Lecturer:	FRANCISCO JAVIER CALLE GOMEZ		
Group:	89	Lab User	fsdb279
Student:	Eduardo Alarcón Navarro	NIA:	100472175
Student:	Salvador Ayala Iglesias	NIA:	100495832
Student:	Ines Guillén Peña	NIA:	100495752

1 Introduction

This project aims to optimize a database by reducing secondary storage accesses and shortening runtime. The process begins by analyzing the current database design, focusing on its structure and common workload patterns to identify resource-intensive processes. Using this information, a new design will be proposed to enhance performance and decrease unnecessary storage access. After implementing the new design, performance will be measured and compared against the original setup to evaluate the improvements.

First, we need to remove all existing database objects (tables, views, triggers, packages, procedures and functions) to clear the database. Next, we'll use the scripts from the "aula global" platform to create and populate the database. With the database reset to its initial state, we'll create the specified package and run the "RUN_TEST" procedure with a parameter set to '10'. This will measure baseline time consumption and consistent gets, serving as a reference for comparing the results after implementing the proposed modifications.

Here are the results from the initial test run:

```
SQL>   exec PKG_COSTES.RUN_TEST(10);
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 29/04/2024 17:40:48
TIME CONSUMPTION (run): 40,6 milliseconds.
CONSISTENT GETS (workload):7083 acc
CONSISTENT GETS (weighted average):708,3 acc

Procedimiento PL/SQL terminado correctamente.
```

2 Analysis

Firstly, we will start by reviewing Oracle SQL's default physical design to understand our test results and determine how to implement a more efficient setup. The key characteristics of Oracle's default design include:

- Serial non-consecutive organization: Records are organized within buckets.
- PCTFREE set to 10%
- By default, its bucket size (BS) is 8KB.
- PCTUSED at 60%.
- No auxiliary external structures.

Lowering PCTFREE increases data density within buckets but may lead to more frequent block splits, slowing down insertion and update processes. Regarding block size, increasing it allows more rows to be read with fewer block accesses, potentially enhancing read performance. However, larger block sizes can negatively impact insertion processes due to writing larger blocks and more bytes.

Secondly, we will analyze the workload. This workload consists of 5 processes with different frequencies.

- **Process 1: First query (10%)**

Table involved: Posts

```
SQL> desc posts;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 POSTDATE                                  NOT NULL DATE
 BARCODE                                            CHAR(15)
 PRODUCT                                   NOT NULL VARCHAR2(50)
 SCORE                                     NOT NULL NUMBER(1)
 TITLE                                              VARCHAR2(50)
 TEXT                                      NOT NULL VARCHAR2(2000)
 LIKES                                     NOT NULL NUMBER(9)
 ENDORSED                                           DATE
```

Execution Plan:

This query probably triggers a full table scan because there's no index on the *'barcode'* field, resulting in high consistent gets and slower query performance. Its weakness is that full table scans are inefficient for queries that run frequently.

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3606309814

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     8 |  9072 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |     8 |  9072 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("BARCODE"='OII044550419282')

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


Estadisticas
----------------------------------------------------------
         13  recursive calls
          0  db block gets
        508  consistent gets
          0  physical reads
          0  redo size
       9692  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

After setting the autotrace traceonly to see what is happening during the execution, the results indicate that a significant amount of resources are being used by the Posts table, which is subjected to "FULL ACCESS" operation. To optimize performance, we should consider implementing indexes to reduce the resource usage associated with this full table scan.

- **Process 2: Second query (10%)**

Table involved: Posts

```
SQL> desc posts;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 POSTDATE                                  NOT NULL DATE
 BARCODE                                            CHAR(15)
 PRODUCT                                   NOT NULL VARCHAR2(50)
 SCORE                                     NOT NULL NUMBER(1)
 TITLE                                              VARCHAR2(50)
 TEXT                                      NOT NULL VARCHAR2(2000)
 LIKES                                     NOT NULL NUMBER(9)
 ENDORSED                                           DATE
```

Execution Plan:

An execution plan without an index on *'product'* can lead to a full table scan, causing high consistent gets and degraded performance. The lack of an index forces the database to scan every row, which is highly inefficient for queries, resulting in longer processing times. This weakness can severely impact query performance and overall database efficiency.

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3606309814

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    63 | 71442 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |    63 | 71442 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("PRODUCT"='Compromiso')

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


Estadisticas
----------------------------------------------------------
         97  recursive calls
          2  db block gets
        619  consistent gets
          6  physical reads
          0  redo size
      54499  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

After setting the autotrace traceonly to see what is happening during the execution, the results indicate that a significant amount of resources are being used by the Posts table, which is subjected to "FULL ACCESS" operation. To optimize performance, we should consider implementing indexes to reduce the resource usage associated with this full table scan.

- **Process 3: Third query (10%)**

Table involved: Posts

```
SQL> desc posts;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 POSTDATE                                  NOT NULL DATE
 BARCODE                                            CHAR(15)
 PRODUCT                                   NOT NULL VARCHAR2(50)
 SCORE                                     NOT NULL NUMBER(1)
 TITLE                                              VARCHAR2(50)
 TEXT                                      NOT NULL VARCHAR2(2000)
 LIKES                                     NOT NULL NUMBER(9)
 ENDORSED                                           DATE
```

Execution Plan:

The execution plan for this query involves a range-based condition. Without an index on the *'score'* field, the database may resort to a full table scan, which is inefficient. This lack of an index can lead to slower query performance and higher consistent gets, impacting overall database efficiency.

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3606309814

-------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  1189 |  1316K|   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |  1189 |  1316K|   136   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("SCORE">=4)

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


Estadisticas
----------------------------------------------------------
         20  recursive calls
          0  db block gets
        578  consistent gets
          0  physical reads
          0  redo size
    1098428  bytes sent via SQL*Net to client
       1218  bytes received via SQL*Net from client
         80  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1173  rows processed
```

After setting the autotrace traceonly to see what is happening during the execution, the results indicate that a significant amount of resources are being used by the Posts table, which is subjected to "FULL ACCESS" operation. To optimize performance, we should consider implementing indexes to reduce the resource usage associated with this full table scan.

Process 4: Fourth query (20%)

Table involved: Posts

```
SQL> desc posts;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 POSTDATE                                  NOT NULL DATE
 BARCODE                                            CHAR(15)
 PRODUCT                                   NOT NULL VARCHAR2(50)
 SCORE                                     NOT NULL NUMBER(1)
 TITLE                                              VARCHAR2(50)
 TEXT                                      NOT NULL VARCHAR2(2000)
 LIKES                                     NOT NULL NUMBER(9)
 ENDORSED                                           DATE
```

Execution Plan:

The execution plan for this query indicates a full table scan because it retrieves all records from the table. This approach can become a significant performance bottleneck, especially as the table size grows, leading to longer query times and increased resource usage.

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3606309814

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  3582 |  3966K|   136   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| POSTS |  3582 |  3966K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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


Estadisticas
----------------------------------------------------------
          7  recursive calls
          0  db block gets
        711  consistent gets
          0  physical reads
          0  redo size
    3222320  bytes sent via SQL*Net to client
       2853  bytes received via SQL*Net from client
        230  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       3429  rows processed
```

After setting the autotrace traceonly to see what is happening during the execution, the results indicate that a significant amount of resources are being used by the Posts table, which is subjected to "FULL ACCESS" operation. To optimize performance, we should consider implementing a hint.

- **Process 5: Fifth query (50%)**

Tables involved: CLIENT_LINES, ORDERS_CLIENTS

```
SQL> desc client_lines;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 ORDERDATE                                 NOT NULL DATE
 USERNAME                                  NOT NULL VARCHAR2(30)
 TOWN                                      NOT NULL VARCHAR2(45)
 COUNTRY                                   NOT NULL VARCHAR2(45)
 BARCODE                                   NOT NULL CHAR(15)
 PRICE                                     NOT NULL NUMBER(12,2)
 QUANTITY                                  NOT NULL VARCHAR2(2)
 PAY_TYPE                                  NOT NULL VARCHAR2(15)
 PAY_DATETIME                                       DATE
 CARDNUM                                            NUMBER(20)

SQL> desc orders_clients;
 Nombre                                    ?Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 ORDERDATE                                 NOT NULL DATE
 USERNAME                                  NOT NULL VARCHAR2(30)
 TOWN                                      NOT NULL VARCHAR2(45)
 COUNTRY                                   NOT NULL VARCHAR2(45)
 DLIV_DATETIME                                      DATE
 BILL_TOWN                                 NOT NULL VARCHAR2(45)
 BILL_COUNTRY                              NOT NULL VARCHAR2(45)
 DISCOUNT                                           NUMBER(2)
```

Execution Plan:

The execution plan for this query includes a join operation, but there's no indication of appropriate indexing or clustering. Without indexes or clustering, the join process can be highly inefficient, requiring more resources and time to complete. This lack of optimization can significantly affect query performance, leading to longer execution times and increased database workload.

```
Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1654569925

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |    12 |  2544 |   223   (1)| 00:00:01 |
|   1 |  NESTED LOOPS                |                 |    12 |  2544 |   223   (1)| 00:00:01 |
|   2 |   NESTED LOOPS               |                 |    24 |  2544 |   223   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | CLIENT_LINES    |    24 |  2160 |   205   (1)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_CLIENTORDERS |     1 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| ORDERS_CLIENTS  |     1 |   122 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   3 - filter("CLIENT_LINES"."USERNAME"='chamorro')
   4 - access("ORDERS_CLIENTS"."ORDERDATE"="CLIENT_LINES"."ORDERDATE" AND
              "ORDERS_CLIENTS"."USERNAME"='chamorro' AND "ORDERS_CLIENTS"."TOWN"="CLIENT_LINES"."TOWN"
               AND "ORDERS_CLIENTS"."COUNTRY"="CLIENT_LINES"."COUNTRY")

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


Estadisticas
----------------------------------------------------------
        129  recursive calls
          0  db block gets
       1166  consistent gets
          5  physical reads
          0  redo size
       2171  bytes sent via SQL*Net to client
        548  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

It has been set the autotrace traceonly to see what is happening during the execution, showing the performance in the execution plan that can be seen above. The autotrace results indicate that a significant amount of resources are being used by the Client_Lines table, which is subjected to "FULL ACCESS" operation. To optimize performance, we should consider implementing indexes and clusters to reduce the resource usage associated with this full table scan.

Possible optimizations

Element	Expected Benefits	Expected Drawbacks
Index on 'barcode' in 'Posts'	We'll improve queries' performance based on the barcode. Reduced consistent gets and quicker retrieval.	Potential overhead for updates and insertions.
Index on 'product' in 'Posts'	Queries' performance based on product will be faster. Reduced consistent gets.	Increased index maintenance during insertions/updates.
Index on 'score' in 'Posts'	Efficient range-based queries. Reduced consistent gets and quicker performance.	Additional overhead for updates/insertions.
Index on 'Client_Lines' by username	Faster joins with orders_clients. Improved retrieval for frequent join-based queries.	Increased complexity in maintaining composite indexes.
Multi-table cluster for orders_clients and client_lines	We'll improve the join performance due to physical clustering of related data. Reduced consistent gets during join operations.	Potential impact on insertion performance. Increased storage requirements.

3 Physical Design

```
Unset
DROP ...
DROP INDEX idx_posts_barcode;
DROP INDEX idx_posts_product;
alter table Client_Lines drop clustering;
DROP INDEX idx_client_lines_01;

-- - VALIDATION TABLES - - - - - - - -
CREATE TABLE ...

-- - TABLES CREATION - - - - - - - - -
CREATE TABLE ...
select table_name from user_tables;

-- Indexes
CREATE INDEX idx_posts_barcode ON posts (barcode);
CREATE INDEX idx_posts_product ON posts (product);

SELECT /*+ FULL(posts) PARALLEL(posts, 2) */ * from posts;
```

```
-- CREATE CLUSTER
ALTER TABLE Client_Lines add clustering by linear order (username);
ALTER TABLE Client_Lines move online;
-- CREATE INDEX FOR SUCH CLUSTER
CREATE INDEX idx_client_lines_01 ON client_lines (username);
```

4 Evaluation

Process 1: First query (10%)

As we decided before, we created an index for the barcode in the Posts table, we expected the consistent gets and the time to get reduced. That was the conclusion:

```
Unset
                CREATE INDEX idx_posts_barcode ON posts (barcode);
```

Execution plan without the index:

```
Estadisticas
-----------------------------------------------------------
         13  recursive calls
          0  db block gets
        508  consistent gets
          0  physical reads
          0  redo size
       9692  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

We'll use a hint to force ORACLE to use the index when doing the SELECT statement:

```
Unset
   SELECT /*+ INDEX(Posts idx_posts_barcode) */ * FROM Posts WHERE barCode =
                                'OII044550419282';
```

Execution plan with the index:

```
Estadisticas
----------------------------------------------------------
         16  recursive calls
          0  db block gets
         20  consistent gets
          0  physical reads
          0  redo size
       9712  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

At first, it wasn't clear which option was more efficient. However, after analyzing the data, we found that the index insertion resulted in fewer consistent gets. Although there was an increase in recursive calls and bytes sent via SQL*Net to client, the overall operations seemed less costly after the index insertion.

Then, we run the process run test and obtain this result:

```
SQL> exec PKG_COSTES.RUN_TEST(10);
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 29/04/2024 17:31:36
TIME CONSUMPTION (run): 35,9 milliseconds.
CONSISTENT GETS (workload):6594 acc
CONSISTENT GETS (weighted average):659,4 acc

Procedimiento PL/SQL terminado correctamente.
```

The time consumption and the consistent gets (from 7083 acc to 6594 acc) have been reduced.

Process 2: Second query (10%)

As we decided before, we created an index for the product in the Posts table, we expected the consistent gets to get reduced. That was the conclusion:

```
Unset
CREATE INDEX idx_posts_product ON posts (product);
```

Execution plan without the index:

```
Estadisticas
----------------------------------------------------------
         97  recursive calls
          2  db block gets
        619  consistent gets
          6  physical reads
          0  redo size
      54499  bytes sent via SQL*Net to client
        405  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

We'll use a hint to force ORACLE to use the index when doing the SELECT statement:

```
Unset
   SELECT /*+ INDEX(Posts idx_posts_product) */ * FROM Posts WHERE product =
                                    'Compromiso';
```

Execution plan with the index:

```
Estadisticas
----------------------------------------------------------
         19  recursive calls
          0  db block gets
         72  consistent gets
          1  physical reads
          0  redo size
      54606  bytes sent via SQL*Net to client
        409  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

It's evident that using this index makes the query more efficient. Although there was an increase in bytes sent via SQL*Net to client and bytes received via SQL*Net from client, the overall operations seemed less costly after the index insertion.

Then, we executed the 'RUN_TEST' and got the following result:

```
SQL> exec PKG_COSTES.RUN_TEST(10);
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 29/04/2024 17:49:36
TIME CONSUMPTION (run): 34,4 milliseconds.
CONSISTENT GETS (workload):6147 acc
CONSISTENT GETS (weighted average):614,7 acc

Procedimiento PL/SQL terminado correctamente.
```

The time consumption and the consistent gets (from 6594 acc to 6147 acc) have been reduced.

Process 3: Third query (10%)

As we decided before, we created an index for the score in the Posts table, we expected the consistent gets and the time execution to get reduced. That was the conclusion:

```
Unset
CREATE INDEX idx_posts_score ON posts (score);
```

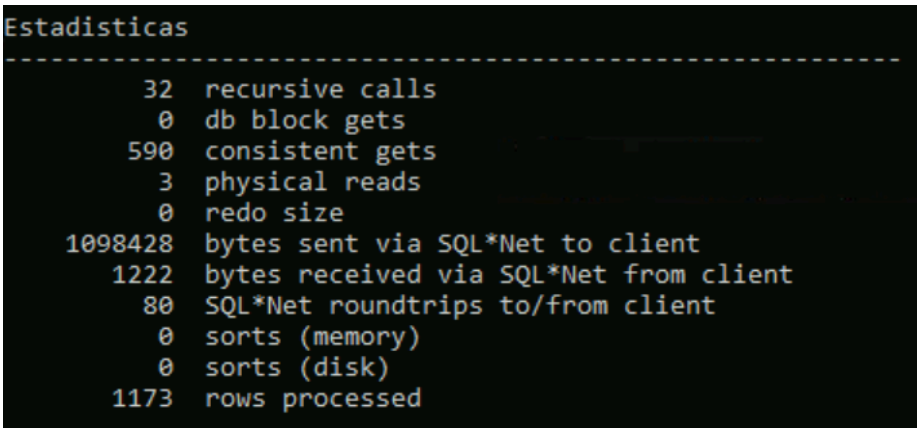
Execution plan without the index:

```
Estadisticas
----------------------------------------------------------
         20  recursive calls
          0  db block gets
        578  consistent gets
          0  physical reads
          0  redo size
    1098428  bytes sent via SQL*Net to client
       1218  bytes received via SQL*Net from client
         80  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1173  rows processed
```

We'll use a hint to force ORACLE to use the index when doing the SELECT statement:

```
Unset
  SELECT /*+ INDEX(posts idx_posts_score ) */ * FROM posts WHERE score >= 4;
```

Execution plan with the index:



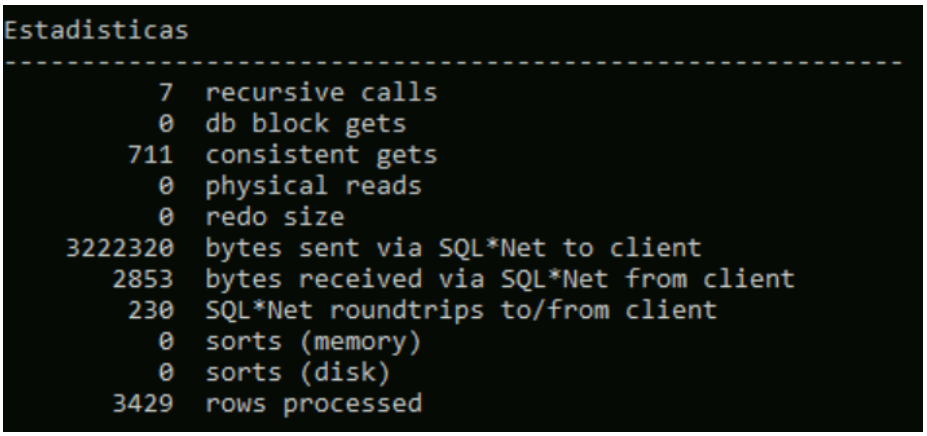
```
Estadisticas
----------------------------------------------------------
         32  recursive calls
          0  db block gets
        590  consistent gets
          3  physical reads
          0  redo size
    1098428  bytes sent via SQL*Net to client
       1222  bytes received via SQL*Net from client
         80  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1173  rows processed
```

After executing the SELECT statement, as we can see, the values of each variable after execution have either stayed the same or increased, indicating that **this optimization method isn't effective**. Given that the number of rows with a score of 4 or higher is quite large, we'll still do a Range Scan throughout the index and this will make it less effective.

This indicates that there's no index that can optimize this query. The problem is the data distribution (too many rows meet the query's condition), reducing the effectiveness of any index.

Process 4: Fourth query (20%)

Execution plan without any optimization:



```
Estadisticas
----------------------------------------------------------
          7  recursive calls
          0  db block gets
        711  consistent gets
          0  physical reads
          0  redo size
    3222320  bytes sent via SQL*Net to client
       2853  bytes received via SQL*Net from client
        230  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       3429  rows processed
```

An index won't be much help for this query because it selects every row; therefore, to improve efficiency we'll use a hint:

```
Unset
SELECT /*+ FULL(posts) PARALLEL(posts, 2) */ * from posts;
```

Execution plan with optimization:

```
Estadisticas
-----------------------------------------------------------
          6  recursive calls
          0  db block gets
        601  consistent gets
          0  physical reads
          0  redo size
    3215804  bytes sent via SQL*Net to client
       2895  bytes received via SQL*Net from client
        230  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       3429  rows processed
```

As we can observe, the number of consistent gets has been reduced. Although there was an increase in bytes received via SQL*Net from client, the overall operations seemed less costly after the index insertion.

Then, we executed the 'RUN_TEST' and got the following result:

```
SQL>  exec pkg_costes.run_test(10);
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 30/04/2024 13:40:11
TIME CONSUMPTION (run): 31,4 milliseconds.
CONSISTENT GETS (workload):5907 acc
CONSISTENT GETS (weighted average):590,7 acc

Procedimiento PL/SQL terminado correctamente.
```

The time consumption and the consistent gets (from 6147 acc to 5907 acc) have been reduced.

Process 5: Fifth query (50%)

Without any index and cluster:

```
Estadisticas
----------------------------------------------------------
        129  recursive calls
          0  db block gets
       1166  consistent gets
          5  physical reads
          0  redo size
       2171  bytes sent via SQL*Net to client
        548  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

For this query, first we will use a cluster to optimize the Join between the two tables (orders_clients and client_lines) and the corresponding index:

```
Unset
      ALTER TABLE Client_Lines add clustering by linear order (username);
                    ALTER TABLE Client_Lines move online;

          CREATE INDEX idx_client_lines_01 ON client_lines (username);
```

The output obtained after creating such cluster is the following:

```
Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        159  consistent gets
          0  physical reads
          0  redo size
       2171  bytes sent via SQL*Net to client
        594  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

As we can see, the consistent gets have been highly reduced, which indicates that the cluster and index have optimized the code properly.

Then, we executed the 'RUN_TEST' and got the following result:

```
SQL>  exec pkg_costes.run_test(10);
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
RESULTS AT 30/04/2024 14:06:49
TIME CONSUMPTION (run): 20,3 milliseconds.
CONSISTENT GETS (workload):2357 acc
CONSISTENT GETS (weighted average):235,7 acc

Procedimiento PL/SQL terminado correctamente.
```

The time consumption and the consistent gets (from 5907 acc to 2357 acc) have been reduced.

5 Concluding Remarks

In summary, we have developed an improved physical design based on a given workload. We began by analyzing the default setup and identified areas for improvement. Based on this analysis, we proposed a new design and selected specific structures to implement. After testing the changes, the results aligned with our expectations. We successfully improved the efficiency of workload processes by reducing the number of block accesses (7083 → 2357).

This lab work has helped us understand how data is organized within file structures, along with methods for improving the efficiency of various processes.