Practica 5
Programas Almacenados y Vistas

- En esta/s sesión/es se aprenderá a gestionar tanto los programas almacenados como las vistas: objetos de la base de datos definidos en términos de código SQL que se almacenan en el servidor para su posterior ejecución. Se verá:
  - Programas almacenados: Procedimientos y Funciones
  - Handler o manejadores
  - Cursores
  - Gestión de errores
  - Triggers / Disparadores
  - Vistas

- Se recomienda la lectura de la documentación de MYSQL referida a este tema (capítulo 20).
  - http://dev.mysql.com/doc/refman/5.6/en/security.html
  - http://www.java2s.com/Code/SQL/CatalogSQL.htm

1. Introducción

- A partir de la versión 5.0, MySQL permite la extensión procedimental.
  - Se puede almacenar procedimientos y funciones dentro del servidor.
  - Estos procedimientos son un conjunto de comandos SQL que los clientes pueden ejecutar conjuntamente refiriéndose al procedimiento que los contiene.

- Una de las principales ventajas es la mejora en el rendimiento pues se ejecutan las funciones dentro del servidor. Además estas funciones son independientes del lenguaje de las aplicaciones clientes que usen las bases de datos del servidor.

2. Programas almacenados

- Los programas almacenados incluyen estos objetos:
  - Programas almacenados: procedimientos almacenados y funciones.
    - ✓ Un procedimiento resuelven un determinado problema cuando son llamados y pueden aceptar varios parámetros de entrada así como devolver varios de salida.
    - ✓ Una función almacenada se utiliza mucho como una función incorporada, se invoca en una expresión y devuelve un valor durante la evaluación de la expresión.
  - Triggers. Un disparador es un objeto de base de datos que se asocia con una tabla y que se activa cuando se produce un evento en particular en la tabla, como una inserción o actualización.
  - Events. Un evento es una tarea que el servidor se ejecuta de acuerdo a lo programado.

2. Programas almacenados

- Programas almacenados son útiles en determinadas situaciones:
  - Cuando existen múltiples aplicaciones clientes que están escritas en diferentes lenguajes o trabajan en diferentes plataformas, pero tienen que realizar las mismas operaciones. Con programas almacenados:
    - ✓ Mejor mantenimiento de las aplicaciones que acceden a los programas pues sólo debe modificarse en el servidor.
    - ✓ Disminución de la posibilidad de aparición de errores.
    - ✓ Mayor portabilidad de las aplicaciones. No es necesario usar conectores como ODBC (Open DataBase Connectivity) o JDBC (Java DataBase Connectivity) sino establecer el procedimiento y realizar su llamada.
    - ✓ Reducción del tráfico de red.
  - Cuando la seguridad es primordial.
    - ✓ Las aplicaciones y los usuarios no tendrían acceso a las tablas de la bases de datos directamente, sólo pueden ejecutar procedimientos almacenados específicos y recuperar su valor. Por ejemplo: los bancos
- **Desventaja**:
  - Aumenta la carga en el servidor de base de datos porque la mayoría del trabajo se hace en el lado del servidor y es una fuente de ataques

2. Programas almacenados

- Un procedimiento o función se asocia siempre con una base de datos concreta. Esto tiene varias implicaciones:
  - No se permite “USE” en el programa almacenado
  - Se puede llamar a partir de su base de datos. CALL test.p() o test.f() .
  - Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se eliminan.
- Procedimiento vs Funciones
  - CREATE PROCEDURE
    - ✓ Un procedimiento se invoca utilizando **CALL**
    - ✓ Puede devolver valores pero **usando variables de salida**. El procedimiento en sí no devuelve nada.
  - CREATE FUNCTION
    - ✓ Puede llamarse desde **dentro de una sentencia**
      - select * from XX where text.f()=10
    - ✓ Como cualquier otra función **devuelve un valor escalar**.
    - ✓ No puede ser recursiva

2. Programas almacenados

- Privilegios asociados a los programas almacenados:
  - CREATE ROUTINE para crear programas almacenados
  - ALTER ROUTINE para modificar o borrar programas almacenados.
  - EXECUTE para ejecutar los programas almacenados

- Para mostrar el código fuente de un programa almacenado
  - SHOW PROCEDURE CODE <<nombre_prog_almacenado>>

- **Ejercicio**: Crear un usuario con acceso desde localhost que se llame Sesión 5 que tenga acceso a la BD world y tenga los siguientes privilegios:
  - CREATE ROUTINE, ALTER ROUTINE, EXECUTE
  - SELECT, INSERT, DELETE, UPDATE

2. Programas almacenados

- Cada programa contiene un conjunto de sentencias SQL.
- **IMPORTANTE**:
  - MySQL utiliza el **carácter ";" para finalizar cada sentencia SQL**.
  - Como dentro del cuerpo las instrucciones van separadas por ";", para distinguirlas de las sentencias SQL se **necesita otro carácter delimitador**
  - La primera y última línea siempre establecerá el delimitador

```
mysql> CREATE FUNCTION hello (s CHAR(20))
mysql> RETURNS CHAR(50) DETERMINISTIC
    -> RETURN CONCAT('Hello, ',s,'!');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

```
mysql> delimiter //

mysql> CREATE PROCEDURE dorepeat(p1 INT)
    -> BEGIN
    ->   SET @x = 0;
    ->   REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
    -> END
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL dorepeat(1000);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x;
+------+
| @x   |
+------+
| 1001 |
+------+
1 row in set (0.00 sec)
```

2. Programas almacenados

- Sintaxis
  - **Declaración de variables**: DECLARE nom_var1 [,nom_var2...] tipo [DEFAULT valor];
    - ✓ Declare edad INT;
    - ✓ Declare nombre varchar(30);
    - ✓ Declare fechaNacimiento date default CURRENT_DATE;
  - **Asignación de valores**: Sentencia set:
    - ✓ Set edad=0;
    - ✓ Set nombre='Luis';
    - ✓ Set fechaNacimiento = '1975-10-10';
  - **Variables del usuario**: pueden ser manipuladas dentro y fuera de los programas almacenados
    - ✓ Este tipo de variables no necesitan declaración y van precedidas del carácter @.
    - ✓ Son de un tipo de datos variant y pueden almacenar texto, fechas y números.
      - set @v1= edad * 3;

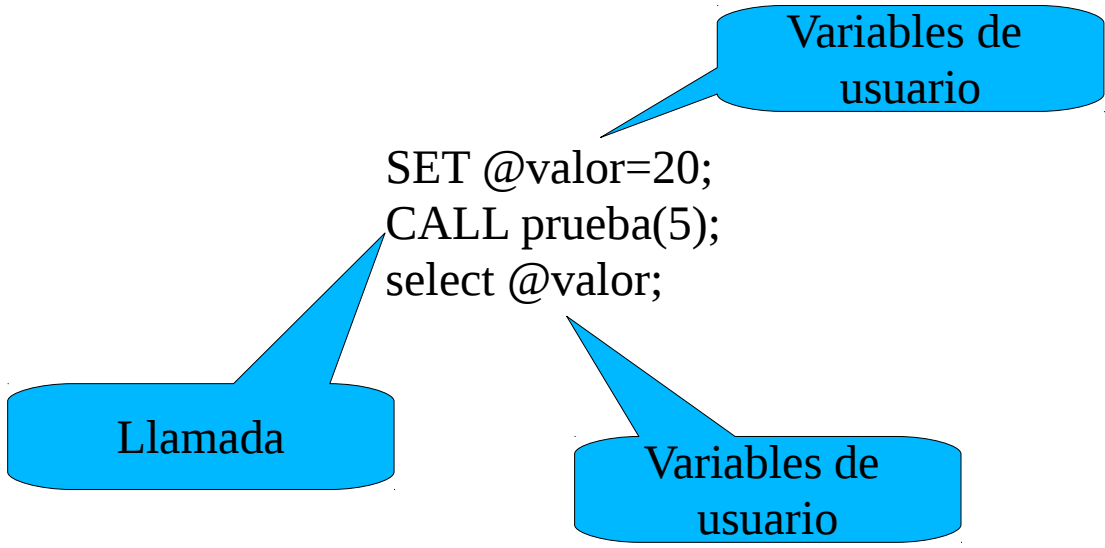
2. Programas almacenados

- Sintaxis. Ejemplo

Definición

```
delimiter //
DROP PROCEDURE IF EXISTS prueba //
CREATE PROCEDURE prueba(p1 INT)
BEGIN
  DECLARE x INT;
  SET x = p1;
  SET @valor = x * @valor;
END
//
delimiter ;
```

Ejecución



Ejercicio: Haga la suma de dos números recibidos por parámetros y lo devuelva en @suma

2. Programas almacenados

- Sintaxis
  - **Parámetros**: CREATE PROCEDURE nombre ([[IN |OUT |INOUT] nombre_parámetro tipo de datos...])
    - ✓ Los parámetros son variables que se envían y reciben de los programas a los que se llaman.
      - IN: Opción por defecto. Paso de parámetros por valor
      - OUT: Paso de parámetros por variable
        - » Las modificaciones del parámetro modifican su valor
        - » Hasta que **no se le asigne un valor determinado**, su valor dentro de él será nulo.
        - » Se suele utilizar para devolver si la ejecución ha sido exitosa o no
      - INOUT: Recibe un valor inicial y es modificable
  - **Operadores**:
    - ✓ Aritméticos: +, -, *, /, div, %,
    - ✓ Comparación: <, >, >=, <=,=, <>, !=,between, is null, is not null, not between, in, not in, like
    - ✓ Lógicos: and, or y not
  - **Funciones**: matemáticas, cadenas, fechas, … ver documentación MySQL – capítulo 12

Ejercicio2: Suma y se devuelva por parámetro

2. Programas almacenados

- Sintaxis: Programación estructurada
  - Estructura condicionales:
    - ✓ If-then
    - ✓ Case-when
      - Si no se quisiera añadir ninguna sentencia en el bloque ELSE, añadiremos BEGIN END;
  - Estructuras iterativas
    - ✓ LOOP
      - Sentencias LEAVE e ITERATE.
    - ✓ REPEAT
    - ✓ WHILE.

Ejercicio: Dado un entero, devolver si es “menor”, “mayor”, “Jubilado” de edad por parámetro

```
IF condición THEN
    Sentencias;
   ELSEIF condición THEN
        Sentencias;
       ELSE
        Sentencias;
END IF;
```

```
CASE variable
   WHEN valor1 THEN sentencias;
   WHEN valor2 THEN sentencias;
   ELSE sentencias;
END CASE;
```

```
[etiqueta:] LOOP
   Sentencias;
END LOOP [etiqueta];
```

```
[etiqueta:] REPEAT
    sentencias;
   UNTIL condición
END REPEAT [etiqueta];
```

```
[etiqueta:] WHILE condición
DO
    sentencias;
END WHILE [etiqueta];
```

2. Programas almacenados

- Sintaxis: Programación estructurada

Definición Loop

```
delimiter //
CREATE PROCEDURE bucle_loop (p1 INT)
  BEGIN
   declare cont INT;
   set cont=0;
   label1: LOOP
     IF cont<p1 THEN
        SET cont = cont + 1;
         set @suma = @suma + cont;
         ITERATE label1;
     END IF;
     LEAVE label1;
   END LOOP label1;
END
//
```

Ejecución

```
set @suma =0;
call bucle_while(10);
select @suma;
```

Definición Repeat

```
delimiter //
CREATE PROCEDURE bucle_repeat (p1 INT)
  BEGIN
    declare cont INT;
    set cont=0;
    repeat
          SET cont = cont + 1;
          set @suma = @suma + cont;
          until cont>=p1
    END repeat;
END
//
```

Definición while

```
delimiter //
CREATE PROCEDURE bucle_while (p1 INT)
  BEGIN
    declare cont INT;
    set cont=0;
    while cont<p1 do
       SET cont = cont + 1;
       set @suma = @suma + cont;
    END while;
END
//
```

3. Procedimiento almacenados

- Sintaxis de Procedimientos Almacenados

```
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
```

3. Procedimiento almacenados

- Sintaxis
  - LANGUAGE SQL indica el lenguaje utilizado en los procedimientos cumple el estándar SQL:PSM. Cláusula innecesaria en MYSQL
  - [NOT] DETERMINISTIC:
    - ✓ DETERMINISTIC, siempre ante una misma entrada devuelve una misma salida.
    - ✓ NOT DETERMINISTIC: su valor depende de la fecha o momento en el que se ejecute.
  - [{CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL}]: Tipo de acceso a la base de datos
    - ✓ READ SQL DATA: Sólo lectura
    - ✓ MODIFIES SQL DATA: Modificar los datos
    - ✓ NO SQL: No accede a la BD
    - ✓ CONTAINS SQL: contiene consultas SQL.
  - SQL SECURITY {DEFINER | INVOKER}]: Si se ejecutará con los permisos del usuario que lo creó (DEFINER) o con los permisos del usuario que llama al procedimiento (INVOKER).

3. Procedimiento almacenados

- Ejemplo de Función Almacenada

Definición

```
DELIMITER //
CREATE FUNCTION myFunction(normal_price NUMERIC(8,2))
        RETURNS NUMERIC(8,2)
BEGIN
  DECLARE discount_price NUMERIC(8,2);
  IF (normal_price>500) THEN
        SET discount_price=normal_price*.8;
  ELSEIF (normal_price>100) THEN
        SET discount_price=normal_price*.9;
  ELSE
        SET discount_price=normal_price;
  END IF;
  RETURN(discount_price);
END
//
delimiter ;
```

Ejecución

```
select myFunction(123.123);
```

3. Procedimiento almacenados

- Modificación de un programa almacenado

```
ALTER PROCEDURE|FUNCTION proc_name [characteristic ...]
characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

- Borrado de un programa almacenado

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Procedimientos almacenados: Ejercicios

- Devuelva una cadena que indique si un valor recibido por parámetro es menor/mayor/adulto(65) de edad cómo función
- Desarrollar una función que devuelva el número de años completos que hay entre dos fechas que se pasan como parámetros. Utiliza la función DATEDIFF .
- Escribir una función que, haciendo uso de la función anterior, devuelva los trienios que hay entre dos fechas.

3. Procedimiento almacenados

- Sin duda alguna, una de las ventajas más importantes de los programas almacenados, es la posibilidad de ejecutar instrucciones DML así como DDL.
  - MySQL devuelve como resultado de la ejecución del procedimiento un conjunto de filas originado por un Select.
  - **Cuidado**: Un procedimiento MySQL no puede reutilizar el resultado de la ejecución de otro procedimiento almacenado → Tabla temporal
  - **Importante**: Si el resultado es una única fila, se puede recuperar

Definición

```
 DELIMITER //
CREATE PROCEDURE CiudadesEspanolas()
  BEGIN
  select * from City where CountryCode='ESP';
  END //
DELIMITER ;
```

Ejecución

```
call CiudadesEspanolas;
```

3. Procedimiento almacenados

- Sin duda alguna, una de las ventajas más importantes de los programas almacenados, es la posibilidad de ejecutar instrucciones DML así como DDL.

Definición

```
DELIMITER //
CREATE PROCEDURE CreateCiudadesEspanolas()
  BEGIN
  drop temporary table if exists ciudadesEspanolas;
  create temporary table ciudadesEspanolas(
            `ID` int(11) NOT NULL,
            `Name` char(35) NOT NULL DEFAULT '',
            `Population` int(11) NOT NULL DEFAULT '0',
            PRIMARY KEY (`ID`))ENGINE MEMORY;
  insert into ciudadesEspanolas
            select ID,Name,Population from City where
                        CountryCode='ESP';
  END //
DELIMITER ;
```

Ejecución

```
call CreateCiudadesEspanolas;
Select * from ciudadesEspanolas;
```

3. Procedimiento almacenados

- Sentencias preparadas. SQL dinámico.
  - MySQL permite realizar sentencias SQL para que sean ejecutadas varias veces de manera eficiente → Sentencias preparadas
  - Crear una sentencia preparada
    - ✓ PREPARE nombre_sentencia FROM texto_sql
  - Ejecución
    - ✓ EXECUTE nombre_sentencia [USING variable [, variable …]]
  - Eliminación de la sentencia preparada
    - ✓ DEALLOCATE PREPARE sentencia_preparada
  - SQL dinámico no es muy usado pues es menos eficiente que SQL estático. Debe emplearse para realizar tareas o implementar utilidades que no pueden realizarse de otra manera.

3. Procedimiento almacenados

- Sentencias preparadas. SQL dinámico.

```
PREPARE selectCityForCountry from
            "SELECT * FROM City WHERE countrycode=?";

set @value='ESP';
EXECUTE selectCityForCountry USING @value;

set @value='USA';
EXECUTE selectCityForCountry USING @value;

DEALLOCATE PREPARE selectCityForCountry;
```

Procedimientos almacenados: Ejercicios

- Crear un PA que cambie el surfaceArea de un pais por otro que se pasará como parámetro. El PA recibirá dos parámetros, el identificador del país (char(3))y el nuevo surface (float(10,2))
  - Call cambioSurface('ABW',194):
- Crear un PA que tenga como parámetros nombre de un pais, continente y SurfaceArea de un pais y se debe modificar el surface a partir del nombre del país usando el procedimiento anterior
- Crea un procedimiento que reciba como parámetros de entrada el continente y la lengua y obtenga todos los países de ese continente que hablen esa lengua.
  - Nota, a pesar de que el campo continente es de tipo enum, podemos pasar el continente como tipo varchar porque es compatible.
- En la tabla country de la BD World, actualizar el GPN de un país cuyo nombre se pasará como parámetro: si es de África, se aumentará un 1%, si es de Antartida un 1,5%, si es de Asia o Sudamérica un 1,7%, si es de Europa o NorteAmérica un 1,9% y si es de Oceania, un 1,6%.

3. Handler

- Handler
  - Se invoca cuando se produce una condición previamente definida.
  - Esta condición suele estar asociado con una condición de error.
  - El handler lleva un comando específico asociado, que se ejecutará inmediatamente después de que se cumpla la condición.
  - Se pueden realizar tres acciones
    - ✓CONTINUE , continúa la rutina actual tras la ejecución del comando del handler.
    - ✓EXIT, termina la ejecución del comando compuesto BEGIN...END actual.
    - ✓UNDO todavía no se soporta.

3. Handler

- Si una sentencia SQL falla en un programa almacenado se interrumpe la ejecución del programa en ese punto y finaliza.
- Por ejemplo:
  - Clave primaria repetida o nula
  - Inserción de datos incorrecto
  - Recuperación de la una columna inexistente
  - Etc

- El control de estos errores se realiza mediante **handler**.

3. Handler

Handler

- Se pueden realizar tres acciones
  - ✓CONTINUE , continúa la rutina actual tras la ejecución del comando del handler.
  - ✓EXIT, termina la ejecución del comando compuesto BEGIN...END actual.
  - ✓UNDO todavía no se soporta.
- Condition_value tres condiciones:
  - ✓Número de error de MySQL.
  - ✓Código SQLSTATE estándar ANSI.
  - ✓Condiciones de error con nombre.

```
DECLARE handler_action HANDLER
    FOR condition_value
    Statemen

handler_action:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    mysql_error_code
  | SQLSTATE [VALUE]
          sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
```

3. Handler

Handler

DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET has_error = 1;

DECLARE CONTINUE HANDLER FOR 1062
SELECT 'Error, duplicate key occurred';

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
ROLLBACK;
SELECT 'An error has occurred, operation rollbacked and the stored procedure was terminated';
END;

```
DECLARE handler_action HANDLER
    FOR condition_value
    Statemen

handler_action:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    mysql_error_code
  | SQLSTATE [VALUE]
        sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
```

3. Handler

- Handler: Condition_value permite tres condiciones:
  - Número de error de MySQL: No suele usarse pues el código sería dependiente de MySQL
    - ✓ Códigos de error: http://dev.mysql.com/doc/refman/5.6/en/error-messages-server.html
  - Código SQLSTATE estándar ANSI: No suele usarse pues el código no es totalmente independiente (Microsoft SQL Server usa los suyos propios)
  - Condiciones de error con nombre: Existen las siguientes predefinidas
    - ✓ SQLWARNING es una abreviación para todos los códigos SQLSTATE que comienzan con 01.
    - ✓ NOT FOUND es una abreviación para todos los códigos SQLSTATE que comienzan con 02.
    - ✓ SQLEXCEPTION es una abreviación para todos los códigos SQLSTATE no tratados por SQLWARNING o NOT FOUND.

3. Handler

- ¿Qué sucede con @x y @x2 en los siguientes casos???

```
drop table prueba;
create table prueba (
   id int,
   primary key(id)
);
/*Insertamos dos valores repetidos*/
DELIMITER //
drop procedure IF EXISTS ProcPrueba //
CREATE PROCEDURE ProcPrueba ()
    BEGIN
    DECLARE CONTINUE HANDLER
          FOR sqlwarning SET @x2 = 1;
     SET @x = 1;
     INSERT INTO prueba VALUES (1);
     SET @x = 2;
     INSERT INTO prueba VALUES (1);
     SET @x = 3;
END
//
DELIMITER ;
call ProcPrueba();
select @x,@x2;
```

```
drop table prueba;
create table prueba (
    id int,
   primary key(id)
);
/*Insertamos dos valores repetidos*/
DELIMITER //
drop procedure IF EXISTS ProcPrueba //
CREATE PROCEDURE ProcPrueba ()
    BEGIN
    DECLARE EXIT  HANDLER
      FOR sqlwarning SET @x2 = 1;
    SET @x = 1;
    INSERT INTO prueba VALUES (1);
    SET @x = 2;
    INSERT INTO prueba VALUES (1);
    SET @x = 3;
END
//
DELIMITER ;
call ProcPrueba();
select @x,@x2;
```

```
drop table prueba;
create table prueba (
   id int,
   primary key(id)
);
/*Insertamos dos valores repetidos*/
DELIMITER //
drop procedure IF EXISTS ProcPrueba //
CREATE PROCEDURE ProcPrueba ()
    BEGIN
     DECLARE EXIT  HANDLER
       FOR sqlwarning SET @x2 = 1;
     SET @x = 1;
     INSERT INTO prueba VALUES (1);
     SET @x = 2;
     INSERT INTO prueba VALUES (1);
     SET @x = 3;
END
//
DELIMITER ;
call ProcPrueba();
select @x,@x2;
```

3. Handler

- Si se produce una circunstancia para la que se ha declarado ningún controlador, la acción depende de la clase de condición:
  - SQLEXCEPTION, el programa almacenado termina en la declaración que elevó la condición. Si el programa es llamado por otro programa almacenado, el programa de llamada se encarga de la condición utilizando las reglas del controlador aplicadas a sus propios controladores.
  - SQLWARNING, el programa continúa ejecutandose, como si no hubiera un manejador CONTINUE.
  - NOT FOUND, si la condición se elevó normalmente, la acción es CONTINUE . Si fue criado por SIGNAL o RESIGNAL , la acción es EXIT .

3. Handler

- Handler: Devolver un mensaje si ha habido error

```
CREATE TABLE article_tags(
   article_id INT,  tag_id     INT,
   PRIMARY KEY(article_id,tag_id)
);

DELIMITER $$
CREATE PROCEDURE insert_article_tags(IN article_id INT, IN tag_id INT)
BEGIN

 DECLARE CONTINUE HANDLER FOR 1062
 SELECT CONCAT('duplicate keys (',article_id,',',tag_id,') found') AS msg;

 -- insert a new record into article_tags
 INSERT INTO article_tags(article_id,tag_id)
 VALUES(article_id,tag_id);
 -- return tag count for the article
 SELECT COUNT(*) FROM article_tags;
END
```

```
CALL insert_article_tags(1,1);
CALL insert_article_tags(1,2);
CALL insert_article_tags(1,3);

……
CALL insert_article_tags(1,3);
```

¿Problema?

3. Handler

- Handler: Devolver un mensaje si ha habido error

```
DELIMITER $$

CREATE PROCEDURE insert_article_tags_2(IN article_id INT, IN tag_id INT)
BEGIN

DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate keys error encountered';
 DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'SQLException encountered';
 DECLARE EXIT HANDLER FOR SQLSTATE '23000' SELECT 'SQLSTATE 23000';

 -- insert a new record into article_tags
 INSERT INTO article_tags(article_id,tag_id)
   VALUES(article_id,tag_id);

 -- return tag count for the article
 SELECT COUNT(*) FROM article_tags;
END
```

Handler: Ejercicios

- Crear un procedimiento para actualizar la población de un determinado país. Se pasarán dos parámetros, la nueva población de tipo float y el nombre del país.
  - Realiza el procedimiento primero sin hacer el tratamiento de errores con los siguientes valores
    - ✓ call ('Alicia',10);
    - ✓ call ('Angola',1000);
    - ✓ Call('Angola', 1234567891234) ¿Qué ocurre y porqué? → No permite la población pues supera el valor permitido que es int(11)
  - Realiza ahora el tratamiento de los errores de modo que si se introduce un valor para la población mayor del permitido, se actualizará la población de ese país aumentándola un 10%. Si se introduce un país que no existe, se acabará el procedimiento con un mensaje indicando que el país no existe.
- Definir un procedimiento almacenado que reciba el código y la población y llame al anterior procedimiento almacenado

Handler: Ejercicios

- Realiza un procedimiento en el que se pasen como parámetros el Code, nombre y Continente de un Pais y lo inserte en la tabla Country.
  - ¿qué ocurre si tratas de insertar un pais con un identificador que ya existe?
  - Modifica el procedimiento anterior tratándose esta condición (clave duplicada).
  - En ese caso el procedimiento deberá finalizar y mostrar el mensaje el identificador de pais ya existe.

  - call insertarPais('ABW',"Aruba","North America"); //Existe
  - call insertarPais('AWW',"Arubaito","North America"); //No existe

Handler: Ejercicios

- Crea un procedimiento para añadir un nuevo registro a la tabla city haciendo el tratamiento de errores y mostrando los mensajes pertinentes.
  - Aplica el procedimiento con los siguientes datos para comprobar el funcionamiento del procedimiento:
    - ✓ set @valor:= (select MAX(ID) FROM City);
    - ✓ call insertarCiudad(1, 'Merida','AAA');
    - ✓ call insertarCiudad(601, "Merida", "ESP");
    - ✓ call insertarCiudad(@valor+1, "Merida", "ESP");
- Crea un procedimiento para añadir un nuevo registro a la tabla city haciendo el tratamiento de errores pero en el caso que exista el ID de la ciudad, se inserte con el valor inmediatamente superior al valor mayor de las claves

4. Cursores

- **Cursores**: Como se ha dicho anteriormente, la sentencias DML de los procedimientos almacenados son recuperadas en la llamada
  - Pero no puede ser usada en otro procedimiento almacenado → Tabla temporal
  - Aunque si el resultado es una única fila, si se puede recuperar
  - Si se desea procesar varías filas → Cursores

Definición

```
DELIMITER //
CREATE PROCEDURE CiudadMadrid()
  BEGIN
    select Name,Population
           into @Name, @Population
           from City
           where CountryCode='ESP' and id=653;

  END //
DELIMITER ;
```

Ejecución

```
call CiudadMadrid;
select @Name, @Population
```

4. Cursores

- Cursores
  - Contiene un conjunto de filas resultantes de una consulta SQL que nos permite recorrer y manipular una a una cada una de esas filas.
  - Son de solo lectura y pueden recorrerse únicamente en una dirección.

- Los pasos habituales para trabajar con cursores
  - Definición:
  - Apertura del cursor: OPEN
  - Un Handler que nos alerte cuando hayamos llegado al último registro.
  - Un bucle que nos permita iterar a través del ResultSet devuelto por el cursor.
  - Cierre del cursor

4. Cursores

- Cursores: Si hacer recorrido

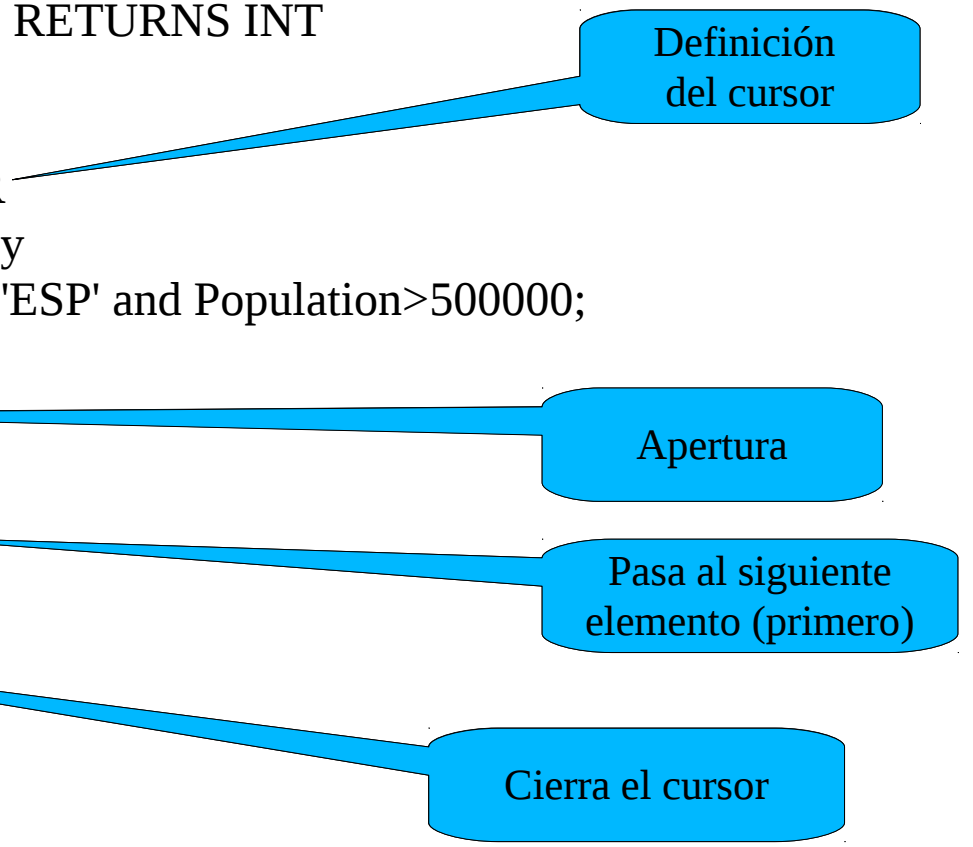
```
DELIMITER //
DROP function IF EXISTS countCityList //
CREATE FUNCTION countCityList() RETURNS INT
BEGIN
   DECLARE valor integer;
   DECLARE city_cur CURSOR FOR
          select count(Name) from City
                 where CountryCode='ESP' and Population>500000;

   OPEN city_cur;

   FETCH city_cur INTO valor;

   CLOSE city_cur;

   RETURN valor;
END
//
DELIMITER ;
SELECT countCityList() AS cities;
```



4. Cursores

Cursores

```
DELIMITER //
CREATE FUNCTION city_list() RETURNS VARCHAR(255)
BEGIN
   DECLARE city_name VARCHAR(50) DEFAULT "";
   DECLARE list VARCHAR(255) DEFAULT "";
   DECLARE city_cur CURSOR FOR
            select Name from City where CountryCode='ESP' and Population>500000;

   OPEN city_cur;

   get_city: LOOP
      FETCH city_cur INTO city_name;
      SET list = CONCAT(list,", ",city_name);
   END LOOP get_city;

   CLOSE city_cur;

   RETURN SUBSTR(list,3);
END
//
DELIMITER ;
SELECT city_list() AS cities;
```

Definición del cursor

Apertura

Bucle

Cierra el cursor

Problema: No hay control de fin de cursor

4. Cursores

Cursores

```
DELIMITER //
CREATE FUNCTION city_list() RETURNS VARCHAR(255)
BEGIN
   DECLARE finished INTEGER DEFAULT 0;
   DECLARE city_name VARCHAR(50) DEFAULT "";
   DECLARE list VARCHAR(255) DEFAULT "";
   DECLARE city_cur CURSOR FOR
            select Name from City where CountryCode='ESP' and Population>500000;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

   OPEN city_cur;
    get_city: LOOP
         FETCH city_cur INTO city_name;
         IF finished THEN
              LEAVE get_city;
         END IF;
         SET list = CONCAT(list,", ",city_name);
     END LOOP get_city;
   CLOSE city_cur;
   RETURN SUBSTR(list,3);
END
//
```

Definición de la bandera

Definición Del handler

Control del fin

4. Cursores

- Cursores: Implementación con un bucle repeat

```
DELIMITER //
CREATE FUNCTION city_list() RETURNS VARCHAR(255)
BEGIN
  DECLARE finished INTEGER DEFAULT 0;
   DECLARE city_name VARCHAR(50) DEFAULT "";
   DECLARE list VARCHAR(255) DEFAULT "";
   DECLARE city_cur CURSOR FOR
            select Name from City where CountryCode='ESP' and Population>500000;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
   OPEN city_cur;
   repeat
      FETCH city_cur INTO city_name;
      IF not finished THEN
            SET list = CONCAT(list,", ",city_name);
      END IF;
   UNTIL finished=1
   END REPEAT;
   CLOSE city_cur;
   RETURN SUBSTR(list,3);
END
//
DELIMITER ;
SELECT city_list() AS cities;
```

4. Cursores: Ejercicios

- Crea un procedimiento almacenado que suma la población de todas las ciudades de España
- Crea un procedimiento que muestre para cada continente, los nombres y la población de los 5 paises más poblados.
  - Utiliza un cursor que recorra los distintos continentes que aparecen en la tabla country.
  - Observa que para cada fetch, mysql muestra el resultado en una ventana nueva.

5. Disparadores

- Los triggers o disparadores son procedimientos que se ejecutan como respuesta a un evento producido sobre una tabla concreta de la base de datos.
  - Los eventos causantes del trigger son las operaciones INSERT, UPDATE o DELETE.
- Los triggers son ejecutados automáticamente por la base de datos, a diferencia de las funciones y los procedimientos que es el propio usuario.
- Cuando se crea un trigger, debemos especificar su nombre, la tabla a la que se asocia, y la operación que hará que se ejecute el trigger.
- Además, hay que decir en qué momento queremos que se ejecute el código del trigger: antes de realizar la operación INSERT, UPDATE o DELETE que lo desencadena, o después.

5. Disparadores

- Sintaxis:
  - TriggerTime: especifica cuándo debe ejecutarse el código asociado al trigger. Admite los valores BEFORE y AFTER.
  - Trigger_event: operación que desencadena el trigger: INSERT, UPDATE o DELETE.

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

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }
```

5. Disparadores

- Ejemplo: Comprobación de integridad referencial en Tablas MyISAM
  - Al añadir una ciudad, que exista el País
  - Al borrar un país, borrar sus ciudades

```
DELIMITER //
DROP TRIGGER IF EXISTS City_bi //
CREATE TRIGGER City_bi BEFORE INSERT ON City
FOR EACH ROW
BEGIN
 IF (NOT EXISTS (SELECT NULL FROM Country WHERE Code=NEW.CountryCode)) THEN
  SELECT 0 FROM `CountryCode does not exist in Country table` INTO @error;
 END IF;
END //
DROP TRIGGER IF EXISTS Country_ad //
CREATE TRIGGER Country_ad AFTER DELETE ON Country
FOR EACH ROW
BEGIN
 DELETE FROM City WHERE CountryCode = OLD.Code;
END //
DELIMITER ;
```

5. Disparadores

- Ejemplo: Un acumulador del dinero que se va a insertando en una cuenta

```
CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));

CREATE TRIGGER ins_sum BEFORE INSERT ON account
        FOR EACH ROW SET @sum = @sum + NEW.amount;

SET @sum = 0;

INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
SELECT @sum AS 'Total amount inserted';
```

5. Disparadores

- Ejemplo: Se quiere almacenar en una tabla temporal todas las ciudades que son borradas

```
DROP TRIGGER IF EXISTS City_AD;
DROP TABLE IF EXISTS DeletedCity;

CREATE TABLE DeletedCity (
        ID INT UNSIGNED,
        Name VARCHAR(50),
        DeleteDate TIMESTAMP);

DELIMITER //
CREATE TRIGGER City_AD AFTER DELETE ON City  FOR EACH ROW
BEGIN
        INSERT INTO DeletedCity (ID, Name) VALUES (OLD.ID, OLD.Name);
END// DELIMITER;

SHOW TRIGGERS;
DELETE FROM City WHERE Name = 'Dallas';
SELECT * FROM City WHERE Name = 'Dallas';
SELECT * FROM DeletedCity;
```

5. Disparadores

- Crear un trigger de modo que cada vez que se haga una operación de inserción sobre la tabla country, automáticamente se calcule por cada continente correspondiente a ese país, el número de países y la media de la población de esos países.
  - Se introducirán en una tabla temporal estadística (id int primary key auto_increment, continente varchar(50), numero int, media float)
  - insert into Country values ('XXY','Pais??','Asia','dd',18,1901,11111,99,100,1000,'hol','dd','dd',1111,'A3')

```
CREATE TRIGGER AFTER INSERT ON Country for each row begin
declare campo1, campo2 integer;
select count(*), avg(Population) into campo1, campo2 from Country
where Continent=new.Continent;
…..
```

- Crear un disparador que compruebe que la población de una ciudad no puede ser superior a la de su país. Se debe controlar cuando se actualice o inserte la población de puna ciudad. Se devolverá el resultado en una variable del usuario (@error)

6. Gestión de vistas

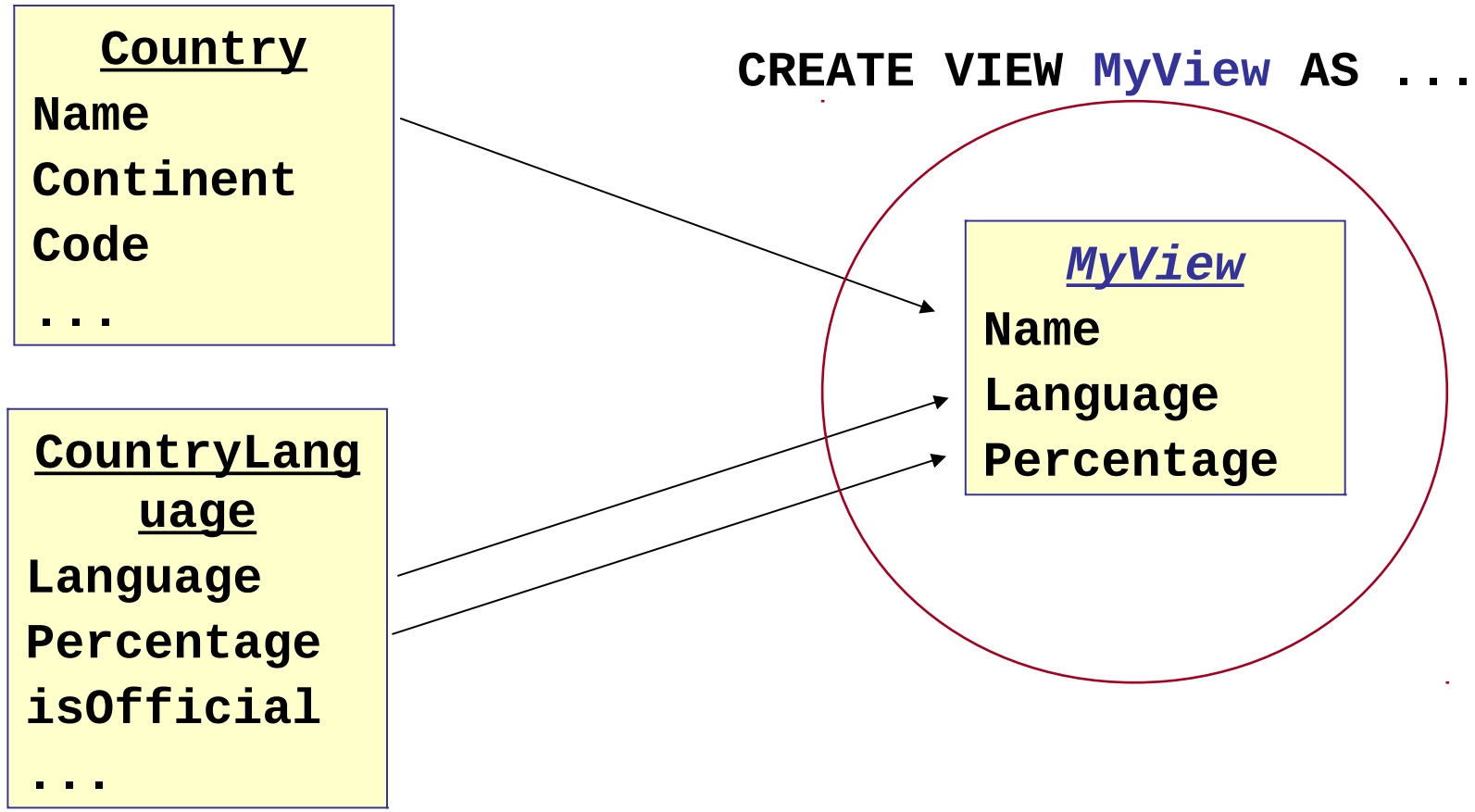
- Una vista es una tabla virtual cuyo contenido está definido por una consulta.
- Una vista es sencillamente un objeto de base de datos que presenta datos de tablas.
- Se trata de una consulta SQL que está permanentemente almacenada en la Base de datos y a la que se le asigna un nombre, de modo que los resultados de la consulta almacenada son visibles a través de la vista
- SQL permite acceder a estos resultados como si fueran de hecho una tabla real en la base de datos.
- Las tablas y las vistas comparten el mismo espacio de nombres en la base de datos, por lo tanto, una base de datos no puede contener una tabla y una vista con el mismo nombre.

6. Gestión de vistas

- Ventajas
  - Seguridad y confidencialidad: permiten a los usuarios obtener acceso a los datos por medio de la vista, pero no les conceden el permiso de obtener acceso directo a las tablas subyacentes de la vista.
  - Facilidad de uso para consultas “complejas”: Las vistas suelen utilizarse para centrar, simplificar y personalizar la percepción de la base de datos para cada usuario.
  - Particionar datos: También se pueden utilizar para realizar particiones de datos.
  - Nexo de unión: Mediante vistas es posible presentar datos de distintos servidores.
- Una diferencia entre vistas y procedimientos almacenados
  - Es que las primeras no aceptan parámetros, no siendo así con los procedimientos almacenados, que si los aceptan.
  - Un procedimiento almacenado suele utilizarse cuando no es suficiente una simple consulta sql. Los procedimientos almacenados contienen variables, bucles y llamadas a otros procedimientos almacenados.

6. Gestión de vistas

- Por ejemplo

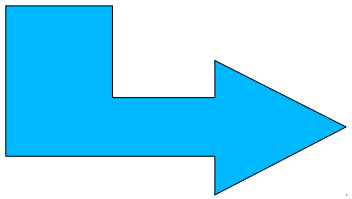


6. Gestión de vistas

- Sintaxis:
  - Para poder crear vistas se requiere el privilegio CREATE VIEW así como privilegios de acceso a las columnas que forman parte de la vista.

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

```
CREATE OR REPLACE VIEW datosInformatica AS
Select * from alumnos
Where cod_titulacion='1514'
```



```
Select * from datosInformatica
Where nombre = 'Juan'
```

6. Gestión de vistas

- Opciones
  - ALGORITHM
    - ✓ MERGE, las consultas que se ejecutan sobre la vista se combinan con la consulta de definición de la vista para crear una nueva consulta.
    - ✓ TEMPTABLE, se utiliza la definición de la vista para crear una tabla temporal donde se almacenan los datos (y por tanto las consultas se ejecutan sobre esta tabla)
    - ✓ UNDEFINED: MySQL escoge la opción más beneficiosa (Por defecto)
  - DEFINER (a partir de MySQL 5.1.2) especifica el propietario de la vista. Por defecto es el usuario actual (CURRENT_USER).
  - SQL SECURITY (a partir de MySQL 5.1.2)
    - ✓ Se puede precisar que las consultas que se efectúen sobre la vista se traten con los privilegios del propietario de la vista (DEFINER)
    - ✓ Los privilegios de quien ejecute las consultas sobre la vista (INVOKER);
  - WITH CHECK OPTION añade una opción de seguridad a la hora de realizar operaciones de manipulación de datos utilizando vistas
    - ✓ LOCAL los datos que se inserten a través de la vista deben respetar solamente las restricciones de la vista.
    - ✓ CASCADED en el momento de la creación de la vista entonces deben respetarse las restricciones de la vista fuente o “padre” (por defecto)

6. Gestión de vistas

- Particularidades:
  - Para conocer que vistas hay definidas sobre una BD
    - ✓ SELECT TABLE_NAME,IS_UPDATABLE FROM information_schema.`VIEWS`;
    - ✓ SHOW FULL TABLES IN database_name WHERE TABLE_TYPE LIKE 'VIEW';
    - ✓ SELECT TABLE_NAME FROM information_schema.`TABLES` WHERE TABLE_TYPE LIKE 'VIEW' AND TABLE_SCHEMA LIKE 'sakila';
  - Para ver el código fuente de la vista
    - ✓ show create view actor_info;

6. Gestión de vistas

- Por ejemplo: Obtener una vista de los países que hablan una determinada lengua

```
CREATE VIEW lang  AS
  SELECT name, language, percentage FROM Country C, CountryLanguage L
        WHERE C.code = L.countrycode ORDER BY language ASC;

SELECT * FROM lang WHERE language='Spanish';
```

6. Gestión de vistas

- Ejercicio: Crear una vista que muestre los siguientes campos:
  - City.name as name
  - Country.name as country
  - Region
  - Population of the city
  - Official language
  - id of the city

- Ejercicio: Crear una vista que muestre las ciudades en Southest Asia donde el Inglés es el idioma oficial y la población es más de 100.000.

6. Gestión de vistas

- Sintaxis de modificación (similar a la creación)
  - Lo que hace es modificar la especificación de la vista.
  - Realmente es como si se borrara y se volviera a crear con la nueva especificación de la sentencia SELECT
  - Necesita de los privilegios de: CREATE VIEW and DROP

```
ALTER
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

- En el caso de borrado de una vista

```
DROP VIEW [IF EXISTS] view_name [, view_name] ...
    [RESTRICT | CASCADE]
```

6. Gestión de vistas

- Restricciones
  - Habitualmente se prefiere MERGE, si es posible, porque MERGE es generalmente más eficientes y porque el uso de una tabla temporal impide una vista actualizable. Merge no puede usar:
    - ✓ Funciones agregadas(SUM(), MIN(), MAX(), COUNT(), …), DISTINCT, GROUP BY, HAVING, LIMIT, UNION or UNION ALL, subconsultas
  - Una razón para elegir TEMPTABLE explícitamente es que los bloqueos pueden ser liberados en las tablas subyacentes después de la tabla temporal haya sido creada y antes de que se utilice para terminar de procesar la sentencia.
  - La sentencia SELECT de creación de la vista no puede hacer referencia en la cláusula FROM ni a una tabla temporal (TEMPORARY TABLE) ni contener una subconsulta.
  - La cláusula WHERE no debe incluir una subconsulta.
  - La consulta no debe incluir una cláusula GROUP BY o HAVING.

6. Gestión de vistas

- El principal problema con las vistas es la actualización de los datos bien por inserciones, actualizaciones o borrados.
  - Si una vista está basada en una sola tabla, al modificar la vista se está modificando directamente la tabla (en estos casos habitualmente no hay problema)

```
CREATE TABLE t1 (a INT);
 CREATE VIEW v1 AS SELECT * FROM t1 WHERE a < 2
    WITH CHECK OPTION;
```

UPDATE T1 SET a=0 where a=1

UPDATE V1 SET a=0 where a=1

6. Gestión de vistas

- El principal problemas con las vistas es la actualización de los datos. Restricciones:
  - Contiene funciones de agregado como SUM(), MIN(), MAX(), COUNT()
  - Contiene la cláusula DISTINCT o GROUP BY o HAVING o UNION o JOIN u ORDER BY
  - Una consulsta en la cláusula FROM o una vista no actualizable
  - Una subconsulta en la cláusula WHERE que se refiere a una tabla en la cláusula FROM
  - Se refiere sólo a valores literales (en este caso, no hay ninguna tabla subyacente para actualizar)
  - Utiliza ALGORITHM = TEMPTABLE (uso de una tabla temporal siempre hace una vista no actualizable)
  - Múltiples referencias a cualquier columna de una tabla base.

6. Gestión de vistas

- Inserciones en vistas:
  - Los campos de la operación “insert into” sobre la vista implica que el resto de campos se añaden con null (si hubiera alguna restricción daría un fallo)
  - Siempre que no nos dejemos de rellenar los campos obligatorios de la tabla base podremos hacer inserciones a través de la vista.
  - Si una tabla contiene una columna AUTO_INCREMENT, la inserción en una vista insertable no debe incluir esta columna en la vista.

6. Gestión de vistas

- With check-option: Realiza la comprobación de las operaciones

```
CREATE TABLE t1 (a INT);
 CREATE VIEW v1 AS SELECT * FROM t1 WHERE a < 2
    WITH CHECK OPTION;
 CREATE VIEW v2 AS SELECT * FROM v1 WHERE a > 0
     WITH LOCAL CHECK OPTION;
 CREATE VIEW v3 AS SELECT * FROM v1 WHERE a > 0
    WITH CASCADED CHECK OPTION;
```

```
mysql> INSERT INTO v2 VALUES (2);
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO v3 VALUES (2);
ERROR 1369 (HY000): CHECK OPTION failed 'test.v3'
```

Con el check option da error la segunda porque su valor es 2 y v1 no puede almacenar valores <2