2055 (1)| 00:00:01 |

------------------------------------------------------------------------------

Statistics

----------------------------------------------------------

42 recursive calls

87 db block gets

23119 consistent gets

3071 physical reads

17668 redo size

57147402 bytes sent via SQL\*Net to client

176814 bytes received via SQL\*Net from client

16044 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

240632 rows processed



As in previous queries we focus on the consistent gets and physical reads, there is a total of 23119 consistent gets and 3071 physical reads

# Physical Design

## QUERY 1: Physical design

In order to optimize the query we’ll see three methods used.

### Index

We’ll create a secondary index on editions. After this we’ll perform our queries with the index hint so that the index search is forced and we have no doubts that we are using the index we just created.

create index idx\_ed1 on editions(pub\_place);

-- Check the query is being executed with the index

select \* from editions where pub\_place='Madrid';

-- If not, force the query to execute with the index

select /\*+ index(editions) \*/ \* from editions

where pub\_place='Madrid';

-- For trying with segovia

select /\*+ index(editions) \*/ \* from editions

where pub\_place='Segovia';

-- For madrid, it is better to do a fullscan

select /\*+ full(editions) \*/ \* from editions

where pub\_place='Madrid';



Now we need to analyze what this index did by running again the autotrace on both queries. After doing so we find the following results:

1. For madrid with index:

SQL> select /\*+ index(editions) \*/ \* from editions where pub\_place='Madrid';

82450 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 4255503933

------------------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 192 | 41280 | 78 (0)| 00:00:01 |

| 1 | TABLE ACCESS BY INDEX ROWID BATCHED| EDITIONS | 192 | 41280 | 78 (0)| 00:00:01 |

|\* 2 | INDEX RANGE SCAN | IDX\_ED1 | 192 | | 3 (0)| 00:00:01 |

------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("PUB\_PLACE"='Madrid')

Statistics

----------------------------------------------------------

1 recursive calls

0 db block gets

18261 consistent gets

7721 physical reads

0 redo size

19595058 bytes sent via SQL\*Net to client

60859 bytes received via SQL\*Net from client

5498 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

82450 rows processed



1. For segovia with index:

SQL> select /\*+ index(editions) \*/ \* from editions where pub\_place='Segovia';

71 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 4255503933

------------------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 192 | 41280 | 78 (0)| 00:00:01 |

| 1 | TABLE ACCESS BY INDEX ROWID BATCHED| EDITIONS | 192 | 41280 | 78 (0)| 00:00:01 |

|\* 2 | INDEX RANGE SCAN | IDX\_ED1 | 192 | | 3 (0)| 00:00:01 |

------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("PUB\_PLACE"='Segovia')

Statistics

----------------------------------------------------------

1 recursive calls

0 db block gets

80 consistent gets

2 physical reads

0 redo size

18569 bytes sent via SQL\*Net to client

444 bytes received via SQL\*Net from client

6 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

71 rows processed



1. For madrid with a full scan:

select /\*+ full(editions) \*/ \* from editions where pub\_place='Madrid';

82450 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 1741989577

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 192 | 41280 | 2054 (1)| 00:00:01 |

|\* 1 | TABLE ACCESS FULL| EDITIONS | 192 | 41280 | 2054 (1)| 00:00:01 |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("PUB\_PLACE"='Madrid')

Statistics

----------------------------------------------------------

1 recursive calls

0 db block gets

12889 consistent gets

7552 physical reads

0 redo size

17923258 bytes sent via SQL\*Net to client

60854 bytes received via SQL\*Net from client

5498 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

82450 rows processed

SQL>



* We see that for Madrid, with the inde, the number of consistent gets is higher than the initial one, this is not good.   
  Value of consistent gets before: 12927  
  Value of consistent gets after: 18261
* However it works for low-coincidence values, it is more efficient, see with segovia where the number of entries decreases  
  Value of consistent gets before: 7566  
  Value of consistent gets after:80

After looking at the results we can also think about performing a full scan with Madrid (as shown in the code above), with that we see that the full scan is more optimal than the secondary index search.

### Index clustering

Because the index is not really making that much of a difference for Madrid we’ll implement a cluster on the pub\_place property of the editions table.

drop cluster places;

create cluster places(pub\_place varchar2(50));

create table editions(..) cluster places(pub\_place);

create index idx\_places on cluster places;



The editions table of the NEW\_creation.sql script is then modified as:

CREATE TABLE Editions(

ISBN VARCHAR2(20),

TITLE VARCHAR2(200) NOT NULL,

AUTHOR VARCHAR2(100) NOT NULL,

LANGUAGE VARCHAR2(50) default('Spanish') NOT NULL,

ALT\_LANGUAGES VARCHAR2(50),

EDITION VARCHAR2(50),

PUBLISHER VARCHAR2(100),

EXTENSION VARCHAR2(50),

SERIES VARCHAR2(50),

COPYRIGHT VARCHAR2(20),

PUB\_PLACE VARCHAR2(50),

DIMENSIONS VARCHAR2(50),

PHY\_FEATURES VARCHAR2(200),

MATERIALS VARCHAR2(200),

NOTES VARCHAR2(500),

NATIONAL\_LIB\_ID VARCHAR2(20) NOT NULL,

URL VARCHAR2(200),

CONSTRAINT pk\_editions PRIMARY KEY(isbn),

CONSTRAINT uk\_editions UNIQUE (national\_lib\_id),

CONSTRAINT fk\_editions\_books FOREIGN KEY(title,author) REFERENCES books(title,author)

) cluster places(pub\_place);



See that the cluster is added in the last line.

After running the cluster, the results are the following:

1. For madrid with a cluster index:

 select \* from editions where pub\_place='Madrid';

no rows selected

Execution Plan

----------------------------------------------------------

Plan hash value: 9953560

-----------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

-----------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | 989 | 1 (0)| 00:00:01 |

| 1 | TABLE ACCESS CLUSTER| EDITIONS | 1 | 989 | 1 (0)| 00:00:01 |

|\* 2 | INDEX UNIQUE SCAN | IDX\_PLACES | 1 | | 1 (0)| 00:00:01 |

-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("PUB\_PLACE"='Madrid')

Note

-----

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

Statistics

----------------------------------------------------------

8 recursive calls

0 db block gets

17 consistent gets

5 physical reads

0 redo size

1467 bytes sent via SQL\*Net to client

367 bytes received via SQL\*Net from client

1 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

0 rows processed



1. For segovia with cluster index:

select \* from editions where pub\_place='Segovia'

2 ;

no rows selected

Execution Plan

----------------------------------------------------------

Plan hash value: 9953560

-----------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

-----------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | 989 | 1 (0)| 00:00:01 |

| 1 | TABLE ACCESS CLUSTER| EDITIONS | 1 | 989 | 1 (0)| 00:00:01 |

|\* 2 | INDEX UNIQUE SCAN | IDX\_PLACES | 1 | | 1 (0)| 00:00:01 |

-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("PUB\_PLACE"='Segovia')

Note

-----

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

Statistics

----------------------------------------------------------

5 recursive calls

0 db block gets

11 consistent gets

5 physical reads

0 redo size

1467 bytes sent via SQL\*Net to client

364 bytes received via SQL\*Net from client

1 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

0 rows processed

SQL>



With clustering we are obtaining an incredible result out of the number of consistent gets both from Madrid and Segovia. This is promising, however when analysing the table blocks we find that the blocks have increased from 7 thousand to 11 thousand.

SQL> select table\_name, avg\_row\_len, num\_rows, blocks from user\_tables;

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

ASSIGN\_BUS

ASSIGN\_DRV

BIBUS 0 0 5

BIBUSERO 0 0 5

BIBUSERO\_STATE 0 0 5

BIBUSES

BIBUS\_STATE 0 0 5

BOOKS

BOOK\_ENTRIES 0 0 5

BOOK\_LOANS 0 0 0

BOOK\_RESERVATIONS 0 0 5

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

COPIES

DRIVERS

EDITIONS 243 240632 11754

ID\_ROUTES 6 150 5

MORE\_AUTHORS

MUNICIPALITIES

MUNICIPALITY 0 0 0

M\_LIBRARY 0 0 0

POSTS

BOOK\_EDITIONS 0 0 0

COURSES 0 0 5

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

ENROLLMENTS 0 0 5

LOANS

PROFESSORS 0 0 5

ROUTES

SERVICES

SROUTES 0 0 5

STOPS

STUDENTS 0 0 5

USERS

31 rows selected.

SQL>



### Hashkeys cluster

Finally, we’ll analyze the performance after using a hashkeys table to see if it is better or worse than when using the index, for that we first modify the creation of the cluster to use the hashkeys instead of the index.

* Hashkeys are used because they’ll help with the number of blocks problem that appeared in the last step.

drop cluster places;

create cluster places(pub\_place varchar2(50))

single table Hashkeys 251;

create table editions(..) cluster places(pub\_place);



After performing this change the analysed results are:

1. For madrid with cluster hashkeys

SQL> select \* from editions where pub\_place='Madrid';

82450 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 3284490970

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 188 | 40420 | 30 (0)| 00:00:01 |

|\* 1 | TABLE ACCESS HASH| EDITIONS | 188 | 40420 | 30 (0)| 00:00:01 |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - access("PUB\_PLACE"='Madrid')

Statistics

----------------------------------------------------------

2 recursive calls

0 db block gets

8090 consistent gets

1792 physical reads

0 redo size

17922933 bytes sent via SQL\*Net to client

60833 bytes received via SQL\*Net from client

5498 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

82450 rows processed



1. For Segovia with cluster hashkeys:

SQL> select \* from editions where pub\_place='Segovia'

2 ;

71 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 3284490970

------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 188 | 40420 | 30 (0)| 00:00:01 |

|\* 1 | TABLE ACCESS HASH| EDITIONS | 188 | 40420 | 30 (0)| 00:00:01 |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - access("PUB\_PLACE"='Segovia')

Statistics

----------------------------------------------------------

2 recursive calls

0 db block gets

14 consistent gets

1 physical reads

0 redo size

16917 bytes sent via SQL\*Net to client

419 bytes received via SQL\*Net from client

6 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

71 rows processed



We can see how both consistent get values have really improved, Madrid has gone from 12 thousand to 8 thousand while segovia is even better than with indexes at 14 vs 80, and not to mention compared to the original data of thousands. This is the one that will be pointed to at the evaluation.

We can also analyze the table data in order to se if blocks are down or up as with the last cluster optimization:

SQL> select table\_name, avg\_row\_len, num\_rows, blocks from user\_tables;

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

ASSIGN\_BUS

ASSIGN\_DRV

BIBUS 0 0 5

BIBUSERO 0 0 5

BIBUSERO\_STATE 0 0 5

BIBUSES

BIBUS\_STATE 0 0 5

BOOKS

BOOK\_ENTRIES 0 0 5

BOOK\_LOANS 0 0 0

BOOK\_RESERVATIONS 0 0 5

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

COPIES

DRIVERS

EDITIONS 223 241071 7692

ID\_ROUTES 6 150 5

LOANS

MORE\_AUTHORS

MUNICIPALITIES

MUNICIPALITY 0 0 0

M\_LIBRARY 0 0 0

POSTS

ROUTES

TABLE\_NAME AVG\_ROW\_LEN NUM\_ROWS BLOCKS

-------------------------------------------------------------------------------------------------------------------------------- ----------- ---------- ----------

SERVICES

SROUTES 0 0 5

STOPS

STUDENTS 0 0 5

BOOK\_EDITIONS 0 0 0

COURSES 0 0 5

ENROLLMENTS 0 0 5

PROFESSORS 0 0 5

USERS

31 rows selected.

SQL>



The number of blocks is kept at the 7 thousand mark, this means that this optimization is good.

## QUERY 2: Physical Design

In order to optimize the query we use three methods:

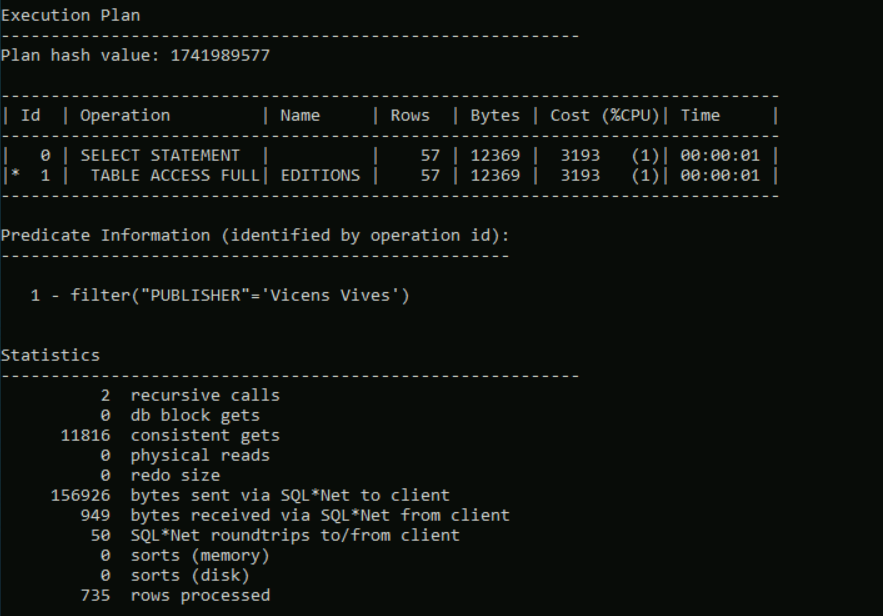
### 1.Index

To optimize the query 2, we created a secondary index named idx\_ed\_publisher on the publisher attribute of the editions table:

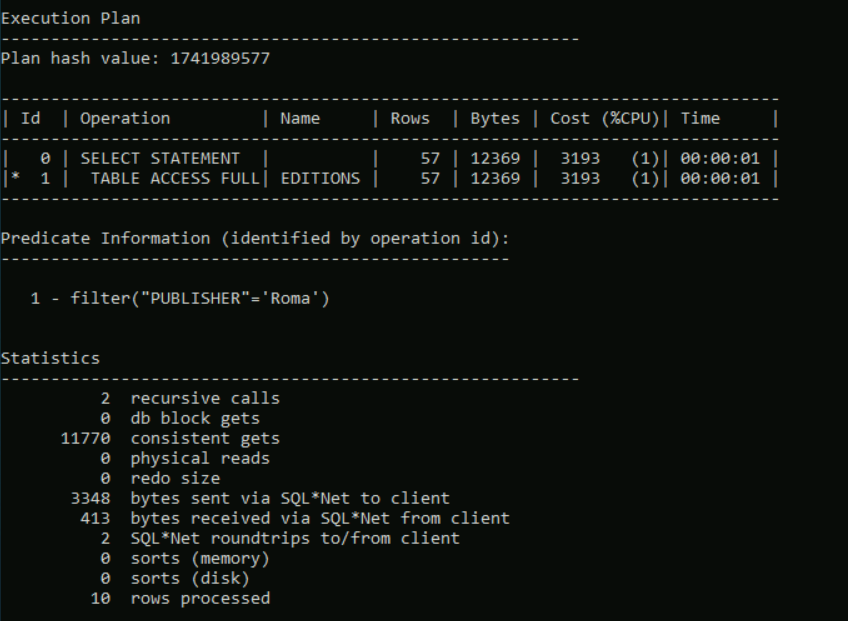
create index idx\_ed\_publisher on editions(publisher);

After creating the index, we compared the behavior of the system before and after the creation of the index by analyzing the consistent gets and physical reads.

* For 'Vicens Vives' (735 rows returned): 11816 consistent gets, 0 physical reads.



* For 'Roma' (10 rows returned): 11770 consistent gets, 0 physical reads



Although physical reads were eliminated thanks to Oracle’s caching mechanism, the logical reads (consistent gets) increased significantly when forcing the use of the index idx\_ed\_publisher. This behavior was consistent for both frequent and infrequent values of publisher. So, for this specific query and dataset the index does not provide better performance and may even add unnecessary overhead.

### 2. Index clustering

Because the index is not really making that much of a difference for we’ll implement a cluster on the publisher property of the editions table.

drop cluster publishers;

create cluster publishers (publisher varchar2(100));

CREATE TABLE Editions(

ISBN VARCHAR2(20),

TITLE VARCHAR2(200) NOT NULL,

AUTHOR VARCHAR2(100) NOT NULL,

LANGUAGE VARCHAR2(50) DEFAULT 'Spanish' NOT NULL,

ALT\_LANGUAGES VARCHAR2(50),

EDITION VARCHAR2(50),

PUBLISHER VARCHAR2(100),

EXTENSION VARCHAR2(50),

SERIES VARCHAR2(50),

COPYRIGHT VARCHAR2(20),

PUB\_PLACE VARCHAR2(50),

DIMENSIONS VARCHAR2(50),

PHY\_FEATURES VARCHAR2(200),

MATERIALS VARCHAR2(200),

NOTES VARCHAR2(500),

NATIONAL\_LIB\_ID VARCHAR2(20) NOT NULL,

URL VARCHAR2(200),

CONSTRAINT pk\_editions PRIMARY KEY (ISBN),

CONSTRAINT uk\_editions UNIQUE (NATIONAL\_LIB\_ID),

CONSTRAINT fk\_editions\_books FOREIGN KEY (TITLE, AUTHOR) REFERENCES books(TITLE, AUTHOR)

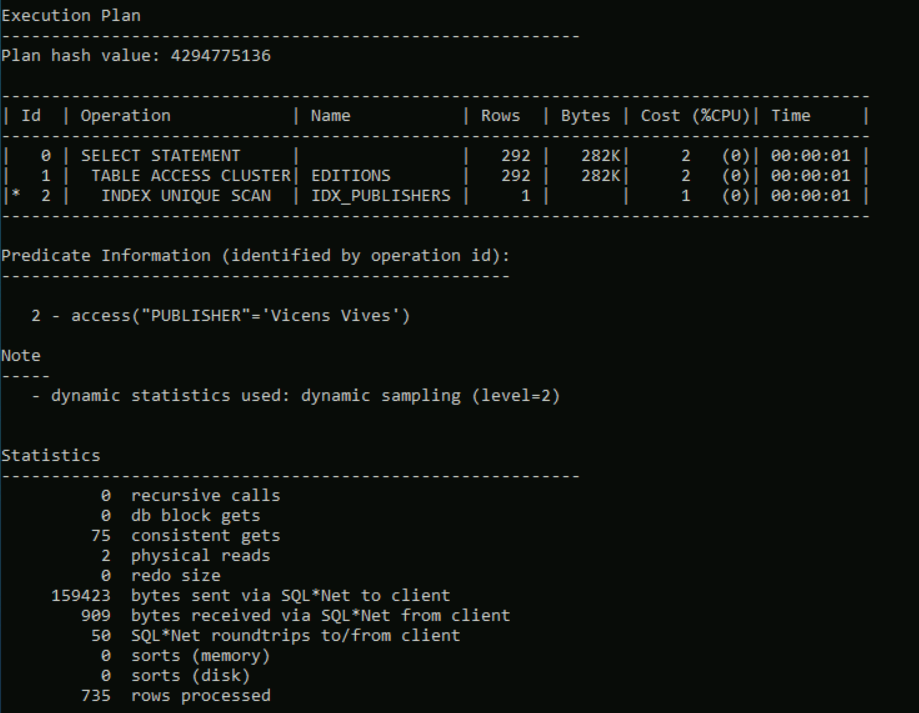
) CLUSTER publishers(publisher);

create index idx\_publishers on cluster publishers;

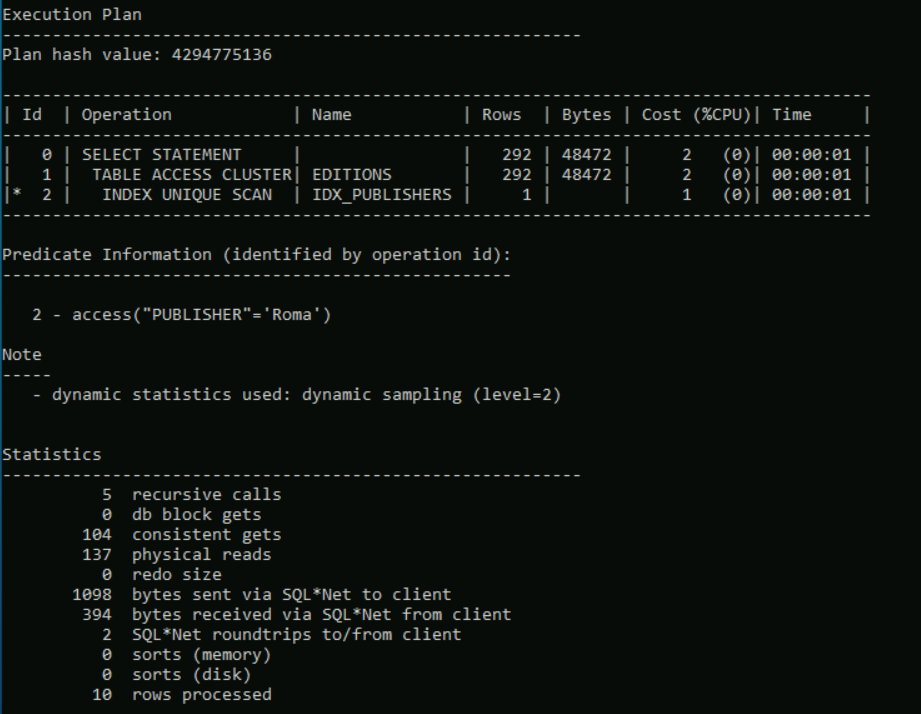


We compared the behavior of the system before and after the creation of the index clustering by analyzing the consistent gets and physical reads.

* For 'Vicens Vives' (735 rows returned):



* For 'Roma' (10 rows returned):



Compared to both the full scan and secondary index access, the cluster significantly reduced the number of consistent gets for both values. While 'Vicens Vives' had fewer logical and physical reads, 'Roma' still required more physical access, possibly due to disk layout or caching behavior.

### Hashkeys cluster

We evaluated the use of hash clustering on the publisher column. This technique stores rows based on a hash value calculated from the clustering key, and aims to provide constant-time access for equality-based queries. The cluster was created as follows:

CREATE CLUSTER publishers (

publisher VARCHAR2(100)

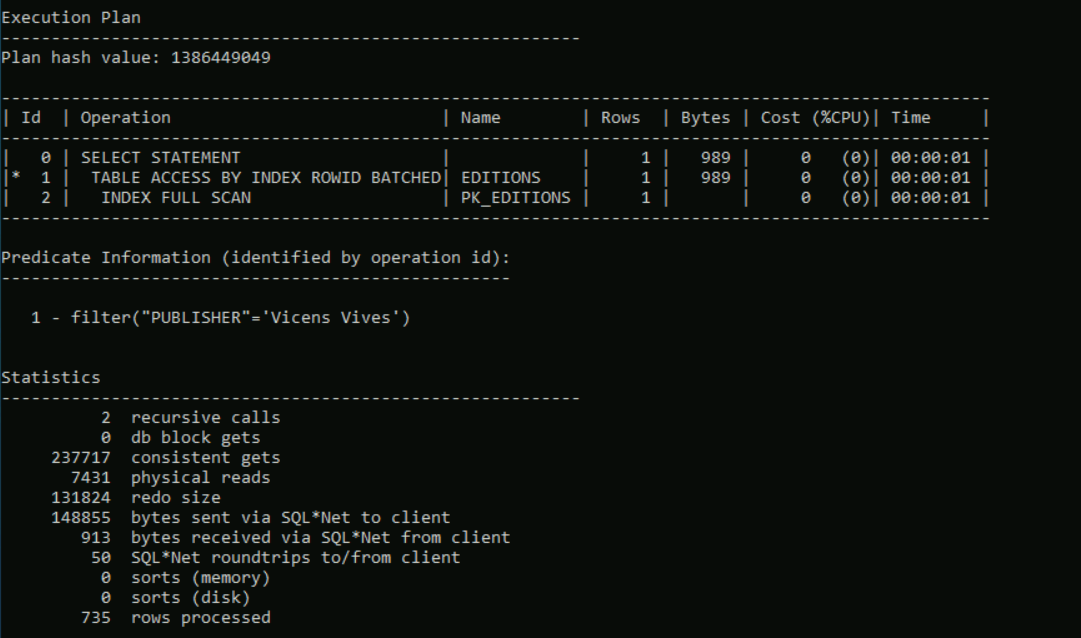
)

SINGLE TABLE HASHKEYS 251;

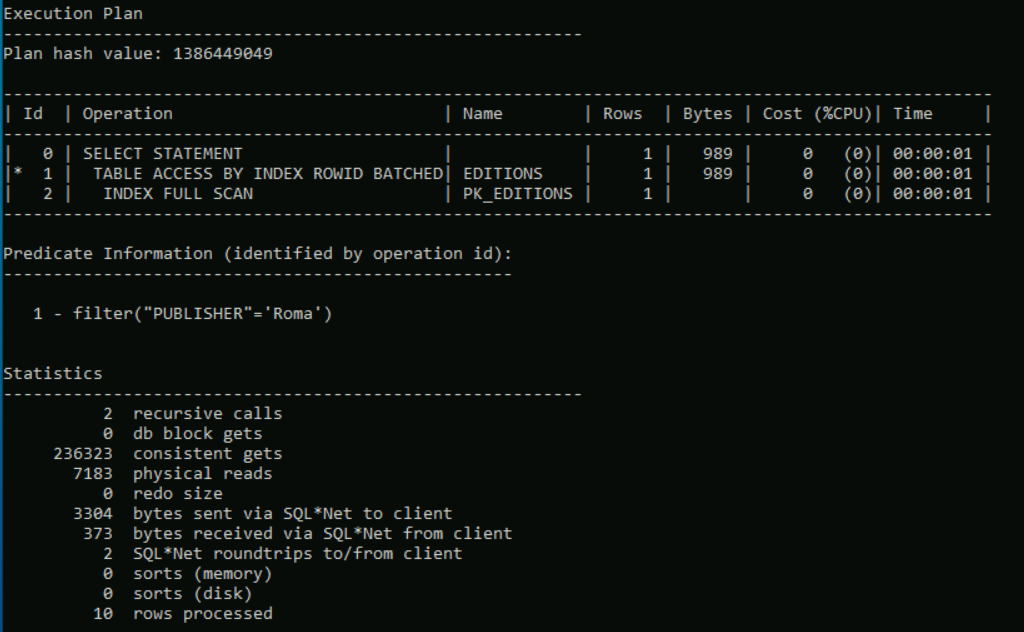
CREATE TABLE editions (...) CLUSTER publishers(publisher);



* For 'Vicens Vives' (735 rows returned):



* For 'Roma’(10 rows returned):



The hash cluster unexpectedly performed worse than the full scan, index, and index clustering options, since the number of consistent gets increased drastically

## QUERY 3: Physical Design

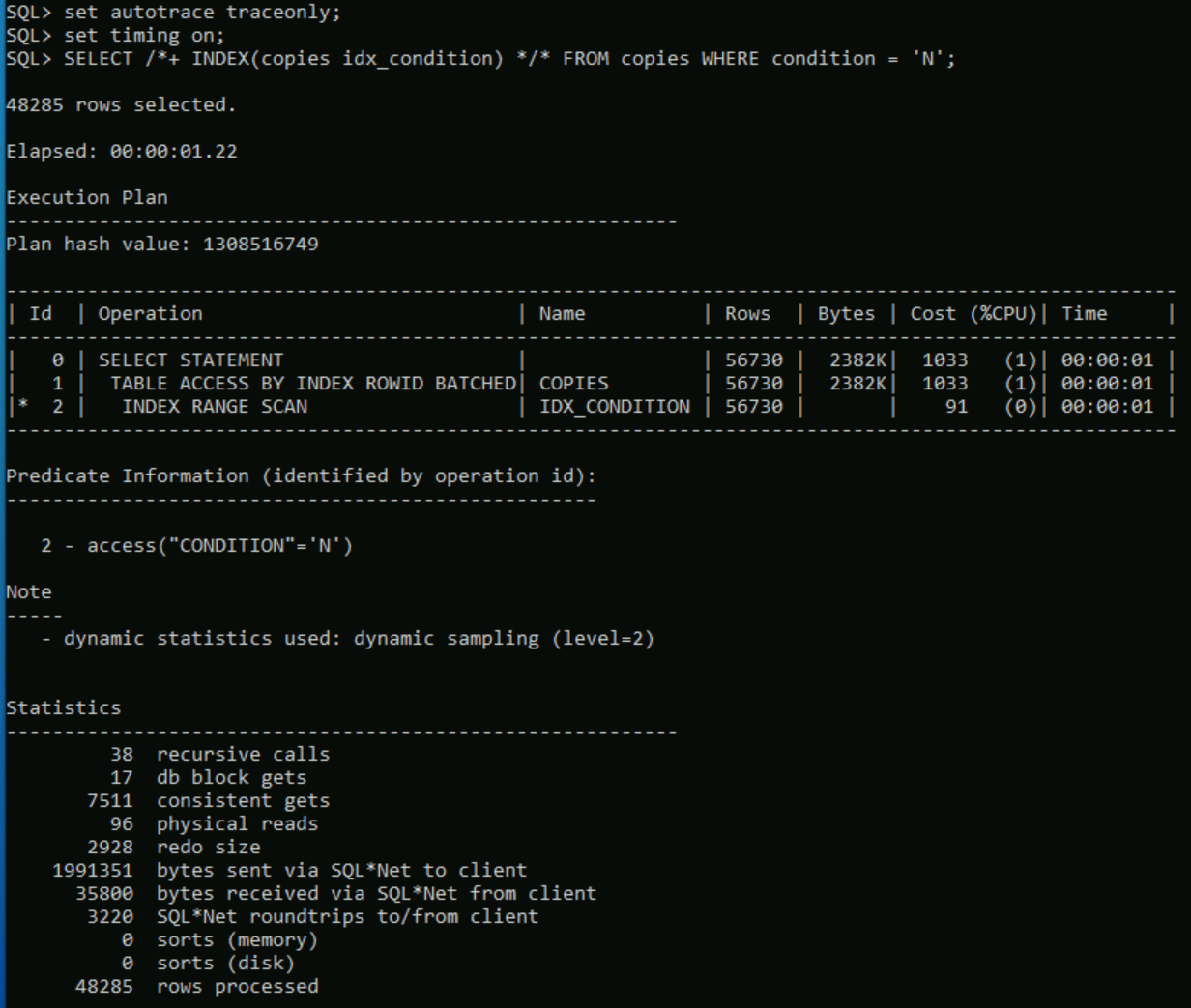
In order to optimize the query 3, we have created the following index:

k

After applying the index, we ran the query again using a hint to ensure the index is being used.

SELECT /\*+ INDEX(copies idx\_condition) \*/ \* FROM copies WHERE condition = 'N';





After applying the index, we observe that the execution plan is now access by index, the cost is higher than in the previous analysis, we can also see that the consistent gets has also significantly augmented, however on the other hand we can see that the physical reads has decreased passing from 979 in the full table scan, to 96 in the indexed query.

## QUERY 4: Physical Design

The fourth query is a fullscan, this means that there isn’t anything that can be done to improve it by means of changing how to look, as everything has to be looked at.

However we can actually do changes on how fast we can access the memory by multithreading, we assign four threads as a hint, by doing this we can see that the number of physical reads decreases from 23 thousand to 7 thousand.

SQL> set autotrace traceonly

SQL> SELECT /\*+ PARALLEL(editions, 4) \*/ \* FROM editions;

240632 rows selected.

Execution Plan

----------------------------------------------------------

Plan hash value: 1792949243

--------------------------------------------------------------------------------

------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time

| TQ |IN-OUT| PQ Distrib |

--------------------------------------------------------------------------------

------------------------------

| 0 | SELECT STATEMENT | | 203K| 191M| 570 (1)| 00:00:01

| | | |

| 1 | PX COORDINATOR | | | | |

| | | |

| 2 | PX SEND QC (RANDOM)| :TQ10000 | 203K| 191M| 570 (1)| 00:00:01

| Q1,00 | P->S | QC (RAND) |

| 3 | PX BLOCK ITERATOR | | 203K| 191M| 570 (1)| 00:00:01

| Q1,00 | PCWC | |

| 4 | TABLE ACCESS FULL| EDITIONS | 203K| 191M| 570 (1)| 00:00:01

| Q1,00 | PCWP | |

--------------------------------------------------------------------------------

------------------------------

Note

-----

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

- Degree of Parallelism is 4 because of table property

Statistics

----------------------------------------------------------

12 recursive calls

0 db block gets

7737 consistent gets

7552 physical reads

0 redo size

53740051 bytes sent via SQL\*Net to client

176839 bytes received via SQL\*Net from client

16044 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

240632 rows processed

SQL>



# Evaluation

*You have measured the performance of the initial physical design and stated it. After implementing your improved physical design and measured the new performance, compare both and analyze the results obtained (comment divergences with expected results). Add screenshots for backing your evaluation.*

## QUERY 1: Evaluation

For the first query many methods we’ve tried, all the screenshots and after analysis is explained on the physical design point of that query.

We can conclude that the final and better alternative is to use a cluster with a Hashkey table if we care about the number of blocks that the table takes over the consistent gets of values with many entries.

However, if it was preferred to maintain the number of consistent gets as low as possible without any concern about the number of blocks, then the clustering algorithm with an index would be the best.

For the final RUN\_TEST procedure the clustering algorithm with indexes will be used as the procedure measures the consistent gets not the number of blocks.

The code for indexes is the following:

drop table editions;

drop cluster places;

create cluster places(pub\_place varchar2(50));

CREATE TABLE Editions(

ISBN VARCHAR2(20),

TITLE VARCHAR2(200) NOT NULL,

AUTHOR VARCHAR2(100) NOT NULL,

LANGUAGE VARCHAR2(50) default('Spanish') NOT NULL,

ALT\_LANGUAGES VARCHAR2(50),

EDITION VARCHAR2(50),

PUBLISHER VARCHAR2(100),

EXTENSION VARCHAR2(50),

SERIES VARCHAR2(50),

COPYRIGHT VARCHAR2(20),

PUB\_PLACE VARCHAR2(50),

DIMENSIONS VARCHAR2(50),

PHY\_FEATURES VARCHAR2(200),

MATERIALS VARCHAR2(200),

NOTES VARCHAR2(500),

NATIONAL\_LIB\_ID VARCHAR2(20) NOT NULL,

URL VARCHAR2(200),

CONSTRAINT pk\_editions PRIMARY KEY(isbn),

CONSTRAINT uk\_editions UNIQUE (national\_lib\_id),

CONSTRAINT fk\_editions\_books FOREIGN KEY(title,author) REFERENCES books(title,author)

) cluster places(pub\_place);

create index idx\_places on cluster places;



## 

## Query 2: Evaluation

In the analysis of different physical design strategies to optimize the query SELECT \* FROM editions WHERE publisher = '...', we explored four alternatives: full table scan, secondary index, index clustering, and hash clustering.

* The full table scan, while not optimized, performed relatively well in some scenarios, particularly when a large number of rows matched the condition.
* The secondary index on the publisher column succeeded in eliminating physical reads but increased the number of logical accesses (consistent gets), which led to worse overall performance, even for selective queries.
* The most effective improvement was achieved with index clustering by publisher, which significantly reduced both logical and physical reads, especially for frequent values such as 'Vicens Vives'.
* Lastly, the hash clustering strategy, generated an extremely high number of logical reads in all cases and did not lead to any performance gains.

As a result, index clustering was identified as the most balanced and efficient solution for this query within our workload context. Here there is a comparative table:

| **Method** | **Value Tested** | **Consistent Gets** | **Physical Reads** |
| --- | --- | --- | --- |
| **Full Scan** | Vicens Vives | 7606 | 7555 |
| **Full Scan** | Roma | 7557 | 7553 |
| **Index** | Vicens Vives | 11816 | 0 |
| **Index** | Roma | 11770 | 0 |
| **Index Clustering** | Vicens Vives | 75 | 2 |
| **Index Clustering** | Roma | 104 | 137 |
| **Hash Clustering** | Vicens Vives | 237717 | 7431 |
| **Hash Clustering** | Roma | 236323 | 7183 |

The code for index clustering:

DROP TABLE editions CASCADE CONSTRAINTS;

DROP CLUSTER publishers;

CREATE CLUSTER publishers (publisher VARCHAR2(100));

CREATE TABLE editions(

ISBN VARCHAR2(20),

TITLE VARCHAR2(200) NOT NULL,

AUTHOR VARCHAR2(100) NOT NULL,

LANGUAGE VARCHAR2(50) DEFAULT 'Spanish' NOT NULL,

ALT\_LANGUAGES VARCHAR2(50),

EDITION VARCHAR2(50),

PUBLISHER VARCHAR2(100),

EXTENSION VARCHAR2(50),

SERIES VARCHAR2(50),

COPYRIGHT VARCHAR2(20),

PUB\_PLACE VARCHAR2(50),

DIMENSIONS VARCHAR2(50),

PHY\_FEATURES VARCHAR2(200),

MATERIALS VARCHAR2(200),

NOTES VARCHAR2(500),

NATIONAL\_LIB\_ID VARCHAR2(20) NOT NULL,

URL VARCHAR2(200),

CONSTRAINT pk\_editions PRIMARY KEY (ISBN),

CONSTRAINT uk\_editions UNIQUE (NATIONAL\_LIB\_ID),

CONSTRAINT fk\_editions\_books FOREIGN KEY (TITLE, AUTHOR) REFERENCES books(TITLE, AUTHOR)

) CLUSTER publishers(publisher);

CREATE INDEX idx\_publishers ON CLUSTER publishers;



## Query 3: Evaluation

We have compared the performance of the Query 3 before and after creation of the index on the condition columns of the copies table.

As we can see in the screenshots the query without the index returned 48,591 rows using a full table scan. The execution required 4,221 consistent gets, 979 physical reads, and was completed in 1.13s. Although the performance is acceptable, the high number of disc reads means that the query has room for improvement.

After creating the index (idx\_condition), we forced Oracle to use it using a hint. The query returned 48,285 rows, with 7,511 consistent gets, 96 physical reads, and completed in 1.22s.

The number of physical reads dropped which confirms that the index reduced disk I/O. The consistent gets increased as it is expected when accessing via the index.

Considering the pros and cons of the index in the query, we have concluded that the index does improve the performance by reducing the disk usage and should be in the final physical design.

## Query 4:

After applying the only method that occurred to us for a full scan query (multithreading) we obtained three times less gets in the query, optimizing it as much as we could.

We will therefore use the query with the hint for the final run tests procedure.

SELECT /\*+ PARALLEL(editions, 4) \*/ \* FROM editions;

# Concluding Remarks

Firstly, make conclusions on the work and the results obtained. Reflex on (defend or criticize) the achieved result (if you think it is good, explain why).

After stating your results, comment on your achievement through this labwork, and all assignments in general: required effort, knowledge gain, progress, etc. You can also propose improvements for further editions (focus, size of the problem, requested items, deadlines, supporting materials, etc.). Finally, you can add comments on the whole course (lacks in the syllabus, issues you would like to study more deeply, non-useful issues, etc.).