

Bases de Datos 1

Alejandra Lliteras

alejandra.lliteras@lifa.info.unlp.edu.ar



MySQL

Stored Procedures

- Conjunto de sentencias SQL que se almacenan el servidor
- Se lo crea indicando
 - Nombre
 - Parámetros (de ser necesario)
 - IN, OUT, INOUT
 - Cuerpo del Procedimiento almacenado
- Se lo invoca mediante
 - `CALL nombreSP`
- Eliminación
 - Al borrar toda la base, los stored procedures se eliminan
 - Drop
 - `DROP PROCEDURE IF EXISTS nombreSP`

Stored Procedures

- **Acceso homogéneo:**
 - Cuando es necesario realizar las mismas operaciones en la base de datos desde diferentes aplicaciones clientes, escritas en diferentes lenguajes y quizás hasta en diferentes plataformas.
- **Asegurar consistencia en las operaciones**
 - Escribiendo una vez y asegurando para cada ejecución que la secuencia de instrucciones retornara de manera consistente los resultados. Puede usarse para no permitir acceder a tablas directamente, sino que solo por medio de estas rutinas
- **Pueden ayudar a la performance**
 - porque se disminuye el tráfico entre el servidor y el cliente
 - Sin embargo, incrementa la actividad en el servidor de la bases de datos ya que el trabajo se ejecuta en el
 - Esto empeora si múltiples cliente peticionan a un solo servidor
- **Consultas complejas**
 - Es posible usar estructuras y funciones adicionales para resolver consultas complejas dentro de un sp

Stored Procedures

- ▶ Privilegios que tiene que tener el usuario para:
 - Crearlo
 - CREATE ROUTINE
 - Modificarlo
 - ALTER ROUTINE
 - Ejecutarlo
 - EXECUTE

```
CREATE USER 'usuarioAdministrativo'@'localhost' IDENTIFIED BY '1234';  
Grant create routine on pruebajson.* to 'usuarioAdministrativo'@'localhost';  
revoke create routine on pruebajson.* from 'usuarioAdministrativo'@'localhost';|
```


```
CREATE TABLE `pruebasjson`.`empleado` (  
  `nroEmpleado` INT NOT NULL ,  
  `nombre` VARCHAR(45) NULL ,  
  `apellido` VARCHAR(45) NULL ,  
  `edad` INT NULL ,  
  PRIMARY KEY (`nroEmpleado`));
```

```
ALTER TABLE `pruebasjson`.`empleado` CHANGE COLUMN `nroEmpleado`  
  `nroEmpleado` INT(11) NOT NULL AUTO_INCREMENT ;
```

```
INSERT INTO `pruebasjson`.`empleado` (`nombre`, `apellido`, `edad`)  
VALUES ('Juan', 'Pedro', 25);
```

```
INSERT INTO `pruebasjson`.`empleado` (`nombre`, `apellido`, `edad`)  
VALUES ('Maria ', 'Lopez', 35);
```

```
INSERT INTO `pruebasjson`.`empleado` (`nombre`, `apellido`, `edad`)  
VALUES ('José', 'Gonzalez', 29);
```



SP –Parámetro IN

- ▶ Son parámetros de entrada
- ▶ Se puede usar y modificar su valor dentro del SP, pero los cambios no se verán reflejados fuera de este

```
1 delimiter //
2 • create procedure recuperarEmpleado (in nroEmpleadoParam int)
3 begin
4 select * from empleado where nroEmpleado = nroEmpleadoParam;
5 end//
6
7
```

```
call recuperarEmpleado(3)
```

	nroEmpleado	nombre	apellido	edad
▶	3	José	Gonzalez	29

SP –Parámetro OUT

- ▶ Son parámetros de salida
- ▶ Se puede asignar un valor dentro del SP, y usarlo dentro del mismo. Los cambios se verán reflejados fuera del SP

```
DROP procedure IF EXISTS `pruebajson`.`parametrosSalida2`;  
  
DELIMITER //  
CREATE PROCEDURE `pruebajson`.`parametrosSalida2` (out valor int)  
BEGIN  
set valor = 6 *2;  
END//  
  
delimiter ;
```

```
set @valor = 8;  
CALL parametrosSalida2 (@valor);  
select @VALOR;
```

	@VALOR
▶	12

SP –Parámetro OUT

- ▶ Son parámetros de salida
- ▶ Se puede asignar un valor dentro del SP, y usarlo dentro del mismo. Los cambios se verán reflejados fuera del SP

```
DELIMITER //
```

```
CREATE PROCEDURE `pruebajson`.`parametrosSalida` (out valor int)
```

```
BEGIN
```

```
select count(*) into valor from empleado;
```

```
END//
```

```
delimiter ;|
```

```
call parametrosSalida (@param);
```

```
SELECT @param|
```

	@param
▶	3

SP –Parámetro INOUT

- ▶ Son parámetros de entrada salida
- ▶ Se puede usar y modificar su valor dentro del SP, y los cambios se verán reflejados fuera de este

```
DELIMITER //
```

```
CREATE PROCEDURE `pruebajson`.`parametrosEntradaSalida` (inout valor int)  
BEGIN  
  set valor = valor *2;  
END//  
delimiter ;
```

```
set @valor = 8;  
CALL parametrosEntradaSalida (@valor);  
CALL parametrosEntradaSalida (@valor);  
select @VALOR;
```

	@VALOR
▶	32

SP – Función LAST_INSERT_ID

- ▶ Retorna el valor del último autoincremental agregado inmediatamente anterior a su invocación
- ▶ Si la inserción es errónea, el valor que retorna es indefinido

```
DROP procedure IF EXISTS `pruebajson`.`pruebaLastInsert`;

DELIMITER //
CREATE PROCEDURE `pruebajson`.`pruebaLastInsert` (out ultimoValorInsertado int)
BEGIN
    insert into empleado (nombre, apellido, edad) values ("Rosa", "Gimenez", 38);
    select LAST_INSERT_ID() into ultimoValorInsertado;
END//

delimiter ;
```

```
CALL pruebaLastInsert (@valor);
select @VALOR;
```

	@VALOR
▶	4

16 • `select * from empleadd`

	nroEmpleado	nombre	apellido	edad
▶	1	Juan	Pedro	25
	2	Maria	Lopez	35
	3	José	Gonzalez	29
	4	Rosa	Gimenez	38

Sp- Estructuras de control

```
delimiter //
CREATE PROCEDURE estructuraDeControl()
BEGIN
    DECLARE cantidadDeIteraciones INT; -- declaracion de un a variable de tipo entero
    SET cantidadDeIteraciones = 0; -- variable cantidadDeIteraciones con valor cero
    loop_label: LOOP -- el loop se indica con la palabra clave y una etiqueta (loop_label)

        SET cantidadDeIteraciones = cantidadDeIteraciones + 1; -- incremento en una la cantidad de iteraciones
        IF cantidadDeIteraciones = 5 THEN
            ITERATE loop_label; -- si iteró 5 veces, se detiene en este punto y regresa hasta la etiqueta definida
        END IF; -- del if que controla cantidadDeIteraciones = 5

        SELECT cantidadDeIteraciones;

        IF cantidadDeIteraciones >= 6 THEN
            LEAVE loop_label; -- termina el ciclo loop
        END IF; -- fin del if que controla cantidadDeIteraciones >= 6
    END LOOP; -- fin del loop
END; // -- fin del sp

call estructuraDeControl();
```

Sp- Estructuras de control

```
delimiter //
CREATE PROCEDURE estructuraDeControl()
BEGIN
    DECLARE cantidadDeIteraciones INT; -- declaracion de un a
    SET cantidadDeIteraciones = 0; -- variable cantidadDeIter
    loop_label: LOOP -- el loop se indica con la palabra clav

        SET cantidadDeIteraciones = cantidadDeIteraciones + 1;
        IF cantidadDeIteraciones = 5 THEN
            ITERATE loop_label; -- si iteró 5 veces, se detiene
        END IF; -- del if que controla cantidadDeIteraciones =

        SELECT cantidadDeIteraciones;

        IF cantidadDeIteraciones >= 6 THEN
            LEAVE loop_label; -- termina el ciclo loop
        END IF; -- fin del if que controla cantidadDeIteracion
    END LOOP; -- fin del loop
END; // -- fin del sp

call estructuraDeControl();
```

Transacciones

```
start transaction;
```

```
insert into empleado (nombre, apellido, edad) values ("Rosalina", "Moldes", 27);  
update empleado set edad = 29 where nombre = "Rosalina" and apellido = "Moldes";  
select * from empleado;
```

```
commit;  
rollback;
```

	nroEmpleado	nombre	apellido	edad
►	1	Juan	Pedro	25
	2	Maria	Lopez	35
	3	José	Gonzalez	29
	4	Rosa	Gimenez	38
	5	Rosalina	Moldes	29

SP- Transacciones

```
• DROP procedure IF EXISTS `pruebajson`.`pruebaLastInsert`;  
  
DELIMITER //  
• CREATE PROCEDURE `pruebajson`.`pruebaLastInsert` (out ultimoValorInsertado int)  
  BEGIN  
    start transaction;  
  
    insert into empleado (nombre, apellido, edad) values ("Rosa", "Gimenez", 38);  
    select LAST_INSERT_ID() into ultimoValorInsertado;  
  
    commit;  
    rollback;  
  
  END//  
  
delimiter ;
```

SP-Cursores

- ▶ Permiten guardar en ellos valores obtenidos de ejecutar una sentencia SQL
- ▶ Es posible recorrerlos e ir recuperando de a uno sus valores
- ▶ Operaciones:
 - **DECLARE:** permite declarar un cursor
 - **OPEN:** permite abrir un cursor que ya ha sido declarado
 - **FETCH:** permite recuperar el valor de un cursor que ya ha sido abierto previamente
 - **CLOSE:** permite cerrar un cursor que ha sido al menos, declarado previamente.


```

drop procedure if exists `pruebajson`.`iteracionDeCursor`;
delimiter //
create procedure iteracionDeCursor()
begin
    declare fin int default 0;
    declare nombreC varchar(250);
    declare apellidoC varchar(250);
    declare nroEmpleadoC int;
    declare empleadoReducido cursor for -- declaracion del cursor que se carga con los resultados del select
        select nombre,apellido,nroEmpleado from empleado
        where edad > 20 ;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1; -- para manejar la excepcion de cuando se
    -- termine de iterar el cursor
start transaction;
CREATE TABLE if not exists `pruebajson`.`ejemplo`
( `idEjemplo` INT NOT NULL AUTO_INCREMENT ,

    `nombreApellido` VARCHAR(700) not NULL ,
    PRIMARY KEY (`idEjemplo`)
);
open empleadoReducido; -- abro el cursor
ciclo_loop: LOOP -- inicio el loop y pongouna etiqueta
    fetch empleadoReducido into nombreC,apellidoC,nroEmpleadoC;
    -- recupero los valores, en variables, de una tupla guardada en el cursor

    if ( fin=1) then -- quiere decir que el fetch anterior levanto una excepcion por no tener mas tuplas
        LEAVE ciclo_loop;-- Termina el loop
    end if;
    insert into `pruebajson`.`ejemplo` (nombreApellido) values (concat (nombreC,apellidoC));

end LOOP ciclo_loop ;
CLOSE empleadoReducido;
select * from `pruebajson`.`ejemplo`;
commit;
rollback;
end //

```

```

drop procedure if exists `pruebajson`.`iteracionDeCursor`;
delimiter //
create procedure iteracionDeCursor()
begin
    declare fin int default 0;
    declare nombreC varchar(250);
    declare apellidoC varchar(250);
    declare nroEmpleadoC int;
    declare empleadoReducido cursor for -- declaracion del cursor que se carga con los datos
        select nombre,apellido,nroEmpleado from empleado
        where edad > 20 ;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1; -- para manejar la excepcion de
    -- termine de iterar el cursor

    ( `idEjemplo` INT NOT NULL AUTO_INCREMENT ,
      `nombreApellido` VARCHAR(700) not NULL ,
      PRIMARY KEY (`idEjemplo`)
    );
    open empleadoReducido; -- abro el cursor
    ciclo_loop: LOOP -- inicio el loop y pongouna etiqueta
        fetch empleadoReducido into nombreC,apellidoC,nroEmpleadoC;
        -- recupero los valores, en variables, de una tupla guardada en el cursor

        if ( fin=1) then -- quiere decir que el fetch anterior levanto una excepcion por no tener mas tuplas
            LEAVE ciclo_loop;-- Termina el loop
        end if;
        insert into `pruebajson`.`ejemplo` (nombreApellido) values (concat (nombreC,apellidoC));

    end LOOP ciclo_loop ;
    CLOSE empleadoReducido;
    select * from `pruebajson`.`ejemplo`;
commit;
rollback;
end //

```

```

drop procedure if exists `pruebajson`.`iteracionDeCursor`;
delimiter //
create procedure iteracionDeCursor()
begin
    declare fin int default 0;
    declare nombreC varchar(250);
    declare apellidoC varchar(250);
    declare nroEmpleadoC int;
    declare empleadoReducido cursor for -- declaracion del cursor que se carga con los resultados del select
        select nombre,apellido,nroEmpleado from empleado
        where edad > 20 ;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1; -- para manejar la excepcion de cuando se

start transaction;
    CREATE TABLE if not exists `pruebajson`.`ejemplo`
        ( `idEjemplo` INT NOT NULL AUTO_INCREMENT ,

            `nombreApellido` VARCHAR(700) not NULL ,
            PRIMARY KEY (`idEjemplo`)
        );
    open empleadoReducido; -- abro el cursor

    if ( fin=1) then -- quiere decir que el fetch anterior levanto una excepcion por no tener mas tuplas
        LEAVE ciclo_loop;-- Termina el loop
    end if;
    insert into `pruebajson`.`ejemplo` (nombreApellido) values (concat (nombreC,apellidoC));

end LOOP ciclo_loop ;
CLOSE empleadoReducido;
select * from `pruebajson`.`ejemplo`;
commit;
rollback;
end //

```

```

drop procedure if exists `pruebajson`.`iteracionDeCursor`;
delimiter //
create procedure iteracionDeCursor()
begin
    declare fin int default 0;
    declare nombreC varchar(250);
    declare apellidoC varchar(250);
    declare nroEmpleadoC int;
    declare empleadoReducido cursor for -- declaracion del cursor que se carga con los resultados del select
                                        select nombre,apellido,nroEmpleado from empleado
                                        where edad > 20 ;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1; -- para manejar la excepcion de cuando se
    -- termine de iterar el cursor
start transaction;
CREATE TABLE if not exists `pruebajson`.`ejemplo`
( `idEjemplo` INT NOT NULL AUTO_INCREMENT ,
  `nombreApellido` VARCHAR(700) not NULL ,
  PRIMARY KEY (`idEjemplo`)
);

ciclo_loop: LOOP -- inicio el loop y pongouna etiqueta
    fetch empleadoReducido into nombreC,apellidoC,nroEmpleadoC;
    -- recupero los valores, en variables, de una tupla guardada en el cursor

    if ( fin=1) then -- quiere decir que el fetch anterior levanto una excepcion por no tener
        LEAVE ciclo_loop;-- Termina el loop
    end if;
    insert into `pruebajson`.`ejemplo` (nombreApellido) values (concat (nombreC,apellidoC));

end LOOP ciclo_loop ;
select * from pruebajson . ejemplo ;
commit;
rollback;
end //

```



```

drop procedure if exists `pruebajson`.`iteracionDeCursor`;
delimiter //
create procedure iteracionDeCursor()
begin
    declare fin int default 0;
    declare nombreC varchar(250);
    declare apellidoC varchar(250);
    declare nroEmpleadoC int;
    declare empleadoReducido cursor for -- declaracion del cursor que se carga con los resultados del select
        select nombre,apellido,nroEmpleado from empleado
        where edad > 20 ;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1; -- para manejar la excepcion de cuando se
    -- termine de iterar el cursor
start transaction;
CREATE TABLE if not exists `pruebajson`.`ejemplo`
( `idEjemplo` INT NOT NULL AUTO_INCREMENT ,
    `nombreApellido` VARCHAR(700) not NULL ,
    PRIMARY KEY (`idEjemplo`)
);
open empleadoReducido; -- abro el cursor
ciclo_loop: LOOP -- inicio el loop y pongouna etiqueta
    fetch empleadoReducido into nombreC,apellidoC,nroEmpleadoC;
    -- recupero los valores, en variables, de una tupla guardada en el cursor

    if ( fin=1) then -- quiere decir que el fetch anterior levanto una excepcion por no tener mas tuplas
        LEAVE ciclo_loop;-- Termina el loop
    end if;
    insert into `pruebajson`.`ejemplo` (nombreApellido) values (concat (nombreC,apellidoC));
end LOOP ciclo_loop;

CLOSE empleadoReducido;
select * from `pruebajson`.`ejemplo`;

commit;
rollback;
end //

```

TRIGGER

- ▶ Son disparadores que se pueden usar
 - BEFORE (antes de)
 - AFTER (después de)

De las siguientes operaciones

- INSERT
 - DELETE
 - UPDATE
- 

TRIGGER

```
CREATE TRIGGER nombre_disp momento_disp evento_disp  
ON nombre_tabla FOR EACH ROW sentencia_disp
```

► Donde

- nombre_disp es el nombre que se le asigna
- momento_disp: BEFORE o AFTER
- evento_disparador: INSERT-DELETE-UPDATE
- nombre_tabla: tabla sobre la cual se generará el evento
- sentencia_disp: conjunto de sentencias del cuerpo del trigger

```
1 CREATE TABLE employees_audit (  
2     id int(11) NOT NULL AUTO_INCREMENT,  
3     employeeNumber int(11) NOT NULL,  
4     lastname varchar(50) NOT NULL,  
5     changedon datetime DEFAULT NULL,  
6     action varchar(50) DEFAULT NULL,  
7     PRIMARY KEY (id)  
8 )
```

```
1 DELIMITER $$  
2 CREATE TRIGGER before_employee_update  
3     BEFORE UPDATE ON employees  
4     FOR EACH ROW BEGIN  
5  
6     INSERT INTO employees_audit  
7     SET action = 'update',  
8         employeeNumber = OLD.employeeNumber,  
9         lastname = OLD.lastname,  
10        changedon = NOW();  
11 END$$  
12 DELIMITER ;
```

- ▶ Este trigger se ejecuta antes de efectivizar el update.
- ▶ Cuando el evento disparador es **UPDATE**, se puede usar:
 - **OLD.nombreColumna** (hace referencia al valor antes de actualizarse de la columna en cuestión)
 - **NEW.nombreColumna** (hace referencia al valor después de actualizarse de la columna en cuestión)

TRIGGER

- ▶ Cuando el evento disparador es **INSERT**, se puede usar:
 - NEW.nombreColumna
- ▶ Cuando el evento disparador es **DELETE**, se puede usar:
 - OLD.nombreColumna

TRIGGER

- ▶ Cómo eliminar un trigger
 - DROP TRIGGER nombreTrigger
 - Al eliminar una tabla, todos los triggers asociados son eliminados
- ▶ ¿Qué privilegios debe tener un usuario para crear triggers?

```
Grant TRIGGER on pruebajson.* to 'usuarioAdministrativo'@'localhost';
```

- ▶ Una columna OLD es de solo lectura y requiere privilegios de SELECT
- ▶ Una columna de NEW requiere privilegio de SELECT, pero además, es posible modificarla si se usa con BEFORE, para ello es necesario además privilegio de UPDATE

TRIGGER

► Limitaciones

- No pueden ejecutar un stored procedure (call)
- No pueden usar sentencias que explicita o implícitamente abran o cierren una transacción

TRIGGER

```
drop trigger if exists ida;
delimiter |
CREATE TRIGGER IDA BEFORE INSERT ON empleado
FOR EACH ROW
BEGIN
    DECLARE AUTOi INT;
    INSERT INTO deporte (NOMBREDEPORTISTA, NOMBREDEPORTE) VALUES (NEW.nombre , "TENIS");
    select LAST_INSERT_ID() into AUTOi;
    DELETE FROM deporte WHERE nombreDeporte = "FULTBOL";
    UPDATE deporte SET nombreDeporte = concat (nombreDeporte, "trigger") WHERE idDeporte <> AUTOi;
END;
|
```

```
delimiter ;
```

```
insert into `pruebajson`.`empleado` (nombre, apellido, edad) values ("Juan", "Zata", 19);
select * from deporte;
select * from empleado;
```

	iddeporte	nombreDeporte	nombreDeportista
	1	TENIStriggertrigger	Juan
	2	TENIStrigger	Juan
	3	TENIS	Juan

Vistas

- Habitualmente el modelo lógico suele ser muy complejo para el usuario.
- Existen consideraciones de seguridad para que un usuario no acceda al todo el modelo.
- En ocasiones es mejor darle al usuario una “versión” personalizada del modelo que se ajuste mejor a sus necesidades de consulta.
- No se debe permitir al usuario realizar operaciones sobre los datos (insert/update/delete).

Vistas

- Una vez creada, la definición de una vista es “congelada”. Esto significa que cambios posteriores a las tablas de la vista no afectarán la vista.
 - Por ejemplo, si luego de crear una vista con `SELECT *`, si luego se agregan nuevas columnas, éstas no aparecerán en la vista.
- Las vistas pertenecen a una base de datos, por lo que si se elimina la base, se elimina la vista.
- Los nombres de las columnas deben ser únicos.

Vistas

- No se pueden utilizar subqueries en la cláusula FROM.
- El SELECT no puede referirse a variables del usuario o del sistema.
- Si se crea dentro de un programa (sp) no se pueden utilizar los parámetros del programa.
- Cuando se está definiendo la vista, todas las tablas y/o otras vistas a las que se menciona deben existir.
- No se pueden utilizar tablas temporales ni crear vistas temporales.
- No se pueden asociar triggers con las vistas.
- Se pueden utilizar ORDER BY, pero se lo ignora si el select viene acompañado de uno propio.

Vistas

- Ejemplo

```
1 CREATE VIEW SalePerOrder
2   AS
3   SELECT orderNumber,
4   SUM (quantityOrdered * priceEach) total
5   FROM orderDetails
6   GROUP by orderNumber
7   ORDER BY total DESC
```

- Ejecución

```
1 SELECT total
2 FROM salePerOrder
3 WHERE orderNumber = 10102
```