Alumno/a:	**Jorge Mejias Donoso**	**NIA:**	**100495807**
Alumno/a:	**Javier Moyano San Bruno**	**NIA:**	**100495884**

1 Introducción

Este proyecto tiene como objetivo analizar el diseño físico actual de una base de datos y desarrollar una propuesta de mejoras que puedan reducir el número de accesos a cubos y mejorar el rendimiento general. Se evaluará el estado inicial de la base de datos, incluyendo su diseño físico y la carga de trabajo prototípica. Se analizarán los procesos frecuentes de la carga de trabajo, identificando la tipología de los procesos, su frecuencia y costo. El objetivo es proponer un diseño físico completo que pueda optimizar el rendimiento de la base de datos y reducir el número de accesos a cubos.

El diseño propuesto será implementado, y su rendimiento será medido frente al diseño original. Los resultados serán comparados y analizados, y se sugerirán mejoras potenciales. Los pasos involucrados en este proyecto incluyen evaluar el estado actual de la base de datos, proponer un diseño físico completo, implementarlo, medir su rendimiento, y analizar y documentar los resultados.

2 Análisis

En primer lugar, debemos analizar el diseño físico predeterminado ofrecido por Oracle SQL. De esta manera, podremos entender los resultados obtenidos de las pruebas e implementar un diseño físico más eficiente.

Las características más importantes del diseño físico predeterminado en Oracle son las siguientes:

- Organización serial no consecutiva: los registros están organizados dentro de cubos.
- PCTFREE a 10%: para cada cubo hay un espacio reservado del 10% del tamaño de los cubos para futuras actualizaciones de los registros de los cubos.
- BS a 8KB: el tamaño de cada cubo es de 8KB.
- PCTUSED a 60%.
- No hay estructuras auxiliares externas.

Un menor PCTFREE aumenta la cantidad de datos que se pueden almacenar en un cubo (densidad), pero también puede provocar divisiones de bloques más frecuentes y un rendimiento más lento para los procesos de inserción y actualización.

En cuanto al tamaño del bloque, aumentarlo implica que se pueden leer más filas de la tabla con menos acceso al bloque, lo que puede mejorar el rendimiento de lectura de la tabla. Por otro lado, podría afectar negativamente los procesos de inserción, ya que tendrán que escribir bloques más grandes y, como consecuencia, más bytes.

Las pruebas de run_test van a evaluar la ejecución de cinco consultas diferentes. Estas consultas se caracterizan por tener frecuencias relativas diferentes entre sí, lo que significa que algunas se ejecutan con más frecuencia que otras. En este caso específico, se menciona que las tres primeras consultas son equiprobables, es decir, tienen la misma probabilidad de ser ejecutadas. La cuarta consulta es el doble de frecuente que las tres primeras, y la última consulta es la más frecuente, representando la mitad de todas las ejecuciones.

Esto se expresa en un conjunto de frecuencias relativas: {0.1, 0.1, 0.1, 0.2, 0.5}, donde cada valor representa la proporción de ejecuciones de cada consulta respecto al total. Por ejemplo, la primera consulta tiene una probabilidad del 10% de ser ejecutada, al igual que las otras dos consultas equiprobables. La cuarta consulta tiene una probabilidad del 20%, ya que es el doble de frecuente que las tres primeras, y la última consulta tiene una probabilidad del 50%, siendo la más frecuente de todas.

Es importante tener en cuenta que las tres tablas involucradas en estas consultas son volátiles, lo que significa que experimentan cambios frecuentes en sus datos. Además, se destaca que las tablas relacionadas con las compras, como orders_clients y client_lines, son extremadamente volátiles. Por esta razón, se debe garantizar que las pruebas de rendimiento no afecten de ninguna manera el rendimiento de los procesos de actualización, especialmente las inserciones, en estas tablas.

Una vez tenemos en cuenta todo esto y ya con la base de datos limpia (solo con las tablas del ejercicio dos y el paquete de las statistics), podemos realizar la ejecución de el run_test y obtenemos esto:

```
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 28/04/2024 13:24:09
TIME CONSUMPTION (run): 35,9 milliseconds.
CONSISTENT GETS (workload):7076 acc
CONSISTENT GETS (weighted average):707,6 acc

Procedimiento PL/SQL terminado correctamente.
```

- **1ª Consulta**

Consulta:

select * from posts where barcode = ' OII04455O419282 ' ;

Esta consulta busca recuperar todas las columnas de la tabla “posts” donde el valor de la columna "barcode" sea igual al “barcode” por el que se ha filtrado

Test de ejecución:

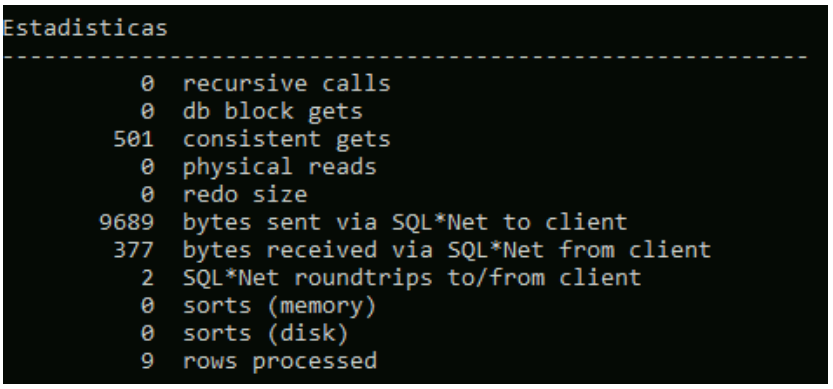
```
SQL> select * from posts where barcode='OII044550419282';

9 filas seleccionadas.


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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     3 |  2742 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |     3 |  2742 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

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


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        501  consistent gets
          0  physical reads
          0  redo size
       9689  bytes sent via SQL*Net to client
        377  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

Podemos ver que se está ejecutando un full scan (TABLE ACCESS FULL) para esta consulta. Esto es por que estamos filtrando por una clave no privilegiada, entonces la base de datos tiene que buscar por toda la tabla para ver que “barcode” coinciden con el buscado

Posibles Optimizaciones:

Índice en la clave “barcode”: Para mejorar la carga de trabajo una posible solución sería implementar un índice en el atributo “barcode”, por que siempre que se ejecuta esta consulta se hace mediante un escaneo total (Full Scan).

Estadísticas:



Como podemos observar en las estadísticas el número de ‘ Consistent Gets ’ es muy alto debido al FULL SCAN que se genera por “barcode” no ser una clave privilegiada y además la consulta tiene un alto coste en la CPU. Por tanto creemos que crear un índice para este atributo reduciría considerablemente los CONSISTENT GETS y el coste en la CPU ya que solo tendría que acceder a aquellos elementos seleccionados, haciendo la consulta mucho más eficiente.

- **2ª consulta**

Consulta:

select * from posts where product = ' Compromiso ' ;

Esta consulta selecciona todas las columnas de la tabla “posts” donde el valor de la columna “product” sea exactamente igual a 'Compromiso'.

Test de ejecución:

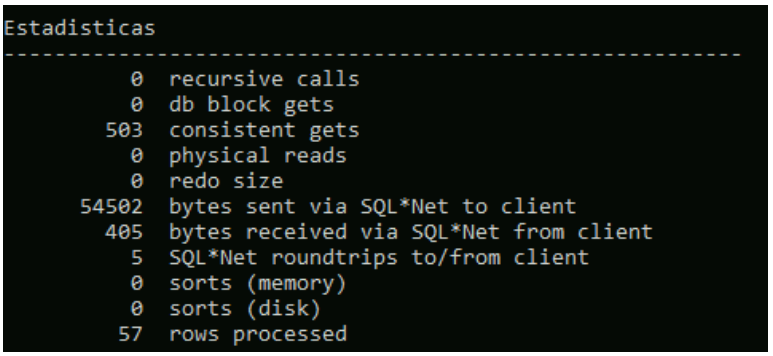
```
SQL> select * from posts where product='Compromiso';

57 filas seleccionadas.


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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    21 | 19194 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |    21 | 19194 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

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


Estadisticas
----------------------------------------------------------
         0  recursive calls
         0  db block gets
       503  consistent gets
         0  physical reads
         0  redo size
     54502  bytes sent via SQL*Net to client
       405  bytes received via SQL*Net from client
         5  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        57  rows processed
```

Podemos ver que se está ejecutando un full scan (TABLE ACCESS FULL) para esta consulta. Esto es por que estamos filtrando por una clave no privilegiada, entonces la base de datos tiene que buscar por toda la tabla para ver que “product” coinciden con el buscado

Posibles Optimizaciones:

Índice en la clave “product”: Para mejorar la carga de trabajo una posible solución sería implementar un índice en el atributo “product”, por que siempre que se ejecuta esta consulta se hace mediante un escaneo total (Full Scan) por lo tanto si creamos este índice no realizaremos un escaneo total si no que sólo accedería cuando coincidiese con dicho atributo.

Estadísticas:



Como podemos observar en las estadísticas el número de ‘ Consistent Gets ’ es muy alto debido al FULL SCAN que se genera por ‘ Product ‘ no ser una clave privilegiada y además la consulta tiene un alto coste en la CPU. Por tanto creemos que crear un índice para este atributo reduciría considerablemente los CONSISTENT GETS y el coste en la CPU ya que solo tendría que acceder a aquellos elementos seleccionados, haciendo la consulta mucho más eficiente.

- **3ª Consulta**

Consulta:

select * from posts where score >= 4 ;

Esta consulta selecciona todas las columnas de la tabla “posts” donde el valor de la columna “score” sea mayor o igual a 4. Esto implica que se están recuperando todas las entradas en la tabla donde la puntuación asociada cumple con este criterio de al menos 4 o más.

Test de ejecución:

```
SQL> select * from posts where score>=4;

1173 filas seleccionadas.


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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  1543 |  1377K|   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |  1543 |  1377K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("SCORE">=4)


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        572  consistent gets
          0  physical reads
          0  redo size
    1098407  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
```

Podemos ver que se está ejecutando un full scan (TABLE ACCESS FULL) para esta consulta. Esto es por que estamos filtrando por una clave no privilegiada, entonces la base de datos tiene que buscar por toda la tabla para ver qué “score” coincide con el buscado.

Posibles Optimizaciones:

Índice en la clave “score”: Para mejorar la carga de trabajo una posible solución sería implementar un índice en el atributo “score”, por que siempre que se ejecuta esta consulta se hace mediante un escaneo total (Full Scan) por lo tanto si creamos este índice no realizaremos un escaneo total si no que sólo accedería cuando coincidiese con dicho atributo.

Estadísticas:

```
Estadisticas
-----------------------------------------------------------
          0  recursive calls
          0  db block gets
        572  consistent gets
          0  physical reads
          0  redo size
    1098407  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
```

Como podemos observar en las estadísticas el número de ‘ Consistent Gets ’ es muy alto debido al FULL SCAN que se genera por “score” no ser una clave privilegiada y además la consulta tiene un alto coste en la CPU. Por tanto creemos que crear un índice para este atributo reduciría considerablemente los CONSISTENT GETS y el coste en la CPU ya que solo tendría que acceder a aquellos elementos seleccionados, haciendo la consulta mucho más eficiente.

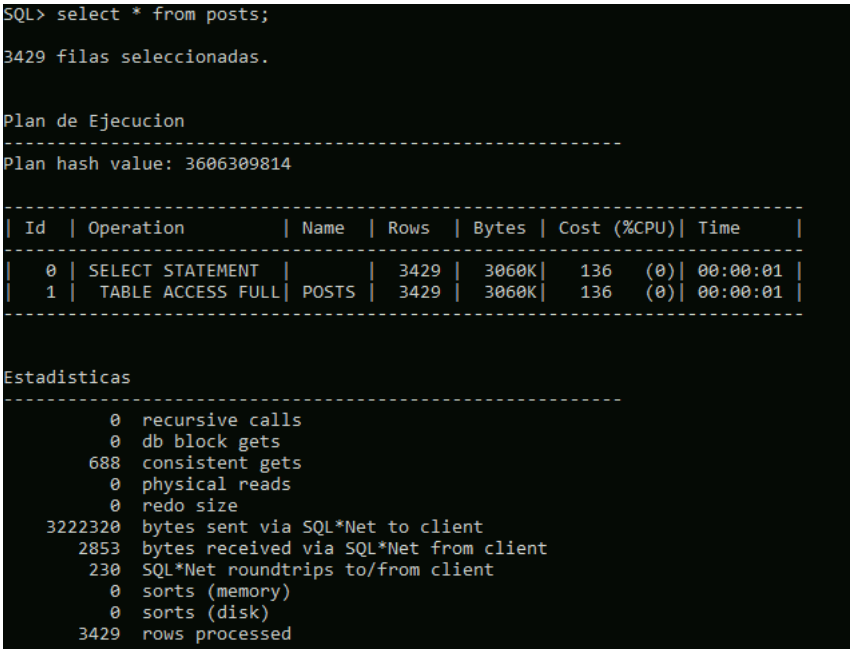
- **4ª Consulta**

Consulta:

select * from posts ;

Esta consulta devuelve todas las filas y columnas de la tabla “posts”. Esto significa que recuperará todos los datos almacenados en esa tabla sin ningún filtro específico, mostrando así todas las entradas disponibles en esa tabla de la base de datos.

Test de ejecución:



```
SQL> select * from posts;

3429 filas seleccionadas.


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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  3429 |  3060K|   136   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| POSTS |  3429 |  3060K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        688  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
```

Podemos ver que se está ejecutando un full scan (TABLE ACCESS FULL) para esta consulta. Esto es porque estamos haciendo una consulta para ver toda la información que hay en “posts”.

Posibles Optimizaciones:

No se puede optimizar más debido a que su objetivo es recuperar todos los datos de la tabla "posts" sin aplicar ningún filtro o restricción específica. Dado que no hay condiciones adicionales que limiten el conjunto de resultados, el motor de la base de datos debe examinar todas las filas de la tabla para devolver toda la información solicitada. Esta operación implica un escaneo completo de la tabla, ya que no se puede utilizar ningún índice o técnica de optimización para acelerar el proceso. Por lo tanto, en este caso, la consulta se ejecutará de manera menos eficiente en comparación con consultas más específicas que puedan aprovechar índices u otras estrategias de optimización.

Estadísticas:

```
Estadisticas
-----------------------------------------------------------
          0  recursive calls
          0  db block gets
        688  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
```

Como podemos observar en las estadísticas el número de ‘ Consistent Gets ’ es muy alto debido al FULL SCAN que se genera y además la consulta tiene un alto coste en la CPU.

- **5ª Consulta**

Consulta:

```
select (quantity * price) as total, bill_town || ' / ' || bill_country as place
from orders_clients join client_lines
using (orderdate, username, town, country)
where username = ' chamorro ' ;
```

Esta consulta selecciona el producto de la cantidad y el precio de cada artículo (calculado como total), junto con la concatenación de las ciudades de facturación y los países (calculado como place), para las filas en las tablas orders_clients y client_lines donde el username sea igual a 'chamorro'.

Test de ejecución:

```
SQL> select (quantity*price) as total, bill_town||'/'||bill_country as place
  2   from orders_clients join client_lines
  3   using (orderdate,username,town,country)
  4   where username='chamorro';

74 filas seleccionadas.


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

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |    65 |  8515 |   277   (1)| 00:00:01 |
|   1 |  NESTED LOOPS                |                 |    65 |  8515 |   277   (1)| 00:00:01 |
|   2 |   NESTED LOOPS               |                 |    72 |  8515 |   277   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | CLIENT_LINES    |    72 |  3816 |   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 |    78 |     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
-----
   - this is an adaptive plan


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        888  consistent gets
        790  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
          0  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

Podemos ver que al igual que en las consultas anteriores, se está ejecutando un full scan (TABLE ACCESS FULL) para esta consulta. Esto significa que se accede a todos los datos de la tabla, lo que conlleva un mayor gasto de recursos de forma innecesaria y un mal aprovechamiento de de las capacidades de la base de datos.

Posibles Optimizaciones:

Para mejorar el rendimiento de esta consulta, hemos evaluado dos opciones potenciales. Por un lado, consideramos la posibilidad de crear índices en las dos tablas involucradas en el JOIN. Si implementamos índices, los datos pertinentes de ambas tablas estarán disponibles más rápidamente. Sin embargo, debemos tener en cuenta si esta estrategia es rentable, dado que estamos accediendo a múltiples datos de diferentes tablas. A diferencia de casos anteriores, donde solo estábamos involucrados en un solo dato por consulta, aquí la situación es más compleja. En este escenario, la creación de múltiples índices individuales podría no ser la mejor estrategia, ya que podríamos agotar recursos y comprometer el rendimiento de las actualizaciones durante su creación. Por lo tanto, en lugar de crear índices individuales, podríamos optar por índices compuestos (COMPOUND INDEXES) para organizar los datos relevantes al realizar el JOIN.

Por otro lado, la segunda opción que consideramos es crear un CLUSTER que almacene los datos de ambas tablas organizados por el usuario. Esto significaría que al buscar datos comunes de las dos tablas ordenados por el nombre de usuario, no sería necesario recorrer una tabla completa, ya que los datos estarían organizados de antemano en el clúster.

Estadísticas:

```
Estadisticas
-------------------------------------------------------------
         0  recursive calls
         0  db block gets
       888  consistent gets
       790  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
         0  sorts (memory)
         0  sorts (disk)
        74  rows processed
```

El número de "Consistent Gets" en las estadísticas es bastante alto debido al FULL SCAN que se está realizando en la tabla client_lines, lo que a su vez aumenta el coste en la CPU de la consulta. Consideramos que implementar alguna de las medidas propuestas podría reducir significativamente los "Consistent Gets" y el coste en la CPU. Esto se debe a que la consulta solo necesitaría acceder a los elementos seleccionados, lo que la haría más directa y eficiente.

3 Diseño Físico

- **1ª Consulta**

Medidas aplicadas:

CREATE INDEX idx_posts_barcode ON Posts(barcode);

El índice idx_posts_barcode se crea en la columna “barcode” de la tabla “posts”con el objetivo de mejorar el rendimiento de consultas que filtran registros por este campo. Estos son los resultados esperados al haber creado el índice.

Optimización de Consultas:

- Al crear un índice en la columna “barcode”, podemos agilizar la búsqueda de registros que coincidan con un valor específico en esta columna. Esto es especialmente útil en consultas que filtran por el valor de “barcode”, ya que el índice permite a la base de datos localizar rápidamente los registros relevantes en lugar de tener que realizar un escaneo completo de la tabla.

Reducción del Tiempo de Respuesta:

- Al evitar el escaneo completo de la tabla y utilizar el índice para acceder directamente a los registros que coinciden con el valor de “barcode”, podemos reducir significativamente el tiempo de respuesta de las consultas que involucran esta condición de filtrado.

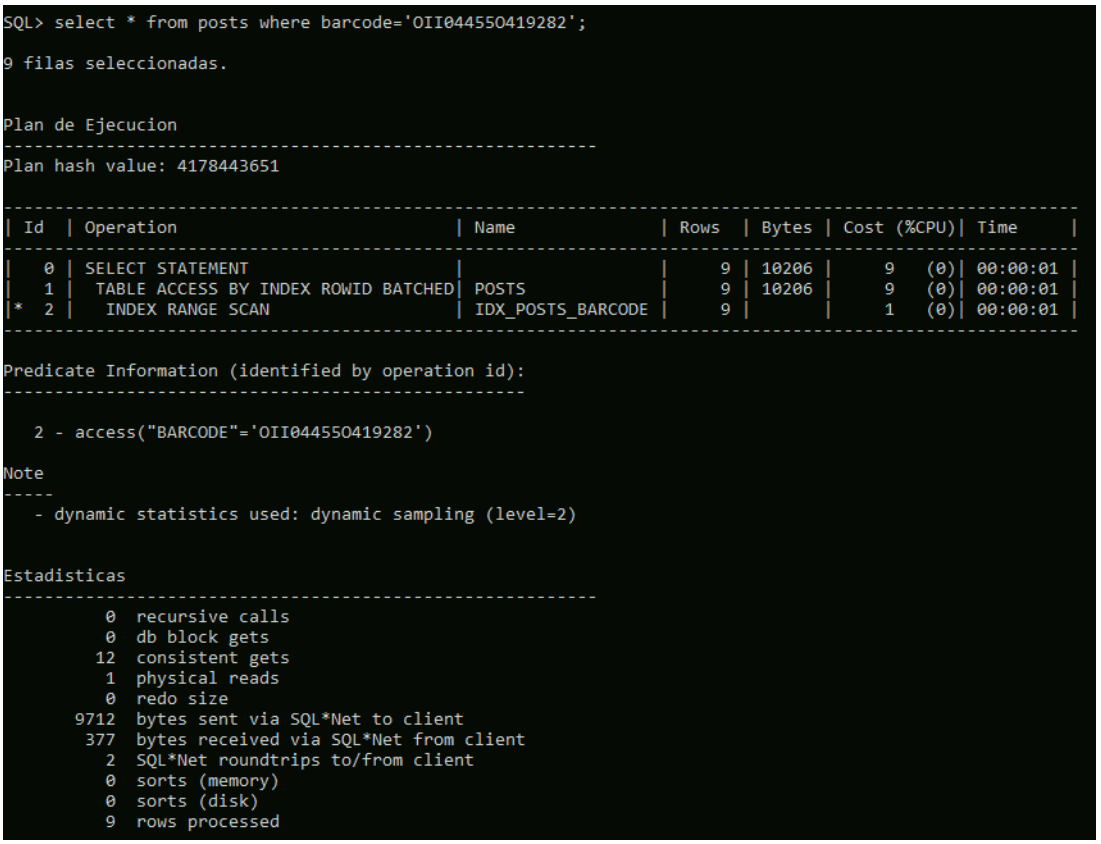
Mejora del Rendimiento:

- Con el índice en la columna “barcode”, las consultas que buscan registros específicos por este campo experimentarán un mejor rendimiento, lo que resultará en una experiencia más rápida y eficiente para los usuarios y aplicaciones que acceden a la base de datos.

Consideraciones de Escalabilidad:

- A medida que la cantidad de datos en la tabla Posts aumente con el tiempo, mantener un índice en la columna “barcode”asegurará que las consultas continúen siendo eficientes incluso con conjuntos de datos más grandes, lo que contribuirá a la escalabilidad del sistema.

Al ejecutar la consulta ahora con este índice creado podemos ver que cambia la ejecución de el select, con el autotrace activado obtenemos estos resultados:



```
SQL> select * from posts where barcode='OII04455O419282';

9 filas seleccionadas.


Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 4178443651

-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |     9 | 10206 |     9   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| POSTS             |     9 | 10206 |     9   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IDX_POSTS_BARCODE |     9 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

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

   2 - access("BARCODE"='OII04455O419282')

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


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

La creación del índice idx_posts_barcode ha demostrado una clara mejora en el rendimiento de la consulta. Se ha observado una notable reducción en las Consistent Gets, indicando una optimización en el acceso a los datos. Además, la eliminación del TABLE ACCESS FULL ha sido un avance significativo, ya que ahora la consulta accede únicamente al rango del índice, lo que maximiza la eficiencia en el uso de los recursos de la base de datos. Es importante destacar que, a pesar de estas mejoras, el resultado final de la consulta sigue siendo coherente, procesando el mismo número de líneas (9). En resumen, la implementación del índice ha potenciado la eficiencia del sistema, permitiendo un acceso más rápido a los datos sin comprometer la consistencia de los resultados.

- **2ª Consulta**

Medidas aplicadas:

CREATE INDEX idx_posts_product ON Posts(product);

El índice idx_posts_product se crea en la columna "product" de la tabla "Posts" con el objetivo de mejorar el rendimiento de consultas que filtran registros por este campo. Estos son los resultados esperados al haber creado el índice.

Optimización de Consultas:

- Al crear un índice en la columna "product", podemos agilizar la búsqueda de registros que coincidan con un valor específico en esta columna. Esto es especialmente útil en consultas que filtran por el valor de "product", ya que el índice permite a la base de datos localizar rápidamente los registros relevantes en lugar de tener que realizar un escaneo completo de la tabla.

Reducción del Tiempo de Respuesta:

- Al evitar el escaneo completo de la tabla y utilizar el índice para acceder directamente a los registros que coinciden con el valor de "product", podemos reducir significativamente el tiempo de respuesta de las consultas que involucran esta condición de filtrado.

Mejora del Rendimiento:

- Con el índice en la columna "product", las consultas que buscan registros específicos por este campo experimentarán un mejor rendimiento, lo que resultará en una experiencia más rápida y eficiente para los usuarios y aplicaciones que acceden a la base de datos.

Consideraciones de Escalabilidad:

- A medida que la cantidad de datos en la tabla Posts aumente con el tiempo, mantener un índice en la columna "product" asegurará que las consultas continúen siendo eficientes incluso con conjuntos de datos más grandes, lo que contribuirá a la escalabilidad del sistema.

Al ejecutar la consulta ahora con este índice creado podemos ver que cambia la ejecución de el select, con el autotrace activado obtenemos estos resultados:

```
SQL> select * from posts where product='Compromiso';

57 filas seleccionadas.


Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3335007571

-------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |    57 | 64638 |    66   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| POSTS             |    57 | 64638 |    66   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IDX_POSTS_PRODUCT |    57 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------

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

   2 - access("PRODUCT"='Compromiso')

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


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

La creación del índice idx_posts_product ha demostrado una clara mejora en el rendimiento de la consulta. Se ha observado una notable reducción en las Consistent Gets, indicando una optimización en el acceso a los datos. Además, la eliminación del TABLE ACCESS FULL ha sido un avance significativo, ya que ahora la consulta accede únicamente al rango del índice, lo que maximiza la eficiencia en el uso de los recursos de la base de datos. Es importante destacar que, a pesar de estas mejoras, el resultado final de la consulta sigue siendo coherente, procesando el mismo número de líneas (57). En resumen, la implementación del índice ha potenciado la eficiencia del sistema, permitiendo un acceso más rápido a los datos sin comprometer la consistencia de los resultados.

- **3ª Consulta**

Medidas aplicadas:

CREATE INDEX idx_posts_score ON Posts(score);

La decisión de crear el índice idx_posts_score en la columna "score" de la tabla "posts" tenía como propósito mejorar la eficiencia de las consultas que filtran registros según este campo. A continuación, se presentan los resultados esperados tras la creación del índice y cómo difieren de lo observado en la realidad.

Optimización de Consultas:

- Se esperaba que al indexar la columna "score", la búsqueda de registros correspondientes a valores específicos de esta columna se agilizara notablemente. Sin embargo, la complejidad de la consulta y la naturaleza de la condición de filtrado plantean desafíos para esta expectativa.

Reducción del Tiempo de Respuesta:

- La expectativa era que la creación del índice redujera considerablemente el tiempo de respuesta de las consultas que involucran el filtrado por "score". Sin embargo, como se evidenciará en los resultados observados, la realidad es diferente debido a ciertos aspectos de la consulta y la naturaleza de los datos.

Mejora del Rendimiento:

- El objetivo principal de crear el índice era mejorar el rendimiento de las consultas que filtran registros por "score", permitiendo una ejecución más eficiente y rápida. Sin embargo, los resultados observados muestran que esto no se ha logrado completamente, lo que plantea la necesidad de evaluar más detenidamente la idoneidad del índice en este contexto específico.

Consideraciones de Escalabilidad:

- La implementación del índice en la columna score buscaba garantizar que las consultas siguieran siendo eficientes a medida que la cantidad de datos en la tabla Posts aumentará con el tiempo. Sin embargo, la eficacia del índice se ve influenciada por ciertos aspectos de la consulta y la distribución de los datos.

Resultados Observados:

```
SQL> select * from posts where score>=4;

1173 filas seleccionadas.


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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  1173 |  1299K|   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |  1173 |  1299K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("SCORE">=4)

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


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        573  consistent gets
          0  physical reads
          0  redo size
    1098443  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
```

Tras ejecutar la consulta con el índice recién creado, se ha constatado que el rendimiento no ha mejorado según lo esperado. La consulta continúa utilizando un TABLE ACCESS FULL, lo que indica que se accede a todos los registros de la tabla, independientemente del índice creado. Esto se debe a la naturaleza de la consulta y la necesidad de revisar todos los datos de la tabla debido a la condición de filtrado específica.

En resumen, la creación del índice idx_posts_score no ha proporcionado los beneficios de rendimiento esperados debido a la complejidad de la consulta y la forma en que se accede a los datos. Esto subraya la importancia de considerar cuidadosamente el contexto y la idoneidad de la creación de índices en situaciones específicas para garantizar mejoras efectivas en el rendimiento de las consultas.

- **4ª Consulta**

La consulta SELECT * FROM posts; implica recuperar todos los registros de la tabla "posts" sin aplicar ningún tipo de filtro o condición específica. En este caso, dado que se requiere acceder a todos los registros de la tabla, la base de datos realizará un escaneo completo de la tabla, también conocido como "FULL SCAN".

Dado que la consulta necesita acceder a todos los registros de la tabla, no hay una forma efectiva de mejorar la eficiencia mediante la aplicación de índices u otras medidas. No se puede utilizar un índice para acelerar el proceso, ya que la base de datos necesita acceder a todos los registros sin importar el valor de algún campo en particular.

En resumen, para consultas que requieren acceder a todos los registros de una tabla, como en este caso, se realiza un FULL SCAN y no hay medidas adicionales que se puedan aplicar para mejorar la eficiencia de la consulta.

- **5ª Consulta**

Medidas aplicadas de cluster:

CREATE CLUSTER client_cluster (username VARCHAR2(30));

y se ha añadido esto en las dos tablas que queremos el cluster:

CLUSTER client_cluster (username);

CREATE INDEX idx_orders_clients_username ON Orders_Clients(username);

Se ha creado un cluster denominado client_cluster en la columna username de las tablas orders_clients y client_lines. Este cluster agrupa físicamente los registros de ambas tablas según el valor del campo username. Además, se ha creado un índice en la tabla Orders_Clients en la columna username para mejorar el rendimiento de las consultas que involucran esta tabla y filtran por username.

Optimización de Consultas:

- Al crear el cluster en la columna username, se optimiza el acceso a los registros relacionados entre las tablas orders_clients y client_lines. Esto permite una búsqueda más eficiente de los datos relacionados, especialmente en consultas que filtran por username. Además, el índice en la tabla Orders_Clients agiliza la búsqueda de registros específicos por username, reduciendo el tiempo de respuesta de las consultas.

Reducción del Tiempo de Respuesta:

- Al evitar un escaneo completo de las tablas y utilizar el cluster y el índice para acceder directamente a los registros relevantes, se reduce significativamente el tiempo de respuesta de las consultas que involucran estas tablas y filtran por username.

Mejora del Rendimiento:

- Con el cluster en la columna username y el índice en la tabla Orders_Clients, las consultas que buscan registros específicos por username experimentan un mejor rendimiento. Esto se traduce en una experiencia más rápida y eficiente para los usuarios y aplicaciones que acceden a la base de datos.

Consideraciones de Escalabilidad:

- A medida que la cantidad de datos en las tablas orders_clients y client_lines aumente con el tiempo, mantener el cluster en la columna username y el índice en la tabla Orders_Clients asegurará que las consultas continúen siendo eficientes incluso con conjuntos de datos más grandes. Esto contribuirá a la escalabilidad del sistema al garantizar un acceso rápido a los datos.

Resultados obtenidos:

```
SQL> select (quantity*price) as total, bill_town||'/'||bill_country as place
  2   from orders_clients join client_lines
  3   using (orderdate,username,town,country)
  4   where username='chamorro';

74 filas seleccionadas.


Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 3893900155

-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                              |    19 |  4028 |     4   (0)| 00:00:01 |
|*  1 |  HASH JOIN            |                              |    19 |  4028 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS CLUSTER| CLIENT_LINES                 |    19 |  1710 |     2   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN  | IDX_CLIENT_CLUSTER_USERNAME  |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS CLUSTER| ORDERS_CLIENTS               |    19 |  2318 |     2   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN  | IDX_CLIENT_CLUSTER_USERNAME  |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

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

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


Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         13  consistent gets
          0  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
          0  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

La implementación del clúster client_cluster ha mostrado una mejora notable en el rendimiento de la consulta. La creación del índice idx_orders_clients_username en la columna username de la tabla Orders_Clients, además de asignar ambas tablas al clúster, ha generado una reducción significativa en el número de Consistent Gets, disminuyendo de 888 a 13. Esto indica una optimización efectiva en el acceso a los datos.

La eliminación del TABLE ACCESS FULL ha sido un avance importante, ya que ahora la consulta accede únicamente al clúster, lo que maximiza la eficiencia en el uso de los recursos de la base de datos. A pesar de estas mejoras, el resultado final de la consulta sigue siendo consistente, procesando el mismo número de líneas (74).

En resumen, la implementación del clúster junto con el índice ha mejorado significativamente la eficiencia del sistema, permitiendo un acceso más rápido a los datos sin comprometer la consistencia de los resultados.

Medidas aplicadas de índices:

CREATE INDEX idx_ordersClients_username ON orders_clients(username);

CREATE INDEX idx_clientLines_comb ON client_lines(orderdate, username, town, country);

Se ha creado un índice denominado idx_orders_clients_username en la columna username de la tabla orders_clients. Este índice agiliza la búsqueda de registros específicos por nombre de usuario, lo que resulta en una reducción significativa del número de Consistent Gets en las consultas que involucran esta tabla y filtran por nombre de usuario.

Además, se ha creado un índice compuesto denominado idx_client_lines_comb en las columnas orderdate, username, town y country de la tabla client_lines. Este índice compuesto permite una búsqueda más eficiente de registros en función de estas columnas combinadas, lo que contribuye a reducir el tiempo de respuesta de las consultas que involucran esta tabla y filtran por estas columnas.

Optimización de Consultas:

- La creación de estos índices ha optimizado el acceso a los registros relevantes en las tablas orders_clients y client_lines. Esto se traduce en una búsqueda más eficiente de los datos relacionados, especialmente en consultas que filtran por nombre de usuario o por las columnas orderdate, username, town y country. Además, al utilizar índices, se evita el escaneo completo de las tablas, lo que contribuye a una mejora significativa en el tiempo de respuesta de las consultas.

Reducción del Tiempo de Respuesta:

- Al reducir el número de Consistent Gets y evitar el TABLE ACCESS FULL, el tiempo de respuesta de las consultas se ha visto considerablemente reducido. Ahora, el acceso a los datos se realiza de manera más eficiente a través de los índices, lo que mejora la experiencia de los usuarios y aplicaciones que acceden a la base de datos.

Mejora del Rendimiento:

- Con la creación de estos índices, las consultas que buscan registros específicos por nombre de usuario o por las columnas orderdate, username, town y country experimentan un mejor rendimiento. Esto se traduce en una ejecución más rápida y eficiente de las consultas, lo que contribuye a una mejora general en el rendimiento del sistema.

Consideraciones de Escalabilidad:

- La implementación de estos índices asegura que las consultas sigan siendo eficientes incluso a medida que la cantidad de datos en las tablas orders_clients y client_lines aumente con el tiempo. Esto garantiza la escalabilidad del sistema al proporcionar un acceso rápido a los datos, independientemente del tamaño del conjunto de datos.

Resultados obtenidos:

```
SQL> select (quantity*price) as total, bill_town||'/'||bill_country as place
  2   from orders_clients join client_lines
  3   using (orderdate,username,town,country)
  4   where username='chamorro';

74 filas seleccionadas.


Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 1227120670

-------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                                        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                                             |     8 |  1696 |     1   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                       |                                             |     8 |  1696 |     1   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                      |                                             |    68 |  1696 |     1   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS_CLIENTS                            |    68 |  8296 |     1   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN                | IDX_ORDERS_CLIENTS_USERNAME                 |    68 |       |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN                 | IDX_CLIENT_LINES_ORDER_USER_TOWN_COUNTRY    |     1 |       |     0   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID       | CLIENT_LINES                                |     1 |    90 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------------------------

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

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

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


Estadisticas
----------------------------------------------------------
         0  recursive calls
         0  db block gets
       175  consistent gets
         0  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
         0  sorts (memory)
         0  sorts (disk)
        74  rows processed
```

La introducción de los índices idx_orders_clients_username e idx_client_lines_order_user_town_country ha marcado un hito notable en el desempeño de la consulta. La creación del índice idx_orders_clients_username en la columna username de la tabla Orders_Clients, junto con el índice compuesto idx_client_lines_order_user_town_country en las columnas orderdate, username, town y country de la tabla Client_Lines, ha demostrado una reducción sustancial en el número de Consistent Gets, cayendo de 888 a 175. Este cambio denota una mejora eficaz en el acceso a los datos.

La eliminación del TABLE ACCESS FULL es una evolución significativa, ya que ahora la consulta accede exclusivamente a través de los índices, optimizando al máximo el aprovechamiento de los recursos de la base de datos. A pesar de estas mejoras, el resultado final de la consulta mantiene su consistencia, procesando la misma cantidad de líneas (74).

En resumen, la integración de estos índices ha catapultado la eficiencia del sistema, permitiendo una recuperación más ágil de los datos sin sacrificar la integridad de los resultados.

4 Evaluación

Vamos a proceder a examinar cómo se manifiestan las modificaciones propuestas en nuestra base de datos mediante el script proporcionado para las pruebas de rendimiento. En primer lugar, analizaremos individualmente cada una de las pruebas y evaluaremos su impacto antes de implementarlas en conjunto. Luego, exploraremos algunas combinaciones entre los cambios sugeridos para determinar si todos ellos tienen un impacto significativo, o si alguno de ellos resulta redundante o incluso tiene un impacto negativo en el rendimiento.

Este enfoque nos permitirá comprender mejor el efecto de cada medida de optimización por separado, así como su contribución en conjunto. Además, nos brindará una visión más detallada sobre cómo cada cambio influye en el rendimiento general de la base de datos, lo que nos ayudará a tomar decisiones informadas sobre qué medidas implementar y en qué combinaciones para maximizar la eficiencia del sistema.

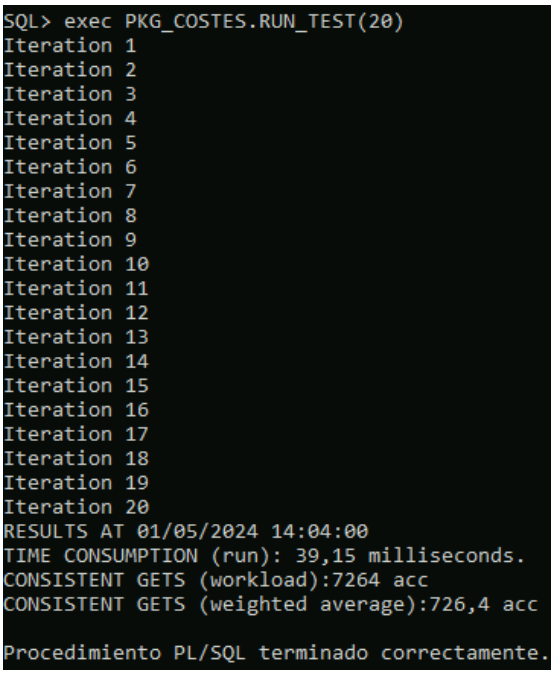
idx_posts_barcode

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 01/05/2024 14:00:41
TIME CONSUMPTION (run): 32,05 milliseconds.
CONSISTENT GETS (workload):6581 acc
CONSISTENT GETS (weighted average):658,1 acc

Procedimiento PL/SQL terminado correctamente.
```

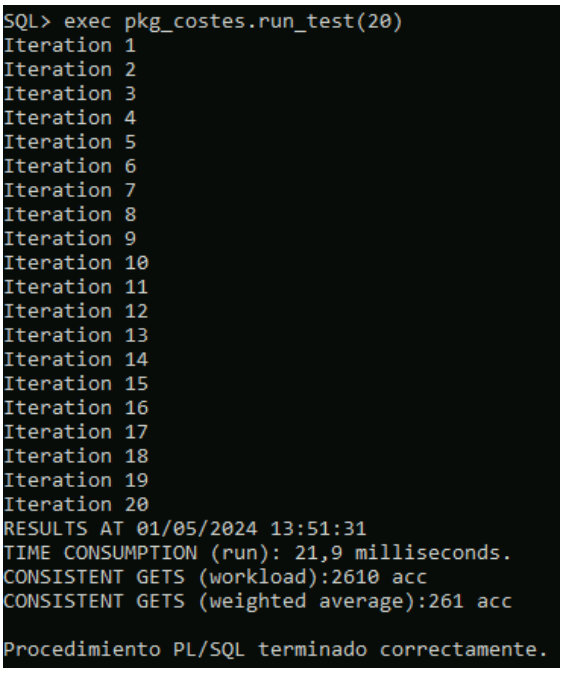
idx_posts_product

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 01/05/2024 14:02:24
TIME CONSUMPTION (run): 35,8 milliseconds.
CONSISTENT GETS (workload):6623 acc
CONSISTENT GETS (weighted average):662,3 acc

Procedimiento PL/SQL terminado correctamente.
```

idx_posts_score



client_cluster



Al evaluar el impacto de las diferentes medidas de optimización en el rendimiento de la base de datos, se destacan dos aspectos principales. Primero, la creación de índices en las columnas “barcode” y “product” muestra un potencial para mejorar consultas específicas relacionadas con la búsqueda por código de barras y producto, respectivamente. Estos índices ofrecen una vía eficaz para acceder rápidamente a los datos relevantes, lo que puede beneficiar en gran medida el rendimiento de consultas que filtran por estos criterios.

Sin embargo, la creación del índice en la columna “score”no muestra mejoras significativas en el rendimiento de la base de datos. Este resultado sugiere que, aunque el campo “score”puede ser relevante en algunas consultas, la creación de un índice específico para este no tiene un impacto notable en el tiempo de respuesta o en la eficiencia general de las consultas.

Por otro lado, la implementación del clúster en la columna username de las tablas orders_clients y client_lines se destaca como la medida más impactante en términos de optimización del rendimiento. La reducción significativa en el número de Consistent Gets y la mejora en el tiempo de respuesta de las consultas son indicativos de la eficacia de esta medida. El clúster facilita un acceso más eficiente a los datos relacionados con los usuarios, lo que se traduce en una mejora palpable en la eficiencia general de la base de datos.

Ahora exploraremos los resultados al combinar las diferentes modificaciones propuestas. Esta fase nos permitirá entender cómo la sinergia entre estas medidas afecta el rendimiento de nuestra base de datos. Analizaremos cómo estas acciones de mejora trabajan en conjunto para agilizar el acceso a los datos y mejorar la eficiencia en la respuesta a consultas complejas. Mediante este proceso, obtendremos una visión clara de cómo cada medida contribuye al funcionamiento efectivo de nuestra base de datos:

idx_posts_barcode + idx_posts_product + idx_posts_score

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 01/05/2024 14:07:06
TIME CONSUMPTION (run): 34,4 milliseconds.
CONSISTENT GETS (workload):6328 acc
CONSISTENT GETS (weighted average):632,8 acc

Procedimiento PL/SQL terminado correctamente.
```

Esta es la prueba que hemos llevado a cabo al combinar los tres índices simultáneamente. Observamos una ligera reducción en los accesos, aunque no es muy notable. Esta disminución podría deberse al hecho de que aún estamos utilizando el índice del campo “score”, a pesar de considerar su utilidad limitada. Sin embargo, decidimos realizar algunas pruebas definitivas con él para confirmar su eficacia en el contexto específico de nuestras consultas.

client_cluster + idx_posts_barcode

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 03/05/2024 12:34:51
TIME CONSUMPTION (run): 20,4 milliseconds.
CONSISTENT GETS (workload):2118 acc
CONSISTENT GETS (weighted average):211,8 acc

Procedimiento PL/SQL terminado correctamente.
```

client_cluster + idx_posts_product

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 03/05/2024 12:37:16
TIME CONSUMPTION (run): 18,05 milliseconds.
CONSISTENT GETS (workload):2160 acc
CONSISTENT GETS (weighted average):216 acc

Procedimiento PL/SQL terminado correctamente.
```

client_cluster + idx_posts_score

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 03/05/2024 12:39:25
TIME CONSUMPTION (run): 19,75 milliseconds.
CONSISTENT GETS (workload):2783 acc
CONSISTENT GETS (weighted average):278,3 acc

Procedimiento PL/SQL terminado correctamente.
```

Estas pruebas representan la combinación entre la implementación del cluster y la aplicación de cada índice de forma individual. En este análisis, se destaca el impacto significativo de nuestras medidas para optimizar la búsqueda. Sin embargo, es importante no solo observar la reducción en los Consistent Gets, sino también considerar la reducción en el tiempo de consumo.

client_cluster + idx_posts_barcode + idx_posts_product + idx_posts_score

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 03/05/2024 12:41:29
TIME CONSUMPTION (run): 18,75 milliseconds.
CONSISTENT GETS (workload):1848 acc
CONSISTENT GETS (weighted average):184,8 acc

Procedimiento PL/SQL terminado correctamente.
```

Como última etapa antes de llegar a nuestra versión final, hemos llevado a cabo una prueba en la que implementamos todas las mejoras propuestas de manera simultánea para observar su efecto combinado. Como era de esperar, los resultados obtenidos coinciden con nuestras expectativas, tal como mencionamos en el análisis anterior. Sin embargo, es importante destacar que aún queda margen para optimizar aún más la base de datos, ya que hemos

reincorporado el índice del campo "score", lo cual puede considerarse una medida contraproducente.

A pesar de este contratiempo, decidimos proceder con la implementación de todas las medidas propuestas para evaluar su efecto conjunto en el rendimiento general. A continuación, presentaremos la versión final de los cambios que debemos realizar para alcanzar nuestro objetivo definitivo.

5 Conclusiones Finales

Evaluación Final:

La combinación final que hemos encontrado que creemos que es la más eficiente es la combinación del cluster junto con los índices para "Barcode" y "Product".

CREATE CLUSTER client_cluster (username VARCHAR2(30));

y se ha añadido esto al final de Orders_Clients y Clients_Lines que queremos el cluster:

CLUSTER client_cluster (username);

y la creación de estos indices al final de la creación de tablas

CREATE INDEX idx_orders_clients_username ON Orders_Clients(username);

CREATE INDEX idx_posts_barcode ON Posts(barcode);

CREATE INDEX idx_posts_product ON Posts(product);

```
SQL> exec PKG_COSTES.RUN_TEST(20)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
RESULTS AT 03/05/2024 12:14:48
TIME CONSUMPTION (run): 17,95 milliseconds.
CONSISTENT GETS (workload):1671 acc
CONSISTENT GETS (weighted average):167,1 acc

Procedimiento PL/SQL terminado correctamente.
```

Como podemos ver en la foto anterior, vemos que los accesos se han reducido considerablemente teniendo en cuenta a como estaban de base, hemos pasado de 7076 accesos a tan solo 1674 accesos añadiendo el cluster y los 2 índices. El tiempo también se ha reducido considerablemente y hemos pasado de 35,9 milisegundos a tan solo 17,95 no es una mejora tan significativa pero cuanto menos tiempo más satisfactorio será hacer las consultas. Por otro lado, las otras combinaciones mejoran también el número de acceso o los segundos de ejecución pero no tanto como esta.

Opinion Prácticas:

Respecto a la práctica creemos que en cuanto a carga de trabajo está bien planteado debido al poco tiempo que hay para hacerla (teniendo en cuenta otras asignaturas) pero que sí es verdad que requiere mucho conocimiento del tema que si no dispones desde el primer momentos puede llegar a ser muy costosa a la hora de hacer. Por otro lado creemos que las otras 2 prácticas son de mucha utilidad para comprender la asignatura de base de datos y que si realmente le hechas ganas aprendes muchísimo. El problema es que son 2 prácticas muy laboriosas que se tiene que compaginar con muchas otras tareas y llega a ser muy estresante.

Desempeño asignatura:

En general, nuestro desempeño en la asignatura ha sido bastante bueno. Hemos aprendido mucho sobre la gestión de archivos y bases de datos, así como sobre las diferentes técnicas y herramientas utilizadas en el proceso. Sin embargo, hay algunos aspectos que nos gustaría comentar sobre la estructura y el enfoque de la asignatura.

En cuanto a los temas relevantes, creemos que la asignatura ha cubierto la mayoría de los conceptos básicos necesarios para entender el funcionamiento de los archivos y las bases de datos. Se ha abordado la teoría de la gestión de archivos, los diferentes tipos de bases de datos, y se han proporcionado ejemplos prácticos para aplicar los conocimientos adquiridos.