CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

EJEMPLOS USANDO BD “CICLISME”

Bases de Datos
CFGS DAW

Pau Miñana Climent
2020/2021

EJEMPLOS USANDO LA BD "CICLISME" (Script en UD6)

CONSULTAS

1. Realiza una consulta que calcule la media de edad de los equipos:

```
SELECT equip,AVG(edat) FROM ciclistes GROUP BY equip;
```

2. Nombre,equipo i edad de los ciclistas con más edad que la media de su equipo (ordenados por equipo):

```
SELECT nom,equip,edat FROM ciclistes a
WHERE edat>(SELECT AVG(edat) FROM ciclistes b GROUP BY b.equip
            HAVING b.equip=a.equip) ORDER BY equip;
```

3. Equipo que más etapas ha ganado y cuántas son:

Primero un intermedio para entender la subconsulta, numero de etapas ganadas por cada equipo

```
SELECT COUNT(equip) as N_etapes, equip FROM etapes, ciclistes
WHERE ciclista=dorsal GROUP BY equip ORDER BY N_etapes DESC;
```

Ahora a la consulta que nos han pedido:

```
SELECT COUNT(equip) as N_etapes, equip FROM etapes, ciclistes
WHERE ciclista=dorsal GROUP BY equip
HAVING COUNT(equip)>=ALL
    (SELECT COUNT(equip) as N_etapes FROM etapes, ciclistes
     WHERE ciclista=dorsal GROUP BY equip);
```

Cuidado con la tendencia a abusar de estas subconsultas, por ejemplo, mucha gente tiende a ofrecer la siguiente solución, donde la consulta principal sólo extrae datos de la tabla de la subconsulta que se podían sacar con el HAVING anterior, con lo que es algo redundante:

```
Select N_etapes,equip FROM (SELECT COUNT(equip) as N_etapes, equip
                            FROM etapes, ciclistes
                            WHERE ciclista=dorsal GROUP BY equip) sub1
WHERE N_etapes>=ALL(SELECT COUNT(equip) as N_etapes FROM etapes, ciclistes
                    WHERE ciclista=dorsal GROUP BY equip);
```

Si este tipo de consultas os marean demasiado usar una vista para la subconsulta puede facilitar las cosas, aunque es cierto que con la vista se incurre en la redundancia de subconsultas que se ha comentado, así que es menos eficiente (en oracle necesitaréis dar permiso para crear vistas primero, no se incluye en CONNECT y RESOURCE que es lo que solemos dar en permisos):

```
CREATE VIEW etapes_equip AS SELECT COUNT(equip) as N_etapes, equip
FROM etapes, ciclistes WHERE ciclista=dorsal GROUP BY equip ORDER BY N_etapes DESC;

SELECT N_etapes,equip FROM etapes_equip
WHERE N_etapes>=ALL (SELECT N_etapes FROM etapes_equip);
```

Otra opción para los que prefieran MAX en vez de >=ALL:

```
SELECT N_etapes,equip FROM etapes_equip
WHERE N_etapes= (SELECT MAX(N_etapes) FROM etapes_equip);
```

4. Ciclista de cada equipo que ha ganado más etapas y cuántas son…

Como paso intermedio esta vez etapas ganadas agrupadas por corredor y equipo:

```
SELECT COUNT(nom) as N_etapes, nom,equip FROM etapes, ciclistes
WHERE ciclista=dorsal GROUP BY nom,equip;
```

Pero esta vez necesitamos no el máximo absoluto, sinó los máximos de cada equipo, luego en el HAVING necesitaremos comparar a la vez el N_etapes y el equipo con los máximos agrupados por cada equipo es decir que nuestra consulta queda:

```
SELECT COUNT(nom) as N_etapes, nom,equip FROM etapes, ciclistes
WHERE ciclista=dorsal GROUP BY nom,equip
HAVING (COUNT(nom),equip)= ANY
    (SELECT MAX(N_etapes),equip FROM
        (SELECT COUNT(nom) as N_etapes,equip FROM etapes, ciclistes
        WHERE ciclista=dorsal GROUP BY nom,equip) GROUP BY equip)
ORDER BY EQUIP;
```

Es decir que se compara a la vez el conjunto (COUNT(nom),equip) y puesto que el resultado de la subconsúlta es múltiple (al menos uno por cada equipo que haya ganado carreras, no el máximo absoluto como antes) la comparación necesita un ANY.

Obviamente en equipos donde el máximo número de etapas ganadas sea compartido por varios ciclistas, deben aparecer todos ellos. Se ordena el resultado por equipos para poder observarlo con claridad.

5. Ciclistas que han ganado una etapa cuando alguien de su equipo llevaba algún mallot (no ellos mismos) y qué etapa es:

Para ofrecer la información completa se añade el equipo y el nombre y mallot del compañero, con lo que para algunas etapas sale un corredor varias veces al tener varios compañeros añlgún mallot.

Se pueden adoptar 2 puntos de vista aquí, el primero es más directo y realiza una búsqueda más eficiente y directa. Primero vemos que tablas contienen la información que necesitamos, creamos las condiciones para que se unan y ponemos las condiciones para restringir los resultados a los deseados. En este caso necesitamos etapas y ciclistas (para conocer a los ganadores de etapa y sus nombres) y ciclistas,mallots y portar para conocer el color dell mallot, nombre y equipo de los "portadores" de los mallots. Se debe tener en cuenta que se necesitan 2 copias de la tabla "Ciclistes", para poder buscar a los "compañeros de equipo del ganador" :

```
SELECT e.numero as etapa, c1.nom as ganador, c2.nom as portador,
m.color as mallot, c1.equip
FROM etapes e,ciclistes c1,ciclistes c2,portar p, mallots m
WHERE e.ciclista=c1.dorsal
AND p.etapa=e.numero AND c2.dorsal=p.ciclista AND m.codi=p.mallot
AND c1.equip=c2.equip AND c1.dorsal<> c2.dorsal
ORDER BY e.numero;
```

Aunque parezca larga es porqué se han organizado por filas las condiciones en el WHERE.

La primera une etapas y ciclistas1 para restringir los ganadores de etapa; En la segunda se unen las condiciones para unir portar,etapas,ciclistas2 y mallots, añadiendo así a los portadores de mallots; En la tercera se añaden las condiciones que pide el enunciado, que los ciclistas que portan mallots (c2) sean compañeros de equipo de los ganadores de etapa (c1) y que no sea el mismo ciclista.

En cambio el segundo punto de vista, usando subconsultas, puede resultar más fácil de ver y comprender para ciertas personas, al separar cada parte en subconsultas. Es decir, se busca por un lado la lista de ciclistas ganadores de etapa en una subconsulta, por el otro la lista de portadores de mallots en otra, y se usan estas 2 subconsultas como tablas para unir los resultado Con esto se evita el "problema" de tener que llamar 2 veces a la tabla ciclistas usando alias, ya que cada una queda en una subconsulta. Como defectos, esta consulta es un poco menos eficiente, pues primero hace las 2 listas completas (ganadores y portadores de mallots), luego las une y luego restringe los resultados mostrados y queda más larga de escribir, aunque esto segundo es poco relevante. Por pasos, consulta para ver los ganadores de las etapas (se pone el equipo para unirlas después):

```
SELECT numero as etapa, nom as ganador, equip FROM etapes, ciclistes
WHERE ciclista=dorsal;
```

Consulta para ver los ciclistas que portan mallots en cada etapa (se llama a los campos etapa2 y equip2 para no confundir los nombres con los anteriores al unir las consultas, aunque se podría dar nombre a cada subconsulta y entonces sí podrían coincidir):

```
SELECT etapa as etapa2, nom as portador, color as mallot, equip as equip2
FROM ciclistes,portar,mallots WHERE dorsal=ciclista and codi=mallot
```

Con esto, usando en el FROM las consultas anteriores y uniendo que la etapa y equipo debe ser la misma y portador y ganador no deben coincidir, queda la consulta definitiva:

```
SELECT etapa,ganador,portador,mallot,equip FROM
    (SELECT numero as etapa, nom as ganador, equip FROM etapes, ciclistes
     WHERE ciclista=dorsal),
    (SELECT etapa as etapa2, nom as portador, color as mallot, equip as equip2
     FROM ciclistes,portar,mallots WHERE dorsal=ciclista and codi=mallot)
WHERE etapa=etapa2 AND equip=equip2 AND portador<>ganador
ORDER BY etapa;
```

PL/SQL

Para este apartado vamos a suponer que queremos implementar una Clasificación de la Montaña que se gestione de forma automática para esta BD. En la tabla Ports, se marca el ganador de cada puerto, su categoría y la etapa en que aparece; para los entendidos en ciclismo, con esto no se puede hacer una clasificación real pues sólo conocemos al primero en superar el puerto, así que simplificando, se supone que sólo el ganador de cada puerto puntúa y se suman 10 puntos por cada puerto de categoría especial, 7 puntos por los de primera y 5 por los de segunda. La clasificación de la montaña obviamente es la suma de las puntuaciones de los ciclistas en los puertos.

1. Crear una función que devuelva el número de puntos ganados según el puerto.

```
CREATE OR REPLACE FUNCTION puntos (port IN VARCHAR2)
RETURN NUMBER
IS
cat ports.categoria%TYPE:='0';
BEGIN
    SELECT categoria INTO cat FROM ports WHERE nom=port;
    CASE cat
        WHEN 'E' THEN
            RETURN 10;
        WHEN '1' THEN
            RETURN 7;
        WHEN '2' THEN
            RETURN 5;
        ELSE
            DBMS_OUTPUT.PUT_LINE('La categoria '||cat||' no es correcta.');
            RETURN 0;
    END CASE;
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('El puerto '||port||' no es correcto');
        RETURN 0;
END;
/
```

2. Mostrar mediante PL/SQL la clasificación de la montaña con la ayuda de la función anterior.

Nos basta con un bucle de cursor y una consulta que seleccione los corredores y la suma de sus puntos en cada montaña usando la función anterior y agrupando por ciclista. Se ordena de forma descendente para que el que más puntos tenga sea el primero…

```
BEGIN
        DBMS_OUTPUT.PUT_LINE('CLASSIFICACIO DE LA MUNTANYA');
        FOR i IN (SELECT p.ciclista,c.nom,SUM(puntos(p.nom)) as puntos
                    FROM ports p,ciclistes c
                    WHERE p.ciclista=c.dorsal GROUP BY p.ciclista,c.nom
                    ORDER BY puntos DESC)
        LOOP
                DBMS_OUTPUT.PUT_LINE(i.nom||': '||i.puntos);
        END LOOP;
    END;
    /
```

3. Guardar en una tabla auxiliar la clasificación de la montaña mediante un procedimiento.

Para empezar hay que crear la tabla. Se podría crear una tabla vacía con los campos que se usan y ya está pero esta solución usará la siguiente formula para mostrar como se puede crear la clasificación de la montaña directamente en el propio comando sin nada más:

```
CREATE TABLE cl_mont AS SELECT p.ciclista,c.nom,SUM(puntos(p.nom)) as puntos
                            FROM ports p,ciclistes c
                            WHERE p.ciclista=c.dorsal GROUP BY p.ciclista,c.nom
                            ORDER BY puntos DESC;
```

Ahora procedemos a borrar la clasificación que hemos introducido para dejar toda la tabla límpia, para ello hay 2 posibilidades, un DELETE o el comando TRUNCATE TABLE (tened en cuenta que este segundo no nos sirve dentro de PL/SQL), las 2 siguientes líneas tienen el mismo resultado:

```
TRUNCATE TABLE cl_mont;
DELETE FROM cl_mont;
```

Obviamente la orden para rellenar la tabla es un simple insert que puede ejecutarse directamente, pero como se ha pedido crear un procedimiento se pone dentro de uno. Antes del insert se limpian los valores anteriores que pudiesen existir en la tabla puesto que hemos creado una tabla cl_mont sin restricciones ni claves, así que en caso de contener datos se duplicarían:

```
CREATE OR REPLACE PROCEDURE cl_mont_tabla
IS
BEGIN
        DELETE FROM cl_mont;
        INSERT INTO cl_mont SELECT p.ciclista,c.nom,SUM(puntos(p.nom)) as puntos
        FROM ports p,ciclistes c
        WHERE p.ciclista=c.dorsal GROUP BY p.ciclista,c.nom ORDER BY puntos DESC;
END;
/
```

4. Crear un Trigger para que cada vez que se insertan,borran o actualizan datos en PORTS se actualize la clasificación de la montaña.

Realmente se podría hacer todo directamente en el trigger, sin los puntos anteriores, pero aprovecharemos las funciones procedimientos creados para simplificarlo.

```
CREATE OR REPLACE TRIGGER act_clasificacion
AFTER INSERT OR UPDATE OR DELETE
ON ports
BEGIN
    cl_mont_tabla;
END;
/
```

Aquí se podría considerar diferenciar entre cuando se esté actualizando y los otros ya que un insert/delete siempre implica cambiar la clasificación y un update, mientras no cambie ciclista o categoría, no lo hará. Pero para conocer los campos que se actualizan el trigger debería ser a nivel de fila y no parece necesario actualizar la clasificación durante el proceso varias veces. Se podrían hacer 2 triggers distintos, uno a nivel de tabla para insert/delete y otro para update, pero de nuevo, el tiempo ahorrado por no actualizar la clasificación cuando los cambios no la afecten no parece compensar el análisis a nivel de fila y actualizar varias veces. Además las posibilidades de cambiar el nombre/altura/etapa de un puerto sin que impliquen cambios de categoría o ganador son bastante escasas, así que no se considera necesario.

EXTRA

Los contenidos desarrollados en este apartado no se pedirán para el examen del módulo. No obstante cubren un aspecto que no se ha visto y lo suficientemente relevante para incluirlo aquí.

La cuestión es que se ha visto como usar cursores o vectores en procedimientos/funciones, pero no cómo pasarlos. Es decir, por ejemplo, como aprovechar los resultados de un cursor/vector creado en un procedimiento/función fuera del mismo o como pasar el contenido de un cursor/vector a una función/procedimiento.

Para ello se usa el mismo ejemplo de la clasificación de la motaña, pero esta vez sin usar una tabla auxiliar. Se guardará la clasificación en un cursor/vector mediante un procedimiento y se leerá mediante otro.

1. Cursor

Puesto que los cursores se definen en el mismo momento de su declaración, no tendría sentido declarar un cursor para después manejarlo dentro de otro procedimiento. Por tanto los cursores explícitos propiamente no se pueden pasar/recuperar de procedimientos. Para ello existen los cursores de referencia, que no son más que punteros sin definir que apuntarán a un cursor y que se definen no en su declaración sinó cuando se abren con OPEN. La forma más sencilla de declararlos es usando SYS_REFCURSOR como tipo de datos, que es un tipo de cursor de referencia predefinido en ORACLE.

En este caso se puededefinir una función que “devuelva” un cursor con la clasificación de la montaña así:

```
CREATE OR REPLACE FUNCTION cl_mont_cur
RETURN SYS_REFCURSOR
IS
      cl_muntanya SYS_REFCURSOR;
BEGIN
   OPEN cl_muntanya FOR SELECT p.ciclista,c.nom,SUM(puntos2(p.nom)) as puntos
                              FROM ports p,ciclistes c
                              WHERE p.ciclista=c.dorsal GROUP BY p.ciclista,c.nom
                              ORDER BY puntos DESC;
   RETURN cl_muntanya;
END;
/
```

Con esto el cursor no se define hasta el OPEN y por tanto se puede poner en los parametros de la función/procedimiento o en el return como SYS_REFCURSOR. Si fuese un procedimiento se podría hacer lo equivalente con:

CREATE OR REPLACE PROCEDURE cl_mont_cur(cl_muntanya OUT SYS_REFCURSOR)

y nos ahorramos la definición y el return de cl_muntanya.

Ahora vamos a crear otro pocedimiento independiente que lea este cursor y lo muestre por la pantalla:

```
CREATE OR REPLACE PROCEDURE leer_mont_cur(cursor1 IN SYS_REFCURSOR)
IS
fila cl_mont%ROWTYPE;
BEGIN
        DBMS_OUTPUT.PUT_LINE('CLASSIFICACIO DE LA MUNTANYA');
        FETCH cursor1 INTO fila;
        WHILE cursor1%FOUND LOOP
                DBMS_OUTPUT.PUT_LINE(fila.nom||': '||fila.puntos);
                FETCH cursor1 INTO fila;
        END LOOP;
        CLOSE cursor1;
END;
/
```

Recordad que cl_mont es la tabla que usamos antes para la clasificación, con los mismos campos que el select así que la reaprovecho para crear el registro con ROWTYPE.

Con SYS_REFCURSOR no se puede usar el bucle FOR de cursor, pues este lleva un OPEN y un CLOSE implicito, y los cursores de referencia ya vienen abiertos pues recordad que se definen con la orden OPEN. Esta es la limitación que tienen; al definirse en la apertura, se pueden “leer” una única vez; después de esto el FETCH ya no devuelve nada y si lo cerramos vuelve a estar indefinido, no se puede volver a abrir con un OPEN a secas como los otros sino que necesita un open que lo defina, como el de la función anterior.

Para usar esta función y proceso que hemos creado, por ejemplo:

```
DECLARE
  a SYS_REFCURSOR;
BEGIN
  a:=cl_mont_cur();
  leer_mont_cur(a);
END;
/
```

Un segundo intento de leer a fallaría, pero se puede volver a lanzar a:=cl_mont_cur(); para “guardar” otra vez la clasificación y poder leerlo de nuevo.

2. Vector

Para usar vectores el problema es que hay que definir el TYPE del vector previamente. Como en las funciones/procedimiento tenemos que especificar el tipo de los parametros y los returns esto resulta un problema. Para vectores de tipos de datos simples como un vector de enteros, o de varchars hay soluciones más fáciles, pero si queremos guardar contenidos de consultas esto es menos directo. En este caso se necesita definir un registro:

```
TYPE reg_mont IS RECORD (
    ciclista number(5),
    nom varchar2(30),
    puntos int);
```

Para el tipo de datos de los elementos del vector (cada tupla que devuelve nuestro SELECT) y después definir el tipo de nuestro vector de registros reg_mont:

```
TYPE vec_mont IS VARRAY(20) OF reg_mont;
```

Entonces necesitamos que oracle reconozca estas definiciones de reg_mont y vec_mont como reconoze un INT o un VARCHAR, para poder usarlas como tipo en los parámetros o returns al crear nuestras funciones/parámetros.

Esto puede lograrse usando paquetes, de forma parecida a como puede hacerse en java. Un parquete es una colección de definiciones y funciones que se podrán llamar desde la BD si se incluye el paquete. Se podrían crear también en el paquete los procedimientos que usaremos, pero por simplificar solo se crearán los tipos de datos y los procedimientos se crearán del modo habitual.

Para definir un paquete que llamaremos pk_mont y que contenga nuestras definiciones de tipos anteriores (recordad tener permiso para poder crear paquetes y dar permiso a los usuarios para poder ejecutar los paquetes):

```
CREATE OR REPLACE PACKAGE pk_mont AS
    TYPE reg_mont IS RECORD (
            ciclista number(5),
            nom varchar2(30),
            puntos int);
    TYPE vec_mont IS VARRAY(20) OF reg_mont;
END pk_mont;
/
```

Y ahora ya podemos usar “pk_mont.vec_mont” directamente como si fuese uno de los tipos de datos por defecto de Oracle.

Así pues crear un procedimiento par almacenar la clasificación de la montaña en un vector y otro para leerlo, ya no debería suponer un problema. Se usan esta vez 2 procedimientos pero se podría usar para el primero una función con un return del vector igualmente.

```
CREATE OR REPLACE PROCEDURE cl_mon_vec(vector1 OUT pk_mont.vec_mont)
IS
BEGIN
    SELECT p.ciclista,c.nom,SUM(puntos2(p.nom)) as puntos BULK COLLECT INTO vector1
    FROM ports p,ciclistes c WHERE p.ciclista=c.dorsal GROUP BY p.ciclista,c.nom
    ORDER BY puntos DESC;
END;
/

CREATE OR REPLACE PROCEDURE leer_mon_vec(vector1 IN pk_mont.vec_mont)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('CLASSIFICACIO DE LA MUNTANYA');
    -- FOR i IN 1 .. vector1.COUNT
    FOR i IN vector1.FIRST .. vector1.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE(vector1(i).ciclista||' '||
        vector1(i).nom||': '||vector1(i).puntos);
    END LOOP;
END;
/
```

En leer_mon_vec se ha usado un bucle FOR distinto al habitual y hay otro equivalente como comentario, para mostrar opciones optimizadas para recorrer vectores. Los nombres son bastante auto descriptivos pero por si acaso

Vector1.COUNT: numero de elementos total de un vector, equivalente al último índice.

Vector1.FIRST: indice del primer elemento con datos del vector.

Vector1.LAST: indice del último elemento con datos del vector

Para llamar a los procedimientos creados se puede usar:

```
DECLARE
    clasificacion pk_mont.vec_mont;
BEGIN
    cl_mon_vec(clasificacion);
    leer_mon_vec(clasificacion);
END;
/
```

Recordad que si otro usuario autorizado quiere usar estas funciones se le deben proporcionar permisos para usar el paquete:

```
GRANT EXECUTE ON ciclisme.pk_mont TO nombreusuario;
```