MySQL: Vistas y Rutinas Almacenadas

Bases de Datos

Contenido

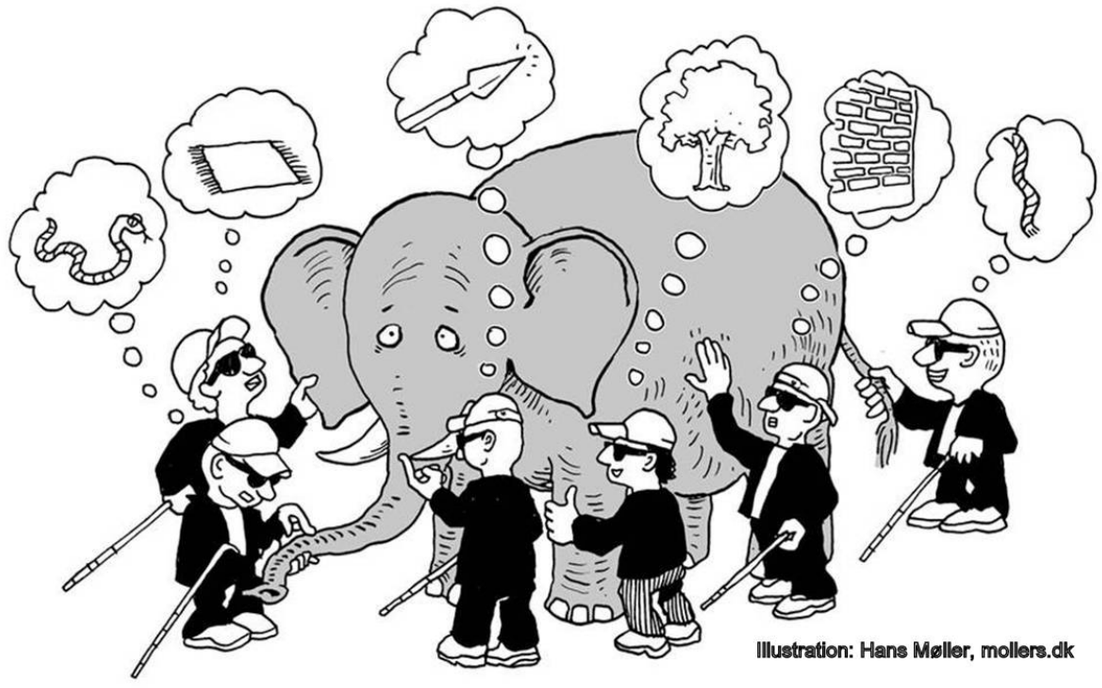
1. Views
2. Stored Procedures
3. Stored Function
4. Triggers

SQL: View

- Una **vista** es una consulta almacenada (**stored query**) que cuando se la invoca produce un conjunto de resultados.
- Una vista actúa como una **tabla virtual**. En contraste, una **relación** cuyo contenido realmente esta en la base de datos, se denomina **tabla ordinaria**
- Se puede crear una **vista** a partir de diferentes tipos de comandos **SELECT** que pueden referir a tablas ordinarias u otras vistas
- Sintaxis:
  CREATE VIEW <name> AS <query>;

SQL: View

- Permite presentar información almacenada de diferentes formas a diferentes usuarios
- Una vista puede ser adaptada a la necesidad del usuario
- Si cambia el esquema conceptual, sólo la consulta SELECT necesaria para construir la vista deberá ser cambiada



Ejemplo: View

- Supongamos que un administrador mantiene una lista de actividades de todos los empleados con la siguiente información: fname, lname, project_name, hours_worked
- Consulta SELECT normal:

```
SELECT fname, lname, pname project_name, hours hours_worked
FROM   employee, works_on, project
WHERE  ssn = essn AND pno = pnumber;
```

- Crear VIEW para el administrador:

```
CREATE  VIEW emp_activity AS
(SELECT fname, lname, pname project_name, hours hours_worked
FROM   employee, works_on, project
WHERE ssn = essn AND pno = pnumber);
```

View: Ventajas

- Se pueden usar las **vistas** como **tablas ordinarias** para definir consultas
- Cuando se utiliza una vista en una consulta SELECT, la tabla virtual se calcula en primer lugar
- Simplifica la definición de consultas complejas ocultándolas del usuario final y de las aplicaciones
- Limita el acceso de datos a usuarios específicos

  (exponiendo sólo los datos no sensibles) y proporciona seguridad adicional para el acceso de lectura/escritura
- Habilita la compatibilidad con versiones anteriores - los cambios en la base de datos no afectarán los cambios en otras aplicaciones.

View: Desventajas

- La consulta de datos a partir de una **vista** puede ser lenta (ya que la vista se calcula cada vez)

- Dependencia de las tablas - actualizaciones de las tablas subyacentes fuerzan cambios a la propia vista para que funcione correctamente

- No todas las vistas son **updatable** (se puede usar en ellas las sentencias INSERT, DELETE, UPDATE para actualizar los datos de las tablas subyacentes)

- Por ejemplo, una vista es no updatable si contiene funciones de agregación, DISTINCT, GROUP BY, etc.

Variables de Sesión

- Una **sesión** comienza con una conexión al servidor SQL y termina cuando se cierra la conexión

- Las **variables de sesión** pueden crease en cualquier momento durante la sesión SQL
  - ✓ Existe para el resto de la sesión SQL
  - ✓ Comienza siempre con el símbolo "@" (por ejemplo, @x, @count)

- No es parte del estándar SQL - por lo tanto pueden variar entre los proveedores

Sintaxis en MySQL: Variables de Sesión

- Asignar un valor
  - ✓ Sintaxis:
    SET <varName> = express;
  - ✓ Ejemplo: SET @count = 100;
- Asignar el resultado de una consulta que produce un solo valor a una variable de sesión
  - ✓ Sintaxis:
    SELECT … INTO @varname
    FROM …
    WHERE …
  - ✓ Ejemplo:
    SELECT max(salary) INTO @maxSal FROM employee;

Sintaxis en MySQL: Variables de Sesión (2)

- Usar una variable de sesión en una consulta

  ✓ Ejemplo:

```
SELECT fname, lname
FROM   employee
WHERE  salary = @maxSal;
```

View vs Temporary Table

- Una **vista** no es una tabla ordinaria, sino sólo una consulta almacenada
- Las **vistas** persisten más allá de una sesión
- Las **tablas temporarias** desaparecen una vez finalizada la sesión
- Las **tablas temporarias** son útiles si la consulta es "larga" y si se esta accediendo a los resultados de varias consultas
- Hacer un "tradeoff" entre **procesamiento** y **storage**

SQL: Stored Procedures

SQL: Stored Procedures

- Generalización de SQL añadiendo la **estructura de un lenguaje de programación** al lenguaje SQL
- Estructuras típicamente disponibles en procedimientos almacenados
  - ✓ Variables Locales
  - ✓ Sentencia IF
  - ✓ Sentencia LOOP
- La mayoría de los proveedores de bases de datos lo incorporan de alguna forma

Stored Procedure: Sintaxis

- Sintaxis:

  CREATE PROCEDURE <sp_name>([proc_parameter[,...]])
  BEGIN
      <routine_body>
  END <DELIMITER>

- proc_parameter:
  [IN | OUT | INOUT] param_name datatype

- <DELIMITER> es un símbolo especial usado por MySQL para finalizar una sentencia - por defecto es el ";"

- Un procedimiento almacenado sólo puede ser usado dentro de la base de datos donde fue definido

Ejemplo: Stored Procedure

- Defina un procedimiento para obtener el nombre y el apellido de todos los empleados

```
DELIMITER //
CREATE PROCEDURE GetAllEmployees()
BEGIN
   SELECT fname, lname  FROM employee;
END; //
DELIMITER ;
```

- Para almacenar el símbolo ";" dentro del procedimiento almacenado, necesitamos redefinir el símbolo delimitador usando el comando DELIMITER //

Llamada a un Stored Procedure

- Invocar (call) un procedimiento almacenado:
  CALL procedureName(parameters);

- Ejemplo:

```
mysql> CALL GetAllEmployees();
+----------+---------+
| fname    | lname   |
+----------+---------+
| John     | Smith   |
| Franklin | Wong    |
| Joyce    | English |
| Ramesh   | Narayan |
| James    | Borg    |
| Jennifer | Wallace |
| Ahmad    | Jabbar  |
| Alicia   | Zelaya  |
+----------+---------+
8 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Información del Stored Procedure

- Mostrar el nombre de los procedimientos almacenados:
  - ✓ Todos los procedimientos:
    SHOW PROCEDURE STATUS;
  - ✓ Sólo los procedimientos con un cierto nombre:
    SHOW PROCEDURE STATUS WHERE name LIKE <pattern>;

- Obtener la definición:
  SHOW CREATE PROCEDURE <procedure name>;

- Eliminar procedimiento almacenado de la base de datos:
  DROP PROCEDURE <procedure name>;

Detalles del Stored Procedure

- Un procedimiento puede tener muchas sentencias.
  - ✓ Ejemplo:

```
DELIMITER //
CREATE PROCEDURE GetAllEmpDepts()
BEGIN
    SELECT fname, lname FROM employee;
    SELECT dname, mgrssn FROM department;
END; //
DELIMITER;
```

- Una linea de comentario empieza con el símbolo --

  - ✓ Ejemplo:

```
-- Esto es una linea de comentario
```

Stored Procedure: Variable Local

- Una variable local sólo existe dentro del procedimiento almacenado (similar al de los lenguajes C o Python)
- No se utiliza el @ como prefijo en una variable local, como en la variable de sesión en MySQL
- Sintaxis:
  DECLARE <var_name> DATATYPE [DEFAULT value];

Ejemplo: Variable Local

```
DELIMITER //

CREATE PROCEDURE Variable1()

BEGIN

    DECLARE  myvar  INT ;

    SET myvar = 1234;

    SELECT concat('myvar = ', myvar ) ;

END; //

DELIMITER ;
```

Stored Procedure: Variables Locales (2)

- Similar al las variables de sesión, se puede asignar un valor o almacenar una consulta que produce un único valor

  ✓ Asignar un valor:

  SET <varname> = expression;

  ✓ Asignar el resultado de una consulta que retorna un valor

  SELECT … INTO <varname>
  FROM …
  WHERE …

- Las palabras claves BEGIN y END definen el alcance de las variables locales

Variable Local desde una consulta

```
DELIMITER //
CREATE PROCEDURE Variable2()
BEGIN
    DECLARE  myvar  INT ;
    SELECT sum(salary) INTO myvar
    FROM    employee
    WHERE  dno = 4;
    SELECT CONCAT('myvar = ', myvar );
END; //
DELIMITER ;
```

Ejemplo: Alcance de la Variable Local

```
DELIMITER //
CREATE PROCEDURE Variable3()
BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'outer';
    SELECT x1;
    BEGIN
        --  x2 only inside inner scope !
        DECLARE x2 CHAR(5) DEFAULT 'inner';
        SELECT x1;
        SELECT x2;
    END;
    SELECT x1;
END; //
DELIMITER ;
```

Ejemplo: Variable Local Shadowing

```
DELIMITER //
CREATE PROCEDURE Variable4()
BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'outer';
    SELECT x1;
    BEGIN
        DECLARE x1 CHAR(5) DEFAULT 'inner';
        SELECT x1;
    END;
    SELECT x1;
END; //
DELIMITER ;
```

¿Qué sucede aquí?

Stored Procedures: Parámetros

- Los procedimientos almacenados puede tener parametros (como las funciones o métodos en los lenguajes de programación)

- Ejemplo: Buscar los empleados con salarios mayor a un cierto valor **sal**

```
DELIMITER //
CREATE PROCEDURE GetEmpWithSal( sal FLOAT )
BEGIN
    SELECT fname, lname, salary
    FROM employee
    WHERE  salary > sal;
 END; //
DELIMITER ;
```

Stored Procedure: Modos de Parámetros

- Existen 3 modos (ways) de pasaje de parámetros

  - ✓ **IN**: parametro pasado por valor, por lo tanto la copia original del valor del parametro no puede modificarse (modo por defecto)

  - ✓ **OUT**: parametro pasado por referencia y puede ser modificado por el procedimiento. Se asume que el parametro OUT no se inicializa

  - ✓ **INOUT**: paramentro pasado por referencia y pude ser modificado pero se asume que ha sido inicializado

- Sintaxis:
  MODE <varname> DataType

Ejemplo: Parámetro OUT

```
DELIMITER //
CREATE PROCEDURE OutParam1( IN  x INT, OUT o FLOAT )
BEGIN
    SELECT max(salary) INTO o
    FROM employee
    WHERE  dno = x;
END; //
DELIMITER ;
```

Stored Procedures: Sentencia IF

- La sentencia IF tiene el mismo significado que cualquier lenguaje de programación

- Sintaxis IF:

```
IF <condition> THEN
    <command>
END IF;
```

- Sintaxis IF-ELSE:

```
IF <condition> THEN
    <command1>
ELSE
    <command2>
END IF;
```

Stored Procedures: Sentencia IF (2)

- Sintaxis IF-ELSEIF

```
IF <condition1> THEN
    <command1>
ELSEIF <condition2> THEN
    <command2>
...
ELSE
    <commandN>
END IF;
```

Ejemplo: Sentencia IF

```
DELIMITER //
CREATE PROCEDURE GetEmpSalLevel( IN essn CHAR(9),
                                 OUT salLevel VARCHAR(9) )
  BEGIN
    DECLARE empSalary DECIMAL(7,2);
    SELECT salary INTO empSalary
    FROM employee
    WHERE ssn = essn;
    IF empSalary < 30000 THEN
      SET salLevel = "Junior";
    ELSEIF (empSalary >= 30000 AND empSalary <= 40000) THEN
      SET salLevel = "Associate";
    ELSE
      SET salLevel = "Executive";
    END IF;
  END //
DELIMITER ;
```

Stored Procedures: Sentencia CASE

- La sentencia **CASE** es una sentencia condicional alternativa

- Makes code more readable and efficient
  Sintaxis:

```
CASE <case expression>
    WHEN <expression1> THEN <command1>
    WHEN <expression2> THEN <command2>
    ...
    ELSE <commandN>
END CASE;
```

Ejemplo: Sentencia CASE

```
DELIMITER //
CREATE PROCEDURE GetEmpBonus( IN essn CHAR(9),
                              OUT bonus DECIMAL(7,2))
  BEGIN
    DECLARE empDept INT;
    SELECT dno INTO empDept
    FROM employee
    WHERE ssn = essn;
    CASE empDept
      WHEN 1 THEN
        SET bonus = 10000;
      WHEN 4 THEN
        SET bonus = 5000;
      ELSE
        SET bonus = 0;
    END CASE;
  END //
DELIMITER ;
```

Stored Procedure: Sentencia LOOP

Existen 3 formas de loops en procedimientos almacenados:

- Sintaxis WHILE:
  WHILE <condition> DO
  <commands>
  END WHILE;

- Sintaxis REPEAT UNTIL:
  REPEAT
  <commands>
  UNTIL <condition>
  END REPEAT;

Stored Procedure: Sentencia LOOP (2)

- Sintaxis LOOP, LEAVE and ITERATE:

```
<LoopLabel>: LOOP                                   loop infinito
    <commands>
    IF <condition1> THEN
        LEAVE <LoopLabel>;      trabaja como el break
    ELSEIF <condition2> THEN
        ITERATE <LoopLabel>; trabaja como el continue
    END IF;
END LOOP;
```

Ejemplo: Sentencia Loop-Leave-Iterate

```
DELIMITER //
CREATE PROCEDURE LOOPLoopProc()
   BEGIN
   DECLARE x  INT ;
   SET  x = 0;
   L:  LOOP
      SET  x = x + 1;
      IF  (x >= 5) THEN
       LEAVE L;
      END IF;
      IF  (x mod 2 = 0) THEN
        ITERATE L;
      END IF;
      SELECT x;
      END LOOP;
   END //
DELIMITER ;
```

Cursores: Procesando Datos

- Estructura de datos usada en procedimientos almacenados que permiten iterar un conjunto de datos (tuplas) producidos para una consulta SQL
- La estructura de datos es Read-only (not updatable)
- Non-scrollable: sólo se puede recorrer en una dirección y no se pueden saltar filas
- Asensitive: server may or may not make a copy of its result table

Trabajando con Cursores

- Declarar un cursor usando la sentencia **DECLARE**:
  DECLARE <cursor_name> CURSOR FOR <query>;

  - ✓ La declaración del cursor debe seguir a todas las declaraciones de variables y antes de la declaracion del handler

  - ✓ El cursor siempre debe estar asociado con un comando SELECT

- Declarar un handler para la condición de error NOT FOUND para que pueda salir cuando no haya mas filas por leer
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

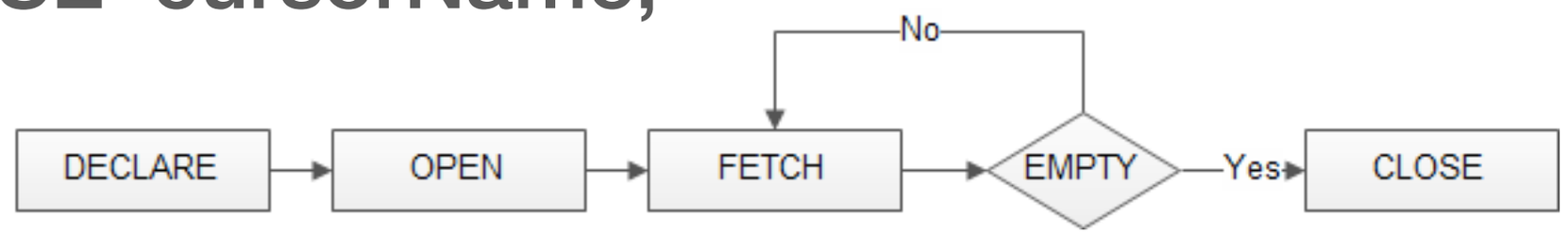
Trabajando con Cursores (2)

- Abrir el cursor usando la sentencia **OPEN**
  OPEN <cursor_name>;

  ✓ Ejecuta la consulta asociada con el cursor

- Usar **FETCH** para recuperar la próxima tupla
  FETCH <cursor_name> INTO list-of-variables;

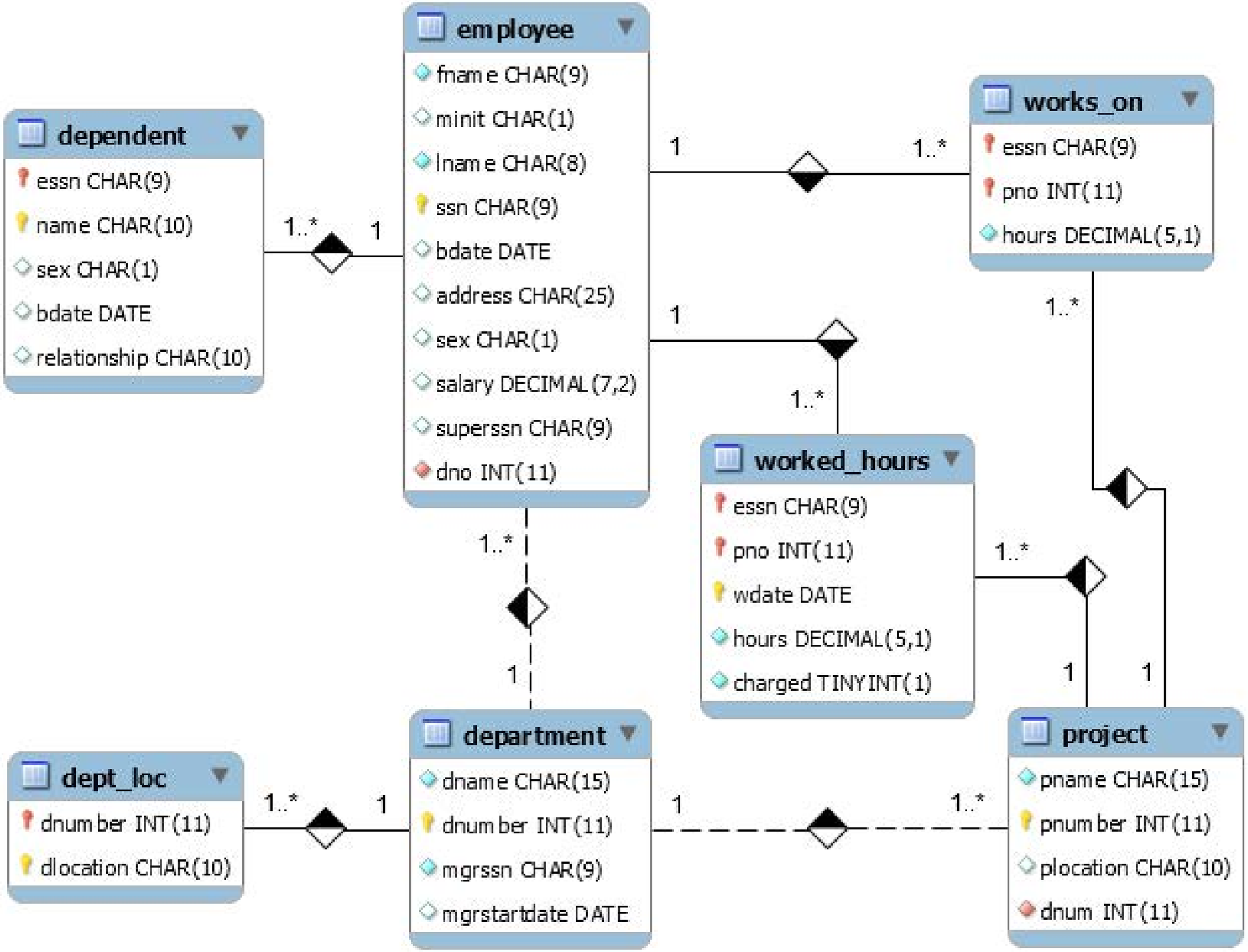
- Cerrar el cursor usando la sentencia **CLOSE**
  CLOSE cursorName;



Ejemplo: Cursor

```
DELIMITER //
CREATE PROCEDURE cursor1()
BEGIN
DECLARE finished  INTEGER   DEFAULT 0;
DECLARE fname1    CHAR(20)  DEFAULT "";
DECLARE lname1    CHAR(20)  DEFAULT "";
DECLARE nameList  CHAR(100) DEFAULT "";
-- 1. Declare cursor for employee
DECLARE emp_cursor CURSOR FOR SELECT fname, lname FROM employee WHERE salary > 40000;
-- 2. Declare NOT FOUND handler
DECLARE CONTINUE HANDLER  FOR NOT FOUND SET finished = 1;
-- 3. Open the cursor
OPEN emp_cursor;
L: LOOP
      -- 4. Fetch next tuple
  FETCH emp_cursor INTO fname1, lname1;
      -- Handler will set finished = 1 if cursor is empty
  IF finished = 1 THEN
     LEAVE L;
  END IF;
   -- build emp list
   SET nameList = CONCAT( nameList, fname1, ' ', lname1, ';' );
   END LOOP ;
-- 5. Close cursor when done
CLOSE emp_cursor;
SELECT nameList ;
END //
DELIMITER ;
```

Company Database Diagram (Actualizado)



SQL: Stored Function

SQL: Stored Function

- Funciones definidas por el usuario

  - ✓ Programa almacenado que retorna un único valor (similar a las funciones de agregación)

  - ✓ Pretende encapsular formulas comunes o reglas de negocios reutilizables

- Sintaxis:

```
CREATE FUNCTION <function_name> ([func_parameter[,...]])
        RETURNS datatype
        [NOT] DETERMINISTIC
        <routine_body>;
```

- func_parameter:

```
    param_name type
```

Ejemplo: Stored Function

```
DELIMITER //
CREATE FUNCTION employeeRaise(salary DECIMAL(7,2))
    RETURNS DECIMAL(7,2) DETERMINISTIC
    BEGIN
        RETURN (1.1 * salary);
    END; //
DELIMITER ;
```

SQL: Triggers

Triggers

- Un **trigger** es una **rutina almacenada** que está asociada con una **tabla** y que se activa cuando se produce un **evento** particular para esa tabla.
- Algunos usos de los trigger son:
  - ✓ Realizar comprobaciones de los valores que se van a insertar en una tabla
  - ✓ Realizar cálculos sobre los valores involucrados en una actualización.
- Si bien el trigger puede acceder a otras tablas para realizar su trabajo, el trigger está siempre asocidada a una sola tabla.

Triggers (2)

- Un trigger se activa cuando un comando **INSERT**, **UPDATE**, o **DELETE** (eventos básicos de triggers) impacta sobre los registros en la tabla asociada.

- Se puede configurar un trigger para que se active o bien antes (**before**) o después (**after**) del evento del trigger

- Sintaxis:

```
CREATE TRIGGER <tgr_name>  trigger_time trigger_event

 ON <tbl_name> FOR EACH ROW
 BEGIN
        <trigger_body>
 END;
```

- trigger_time: { BEFORE | AFTER }
  trigger_event: { INSERT | UPDATE | DELETE }

Ejemplo: Trigger

- Defina un trigger que actualice las horas trabajadas por el empleado en cada proyecto cada vez que se cree un nuevo registro en worked_hours que no haya sido cargado aun

```
DELIMITER //
CREATE TRIGGER chargeHours BEFORE INSERT
ON worked_hours FOR EACH ROW
BEGIN
    IF NEW.charged = 0 THEN
        UPDATE works_on SET hours = hours + NEW.hours
        WHERE essn = NEW.essn AND pno = NEW.pno;
        SET NEW.charged = 1;
    END IF;
END; //
DELIMITER ;
```

Vistas y Rutinas Almacenadas de MySQL: Resumen

- Views
- Stored Procedures
  - ✓ Variables Locales
  - ✓ Parámetros
  - ✓ IF / CASE / Loop
- Stored Function
- Triggers

Referencias

- SQL: Advanced Queries. http://joyceho.github.io/cs377_s17/slide/10-11-adv-sql.pdf
- MySQL: Session Variables & Stored Procedures. http://joyceho.github.io/cs377_s16/slides/mysql-11.pdf
- Chapter 23 Stored Programs and Views. https://dev.mysql.com/doc/refman/5.7/en/stored-programs-views.html