

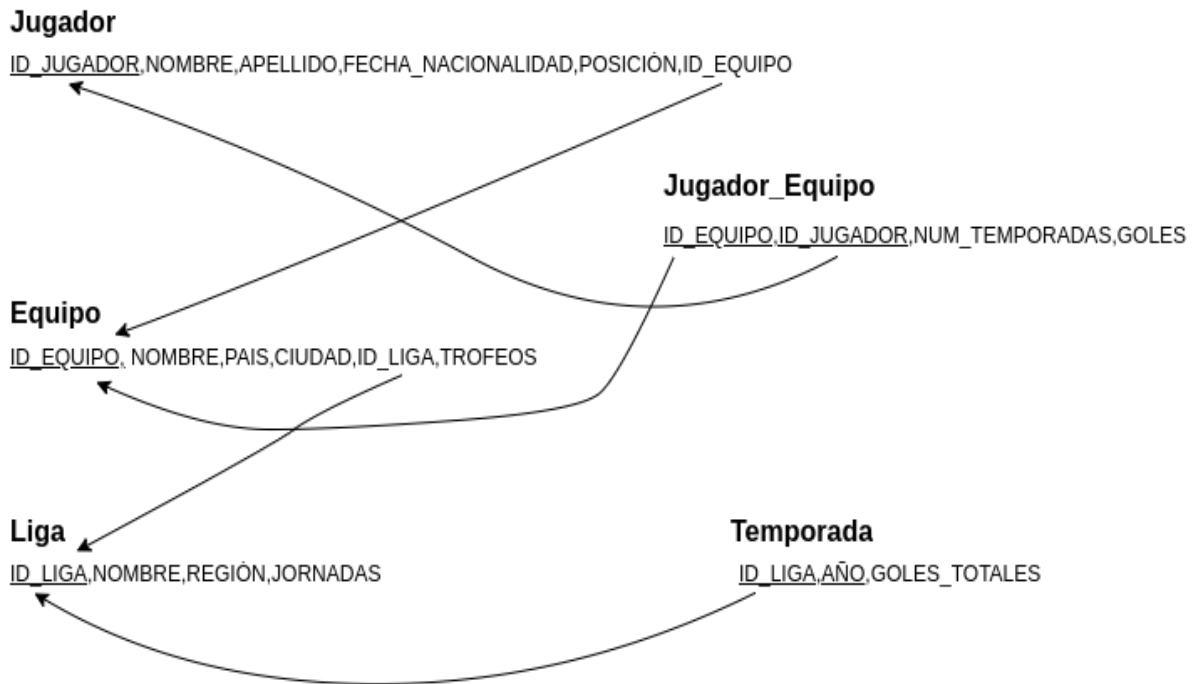
# ADMINISTRACION DE BASE DE DATOS TRABAJO TUTELADO 2



Brais González Piñeiro : [brais.gonzalezp@udc.es](mailto:brais.gonzalezp@udc.es) : 54230875A

Izan Montaos Rodríguez: [izan.montaos@udc.es](mailto:izan.montaos@udc.es) : 46292906R

# ESQUEMA RELACIONAL



## OPTIMIZACIÓN DE CONSULTAS

Cabe destacar primero de todo que para elegir las consultas nos basamos en las consultas más utilizadas para una aplicación de estadísticas de fútbol, con el objetivo de conseguir la mayor cercanía posible a un entorno real. En cuanto a la carga de trabajo, estamos ante una base de datos pensada para usarse en una aplicación de estadísticas deportivas, por lo cual es muy posible que en ciertos momentos la carga de trabajo de esta bd sea muy alta, todo esto dependerá de la cantidad de gente que esté utilizando la aplicación, y por tanto esta carga de trabajo será variable.

-Consulta 1: Esta consulta pretende obtener los datos de un jugador de nuestra base de datos. La consulta sería la siguiente:

```
SELECT J.NOMBRE, J.APELLIDO, J.FECHA_NAC, J.NACIONALIDAD, J.POSICION, E.NOMBRE  
FROM jugador J JOIN EQUIPO e ON J.ID_EQUIPO = E.ID_EQUIPO  
WHERE J.NOMBRE = 'Trude' AND J.APELLIDO = 'Fachini';
```

El nombre del jugador es un mero ejemplo, realmente en el entorno real podría ser cualquier jugador. El resultado de ejecutar la siguiente consulta sería lo siguiente:

ABC NOMBRE	ABC APELLIDO	FECHA NAC	ABC NACIONALIDAD	ABC POSICION	ABC NOMBRE
Trude	Facchini	2002-04-06 00:00:00.000	PL	DEL	RealMadrid

En cuanto a la ejecución en el planificador de ejecución obtenemos los siguientes resultados:

```

SELECT J.NOMBRE,J.APELLIDO,J.FECHA_NAC,J.NACIONALIDAD,J.POSICION,E.NOMBRE FROM JUGADOR J
2 JOIN EQUIPO E ON J.ID_EQUIPO = E.ID_EQUIPO WHERE J.NOMBRE = 'Trude' AND J.APELLIDO = 'Facchini';

```

Plan de Ejecución  
Plan hash value: 1240176962

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1	48	20 (0)
1	NESTED LOOPS		1	48	20 (0)
2	NESTED LOOPS		1	48	20 (0)
* 3	TABLE ACCESS FULL	JUGADOR	1	36	19 (0)
* 4	INDEX UNIQUE SCAN	PK_EQUIPO	1		0 (0)
5	TABLE ACCESS BY INDEX ROWID	EQUIPO	1	12	1 (0)

Como podemos ver, el coste de esta consulta es de 20, esto es debido a que como estamos haciendo una búsqueda por nombre y apellidos, que no son ninguna clave primaria y, por tanto, esta fila a buscar no está indexada, hace que el planificador tenga que recorrer todos los datos de esa tabla hasta encontrar la fila deseada que es lo que produce la mayor carga de trabajo en esta consulta.

Para la optimización de esta primera consulta la mejor solución sería crear un índice para el campo Nombre y Apellido ya que de esta forma la búsqueda en este caso sería muchísimo más rápida por no decir instantánea ya que ahora accedemos directamente a la fila deseada sin tener que ir fila por fila. Para probarlo primero creamos el índice con:

***CREATE INDEX inidiceJugador ON JUGADOR (NOMBRE,APELLIDO);***

Una vez creado observamos en el planificador el coste ganado.

```

SELECT J.NOMBRE,J.APELLIDO,J.FECHA_NAC,J.NACIONALIDAD,J.POSICION,E.NOMBRE
  2   FROM jugador J JOIN EQUIPO e ON J.ID_EQUIPO = E.ID_EQUIPO
  3   WHERE J.NOMBRE = 'Trude' AND J.APELLIDO = 'Fachini';
Transcurrido: 00:00:00.00

Plan de Ejecución
-----
Plan hash value: 320869521

-----
--
| Id | Operation                      | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
|----|-----|-----|-----|-----|-----|-----|
--
|  0 | SELECT STATEMENT                |                     |      1 |    48 |     3  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+-----+
|  1 |   NESTED LOOPS                  |                     |      1 |    48 |     3  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
|  2 |     NESTED LOOPS                |                     |      1 |    48 |     3  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
|  3 |      TABLE ACCESS BY INDEX ROWID BATCHED | JUGADOR             |      1 |    36 |     2  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
|*  4 |         INDEX RANGE SCAN        | INDICEJUGADOR       |      1 |      |     1  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
|*  5 |         INDEX UNIQUE SCAN       | PK_EQUIPO           |      1 |      |     0  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
|  6 |      TABLE ACCESS BY INDEX ROWID | EQUIPO              |      1 |    12 |     1  (0)| 00:00:01 |
+----+-----+-----+-----+-----+-----+
--
--

Predicate Information (identified by operation id):
-----
  4 - access("J"."NOMBRE"='Trude' AND "J"."APELLIDO"='Fachini')
  5 - access("J"."ID_EQUIPO"="E"."ID_EQUIPO")

```

Como podemos observar el coste se reduce se reduce muchísimo de 20 a 3, esto gracias a que ahora la búsqueda en la tabla es instantánea al haber creado el índice nuevo.

-2º Consulta: Esta consulta obtiene toda la plantilla de un equipo determinado. La consulta sería la siguiente:

```

SELECT J.NOMBRE,J.APELLIDO,E.NOMBRE FROM JUGADOR J JOIN EQUIPO E
  ON E.ID_EQUIPO = J.ID_EQUIPO WHERE E.NOMBRE = 'RealMadrid';

```

El nombre del equipo es un mero ejemplo, realmente en el entorno real podría ser cualquier equipo. El resultado de ejecutar la siguiente consulta sería lo siguiente:

	ABC NOMBRE	ABC APELLIDO	ABC NOMBRE
1	Doretta	Stansell	RealMadrid
2	Meredeth	Kuller	RealMadrid
3	Mellisa	Vedenyapin	RealMadrid
4	Sophey	Scholler	RealMadrid
5	Trude	Facchini	RealMadrid
6	Tina	Goodlad	RealMadrid
7	Carey	Clendening	RealMadrid
8	Aurlie	Kelshaw	RealMadrid
9	Rozelle	Neward	RealMadrid
10	Mollie	Rotham	RealMadrid

En cuanto a la ejecución del planificador podemos observar lo siguiente:

```

SQL> SELECT J.NOMBRE,J.APELLIDO,E.NOMBRE FROM JUGADOR J JOIN EQUIPO E
2      ON E.ID_EQUIPO = J.ID_EQUIPO WHERE E.NOMBRE = 'RealMadrid';

Plan de Ejecución
-----
Plan hash value: 3863127603

-----
| Id | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |         |     10 |    330 |      22  (0)| 00:00:01 |
|*  1 |  HASH JOIN         |         |     10 |    330 |      22  (0)| 00:00:01 |
|*  2 |    TABLE ACCESS FULL| EQUIPO  |        1 |     12 |        3  (0)| 00:00:01 |
|  3 |    TABLE ACCESS FULL| JUGADOR |    10000 |   205K |       19  (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
   1 - access("E"."ID_EQUIPO"="J"."ID_EQUIPO")
   2 - filter("E"."NOMBRE"='RealMadrid')

```

Cabe destacar que el índice previamente creado no fue eliminado para esta consulta, sin embargo, tampoco tiene ningún efecto ya que en esta consulta no hacemos un filtrado por apellido y nombre en la tabla jugador.

Por lo que podemos observar el planificador el coste de la consulta es de 22, esto debido a dos accesos completos a dos tablas, por un lado, la de jugadores y por otro la de equipo, aunque cabe destacar que al estar usando un equipo que está al principio de la tabla el coste es relativamente bajo.

Ahora probamos a usar un hint para esta consulta en este caso un use merge para comprobar si un merge fuera más eficiente en esta búsqueda que un hash.

```

SELECT /*+ USE_MERGE(E J) */ J.NOMBRE,J.APELLIDO,E.NOMBRE FROM
JUGADOR J JOIN EQUIPO E ON E.ID_EQUIPO = J.ID_EQUIPO WHERE
E.NOMBRE = 'RealMadrid';

```

La salida del planificador sería:

```
SQL> SELECT /*+ USE_MERGE(E J) */J.NOMBRE,J.APELLIDO,E.NOMBRE FROM JUGADOR J JOIN EQUIPO E
  2   ON E.ID_EQUIPO = J.ID_EQUIPO WHERE E.NOMBRE = 'RealMadrid';
Transcurrido: 00:00:00.01

Plan de Ejecución
-----
Plan hash value: 292743031

-----
| Id | Operation                                | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT                        |           |     10 |   330 |    22  (5)| 00:00:01 |
|  1 | MERGE JOIN                             |           |     10 |   330 |    22  (5)| 00:00:01 |
|*  2 | TABLE ACCESS BY INDEX ROWID          | EQUIPO    |       1 |    12 |     2  (0)| 00:00:01 |
|  3 | INDEX FULL SCAN                        | PK_EQUIPO |    100 |       |     1  (0)| 00:00:01 |
|*  4 | SORT JOIN                             |           |   10000 |  205K |    20  (5)| 00:00:01 |
|  5 | TABLE ACCESS FULL                    | JUGADOR   |   10000 |  205K |    19  (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
  2 - filter("E"."NOMBRE"='RealMadrid')
  4 - access("E"."ID_EQUIPO"="J"."ID_EQUIPO")
    filter("E"."ID_EQUIPO"="J"."ID_EQUIPO")
```

Como podemos observar ahora el planificador usa un merge join en lugar de un hash join, sin embargo, a pesar de cambiar la planificación el coste ,como podemos ver, es el mismo ya que le cuesta lo mismo, en el hash hacer un acceso total a la tabla equipo, que en el caso del merge, primero acceder por el índice de la clave primaria y luego filtrar cual es el índice que quiero.

En cuanto a la optimización de esta segunda consulta, hay dos índices a crear que podrían ayudar a bajar el coste, por un lado, podemos crear un índice en la tabla jugadores para el valor ID\_LIGA lo que hará que la búsqueda sea mucho más rápida al no tener que consultar todos los valores de la tabla liga, los cuales no nos interesan y por otro lado podemos crear otro índice en la tabla equipo para el campo NOMBRE y así acceder directamente a las filas de liga donde el nombre es el que buscamos.

Para crear los índices haríamos:

```
CREATE INDEX indiceEquipo on EQUIPO (NOMBRE);
```

```
CREATE INDEX indiceJugador2 on Jugador (ID_EQUIPO);
```

Ejecutamos otra vez el planificador para ver si los resultados fueron los previstos y efectivamente hemos sido capaces de reducir el tiempo de ejecución de la consulta. El resultado sería el siguiente:

```

SQL> SELECT J.NOMBRE,J.APELLIDO,E.NOMBRE FROM JUGADOR J JOIN EQUIPO E
2      ON E.ID_EQUIPO = J.ID_EQUIPO WHERE E.NOMBRE = 'RealMadrid';
Transcurrido: 00:00:00.01

Plan de Ejecución
-----
Plan hash value: 2604888919

-----
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
|----|-----|-----|-----|-----|-----|-----|
-----
| 0 | SELECT STATEMENT | | 10 | 330 | 13 (0)| 00:00:01 |
| 1 | NESTED LOOPS | | 10 | 330 | 13 (0)| 00:00:01 |
| 2 | NESTED LOOPS | | 10 | 330 | 13 (0)| 00:00:01 |
| 3 | TABLE ACCESS BY INDEX ROWID BATCHED | EQUIPO | 1 | 12 | 2 (0)| 00:00:01 |
|* 4 | INDEX RANGE SCAN | INDICEEQUIPO | 1 | | 1 (0)| 00:00:01 |
|* 5 | INDEX RANGE SCAN | INDICEJUGADOR2 | 10 | | 1 (0)| 00:00:01 |
| 6 | TABLE ACCESS BY INDEX ROWID | JUGADOR | 10 | 210 | 11 (0)| 00:00:01 |
-----
-----

```

Como podemos ver hemos sido capaces de reducir el coste de 22 a 13 por lo que hemos reducido la mitad. Si nos fijamos podemos ver que el planificador cambia radicalmente el plan de ejecución usando ahora bucles anidados. Respectivamente estamos ganando un coste de 8 en el acceso a la tabla Jugadores y un coste de 1 en la tabla equipo, y es que el planificador antes de crear los índices hace 1 acceso completo a cada tabla para realizar un hash join con ambas tablas y ahora usamos bucles anidados para hacer los joins correspondientes, sin embargo, no sobre toda la tabla si no sobre los valores buscados obtenidos directamente gracias a los índices creados.

-Consulta 3: Obtener el número de temporadas y de goles que un jugador ha hecho en un equipo. La consulta sería la siguiente:

```
SELECT J.NOMBRE,J.APELLIDO,E.EQUIPO,D.NUM_TEMPORADAS,D.GOLES
```

```
FROM JUGADOR J JOIN JUGADOR_EQUIPO D ON J.ID_JUGADOR =
D.ID_JUGADOR
```

```
JOIN E.EQUIPO ON D.ID_EQUIPO = E.ID_EQUIPO
```

```
WHERE E.NOMBRE = 'RealMadrid' AND J.NOMBRE = 'Iago' AND J.APELLIDO =
'Dundin'
```

Para esta consulta obtenemos la siguiente salida:

ABC NOMBRE ▼	ABC APELLIDO ▼	ABC NOMBRE ▼	123 NUM TEMPORADAS ▼	123 GOLES ▼
Iago	Dundin	RealMadrid	9	123

En cuanto a la ejecución en el planificador de ejecución obtenemos los siguientes resultados:

```
SQL> SELECT J.NOMBRE,J.APELLIDO,E.NOMBRE,D.NUM_TEMPORADAS,D.GOLES
2  FROM JUGADOR J JOIN JUGADOR_EQUIPO D ON J.ID_JUGADOR = D.ID_JUGADOR
3  JOIN EQUIPO E ON D.ID_EQUIPO = E.ID_EQUIPO
4  WHERE E.NOMBRE = 'RealMadrid' AND J.NOMBRE = 'Iago' AND J.APELLIDO = 'Dundin';
```

Plan de Ejecución  
-----  
Plan hash value: 3805892751  
-----

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	T
0	SELECT STATEMENT		1	47	4 (0)	0
1	NESTED LOOPS		1	47	4 (0)	0
2	NESTED LOOPS		1	47	4 (0)	0
3	MERGE JOIN CARTESIAN		1	33	3 (0)	0
4	TABLE ACCESS BY INDEX ROWID BATCHED	JUGADOR	1	21	2 (0)	0
* 5	INDEX RANGE SCAN	INDICEJUGADOR	1		1 (0)	0
6	BUFFER SORT		1	12	1 (0)	0
7	TABLE ACCESS BY INDEX ROWID BATCHED	EQUIPO	1	12	1 (0)	0
* 8	INDEX RANGE SCAN	INDICEEQUIPO	1		0 (0)	0
* 9	INDEX UNIQUE SCAN	PK_JUGADOR_EQUIPO	1		0 (0)	0
10	TABLE ACCESS BY INDEX ROWID	JUGADOR_EQUIPO	1	14	1 (0)	0

En este caso el plan de ejecución nos da un coste muy bajo de tan solo 5, esto se debe a la creación de índices que venimos haciendo desde la primera consulta, lo que hace que los accesos a estas tablas sean prácticamente instantáneos. Por lo que podemos concluir que en este caso no podemos crear índices ya que los índices que necesitaríamos para esta práctica ya están creados. Podríamos crear un índice para la tabla JUGADOR\_EQUIPO pero



dado a que los dos campos por los que se hacen los joins son claves primarias de la propia tabla, estos ya están indexados directamente sin tener que crear un índice.

Consulta 4: Obtener las estadísticas de una liga en los últimos 10 años. La resolución de esta consulta sería la siguiente:

```
SELECT L.NOMBRE,L.REGION,L.JORNADAS,T.AÑO,T.GOLESTOTALES
FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
WHERE L.NOMBRE = 'LIGA1A';
```

Para esta consulta obtenemos la siguiente salida:

ABC NOMBRE ▼	ABC REGION ▼	123 JORNADAS ▼	123 AÑO ▼	123 GOLES TOTALES ▼
Liga1A	Spain	27	2.020	641
Liga1A	Spain	27	2.019	316
Liga1A	Spain	27	2.018	760
Liga1A	Spain	27	2.017	740
Liga1A	Spain	27	2.016	724
Liga1A	Spain	27	2.015	671
Liga1A	Spain	27	2.014	508
Liga1A	Spain	27	2.013	307
Liga1A	Spain	27	2.012	590
Liga1A	Spain	27	2.011	573

En cuanto al planificador obtenemos el siguiente resultado:

```
SQL> SELECT L.NOMBRE,L.REGION,L.JORNADAS,T.AÑO,T.GOLES_TOTALES
2 FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
3 WHERE L.NOMBRE = 'LIGA1A';
```

#### Plan de Ejecución

Plan hash value: 3482153716

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		10	320	6 (0)	00:00:01
* 1	HASH JOIN		10	320	6 (0)	00:00:01
* 2	TABLE ACCESS FULL	LIGA	1	21	3 (0)	00:00:01
3	TABLE ACCESS FULL	TEMPORADA	1000	11000	3 (0)	00:00:01

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

- 1 - access("L"."ID\_LIGA"="T"."ID\_LIGA")
- 2 - filter("L"."NOMBRE"='LIGA1A')

En este caso podemos observar que los costes tampoco son demasiado altos y el plan de ejecución es el mismo que en la consulta 2. De todas formas, podemos optimizar la consulta, aunque la mejora no será demasiado sustancial. Para ellos decidimos crear un índice en la tabla LIGA, de esta forma en vez de hacer un acceso completo a la tabla liga podemos realizar un acceso a través del índice ahorrándonos consultar muchas filas que no nos interesan. Para crear el índice hacemos:

*CREATE INDEX indiceLiga on LIGA(NOMBRE);*

Después de crear el índice procedemos a ejecutar de nuevo la consulta en el planificador para ver los resultados:

```
SQL> SELECT L.NOMBRE,L.REGION,L.JORNADAS,T.AÑO,T.GOLES_TOTALES
2 FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
3 WHERE L.NOMBRE = 'LIGA9A';
```

#### Plan de Ejecución

Plan hash value: 1800424202

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		10	320	5 (0)	00:00:01
* 1	HASH JOIN		10	320	5 (0)	00:00:01
2	TABLE ACCESS BY INDEX ROWID BATCHED	LIGA	1	21	2 (0)	00:00:01
* 3	INDEX RANGE SCAN	INDICELIGA	1		1 (0)	00:00:01
4	TABLE ACCESS FULL	TEMPORADA	1000	11000	3 (0)	00:00:01

Como podemos observar logramos bajar en 1 el coste de la consulta ya que como hemos dicho la búsqueda de la liga ya no se hace en toda la tabla si no que al crear un índice para la tabla es instantáneo, esto se ve en que antes había que hacer un TABLE FULL ACCESS para la liga sin embargo ahora el acceso lo hacemos por el índice que acabamos de crear como podemos ver en el INDEX RANGE SCAN y el TABLE ACCES BY INDEX.

-Consulta 5: Obtener el número total de goles marcados en una liga a lo largo de toda su historia. La resolución de esta consulta sería la siguiente:

```
SELECT L.NOMBRE, SUM(T.GOLES_TOTALES)
FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
GROUP BY L.NOMBRE
ORDER BY L.NOMBRE;
```

En cuanto al resultado de esta consulta obtenemos:

ABC NOMBRE	123 SUM(T.GOLES TOTAL
Liga1A	5.830
Liga1B	5.835
Liga1C	5.973
Liga1D	5.522
Liga1F	5.022
Liga1G	5.674
Liga1H	5.912
Liga1I	5.813
Liga1J	4.915
Liga10A	5.516
Liga10B	4.676
Liga10C	4.631
Liga10D	5.381
Liga10E	5.090
Liga10F	5.394
Liga10G	4.786
Liga10H	5.871
Liga10I	5.682
Liga10J	5.603
Liga2A	5.207
Liga2B	5.632
Liga2C	5.836
Liga2D	6.358
Liga2E	6.067
Liga2F	5.580
Liga2G	5.227
Liga2H	5.015
Liga2I	5.399
Liga2J	5.024
Liga2K	6.007
Liga3A	5.933
Liga3B	5.403
Liga3C	5.568

En cuanto al plan de ejecución tenemos:

```

SELECT L.NOMBRE, SUM(T.GOLES_TOTALES)
  2 FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
  3 GROUP BY L.NOMBRE
  4 ORDER BY L.NOMBRE;

```

Plan de Ejecución

Plan hash value: 1272565627

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		100	1800	7 (29)	00:00:01
1	SORT ORDER BY		100	1800	7 (29)	00:00:01
2	HASH GROUP BY		100	1800	7 (29)	00:00:01
* 3	HASH JOIN		1000	18000	5 (0)	00:00:01
4	VIEW	index\$_join\$_001	100	1100	2 (0)	00:00:01
* 5	HASH JOIN					
6	INDEX FAST FULL SCAN	INDICELIGA	100	1100	1 (0)	00:00:01
7	INDEX FAST FULL SCAN	PK_LIGA	100	1100	1 (0)	00:00:01
8	TABLE ACCESS FULL	TEMPORADA	1000	7000	3 (0)	00:00:01

Predicate Information (identified by operation id):

```

  3 - access("L"."ID_LIGA"="T"."ID_LIGA")
  5 - access(ROWID=ROWID)

```

En este caso podemos ver que el coste tampoco es demasiado grande en parte porque las tablas no son muy grandes. Analizando la información que nos da el planificador podemos observar cómo primero se accede de forma completa a la tabla temporada, y luego de forma rápida a través de los índices a la tabla de liga. Una vez obtenido los datos, el planificador procede a hacer el join usando el algoritmo hash que seguidamente también utilizará para hacer el group by y finalmente ordenarlos según lo especificado en el order by.

Procedemos ahora a usar un hint para ver si tiene algún efecto en la planificación:

```

SELECT /*+ NO_INDEX(L) */ L.NOMBRE, SUM(T.GOLES_TOTALES)
  2 FROM LIGA L JOIN TEMPORADA T ON L.ID_LIGA = T.ID_LIGA
  3 GROUP BY L.NOMBRE
  4 ORDER BY L.NOMBRE;

Plan de Ejecución
-----
Plan hash value: 590935977

-----
--
| Id  | Operation                      | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
|-----|-----|-----|-----|-----|-----|-----|
--
|  0  | SELECT STATEMENT                |           |    100 | 18000 |    8  (25)| 00:00:01 |
|  1  | SORT ORDER BY                   |           |    100 | 18000 |    8  (25)| 00:00:01 |
|  2  | HASH GROUP BY                   |           |    100 | 18000 |    8  (25)| 00:00:01 |
|* 3  | HASH JOIN                       |           |   1000 | 18000 |    6   (0)| 00:00:01 |
|  4  | TABLE ACCESS FULL              | LIGA      |    100 | 11000 |    3   (0)| 00:00:01 |
|  5  | TABLE ACCESS FULL              | TEMPORADA |   1000 | 70000 |    3   (0)| 00:00:01 |
-----
--

```

En este caso usamos el hint `NO_INDEX` para indicarle al planificador que no use ningún índice para acceder a la tabla `liga`, esto con el objetivo de observar cómo al no acceder a la tabla `liga` con el índice está aumentando el coste de la consulta. Podemos observar como ahora en vez de hacer dos accesos rápidos con 1 de coste a la tabla `liga` tenemos un único acceso a toda la tabla ya que esta no se realiza con índices que hace que tengamos un coste de 3 aumentando así 1 de coste en nuestro contador total.

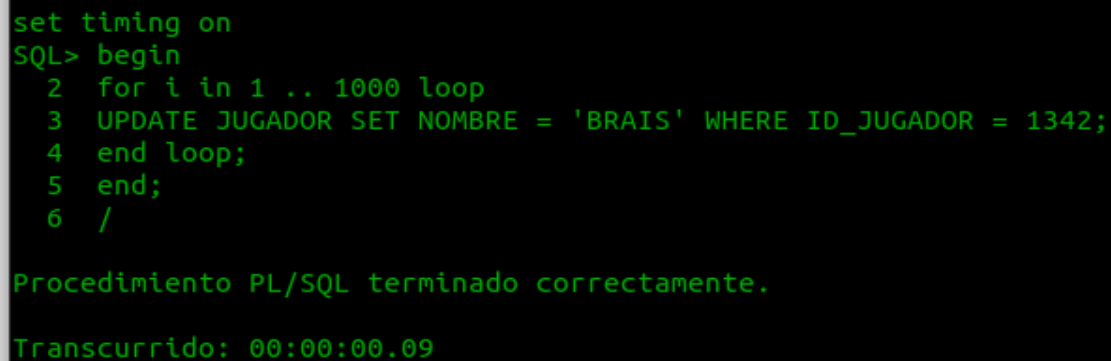
En cuanto a la optimización de esta consulta, podemos ver que realmente la consulta ya está optimizada ya que los índices necesarios para ello ya están creados.

-Consulta DML: Como bien sabemos crear índices puede tener un impacto negativo en la eficiencia de nuestra base de datos a la hora de hacer consultas UML, esto ocurre ya que cuando se insertan nuevos registros en una tabla con índices, se debe actualizar también los índices correspondientes para reflejar los cambios, lo que implica realizar operaciones adicionales de escritura y actualización en los índices, lo que puede llevar más tiempo en comparación con una inserción en una tabla sin índices.

Por ello elegimos una consulta DML de actualización de una tabla a la que mediremos el tiempo de ejecución para de esta forma ver de qué forma se ve afectado la creación de varios índices en nuestra base de datos. Para ello utilizaremos el siguiente script:

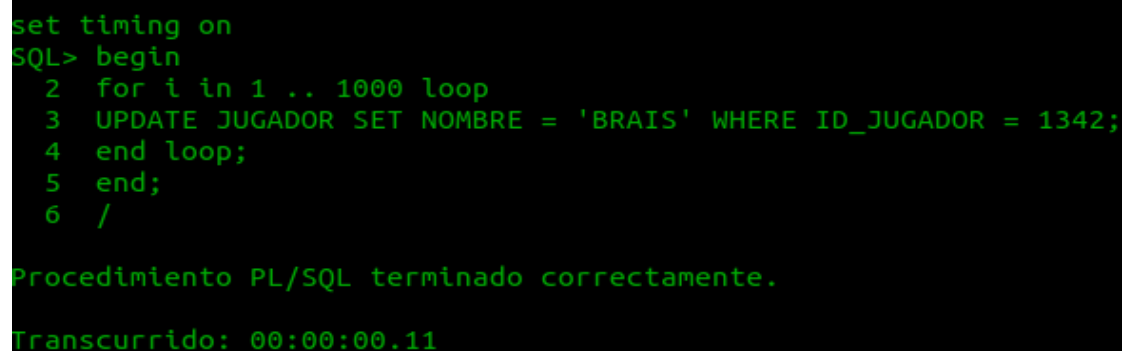
```
set timing on  
  
begin  
  
for i in 1 .. 1000 loop  
  
UPDATE JUGADOR SET NOMBRE = 'BRAIS' WHERE ID_JUGADOR = 1342;  
  
end loop;  
  
end;  
  
/
```

A continuación, una captura con el tiempo de ejecución antes de la creación de índices (la captura fue tomada al iniciar la optimización de la base de datos):



```
set timing on  
SQL> begin  
  2  for i in 1 .. 1000 loop  
  3  UPDATE JUGADOR SET NOMBRE = 'BRAIS' WHERE ID_JUGADOR = 1342;  
  4  end loop;  
  5  end;  
  6  /  
  
Procedimiento PL/SQL terminado correctamente.  
Transcurrido: 00:00:00.09
```

Ahora otra captura después de haber hecho la optimización y haber creado los índices:



```
set timing on  
SQL> begin  
  2  for i in 1 .. 1000 loop  
  3  UPDATE JUGADOR SET NOMBRE = 'BRAIS' WHERE ID_JUGADOR = 1342;  
  4  end loop;  
  5  end;  
  6  /  
  
Procedimiento PL/SQL terminado correctamente.  
Transcurrido: 00:00:00.11
```

Como podemos observar sí que es verdad que el tiempo a la hora de hacer una actualización en nuestra base de datos ha aumentado, sin embargo, solo en 2 centésimas lo que es un tiempo absurdo esto debido a que por un lado nuestras tablas no tienen un tamaño lo suficientemente grande, y por otro lado que tan solo hemos creado unos cuatro índices en total, por lo que realmente como se ha podido ver el tiempo de más no es demasiado relevante.

Con esto, podemos concluir que no sería necesario eliminar ninguno de los índices creados anteriormente ya que no tienen por el momento ningún impacto realmente considerable en

a la hora de hacer operaciones DML en nuestra base de datos, no obstante, en caso de que aumentara o bien el tamaño de la base de datos o el número de índices estaría bien realizar otra comprobación.

## OTRAS POSIBLES OPTIMIZACIONES

En este apartado comentaremos el uso de otras técnicas de optimización y justificaremos si su uso fuera relevante o no en nuestra base de datos:

-Vistas: En cuanto a las vistas es posible que en alguna base de datos fuera interesante su implementación a la hora de realizar ciertas consultas, sin embargo, en nuestro caso creemos que no sería factible. Esto se debe principalmente a que nuestras consultas se centran la mayor parte de ellas en obtener datos específicos de una fila o de un campo de una tabla en concreto, como podría ser obtener un jugador determinado, esto hace que la creación de vistas no sea una solución factible ya que crear una vista para cada jugador sería un consumo de recursos enorme.

-Partición de tablas: Si bien en un entorno donde estemos trabajando con tablas con un gran número de filas esta práctica podría mejorar el rendimiento de la base de datos, en nuestro caso no lo consideramos ya que nuestras tablas son de un tamaño más bien pequeño y, además, no está pensada para hacer consultas masivas de DML.