

# **SQL PSM**

## **(“Persistent Stored Modules”)**

**Bases de Dados (CC2005)**

**Departamento de Ciência de Computadores**  
**Faculdade de Ciências da Universidade do Porto**

**Eduardo R. B. Marques – DCC/FCUP**

# Introdução

## ■ SQL PSM (“Persistent Stored Modules”)

- Standard ISO que permite a **definição de código armazenado na própria BD**, na forma de uma linguagem imperativa que permite ter SQL embebido.
- Esta permite definição de funções e procedimentos designados por **“stored functions”** e **“stored procedures”**.
- Iremos também ver um caso especial de “stored procedure” armazenado na BD, que é executado em associação/reação a operações de manipulação de dados — **“triggers”**.

# Implementações de SQL PSM

- Vários SGBDs suportam um subconjunto de SQL/PSM, ex:
  - MySQL/MariaDB
  - PostgreSQL (PL/pgSQL)
  - Oracle (PL/SQL)
  - Microsoft SQL Server (Transact-SQL)
  - Como em SQL, a aderência ao standard é variável e cada variante tem características /extensões próprias.

# **“Stored functions”**

# MySQL – “stored function”

```
CREATE FUNCTION <NOME>
( <PARÂMETRO 1> <TIPO 1>,
  ... ,
  <PARÂMETRO n> <TIPO n> )
RETURNS <TIPO DE RETORNO>
BEGIN
  <INSTRUÇÕES>
END
```

## ■ Definição geral:

- <NOME>: nome da função
- <PARÂMETRO i> <TIPO i> : parâmetro da função e respectivo tipo SQL
- <TIPO DE RETORNO>: tipo de dados SQL retornado pela função
- <INSTRUÇÕES> : instruções que formam o corpo da função

# Exemplo

- Suponha que na empresa MovieStream se pretende passar a calcular o valor a cobrar pelo “streaming” de um filme segundo a seguinte lógica:
  - O valor base a cobrar é de **0.5** euros mais **0.01** euros por cada minuto de duração de um filme.
  - Acresce **0.75** euros se o filme é visto depois das **21** horas e ainda mais **0.75** euros se o dia da semana for sexta-feira, sábado ou domingo.
- Tendo o filme “**Pulp Fiction**” a duração de **154** minutos, o valor a cobrar será de:
  - **$0.5 + 1.54 = 2.04$**  euros para **11/Abril/2019 (5ª feira) às 20:59** .
  - **$0.5 + 1.54 + 0.75 = 2.79$**  euros para **11/Abril/2019 às 21:00**
  - **$0.5 + 1.54 + 0.75 + 0.75 = 3.54$**  euros para **12/Abril/2019 às 21:00**
- **Vamos implementar esta lógica usando uma função armazenada na BD — “stored function”.**

# Exemplo – “stored function”

```
DROP FUNCTION IF EXISTS getChargeValue;
DELIMITER $
CREATE FUNCTION getChargeValue(stream_time DATETIME, movie_duration INT)
RETURNS DECIMAL(4,2)
BEGIN
-- Declaração de variáveis
DECLARE c DECIMAL(4,2);
... ver próximo slide ...
-- Retorno
RETURN c;
END $
DELIMITER ;
```

**Configuração de delimitadores com DELIMITER:** é necessário redefinir delimitador de instruções para \$ ou outra sequência diferente de ; usado por omissão, antes de **CREATE FUNCTION**, em si uma instrução SQL, para que possamos usar ; como delimitador das instruções na definição da função. O delimitador ; é repostado depois da definição no exemplo.

- **DROP FUNCTION:** remove a função da base de dados caso já exista.
- **CREATE FUNCTION:** cria a função **getCharge()** com argumentos **stream\_time** e **movie\_duration** e retornando um valor com tipo **DECIMAL(4,2)** (nº real com 4 dígitos no total, 2 casas decimais).

# Exemplo – “stored function” (cont.)

```
DROP FUNCTION IF EXISTS getChargeValue;
DELIMITER $
CREATE FUNCTION getChargeValue(stream_time DATETIME, movie_duration INT)
RETURNS DECIMAL(4,2)
BEGIN
    DECLARE c DECIMAL(4,2);
    SET c = 0.5 + 0.01 * movie_duration;
    IF HOUR(stream_time) >= 21 THEN
        SET c = c + 0.75;
        -- Obs.: WEEKDAY retorna valor de 0 a 6 (Seg. a Domingo)
        -- 6ª feira = 4
        IF WEEKDAY(stream_time) >= 4 THEN
            SET c = c + 0.75;
        END IF;
    END IF;
    RETURN c;
END $
DELIMITER ;
```

**Requisito:** “O valor base a cobrar é de **0.5** euros mais **0.01** euros por cada minuto de duração de um filme. Acresce **0.75** euros se o filme é visto depois das **21** horas e ainda mais **0.75** euros se o dia da semana for sexta-feira, sábado ou domingo.”



# Exemplo – uso da “stored function”

- Uma “stored function” pode ser usada numa expressão SQL, por exemplo em associação a uma consulta.

**SELECT**

**Title,**

**getChargeValue('2019-04-11 20:59:59', Duration)**

**AS '5ª 20:59',**

**getChargeValue('2019-04-11 21:00:00', Duration)**

**AS '5ª 21:00',**

**getChargeValue('2019-04-12 21:00:00', Duration)**

**AS '6ª 21:00'**

**FROM MOVIE**

**WHERE Title = 'Pulp Fiction';**

**Requisito:** “O valor base a cobrar é de **0.5** euros mais **0.01** euros por cada minuto de duração de um filme. Acresce **0.75** euros se o filme é visto depois das **21** horas e ainda mais **0.75** euros se o dia da semana for sexta-feira, sábado ou domingo.”

**Duração de “Pulp Fiction”:**  
**154 minutos**

Title	5ª 20:59	5ª 21:00	6ª 21:00
Pulp Fiction	2.04	2.79	3.54

# Exemplo de “stored function” – variante

```
CREATE FUNCTION getChargeValue_v2(stream_time DATETIME, movie_id INT)
RETURNS DECIMAL(4,2)
BEGIN
  DECLARE c DECIMAL(4,2);
  DECLARE movie_duration INT;
  SELECT Duration INTO movie_duration
  FROM MOVIE WHERE MovieId = movie_id;
  SET c = 0.5 + 0.01 * movie_duration;
  IF HOUR(stream_time) >= 21 THEN
    SET c = c + 0.75;
    IF WEEKDAY(stream_time) >= 4 THEN
      SET c = c + 0.75;
    END IF;
  END IF;
  RETURN c;
END $
```

**Variante:** neste caso `getChargeValue()` toma o identificador do filme como parâmetro e faz uma consulta à tabela **MOVIE** para obter a duração do filme.

Instrução da forma `SELECT ... INTO var ...` permite escrever o resultado de uma consulta (tem de ser único) numa variável declarada na função.

# Uso de “stored function” – variante

```
SELECT MovieId INTO @movie_id FROM MOVIE
WHERE Title = 'Pulp Fiction';
SELECT 'Pulp Fiction',
       getChargeValue_v2('2019-04-11 20:59:59', @movie_id)
AS '5ª 20:59',
       getChargeValue_v2('2019-04-11 21:00:00', @movie_id)
AS '5ª 21:00',
       getChargeValue_v2('2019-04-12 21:00:00', @movie_id)
AS '6ª 21:00';
```

	5ª 20:59	5ª 21:00	6ª 21:00
Pulp Fiction	2.04	2.79	3.54

**Requisito:** “O valor base a cobrar é de **0.5** euros mais **0.01** euros por cada minuto de duração de um filme. Acresce **0.75** euros se o filme é visto depois das **21** horas e ainda mais **0.75** euros se o dia da semana for sexta-feira, sábado ou domingo.”

**Duração de “Pulp Fiction”:**  
**154 minutos**

**Nota:** Fora do âmbito de uma “stored function” / “stored procedure” uma instrução da forma **SELECT ... INTO @var ...** (ao invés de **var**) permite introduzir uma variável na sessão SQL em contexto.

# **“Stored procedures”**

# MySQL – “stored procedure”

```
CREATE PROCEDURE <NOME>  
( <QUALIFICADOR 1> <PARÂMETRO 1> <TIPO1>,  
  ...,  
  <QUALIFICADOR n> <PARÂMETRO n> <TIPO n> )  
BEGIN  
  <INSTRUÇÕES>  
END
```

- Um **procedimento na BD** – “**stored procedure**” – não tem um tipo de retorno associado, ao contrário de uma função.
- Contudo um **parâmetro**, além de ter nome e tipo como numa função, **pode ser de entrada e/ou de saída, consoante o qualificador que é usado** – IN : entrada, OUT : saída, INOUT: entrada e saída.
- Um “stored procedure” é invocado usando a instrução **CALL**.

# Exemplo de “stored procedure”

- Suponha que na empresa MovieStream se pretende registrar o visionamento (“stream”) de um filme por parte de um cliente usando um procedimento na BD.
- **Requisito:** deve ser adicionado o registo apropriado à tabela **STREAM** com o valor calculado pela função **getChargeValue()** definida anteriormente.
- Parâmetros de entrada (input):
  - **IN customer\_id INT:** identificador do cliente
  - **IN movie\_id (INT) :** identificador do filme
  - **IN time (DATETIME) :** tempo associado ao visionamento
- Parâmetros de saída (output):
  - **OUT stream\_id (INT):** id do registo inserido em **STREAM**
  - **OUT charge DECIMAL(4,2):** valor cobrado
- Vamos chamar ao procedimento **registerMovieStream** .

# Exemplo de “stored procedure”

```
DROP PROCEDURE IF EXISTS registerMovieStream;  
DELIMITER $
```

```
CREATE PROCEDURE registerMovieStream  
(IN movie_id INT, IN customer_id INT, IN time DATETIME,  
  OUT charge DECIMAL(4,2), OUT stream_id INT)  
BEGIN  
  ... ver próximo slide ...  
END $  
DELIMITER ;
```

- Analogamente a “stored functions”:
  - **DROP PROCEDURE**: remove o procedimento da base de dados caso já exista.
  - **CREATE PROCEDURE**: cria o procedimento **registerMovieStream()**.

# Exemplo de “stored procedure”

```
DROP PROCEDURE IF EXISTS registerMovieStream;
```

```
DELIMITER $
```

```
CREATE PROCEDURE registerMovieStream
```

```
(IN movie_id INT, IN customer_id INT, IN time DATETIME,  
OUT charge DECIMAL(4,2), OUT stream_id INT)
```

```
BEGIN
```

```
DECLARE d INT;
```

```
-- Obtém duração do filme.
```

```
SELECT Duration INTO d FROM MOVIE WHERE MovieId = movie_id;
```

consulta

```
-- Obtém valor a cobrar usando função getChargeValue()
```

```
SET charge = getChargeValue(time, d);
```

```
-- Insere registo na tabela STREAM.
```

```
INSERT INTO STREAM(CustomerId, MovieId, StreamDate, Charge)
```

inserção

```
VALUES(customer_id, movie_id, time, charge);
```

```
-- Obtém id de registo inserido (função LAST_INSERT_ID)
```

```
SET stream_id = LAST_INSERT_ID();
```

```
END $
```

```
DELIMITER ;
```

função “built-in” do MySQL que permite obter último valor gerado usado na sequência **AUTO\_INCREMENT** definida para a coluna **STREAM\_ID**.



# Uso de “stored procedures”

```
CALL registerMovieStream(555, 124, '2018-04-12 20:00:00', @charge, @stream_id);
```

Chamada

```
SELECT @charge, @stream_id;
```

Consulta dos valores obtidos via parâmetros OUT

```
+-----+-----+
| @charge | @stream_id |
+-----+-----+
|      2.19 |          10162 |
+-----+-----+
```

Identificador do registro

```
SELECT * FROM STREAM WHERE StreamId = @stream_id;
```

consulta de dados inseridos

```
+-----+-----+-----+-----+-----+
| StreamId | MovieId | CustomerId | StreamDate | Charge |
+-----+-----+-----+-----+-----+
|      10162 |          555 |          124 | 2018-04-12 20:00:00 |      2.19 |
+-----+-----+-----+-----+-----+
```

- **A invocação de um “stored procedure” requer o uso da instrução CALL.**
- Ao contrário de “stored functions”, **não** podemos usar “stored procedures” em associação a expressões SQL.
- Variáveis no âmbito estrito da sessão SQL, i.e. fora de “stored procedures/functions”, podem ser definidas com a sintaxe @nomeDaVariável.

# “Stored functions” vs. “stored procedures”

## ■ “Stored functions”

- Tipo de retorno definido, parâmetros são todos de entrada
- Invocação em associação a expressões SQL.
- Tipicamente usadas para cálculo de valores, sem afectar o estado da BD. Eventuais operações de alteração do conteúdo da BD são mais restritas do que no caso de “stored procedures”.

## ■ “Stored procedures”:

- Sem tipo de retorno, parâmetros podem ser de entrada e/ou de saída.
- Invocação usando a instrução **CALL**
- A forma e tipo de operações em procedimentos são menos limitadas.

## ■ [Restrições no caso de MySQL](#)

# **“Triggers”**

# Triggers

## ■ “Trigger”:

- “stored procedure” que é executado em função de um evento associado a uma tabela
- **tipo de evento:** inserção, remoção ou actualização de registo da tabela
- **execução do “trigger”:** antes ou depois da operação

## ■ Vantagens / aplicações do uso de triggers:

- Validação de integridade lógica
- Debugging / auditoria de operações na BD

## ■ Desvantagens

- Execução do “trigger” é “invisível” para aplicações cliente
- Tempo acrescido na execução de operações sobre tabelas

# Definição de triggers em MySQL

```
CREATE TRIGGER <NOME DO TRIGGER>  
[BEFORE | AFTER]  
[INSERT | DELETE | UPDATE]  
ON <NOME DA TABELA>  
FOR EACH ROW  
BEGIN  
  <INSTRUÇÕES>  
END;
```

- **<NOME DO TRIGGER>** : nome do “trigger”
- **<NOME DA TABELA>** : tabela à qual fica associado o “trigger”
- **Execução do trigger: antes** (**BEFORE**) **ou depois** do evento (**AFTER**)
- **Evento**: inserção (**INSERT**), remoção (**DELETE**), ou actualização (**UPDATE**)
- **<INSTRUÇÕES>** : bloco de instruções executada para cada registo (**FOR EACH ROW**) da tabela envolvido na operação

# Triggers – exemplo 1

```
DROP TRIGGER IF EXISTS beforeMovieInsertion;  
DELIMITER $
```

```
CREATE TRIGGER beforeMovieInsertion  
BEFORE INSERT ON MOVIE FOR EACH ROW  
BEGIN  
  -- 99999 designa um código de erro da aplicação  
  DECLARE error CONDITION FOR SQLSTATE '99999';  
  -- Valida ano  
  IF NEW.Year < 1900 THEN  
    SIGNAL error SET MESSAGE_TEXT = 'Invalid year!';  
  END IF;  
  -- Valida duração  
  IF NEW.Duration <= 0 THEN  
    SIGNAL error SET MESSAGE_TEXT = 'Invalid duration!';  
  END IF;  
END $  
DELIMITER ;
```

- **Restrições de domínio:** garantir que p/entrada inserida em **MOVIE** que **Duration > 0** e que **Year >= 1900** (2 exemplos de restrições de domínio que se poderiam considerar).
- “Triggers” para validação de restrições de domínio podem ser particularmente úteis em MySQL <= 5.7, que ignora cláusulas do tipo **CHECK** na definição de uma tabela.
- Trigger à esquerda:
  - efectua a validação antes de o registo ser inserido na BD
  - usa instrução **SIGNAL** para abortar a inserção e assinalar o erro.
  - um “trigger” análogo para as mesmas restrições poderia ser definido para actualizações a **MOVIE**

# Triggers – exemplo 1 (cont.)

```
mysql> INSERT INTO MOVIE(Title, Year, Duration) VALUES('Spellbound', 1945, 111);  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO MOVIE(Title, Year, Duration) VALUES('Come Along, Do!', 1898, 1);  
ERROR 1644 (99999): Invalid year!
```

```
mysql> INSERT INTO MOVIE(Title, Year, Duration) VALUES('Less Than Zero', 1987, -98);  
ERROR 1644 (99999): Invalid duration!
```

```
mysql> SELECT * FROM MOVIE ORDER BY MovieId DESC LIMIT 3;
```

MovieId	Title	Year	Duration
1001	Spellbound	1945	111
1000	xXx	2002	124
999	Zootopia	2016	108

```
3 rows in set (0.00 sec)
```

- Apenas 1ª inserção completa. O trigger definido impede a inserção de um filme anterior a 1900 ou com duração negativa.

# Triggers – exemplo 2

```
DROP TRIGGER IF EXISTS beforeDepartmentUpdate,  
beforeDepartmentInsert;  
  
DELIMITER $
```

```
CREATE TRIGGER beforeDepartmentUpdate  
BEFORE UPDATE ON DEPARTMENT FOR EACH ROW  
BEGIN  
    IF NEW.Manager <> OLD.Manager THEN  
        CALL ensureNotDepartmentManager(NEW.Manager);  
    END IF;  
END $
```

```
CREATE TRIGGER beforeDepartmentInsert  
BEFORE INSERT ON DEPARTMENT FOR EACH ROW  
BEGIN  
    CALL ensureNotDepartmentManager(NEW.Manager);  
END $
```

```
DELIMITER ;
```

- **Restrição a validar:** um funcionário (com identificador `staff_id`) não deve ser gestor (**Manager**) de mais do que um departamento (**DEPARTMENT**).
- Note que não conseguimos expressar a restrição no modelo relacional em si.
- Podemos definir dois “triggers” que chamam um procedimento auxiliar chamado `ensureNotDepartmentManager` para validar essa restrição (próximo slide).



# Triggers – exemplo 2 (cont.)

```
CREATE PROCEDURE
ensureNotDepartmentManager(IN staff_id INT)
BEGIN
  DECLARE is_manager BOOL;
  DECLARE error CONDITION FOR SQLSTATE '99999';
  SET is_manager = FALSE;
  SELECT TRUE INTO is_manager
  FROM DEPARTMENT WHERE Manager = staff_id;
  IF is_manager THEN
    SIGNAL error
    SET MESSAGE_TEXT = 'No staff member can supervise more than one department!';
  END IF;
END $
```

- **Acima:** procedimento auxiliar `ensureNotDepartmentManager` usado pelos “triggers” `beforeDepartmentInsert` e `beforeDepartmentUpdate` (slide anterior).

# Triggers – exemplo 2 (cont.)

```
mysql> SELECT * FROM DEPARTMENT;
```

DepId	Name	Manager
1	IT	2
2	Public Relations	7
3	Finance	11
4	Customer	14

```
4 rows in set (0.00 sec)
```

```
mysql> INSERT INTO DEPARTMENT(Name, Manager) VALUES('Misc Operations', 2);  
ERROR 1644 (99999): No staff member can supervise more than one department!
```

```
mysql> UPDATE DEPARTMENT SET Manager = 2 WHERE Name = 'Public Relations';  
ERROR 1644 (99999): No staff member can supervise more than one department!
```

- **Acima:** inserção e actualização falham porque os “triggers” definidos anteriormente detectam que funcionário com id 2 já é gestor do departamento de IT.

# Trigger – exemplo 3

```
DROP TABLE IF EXISTS OPERATION_LOG;  
CREATE TABLE OPERATION_LOG  
(  
    OpLogId INT NOT NULL AUTO_INCREMENT,  
    Time DATETIME NOT NULL,  
    Event ENUM ('INSERT', 'UPDATE', 'DELETE') NOT NULL,  
    TableName VARCHAR(128) NOT NULL,  
    EntryId INT NOT NULL,  
    PRIMARY KEY(OpLogId)  
);
```

- Suponha que queremos registrar as operações sobre tabelas na BD MovieStream.
- A tabela “auxiliar” **OPERATION\_LOG**, com definição dada acima, servirá de registo a essas operações na execução de “triggers” (ver próximo slide).

# Trigger – exemplo 3 (cont.)

```
DROP TRIGGER IF EXISTS afterMovieInsertion;  
DELIMITER $
```

```
CREATE TRIGGER afterMovieInsertion  
AFTER INSERT ON MOVIE FOR EACH ROW  
BEGIN  
    INSERT INTO OPERATION_LOG(Time, Event, TableName, EntryId)  
    VALUES(NOW(), 'INSERT', 'MOVIE', NEW.MovieId);  
END $  
  
DELIMITER ;
```

- O trigger `afterMovieInsertion` insere uma entrada em `OPERATION_LOG` após um inserção em `MOVIE`.
- Triggers análogos poderiam ser definidos para actualizações e remoções em `MOVIE` ou outras tabelas.

# Trigger – exemplo 3 (cont.)

```
mysql> INSERT INTO MOVIE(Title, Year, Duration) VALUES('Spellbound', 1945, 111);  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO MOVIE(Title, Year, Duration) VALUES('The Hunger', 1983, 97);  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM MOVIE ORDER BY MovieID DESC LIMIT 2;
```

MovieId	Title	Year	Duration
1002	The Hunger	1983	97
1001	Spellbound	1945	111

```
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM OPERATION_LOG ORDER BY OpLogId DESC LIMIT 2;
```

OpLogId	Time	Event	TableName	EntryId
2	2019-04-26 18:21:15	INSERT	MOVIE	1002
1	2019-04-26 18:20:24	INSERT	MOVIE	1001

```
2 rows in set (0.00 sec)
```

# Trigger – exemplo 4

```
CREATE TRIGGER beforeCustomerDelete
BEFORE DELETE ON CUSTOMER FOR EACH ROW
BEGIN
  DECLARE error CONDITION FOR SQLSTATE '99999';
  IF OLD.ACTIVE = TRUE THEN
    SIGNAL error
    SET MESSAGE_TEXT = 'Not allowed!';
  ELSE
    DELETE FROM STREAM WHERE CustomerId=OLD.CustomerId;
  END IF;
END $
```

- Trigger garante neste caso duas coisas para a remoção de registos da tabela **CUSTOMER**: **1)** clientes activos não podem ser apagados (têm de ser tornados inactivos primeiro); **2)** clientes inactivos podem ser apagados e nesse caso são também apagados os registos na tabela **STREAM** associados ao(s) cliente(s) em causa (não seria de outra forma possível visto que a chave externa **STREAM.CustomerId** não é declarada com **ON DELETE CASCADE**).

# Trigger – exemplo 4 (cont.)

```
mysql> SELECT CustomerId, Name FROM CUSTOMER
        WHERE Active = TRUE
        AND CustomerId NOT IN
          (SELECT CustomerId FROM STREAM)
        ORDER BY CustomerId
        LIMIT 1;
```

```
+-----+-----+
| CustomerId | Name          |
+-----+-----+
|          59 | Cheryl Murphy |
+-----+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> DELETE FROM CUSTOMER WHERE CustomerId = 59;
ERROR 1644 (99999): Not allowed!
```

- Cliente activo não pode ser apagado, mesmo que não tenha streams associados.

# Trigger – exemplo 4 (cont.)

```
mysql> SELECT CustomerId, Name FROM CUSTOMER WHERE Active = FALSE
        ORDER BY CustomerId LIMIT 1;
```

```
+-----+-----+
| CustomerId | Name          |
+-----+-----+
|          16 | Sandra Martin |
+-----+-----+
```

1 row in set (0.00 sec)

```
mysql> SELECT COUNT(*) FROM STREAM WHERE CustomerId = 16;
```

```
+-----+
| COUNT(*) |
+-----+
|         87 |
+-----+
```

```
mysql> DELETE FROM CUSTOMER WHERE CustomerId = 16;
```

Query OK, 1 row affected (0.00 sec)

```
mysql> SELECT COUNT(*) FROM STREAM WHERE CustomerId = 16;
```

```
+-----+
| COUNT(*) |
+-----+
|         0 |
+-----+
```

- Cliente inactivo é apagado, bem como todos os streams associados a esse cliente.



# **Apontadores para referência MySQL**

# Apontadores para referência MySQL

## ■ Declaração de variáveis

- DECLARE

## ■ Instruções de controlo de fluxo

- IF, WHILE, RETURN, CASE, REPEAT UNTIL, ...

## ■ Erros

- DECLARE CONDITION, SIGNAL, ...

## ■ Cursores

- Permitem iterar os resultados de uma consulta.