Procedimientos almacenados y triggers

Contenido

6. Procedimientos almacenados y triggers 3
6.1. Introducción. 3
6.2. Funciones y procedimientos almacenados 3
6.3. Triggers 7
6.3.1. Desencadenadores BEFORE 8
6.3.2. Desencadenadores AFTER 9
6.4. ANEXO. Combinaciones en la clausula FROM 11
6.4.1. Inner Join 13
6.4.2. Left Outer Join 15
6.4.3. Right Outer Join 17
6.4.4. Full Outer Join 18
6.4.5. Natural Join 19
6.5. Ejercicios 21

6. Procedimientos almacenados y triggers

6.1. Introducción.

Antiguamente existían numerosas tareas que repetíamos constantemente en las bases de datos, como por ejemplo, recalcular campos calculados o recalcular un stock con los datos de los movimientos de una tabla. Tareas que se realizaban en el cliente con el desperdicio de ancho de banda correspondiente para recoger todos los datos. Todas estas tareas y otras pueden ahora realizarse en la parte del servidor con el correspondiente ahorro tiempo. En esta sesión vamos a ver algunos de los diversos mecanismos que existen y como se implementan en MySQL.

6.2. Funciones y procedimientos almacenados

Los procedimientos almacenados en SQL nos proporcionan los mecanismos adecuados para darle a este lenguaje modularidad, que revertirá en claridad para nuestro código así como un mayor grado de reutilización del mismo. Adicionalmente, conseguimos reducir el intercambio de datos con los clientes al realizar parte de los procesos el servidor, ganando en seguridad y en velocidad.

La sintaxis para definir procedimientos almacenados es la siguiente:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

CREATE
    [DEFINER = { user | CURRENT_USER }]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type
```

```
characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }

routine_body:
    Valid SQL routine statement
```

La definición completa de la creación de un procedimiento almacenado se puede consultar en:

http://dev.mysql.com/doc/refman/5.1/en/create-procedure.html

Una función la usaremos en nuestras sentencias SQL mientras que para ejecutar un procedimiento debemos usar el comando CALL:

```
CALL ActualizaStock ()
```

Cuya sintaxis podemos encontrar en:

http://dev.mysql.com/doc/refman/5.1/en/call.html

Para pasar parámetros de entrada o de salida a un procedimiento almacenado debemos hacerlo por posición. Los parámetros irán separados por comas y en el orden de definición. Mientras que los parámetros de los procedimientos pueden ser de entrada, de salida o de entrada-salida, los de las funciones son solo de entrada.

Como ejemplo veamos una versión iterativa del cálculo del factorial de un número:

```
DELIMITER //

CREATE FUNCTION factorial (num bigint)
RETURNS bigint DETERMINISTIC
BEGIN
  DECLARE aux,res bigint default 0;

  IF(num < 1) THEN SET res=0;
  ELSE
    BEGIN
      SET aux=1;
      SET res=1;
      WHILE(aux<=num) DO
          SET res = res * aux;
          SET aux = aux + 1;
      END WHILE;
    END;
  END IF;
  RETURN res;
END//

DELIMITER ;

/*y su llamada*/
SELECT factorial (39);
--En este caso 39 será el máximo numero con el que podemos llamar a
factorial, 40 ya desbordaría.
```

Se puede observar que se ha introducido un cambio de delimitador de sentencias para que pueda ser lanzado desde línea de comandos sin problema.

Vamos ahora a ver lo mismo pero como procedimiento:

```
DELIMITER //

CREATE procedure factorialProc (IN num bigint, OUT resultado bigint)
BEGIN
  DECLARE aux bigint default 0;

  SET resultado = 0;
  IF(num < 1) THEN SET resultado=0;
  ELSE
    BEGIN
      set aux=1;
      set resultado=1;
      WHILE(aux<=num) DO
          SET resultado = resultado * aux;
          SET aux = aux + 1;
      END WHILE;
    END;
  END IF;

END//

DELIMITER ;

/*y su llamada*/
CALL factorialProc (39,@mia);
SELECT @mia;
```

6.3. Triggers

Un desencadenador, disparador o trigger es un procedimiento almacenado asociado con una tabla, el cual se ejecuta automáticamente por ejemplo, cuando se modifica, inserta o elimina un dato de esa tabla.

MySQL soporta triggers **AFTER** o **BEFORE** (trigger_time), es decir que se ejecuten antes o después de insertar, modificar o eliminar cada fila. El estándar SQL incluye otro tipo de trigger **INSTEAD OF** que MySQL no soporta, que reemplazaría la propia acción disparadora por la que le indiquemos, aunque al existir **BEFORE**, algunas de las cosas que pueden hacerse con un **INSTEAD OF** pueden realizarse con **BEFORE** .

Hay que tener en cuenta que MySQL dispone de varias sentencias que tienen como consecuencia la inserción, modificación o eliminación de filas. Los triggers afectan a todas ellas y no sólo a las sentencias **INSERT**, **UPDATE** o **DELETE** aunque al definir trigger_event pueda parecerlo al indicarse en ingles.

Dentro del cuerpo del trigger para hacer referencia al registro viejo (modificación y borrado) utilizaremos **OLD** y para hacer referencia al registro nuevo (inserción y modificación) utilizaremos **NEW**. Como se puede extraer de los registros que utiliza cada acción, una modificación actúa realmente como un borrado y una inserción

```
CREATE
      [DEFINER = { user | CURRENT_USER }]
      TRIGGER trigger_name trigger_time trigger_event
      ON tbl_name FOR EACH ROW trigger_body
```

Vamos a ver ahora como actúa cada uno de los dos tipos de trigger del lenguaje de manipulación de datos:

6.3.1. Desencadenadores BEFORE

Un desencadenador BEFORE se ejecuta antes de la acción desencadenadora. Realmente, se ejecuta una vez creadas las tablas inserted y deleted, pero antes de que se lleve a cabo cualquier otra acción. Se ejecutan antes que las restricciones, de manera que se pueden realizar las tareas de procesamiento previo que complementan a las acciones de restricción.

Por ejemplo, deseamos con un trigger, que al crear una persona de las tablas del anexo esta se cree sin domicilio fiscal de forma que haya que asignarlo luego. Mediante otra función que no implementaremos.

```
DELIMITER //

CREATE TRIGGER InsertaPersona
BEFORE INSERT ON personas
  FOR EACH ROW BEGIN
     NEW.Domfiscal = NULL;
  END//

DELIMITER ;

INSERT INTO Personas (IdNif, Nombre, Ape1, Ape2, DomFiscal)
VALUES("23232300T", "Fernando", "Martínez", "De la Rosa", "00001")
SELECT * FROM Personas;
```

Podemos observar que independientemente de lo introducido en el campo DomFiscal, realmente se graba a nulo.

6.3.2. Desencadenadores AFTER

Un desencadenador AFTER se ejecuta cuando todas las operaciones asociadas a una instrucción han finalizado, es decir, si todo el proceso de inserción, borrado o actualización así como todas las comprobaciones de restricciones se han realizado y su resultado es correcto.

Supongamos que queremos que cuando algún proceso intente borrar un registro de la tabla provincia del ejemplo del anexo, en lugar de hacerlo tal cual, realmente añadiéramos una fecha de borrado a una nueva columna (que habremos creado previamente) llamada Fecha_Borrado.

```
DELIMITER //

CREATE TRIGGER CambiaBorrado
AFTER DELETE ON provincias
  FOR EACH ROW BEGIN
     INSERT INTO Provincias (IdProvincia, Descripcion, Fecha_Borrado)
     VALUES( OLD.IdProvincia, OLD.Descripcion, NOW());
  END//

DELIMITER ;

SELECT * FROM provincias;
DELETE FROM provincias WHERE IdProvincia=44;
SELECT * FROM provincias;
```

Tras ejecutar el ejemplo podemos observar que se produce un error, debido a una limitación de MySQL que por ahora no permite modificar el registro que ha lanzado el trigger cuando en un trigger **AFTER**, las acciones ya se han realizado. Se supone que en versiones posteriores deberían cambiarlo. Puesto que limita mucho la funcionalidad de los triggers.

Para no alterar el ejemplo anterior supongamos que deseamos que cuando se elimina una población del ejemplo del anexo esta pase a una tabla de histórico de poblaciones llamada Poblaciones_OLD que deberemos de crear previamente.

```
DELIMITER //
CREATE TRIGGER CopiaBorrado
  AFTER DELETE ON poblaciones
  FOR EACH ROW BEGIN
     INSERT INTO PoblacionesOLD
     VALUES( OLD.IdPobl,OLD.DescPobl, OLD.idProv);
  END//
DELIMITER ;

SELECT * FROM poblaciones;
DELETE FROM poblaciones WHERE codigo=4;
SELECT * FROM poblacionesOLD;
SELECT * FROM poblaciones;
```

Podemos observar que el resultado es el deseado

6.4. ANEXO. Combinaciones en la clausula FROM

En este punto se van a ver los distintos tipos de combinaciones que podemos aplicar en nuestras consultas. Con algunos de ellos lo que conseguimos es adelantar las condiciones al momento de la combinación de tablas y no necesitar un filtrado posterior, por lo que deberían ser consultas mucho mas optimizadas.

A colación con esto último, en el estándar SQL92 aparece la combinación CROSS JOIN cuya finalidad original era indicar productos cartesianos teniendo así el programador que definirlos, pero en cambio por motivos de compatibilidad se siguió permitiendo el uso en el FROM de tablas separadas por comas (producto cartesiano a no ser por las condiciones del WHERE). Realmente las buenas prácticas deberían llevarnos a no usar en el FROM dichas combinaciones pero a fecha de hoy aun hay mucha gente que las utiliza.

Para ello vamos a utilizar durante este apartado las tablas del ejercicio del tema 5 que nos ayudará a entender mejor cada combinación con los datos mostrados a continuación.

Provincias

IdProvincia	Descripcion
2	Albacete
3	Alicante
12	Castellon
16	Cuenca
44	Teruel
46	Valencia

Localidades

IdPobl	DescPobl	idProv
1	Alcoy	3
2	Elche	3
3	Alicante	3
4	Segorbe	12
5	Barracas	12
6	Valencia	46
7	Catarroja	46

Direcciones

IdDir	Calle	Num	Piso	Puerta	IdPobl
00001	Mayor	Bajo	Izq		2
00002	Del Olmo	25	3	9	2
00003	Mayor	12	1	3	4
00004	Cuba	73	5	18	6
00005	Sueca	27	Ent	A	6
00006	Alba	12	1	1	4
00007	Puerto	15	6	23	5
00008	Vecinos	16	3	15	6
00009	Carcel	1	3	23	7

Propiedades

IdDir	IdNif
00001	0T
00001	23T
00002	0T
00003	345
00004	827
00006	345
00006	0T
00007	345
00008	827

Personas

IdNif	Nombre	Ape1	Ape2	DomFiscal
0T	Pedro	Martínez	Pérez	00001
23T	Esther	García	Pons	00001
46T	Jaime	Martínez	García	**NULL**
345	Potentoso	López		00007
827	Joan	Esteve	Núñez	00007
900	Pedro	Blasco	Hurtado	00004

Para facilitar los ejemplos se facilitan los inserts correspondientes. Como los nombres de campo no estaban prefijados, basta con cambiar la cabecera de los inserts para que funcione con las tablas de los ejercicios del tema 5:

INSERT INTO Provincias (IdProvincia, Descripcion)

VALUES(2, "Albacete"), (3, "Alicante"), (12, "Castellon"), (16, "Cuenca"), (44, "Teruel"), (46, "Valencia");

INSERT INTO Poblaciones (IdPobl, DescPobl, idProv)

VALUES(1, "Alcoy", 3), (2, "Elche", 3), (3, "Alicante", 3), (4, "Segorbe", 12), (5, "Barracas", 12), (6, "Valencia", 46), (7, "Catarroja", 46);

INSERT INTO Direcciones (IdDir, Calle , Num, Piso, Puerta, IdPobl)

VALUES("00001", "Mayor", "Bajo", "Izq", "", 2), ("00002", "Del Olmo", "25", "3", "9", 2), ("00003", "Mayor", "12", "1", "3", 4), ("00004", "Cuba", "73", "5", "18", 6), ("00005", "Sueca", "27", "Ent", "A", 6), ("00006", "Alba", "12", "1", "1", 4), ("00007", "Puerto", "15", "6", "23", 5), ("00008", "Vecinos", "16", "3", "15", 6), ("00009", "Carcel", "1", "3", "23", 7);

INSERT INTO Personas (IdNif, Nombre, Ape1, Ape2, DomFiscal)

VALUES("0T", "Pedro", "Martínez", "Pérez", "00001"), ("23T", "Esther", "García", "Pons", "00001"), ("46T", "Jaime", "Martínez", "García", null), ("345", "Potentoso", "López", "", "00007"), ("827", "Joan", "Esteve", "Núñez", "00007"), ("900", "Pedro", "Blasco", "Hurtado", "00004");

INSERT INTO Propiedades (IdDir, IdNif)

VALUES("00001", "0T"), ("00001", "23T"), ("00002", "0T"), ("00003", "345"), ("00004", "827"), ("00006", "345"), ("00006", "0T"), ("00007", "345"), ("00008", "827");

6.4.1. Inner Join

Esta combinación incluirá todas aquellas filas de la primera tabla y todas aquellas filas de la segunda tabla que cumplan la condición.

Por ejemplo, vamos a seleccionar las personas con sus domicilios fiscales:

```
SELECT *
FROM Personas P
INNER JOIN Direcciones D
ON P.DomFiscal=D.IdDir
```

Nos mostrará:

IdNif	Nombre	Ape1	Ape2	DomFiscal	IdDir	Calle	Num	Piso	Puerta	IdPobl
0T	Pedro	Martínez	Pérez	00001	00001	Mayor	Bajo	Izq		2
23T	Esther	García	Pons	00001	00001	Mayor	Bajo	Izq		2
345	Potentoso	López		00007	00007	Puerto	15	6	23	5
827	Joan	Esteve	Núñez	00007	00007	Puerto	15	6	23	5
900	Pedro	Blasco	Hurtado	00004	00004	Cuba	73	5	18	6

Cabe observar que hemos perdido a la persona de Nif 46T, que no tenía dirección fiscal, así como todas las direcciones que no eran dirección fiscal de nadie.

Esto mismo resultado se podía haber hecho con la siguiente sentencia. Pero ya hemos indicado que no deberíamos utilizarlo y deberíamos utilizar la segunda sintaxis

```
SELECT *
FROM Personas P, Direcciones D
WHERE P.DomFiscal=D.IdDir

SELECT *
FROM Personas P
CROSS JOIN Direcciones D
WHERE P.DomFiscal=D.IdDir
```

Si quisiéramos restringir estos resultados a aquellas personas cuya dirección fiscal está ubicada en la población de identificador 5, podríamos refinar la sentencia inicial de dos formas, aunque cuando encadenemos más de dos tablas, de cara a refinamientos posteriores, en mi opinión queda más clara la segunda puesto que cada condición queda con su tabla:

```
SELECT *
FROM Personas P
INNER JOIN Direcciones D
```

```
ON P.DomFiscal=D.IdDir
WHERE D.IdPobl=5

SELECT *
FROM Personas P
INNER JOIN Direcciones D
ON P.DomFiscal=D.IdDir
AND D.IdPobl=5
```

En este caso son equivalentes, y será el motor correspondiente de MySQL el que se encargará de hacer lo mismo en los dos casos. Incluso en el caso de utilizar un **Cross Join** con estas condiciones, el optimizador de consultas lo debería convertir internamente a **Inner Join**.

6.4.2. Left Outer Join

Esta combinación incluirá todas aquellas filas de la primera tabla y solo aquellas filas de la segunda tabla que cumplan la condición. Es decir, nos podremos encontrar con filas en las que los datos de la segunda tabla sean nulos. Podríamos decir que un **Left Join** es un **Inner Join** al que hemos añadido aquellas filas de la primera tabla que no se relacionan con las de la segunda. La palabra Outer es opcional.

Por ejemplo, vamos a seleccionar las personas y para aquellos que existan sus domicilios fiscales:

```
SELECT *
FROM Personas P
LEFT JOIN Direcciones D
ON P.DomFiscal=D.IdDir
```

Nos mostrará:

IdNif	Nombre	Ape1	Ape2	DomFiscal	IdDir	Calle	Num	Piso	Puerta	IdPobl
0T	Pedro	Martínez	Pérez	00001	00001	Mayor	Bajo	Izq		2
23T	Esther	García	Pons	00001	00001	Mayor	Bajo	Izq		2
46T	Jaime	Martínez	García		**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
345	Potentoso	López		00007	00007	Puerto	15	6	23	5
827	Joan	Esteve	Núñez	00007	00007	Puerto	15	6	23	5
900	Pedro	Blasco	Hurtado	00004	00004	Cuba	73	5	18	6

Cabe observar que la principal diferencia entre esta consulta y la anterior con **Inner Join** es la persona de Nif 46T, que no tenia dirección fiscal.

En este caso el producto cartesiano de las dos tablas no podría incluir dicha fila.

Si quisiéramos restringir estos resultados a mostrar todas las personas y aquellas direcciones fiscales que están ubicadas en la población de identificador 5, podríamos pensar que la sentencia resultante podría ser:

```
SELECT *
FROM Personas P
LEFT JOIN Direcciones D
ON P.DomFiscal=D.IdDir
WHERE D.IdPobl=5
```

Pero dicha sentencia nos devuelve el mismo resultado que un Inner Join perdiendo las personas cuya dirección fiscal no está en la población de identificador 5. ¿Por qué?

Porque una vez seleccionadas las tuplas, las hemos filtrado por población, y han desaparecido los nulos. Entonces la selección debemos hacerla dentro del Left Join, quedando

```
SELECT *
FROM Personas P
LEFT JOIN Direcciones D
ON P.DomFiscal=D.IdDir
AND D.IdPobl=5
```

IdNif	Nombre	Ape1	Ape2	DomFiscal	IdDir	Calle	Num	Piso	Puerta	IdPobl
0T	Pedro	Martínez	Pérez	00001	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
23T	Esther	García	Pons	00001	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
345	Potentoso	López		00007	00007	Puerto	15	6	23	5
46T	Jaime	Martínez	García	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
827	Joan	Esteve	Núñez	00007	00007	Puerto	15	6	23	5
900	Pedro	Blasco	Hurtado	00004	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**

Resumiendo, las condiciones sobre la tabla de la derecha de un **Left Join** se deben realizar en el propio **Left Join** y no en el **Where**.

6.4.3. Right Outer Join

Esta combinación incluirá todas aquellas filas de la segunda tabla y todas aquellas filas de la primera tabla que cumplan la condición. Es decir, nos podremos encontrar con filas en las que los datos de la primera tabla sean nulos. Es exactamente lo mismo que un **Left Join**, pero respetando las filas de la derecha en lugar de las de la izquierda. La palabra **Outer** es opcional.

Por ejemplo, vamos a seleccionar las viviendas y para aquellos que sean domicilio fiscal de alguien sus datos personales:

```
SELECT *
FROM Personas P
RIGHT JOIN Direcciones D
ON P.DomFiscal=D.IdDir
```

Nos mostrará:

IdNif	Nombre	Ape1	Ape2	DomFiscal	IdDir	Calle	Num	Piso	Puerta	IdPobl
0T	Pedro	Martínez	Pérez	00001	00001	Mayor	Bajo	Izq		2
23T	Esther	García	Pons	00001	00001	Mayor	Bajo	Izq		2
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00002	Del Olmo	25	3	9	2
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00003	Mayor	12	1	3	4
900	Pedro	Blasco	Hurtado	00004	00004	Cuba	73	5	18	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00005	Sueca	27	Ent	A	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00006	Alba	12	1	1	4
345	Potentoso	López		00007	00007	Puerto	15	6	23	5
827	Joan	Esteve	Núñez	00007	00007	Puerto	15	6	23	5
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00008	Vecinos	16	3	15	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00009	Carcel	1	3	23	7

Las mismas consideraciones para las condiciones que se han realizado en el **Left Join**, se deben aplicar en el **Right Join**, es decir, los filtros se deben realizar en el propio **Right Join** y no en el **Where**.

6.4.4. Full Outer Join

A diferencia de otras bases de datos, MySQL no soporta la sentencia Full Outer Join, aunque puede ser simulada con la unión de un Left Join y un Right Join

Esta combinación incluirá todas las filas que cumplan la condición más todas las filas de la primera tabla que no la cumplen más todas las filas de la segunda tabla que no la cumplen. O sea sería equivalente a juntar un **Left Join** con un **Right Join**. La palabra **Outer** es opcional.

Por ejemplo, vamos a seleccionar las personas con sus domicilios fiscales:

```
Deberia ser:
SELECT *
FROM Personas P
FULL OUTER JOIN Direcciones D
ON P.DomFiscal=D.IdDir

Pero en MySQL se puede construir con:
SELECT *
FROM Personas P
LEFT OUTER JOIN Direcciones D
ON P.DomFiscal=D.IdDir
UNION
SELECT *
FROM Personas P
RIGHT OUTER JOIN Direcciones D
ON P.DomFiscal=D.IdDir
```

Nos mostrará:

IdNif	Nombre	Ape1	Ape2	DomFiscal	IdDir	Calle	Num	Piso	Puerta	IdPobl
0T	Pedro	Martínez	Pérez	00001	00001	Mayor	Bajo	Izq		2
23T	Esther	García	Pons	00001	00001	Mayor	Bajo	Izq		2
46T	Jaime	Martínez	García		**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00002	Del Olmo	25	3	9	2
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00003	Mayor	12	1	3	4
900	Pedro	Blasco	Hurtado	00004	00004	Cuba	73	5	18	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00005	Sueca	27	Ent	A	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00006	Alba	12	1	1	4
345	Potentoso	López		00007	00007	Puerto	15	6	23	5
827	Joan	Esteve	Núñez	00007	00007	Puerto	15	6	23	5
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00008	Vecinos	16	3	15	6
NULL	**NULL**	**NULL**	**NULL**	**NULL**	00009	Carcel	1	3	23	7

Se puede observar que se han incluido todos los registros que coinciden y los de ambas tablas que no coincidan.

No existe correspondencia de esta combinación con el producto cartesiano.

6.4.5. Natural Join

Es una variación que se puede aplicar tanto a **Left Join** y a **Right Join** como a **Inner Join**, permite cruzar dos tablas por los campos que se llamen igual en ambas tablas, sin necesidad de especificar la igualdad de dichos campos en la clausula **ON.** En el caso del **Inner Join**, inner desaparece, formulándose como **Natural Join** mientras que con los otros dos se mantiene.

Por ejemplo, vamos a cruzar direcciones con propiedades:

```
SELECT *
FROM Direcciones D
LEFT OUTER JOIN Propiedades P
ON D.IdDir = P.IdDir

Podria escribirse con NATURAL como:
SELECT *
FROM Direcciones D
NATURAL LEFT OUTER JOIN Propiedades P
```

Nos mostrará en el primer caso:

IdDir	Calle	Num	Piso	Puerta	IdPobl	IdDir	IdNif
00001	Mayor	Bajo	Izq		2	00001	0T
00001	Mayor	Bajo	Izq		2	00001	23T
00002	Del Olmo	25	3	9	2	00002	0T
00003	Mayor	12	1	3	4	00003	345
00004	Cuba	73	5	18	6	00004	827
00005	Sueca	27	Ent	A	6	**NULL**	**NULL**
00006	Alba	12	1	1	4	00006	0T
00006	Alba	12	1	1	4	00006	345
00007	Puerto	15	6	23	5	00007	345
00008	Vecinos	16	3	15	6	00008	827
00009	Carcel	1	3	23	7	00009	**NULL**

Y en el Segundo:

IdDir	Calle	Num	Piso	Puerta	IdPobl	IdNif
00001	Mayor	Bajo	Izq		2	0T
00001	Mayor	Bajo	Izq		2	23T
00002	Del Olmo	25	3	9	2	0T
00003	Mayor	12	1	3	4	345
00004	Cuba	73	5	18	6	827
00005	Sueca	27	Ent	A	6	**NULL**
00006	Alba	12	1	1	4	0T
00006	Alba	12	1	1	4	345
00007	Puerto	15	6	23	5	345
00008	Vecinos	16	3	15	6	827
00009	Carcel	1	3	23	7	**NULL**

Se puede observar que la única diferencia es que la columna del cruce (IdDir) solo aparece una vez, ya que MySQL ha identificado que son la misma y la ha utilizado para el cruce.

Personalmente este tipo de combinación no es de mi agrado, dado que puede llevar a efectos laterales no deseados y difíciles de localizar. Supongamos que en una aplicación estamos combinando dos tablas por dos campos comunes en cada una, si tras un cambio de esquema uno de los cuatro campos cambia de nombre, con ***Natural Join*** *la consulta seguirá funcionado aunque con resultados distintos, mientras que si hemos tenido que especificar los campos, se producirá un error, que podremos subsanar, pero no obtendremos resultados no deseados sin que se produzca error.*

6.5. Ejercicios

Obligatorios

Basándose en el esquema del anexo 6.4, se pide:

1. Realizar un procedimiento almacenado que para un NIF facilitado como parámetro de entrada, devuelva el numero de propiedades que posee y una cadena que indique si es propietario de la vivienda que figura como su domicilio fiscal indicándose:
    - "Propietario"
    - "Copropietario(x)" siendo x el número total de propietarios de la vivienda
    - "NoPropietario"
2. Añadir los triggers necesarios o realizar los cambios necesarios a los trigers de los ejemplos anteriores para que en el ejemplo de la población si se inserta una población cuya clave ya está en la tabla de respaldo, en lugar de insertar los datos nuevos se recuperen los viejos, eliminado el registro de la tabla de respaldo también**.**
3. Añadir los triggers necesarios o realizar los cambios necesarios a los trigers de los ejemplos anteriores para que en el caso de la inserción de personas, si la dirección del domicilio fiscal ya existe en la tabla de domicilios se permita, pero si no, siga poniéndola a nulo.

Voluntarios

4. Obtener las propiedades que al menos tienen dos propietarios sin utilizar subconsultas en la clausula FROM.
5. Obtener las propiedades que al menos tienen dos propietarios sin utilizar subconsultas en la clausula WHERE.
6. Realizar mediante la operación UNION una vista que incluya para cada persona todas sus direcciones vinculadas, es decir, tanto el domicilio fiscal como las propiedades que posee.