Alumno/a:	**Iván Sebastián Loor Weir**	**NIA:**	**100448737**
Alumno/a:	**Álvaro González Fúnez**	**NIA:**	**100451281**
Alumno/a:	**Arturo Jiménez Tajuelo**	**NIA:**	**100451161**

1 Introducción

Esta tercera, y última, práctica sobre bases de datos, trata sobre los costes asociados a utilizar un sistema de bases de datos, es decir, los recursos de tiempo (y espacio) que son necesarios.

Dependiendo de la organización física de nuestros datos, hay unas operaciones que serán más eficientes que otras, por ejemplo, hacer una consulta sobre la clave privilegiada cuando se almacena de forma direccionada tiene coste unitario, mientras que en otras circunstancias puede suponer un fullscan. Un fullscan es el peor caso posible, sucede cuando tengamos que acceder a varias ocurrencias dispersas en una tabla, sin que las podamos localizar mediante algún índice o clave.

Nuestros objetivos en esta práctica son analizar las operaciones proporcionadas y su frecuencia, y a partir de este punto estudiaremos su plan de ejecución y su coste, para seguidamente proponer una solución (de todas las posibles) que optimice las operaciones.

2 Análisis

El diseño físico actual: por defecto, Oracle SGBD usa una organización serial no consecutiva, y los bloques ocupan 8KB. Dada esta organización, una modificación que podemos hacer en nuestros modelos se basa en usar PCTFREE y PCTUSED, que marcan el porcentaje reservado para modificaciones (por defecto un 10%) y el porcentaje mínimo de ocupación (por defecto un 60%) de cada bloque, así como cambiar el tamaño de bloque (16 KB o 2KB, Oracle no admite más opciones).

La carga de trabajo prototípica consta de 5 procesos, que a su vez son 5 consultas con frecuencias relativas: {0.1, 0.1, 0.1, 0.2, 0.5}. Para realizar estos tests de forma consecutiva disponemos de run_test, que usamos y analizamos más adelante.

Para estudiar cómo ejecuta Oracle y poder estudiar los efectos de nuestros cambios, activamos:

-set autotrace traceonly; Esto nos muestra únicamente el optimizador y las estadísticas.

-set timing on; Para ver los tiempos de ejecución y hacer media.

-set serveroutput on;

Para los análisis tenemos que tener en mente cómo Oracle SQL trata una query. Para empezar la analiza (parsing) para validar la sintaxis. A partir del árbol que obtiene enlaza las entidades con las tablas. Ahora hace un proceso interno de optimización, genera su plan de ejecución y finalmente ejecuta la consulta. Todo esto lo hace por debajo y no es visible al usuario, pero un buen DBA debe comprender qué está ocurriendo detrás de los focos y ser capaz de intuir el por qué de las decisiones tomadas por oracle y cómo se le podría ayudar a ejecutarlas de forma óptima. Tenemos que ser conscientes que gran parte del tiempo el optimizador va a tener razón, pero nosotros podremos ayudarle creando otras estructuras auxiliares como índices o clusters.

Consulta 1

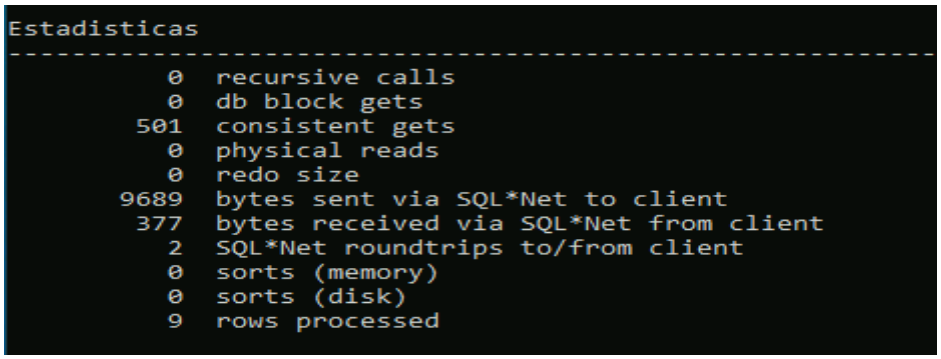
select * from posts where barcode='OII04455O419282';

Se puede apreciar como sólo se buscan los posts que tengan el número de referencia pasado. Al tratarse de una clave secundaria, va a buscar en toda la tabla posts donde el código de barras sea igual al pasado. Podemos ver lo que ocurre por debajo de Oracle SQL al ejecutar el comando:

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

9 filas seleccionadas.

Transcurrido: 00:00:00.04

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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    24 | 27216 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |    24 | 27216 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("BARCODE"='OII04455O419282')

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


Estadisticas
----------------------------------------------------------
         45  recursive calls
         47  db block gets
        600  consistent gets
          8  physical reads
       9188  redo size
       9689  bytes sent via SQL*Net to client
        354  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

Notamos cómo hace un acceso completo a la tabla Posts a pesar del filtrado y el poco tiempo que tarda de por sí la consulta. Realmente, dado el pequeño tamaño de nuestra muestra, las

optimizaciones que hagamos no serán muy significativas. Al repetir la consulta varias veces, las estadísticas se muestran constantes de esta manera:



Hay una gran diferencia entre la primera ejecución y las consecutivas. Mientras que en la primera la mayoría de los gets tienen physical reads, en los consecutivos no.

Tras ejecutar 10 veces esta consulta (recordando que la primera es la más orgánica y significativa, pues no sale de memoria si no que ha de buscar en disco), tenemos los siguientes tiempos:

0.04	0.02	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.02

La consulta es rápida y eficaz, dejándonos como única medidas de mejora tener menos coste.

Consulta 2

select * from posts where product='Compromiso';

Aquí nos enfocamos en buscar los posts que tengan el producto referido. Siguiendo con la misma dinámica que la anterior consulta, va a realizar un acceso completo con el filtrado de la clave secundaria:

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

57 filas seleccionadas.

Transcurrido: 00:00:00.02

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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    39 | 44226 |   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |    39 | 44226 |   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

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

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


Estadisticas
----------------------------------------------------------
         45  recursive calls
         48  db block gets
        612  consistent gets
          9  physical reads
       9244  redo size
      54505  bytes sent via SQL*Net to client
        382  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

Y teniendo en cuenta las estadísticas constantes de las demás repeticiones:

```
Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        503  consistent gets
          0  physical reads
          0  redo size
      54505  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
```

Nos percatamos de que su coste es ligeramente superior a la primera consulta, debiéndose sobre todo a que en la estructura de nuestra práctica ya se sabe que una referencia (barcode) se corresponde con una variedad de productos.

Se ha ejecutado 10 veces y hemos obtenido los siguientes tiempos:

0.02	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01

A pesar de tener un coste más grande con respecto a la consulta anterior, tenemos un tiempo más pequeño. Forzando que haya physical reads (después de resetear) el tiempo de la operación se queda en 0.01s, lo que parece indicar que, independientemente de si los datos se tienen en memoria, la operación se ejecuta con rapidez. Esto nos deja con la opción de optimizar la tabla posts con respecto a los dos atributos: barcode y product.

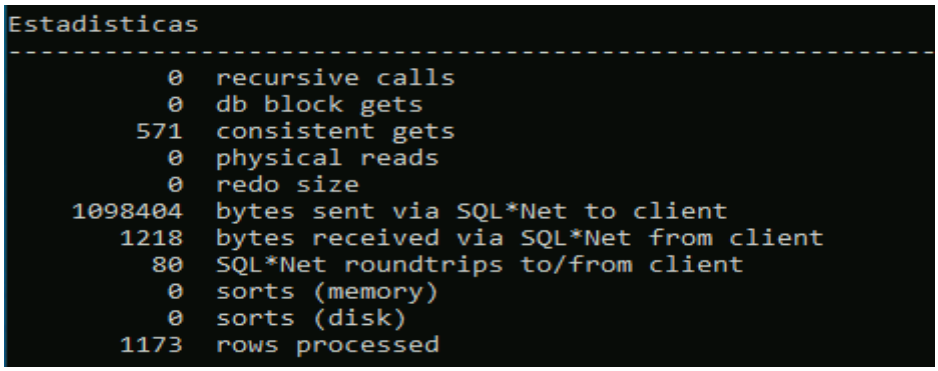
Consulta 3

select * from posts where score>=4;

En este caso, se recorren los posts cuyo score sea mayor o igual a 4. Esta búsqueda sin más información podría requerir fullscan. Nos imprime:

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

1173 filas seleccionadas.

Transcurrido: 00:00:00.10

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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |   882 |   976K|   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |   882 |   976K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("SCORE">=4)

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


Estadisticas
----------------------------------------------------------
         52  recursive calls
         47  db block gets
        666  consistent gets
        509  physical reads
       9316  redo size
    1098404  bytes sent via SQL*Net to client
       1195  bytes received via SQL*Net from client
         80  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1173  rows processed
```

En los datos subsecuentes tenemos:



Se han seleccionado muchas más filas que en las anteriores consultas y por tanto ha tenido un mayor coste nuestra consulta. Si organizamos esta tabla de forma secuencial usando como clave el score, como ordenamiento podríamos usar búsqueda dicotómica extendida. O un índice con un árbol B+.

Hemos realizado la consulta 10 veces y hemos obtenido los siguientes tiempos:

0.10	0.06	0.04	0.05	0.05	0.05	0.05	0.05	0.04	0.05

Empezando con un tiempo mucho mayor pero terminando en una oscilación entre 0.05 y 0.04

Consulta 4

select * from posts;

Se realiza una consulta total a la tabla posts para ver toda su información. A priori, suponemos que su coste va a ser mucho mayor al hacer un escaneo total y a su vez su tiempo de respuesta:

```
SQL> select * from posts;

3429 filas seleccionadas.

Transcurrido: 00:00:00.15

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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |  3078 |  3408K|   136   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| POSTS |  3078 |  3408K|   136   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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


Estadisticas
----------------------------------------------------------
         38  recursive calls
         47  db block gets
        803  consistent gets
          3  physical reads
       9068  redo size
    3222320  bytes sent via SQL*Net to client
       2830  bytes received via SQL*Net from client
        230  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       3429  rows processed
```

En las siguientes iteraciones sobre la consulta nos da:

```
Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        708  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 cabía esperar, todas sus estadísticas son mayores a todas las antecesoras, dejándonos como única duda el cuanto tarda en tiempo.

La ejecutamos 10 veces y conseguimos los siguientes tiempos:

0.15	0.14	0.13	0.15	0.14	0.13	0.13	0.15	0.13	0.12

Nos sorprende que tome tanto tiempo la consulta, aunque en otras sesiones hemos tenido 0.11 o 0.17, la constante en la primera consulta generalmente suele ser 0.15 y luego baja ligeramente sus segundos, llegando en algunos casos a 0.08 excepcionalmente. Se nos ocurre que diseñar un cluster en esta tabla podría ser una buena solución al problema que se nos presenta.

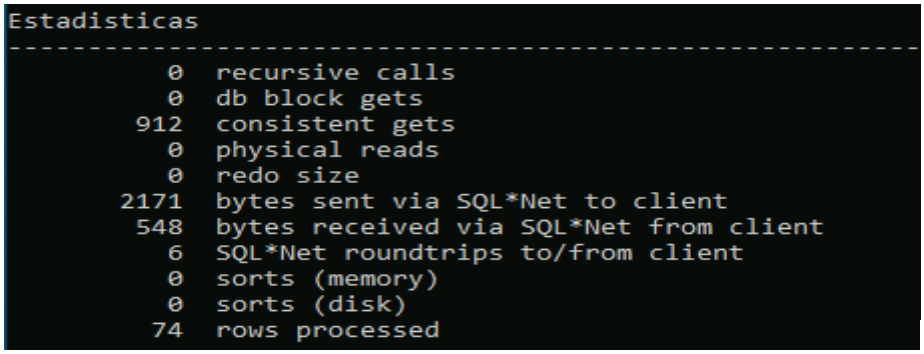
Consulta 5

```
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';
```

En esta consulta, se está calculando el total de cada orden (cantidad multiplicada por precio) y la dirección de facturación (ciudad/país) para todas las órdenes asociadas al usuario 'chamorro'. Tenemos que su respuesta es:

```
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.

Transcurrido: 00:00:00.07

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

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |     8 |  1696 |   213   (1)| 00:00:01 |
|   1 |  NESTED LOOPS                |                 |     8 |  1696 |   213   (1)| 00:00:01 |
|   2 |   NESTED LOOPS               |                 |     8 |  1696 |   213   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | CLIENT_LINES    |     8 |   720 |   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
----------------------------------------------------------
         54  recursive calls
         69  db block gets
       1118  consistent gets
          6  physical reads
      15300  redo size
       2171  bytes sent via SQL*Net to client
        525  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

Las estadísticas siguientes son:



Vemos que empieza recorriendo la tabla de Orders_Clients usando el acceso por índice único, gracias a su clave primaria y luego hace un escaneo completo de la tabla Client_Lines. Para esto se vale de bucles anidados, que buscan coincidencias entre la tabla externa y la tabla interna. Con esto debiéndose al JOIN, también nos damos cuenta de que quizá se podría optimizar si se tiene una organización o un índice que favorezca el acceso a orderdate, username, town y country

Se ha realizado 10 veces y hemos obtenido los tiempos:

0.07	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02

3 Diseño Físico

PCTUSED, PCTFREE

Lo primero que nos llama la atención es el tipo de operaciones. Consultas, nada de operaciones, ni actualizaciones ni borrados. Solo conociendo esto, creemos que aumentar PCTUSED al máximo (99%) podría optimizar nuestras consultas por clave no privilegiada.
Esto no acarrearía ningún tipo de consecuencia al no haber otro tipo de operaciones, pero en un caso real sería más prudente dejar cierto margen (por ejemplo 80%), ya que en el caso de cambios en los registros existentes (incluso si son relativamente pocos) pueden rápidamente aumentar el tiempo de las operaciones, invalidando cualquier optimización previamente obtenida.

Tamaño de bloque

Escoger el tamaño de bloque no es tan obvio. Aumentarlo obviamente reducirá el número de lecturas a almacenamiento secundario, pero reducirá la densidad de bloque en medida similar debido al sistema de organización de ficheros.
Entonces, decidimos probar por cada tamaño, y vimos que 8k y 16k tienen más o menos el mismo comportamiento (con el de 16k teniendo la mitad de consistent gets por motivos obvios), y el de 2k siendo ligeramente más lento. Podríamos justificar esto porque, bajando el tamaño de bloque, llega un momento en el que el incremento de densidad (registros por bloque) no es suficiente para contrarrestar la creciente cantidad de lecturas a almacenamiento secundario, que son muy costosas.

Índices

Los índices nos permiten favorecer búsquedas de clave no privilegiada a cambio de usar estructuras auxiliares en almacenamiento secundario. Tras analizar las consultas, en especial la última, vimos una clara oportunidad de mejora en convertir username (de Orders_Clients) en una clave auxiliar para optimizar la consulta.

Al utilizar los cuatro atributos de la clave primarias de Orders_Clients tuvimos problemas incluyendo orderdate, saltando como error que esa lista de columnas ya estaba indexada, es por eso que fuimos probando con cada uno por separado y luego uniendo todos, hasta quedar con el mejor índice:

Índice 1:
CREATE INDEX ind_compuesto_usuario ON orders_clients(username, town, country);

Al probarlo en la última consulta, cambian los tiempos de respuesta, mejorando de esta forma:

```
  3   using (orderdate,username,town,country)
  4   where username='chamorro';

74 filas seleccionadas.

Transcurrido: 00:00:00.05

Plan de Ejecucion
----------------------------------------------------------
Plan hash value: 2104429266

-------------------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                      |    12 |  2544 |   127   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                         |                      |    12 |  2544 |   127   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                        |                      |    68 |  2544 |   127   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS_CLIENTS       |    68 |  8296 |    22   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN                  | IND_COMPUESTO_USUARIO|    68 |       |     3   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN                   | PK_CLIENTLINES       |     1 |       |     1   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID         | CLIENT_LINES         |     1 |    90 |     2   (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)
   - this is an adaptive plan


Estadisticas
----------------------------------------------------------
         44  recursive calls
         52  db block gets
        509  consistent gets
          5  physical reads
      12404  redo size
       2171  bytes sent via SQL*Net to client
        525  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         74  rows processed
```

Al realizar de nuevo varias veces las consultas, nos muestra:

```
Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        267  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
```

Como principal beneficio, vemos que el coste baja notablemente y lo mismo sucede al analizar nuestras estadísticas.

También ejecutamos 10 veces el proceso, obteniendo los siguientes tiempos:

0.05	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

El resultado es muy favorable, aunque esperado, debido a que sabíamos que no podíamos mejorar considerablemente el tiempo, pero aunque sea lo mejoramos un poco.

Asimismo, decidimos probar con un índice que se ocupara de las cuatro consultas anteriores. Sabíamos que sería un desperdicio añadirle el score como parámetro y nos basamos solamente en los atributos que eran más rebuscados de esa tabla. Debido a eso, optamos por crear un índice compuesto de los atributos barcode, product y username:

Índice 2:
CREATE INDEX ind_post ON posts(barcode, product, username);

El resultado que nos proporciona en las 4 consultas referentes a Posts, es el mismo tiempo de respuesta en cada una de ellas, con la única variación de que el coste mejora ínfimamente, casi milimétrico. Dejándonos con la duda de si vale la pena añadirlo como tal o no.

Cluster

Los clusters son una herramienta peligrosa. Tienen la capacidad de mejorar en gran medida consultas que de otro modo serían extremadamente lentas (joins de decenas de columnas, con decenas de miles de filas), mientras que penalizan enormemente cualquier otra operación (inserciones, actualizaciones, borrados) o incluso lecturas que no usan la clave privilegiada del cluster.

Para nuestro cluster, decidimos enfocarnos en las primeras 4 consultas ya que en la última es irrelevante. A primeras, parecería que lo más lógico sería ir por la última consulta debido a su complejidad y las combinaciones usadas, pero no lo es, al no haber una columna específica de una sola tabla que se esté utilizando para filtrar los resultados.

Para crearlo nos quedamos con el usuario correspondiente a la tabla Posts, por ser clave primaria:

```
CREATE CLUSTER usuario (username VARCHAR2(30));
```

Luego, editamos la tabla Posts para que se cree correctamente:

```
CREATE TABLE Posts (
  username   VARCHAR2(30),
  postdate   DATE,
  barCode    CHAR(15),
  product    VARCHAR2(50) NOT NULL,
  score      NUMBER(1) NOT NULL,
  title      VARCHAR2(50),
  text       VARCHAR2(2000) NOT NULL,
  likes      NUMBER(9) DEFAULT(0) NOT NULL,
  endorsed   DATE, -- null means it isn't endorsed; else, date of last purchase
  CONSTRAINT pk_posts PRIMARY KEY(username,postdate),
  CONSTRAINT fk_posts_clients FOREIGN KEY(username) REFERENCES Clients,
  CONSTRAINT fk_posts_references FOREIGN KEY(product,barcode)
          REFERENCES References(product,barcode),
  CONSTRAINT D_postscore CHECK (score between 0 and 5)
) CLUSTER usuario (username);
```

Y terminamos por crear un índice que marque al cluster creado:

```
CREATE INDEX ind_usuario ON CLUSTER usuario;
```

Al hacer la primera consulta, nos maravillamos del resultado:

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

9 filas seleccionadas.

Transcurrido: 00:00:00.01

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

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    20 | 22680 |   170   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| POSTS |    20 | 22680 |   170   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

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

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


Estadisticas
----------------------------------------------------------
         23  recursive calls
          1  db block gets
        720  consistent gets
          0  physical reads
        184  redo size
       9686  bytes sent via SQL*Net to client
        354  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          9  rows processed
```

Con las estadísticas de las repeticiones siguientes:

```
Estadisticas
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        627  consistent gets
          0  physical reads
          0  redo size
       9686  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
```

Notamos como el coste subió de una manera significativa y quisimos investigar lo relativo al tiempo.

Consultamos 10 veces y obtuvimos los siguientes tiempos:

0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

La mejor noticia fue el mínimo tiempo de respuesta, junto a su constante absoluta. Esto nos dejó con la expectativa de ver cómo funcionaba en las demás consultas.

Sin embargo, en la consulta 2 mejoró muy ligeramente los tiempos, con un coste igual de abultado y ya en las demás consultas se terminó por destruir la expectativa. El coste se mantuvo igual de grande que antes en la consulta 3 y 4, y el tiempo no fue distinto. Nos dejó la única buena noticia de haber mejorado el tiempo de la primera consulta, aunque a costa de empeorar las demás. Nos deja con muchas probabilidades de descarte para las siguientes pruebas.

4 Evaluación

Para normalizar las pruebas y obtener el tiempo medio de ejecución se nos ha facilitado un paquete con varios procedimientos, entre ellos uno que realiza los 5 procesos y finalmente muestra el consumo en tiempo. Para realizar la media del tiempo en los variados tests, los ejecutamos 10 veces y realizamos la división correspondiente. En ningún momento, comparamos los resultados de PCTUSED y PCTFREE porque cuando hicimos las pruebas correspondientes no fueron relevantes en ningún caso, así que optamos por omitirlos.

Diseño Inicial

Este resultado es consistente, y posteriores ejecuciones del test de la carga prototípica dan valores muy cercanos, admisible dentro del margen de error. Independientemente de un número bajo de tests, medio o alto se han mantenido estos valores. Es decir, la media es de 33,515 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 32,1 milliseconds.
CONSISTENT GETS (workload):7063 acc
CONSISTENT GETS (weighted average):706,3 acc

Diseño Inicial con Índice 1

Este resultado es bastante positivo, con el número de lecturas a almacenamiento bajando a la mitad. En cuanto a su media, tuvimos 19,98 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 18,75 milliseconds.
CONSISTENT GETS (workload):3840 acc
CONSISTENT GETS (weighted average):384 acc

Diseño Inicial con Cluster

Este resultado nos pareció bastante pobre, por no decir negativo. El tiempo aumenta considerablemente y no sólo eso, aumenta también el número de lecturas a almacenamiento. Su media nos dejó mucho que desear, con 37,95 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 35,4 milliseconds.
CONSISTENT GETS (workload):7714 acc
CONSISTENT GETS (weighted average):771,4 acc

Diseño Inicial con Índice 1 y Cluster

El resultado mejora al diseño inicial pero empeora al que solo utiliza el índice 1. Notamos como los consistent gets aumentan respecto al diseño del índice 1 y lo mismo pasa con el tiempo. La media que recoge es de 24,335 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 25,05 milliseconds.
CONSISTENT GETS (workload):4453 acc
CONSISTENT GETS (weighted average):445,3 acc

El tema del cluster resultó no ser muy beneficioso en líneas generales. Aunque funcionó a la perfección en la primera consulta, luego en las demás empeora considerablemente el coste y el tiempo, dejándonos con la única opción de prescindir de él.

Diseño Inicial con Tamaño de bloque duplicado (8k -> 16k)

Nos resultaron muy favorables las pruebas, sobre todo por la bajada considerable del número de lecturas, sin necesidad de índices o clusters. El tiempo no se aleja mucho del inicial, pero sigue siendo una mejora con la media de 27,96 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 29 milliseconds.
CONSISTENT GETS (workload):3965 acc
CONSISTENT GETS (weighted average):396,5 acc

Diseño Inicial con Tamaño de bloque duplicado (8k -> 16k) e Índice 1

Este resultado nos alegró demasiado. Los consistent gets bajando a unos mínimos impresionantes y el tiempo haciendo lo mismo. Nos pareció la mejor solución hasta el momento, teniendo una media de 19,87 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 21,1 milliseconds.
CONSISTENT GETS (workload):2210 acc
CONSISTENT GETS (weighted average):221 acc

A pesar de eso, al analizar las consultas específicas de la práctica, el cambio de tamaño de bloque afecta a las mismas. El coste baja como se ve venir, pero el tiempo aumenta significativamente. En las dos primeras consultas el tiempo aumenta ligeramente, pero en la consulta 3 y 4 se da de forma drástica. Llegando a tardar 0.66 segundos y oscilando entre los 0.45 - 0.55 segundos nos deja con una muy mala señal. En el caso de añadir el índice 1, el coste baja pero el tiempo sigue siendo relativamente el mismo. Todo esto nos deja con la única solución de regresar al tamaño de bloque 8k, porque a pesar de que los tests nos den resultados favorables, lo que nos pide la práctica es eficiencia con respecto a las consultas referidas, no eficiencia a nivel general.

Diseño Inicial con Índice 2

Volvemos a tener el tamaño de bloque estándar (8k) y ahora partimos de un resultado similar al diseño original. Se aprecia una bajada ligera del número de lecturas a almacenamiento, pero el tiempo se mantiene igual. Se tiene una media de 33,195 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 32,1 milliseconds.
CONSISTENT GETS (workload):6591 acc
CONSISTENT GETS (weighted average):659,1 acc

Diseño Inicial con Índice 1 e Índice 2

El resultado es el más óptimo posible. Combinando ambos índices, disminuye considerablemente tanto el tiempo como el número de lecturas a almacenamiento. La media es de 20,075 milisegundos por 20 cargas prototípicas.

TIME CONSUMPTION (run): 22,3 milliseconds.
CONSISTENT GETS (workload):2906 acc
CONSISTENT GETS (weighted average):290,6 acc

5 Conclusiones Finales

De entre los resultados obtenidos, el más eficiente ha sido el que usa el modelo inicial (tamaño de bloque 8k) junto a los dos índices. Tiene una mejora significativa en cuanto a tiempo y coste, que son las cosas más importantes a las cuales siempre debemos destinar nuestros recursos. El índice 2 aporta en líneas generales a bajar el coste (sin influir en el tiempo) y el índice 1 influye tanto en tiempo como en coste, disminuyéndolo todo de la mejor manera posible. Lamentablemente, no pudimos dar con una solución óptima del cluster. Sí aportó en la primera consulta, pero afectando a las otras, lo cual nos dejó sin más opción que crear el índice 2 para solventar las 4 consultas que quedaban por optimizar.

Al realizar las pruebas de las consultas, se nos dió muy bien evaluarlas y encontrar una solución. Al igual que el compilador de C crea código máquina mucho más directo y sencillo que lo que programamos, Oracle debe de estar optimizando por debajo y, al ser consciente de las relaciones entre tablas, puede generar un plan muy óptimo. Es por eso que no tuvimos que hacer mucho para idear un mejor plan. Quizá si hubiéramos operado en tablas muy grandes, en las que cueste mucho computacionalmente recorrerlas, se daría más juego a mejoras.

No hemos sido muy fans de sqlplus, pero nos ha ahorrado diversos quebraderos de cabeza como serían instalarnos y ejecutar un servidor y cliente SQL genérico.

Hemos encontrado esta práctica mucho más fácil que la anterior. Aunque por falta de tiempo y acumulación de trabajos de otras asignaturas en esta semana, no trabajamos mucho en encontrar más soluciones (como encontrar un cluster que funcionara adecuadamente).

Hemos estudiado el procesamiento que hace el sistema de Oracle, hemos intentado comprender sus razonamientos y adaptarnos a ellos. También, el resultado de comprensión y aprendizaje sí que lo consideramos provechoso, pues estos conocimientos son muy específicos y adquirirlos de forma autodidacta hubiese requerido de recursos bastante avanzados.

Agradecemos que esta práctica se haya enfocado de esta manera, con procesos sencillos y ligeros. Comprendemos que, como hemos dicho anteriormente, consultas más pesadas hubiesen dado más juego a optimizaciones cuyo resultado fuese más evidente, pero hubiese aumentado mucho la laboriosidad de la práctica. Al final lo que estamos haciendo es intentar ser más perspicaces que el optimizador de Oracle, y que sólo en casos específicos y circunstanciales se obtendrá una mejora considerable.

Respecto a las anteriores prácticas, la primera ha sido de una dificultad adecuada, crear unas tablas e importar datos de otras. En cambio la segunda pide demasiada profundidad. Las tablas son muchas y el vocabulario tan específico en inglés no ha ayudado, pero aceptamos que en esta profesión es necesario prepararse para esta clase de cosas.

Respecto a la asignatura estamos de acuerdo en la separación de las estructuras de bases de datos por un lado y ficheros por otro, pero quizás ambas son demasiado densas y profundas para lo esperado. Quizás enfocarnos más en bases de datos como tal en esta asignatura y dejar ficheros de datos como otra asignatura sería una mejor opción, sobre todo por la falta de tiempo que se necesita para profundizar como se debe en ambos temas.