

Bases de Dados

PL10 – SQL Avançada

Docente: Diana Ferreira

Email: diana.ferreira@algoritmi.uminho.pt

Horário de Atendimento:

5ª feira 16h–17h



Sumário

1

Vistas

2

Funções

3

Procedimentos

4

Triggers

Bibliografia:

- Connolly, T., Begg, C., Database Systems, A Practical Approach to Design, Implementation, and Management , Addison-Wesley, 4a Edição, 2004. **(Chapter 6 e 7)**
- Belo, O., "Bases de Dados Relacionais: Implementação com MySQL", FCA – Editora de Informática, 376p, Set 2021. ISBN: 978-972-722-921-5. **(Capítulo 4 e 5)**

FASE 6: Exploração

➔ VISTAS

Uma vista é uma tabela virtual derivada de uma ou mais tabelas ou vistas existentes. É definida por uma query SQL e tem a aparência de uma tabela, mas não armazena nenhum dado. Em vez disso, recupera os dados dinamicamente das tabelas/vistas subjacentes sempre que é consultada.

- 1) Apresentar valores estatísticos sobre os medicamentos prescritos, como o número de vezes prescrito, a quantidade total, média, mínima e máxima

```
CREATE VIEW vw_medication_stats AS  
SELECT m.nome as 'Medicamento',  
COUNT(*) as 'Nr de vezes prescrito',  
SUM(quantidade) as 'Quantidade Total',  
ROUND(AVG(quantidade),0) as 'Quantidade Média',  
MAX(quantidade) as 'Quantidade Máxima',  
MIN(quantidade) as 'Quantidade Mínima'  
FROM medicamentos m  
INNER JOIN prescricoes p USING (id_med)  
GROUP BY m.nome;
```

FASE 6: Exploração

➔ PREPARED STATEMENTS

Uma *query* pré-compilada permite a criação de declarações SQL para posterior execução, otimizando a execução de consultas repetitivas sem variação sintática, com dinamismo apenas nos parâmetros.

EXEMPLO:

-- Preparar a query pré-compilada

PREPARE ps_medicos_por_especialidade **FROM**

'SELECT f.nome FROM funcionarios f

INNER JOIN medicos m USING(nr_mec)

INNER JOIN especialidades e USING(cod_especialidade)

WHERE e.des_especialidade = ?';

-- Atribuir os valores aos parâmetros (caso existam)

SET @especialidade = 'Neurologia';

-- Executar a query pré-compilada

EXECUTE ps_medicos_por_especialidade USING @especialidade;

-- Executar a query pré-compilada usando outros valores

SET @especialidade = 'Cardiologia'; EXECUTE ps_medicos_por_especialidade USING @especialidade;

FASE 6: Exploração

➔ PROCEDIMENTOS

Um procedimento é uma coleção de instruções SQL pré-compiladas armazenadas na BD que mais tarde podem ser invocadas.

Síntaxe para criação de um procedimento:

```
DELIMITER &&  
CREATE PROCEDURE <procedure_name> ([IN | OUT | INOUT] parameter_name parameter_datatype)  
BEGIN  
    Declaration_section  
    Executable_section  
END &&  
DELIMITER;
```

Nome do parâmetro

Tipo de parâmetro

Tipo de dados e tamanho do parâmetro

IN - É o modo padrão, permite passar parâmetros de entrada.

Tipo de Parâmetro

OUT - É usado para passar um parâmetro como saída. O seu valor pode ser alterado dentro do procedimento armazenado e o valor alterado (novo) é passado de volta para o programa que invoca o procedimento.

INOUT - É uma combinação dos modos IN e OUT.

FASE 6: Exploração

➔ PROCEDIMENTOS

Podemos criar um procedimento sem parâmetros. A rotina a baixo descrita é um procedimento que retorna todos os médicos do hospital.

```
DELIMITER &&  
CREATE PROCEDURE GetMedicos()  
BEGIN  
    SELECT * FROM medicos;  
END &&  
DELIMITER;
```

➔ CALL GetMedicos();

FASE 6: Exploração

➔ PROCEDIMENTOS

Sabemos que a tabela de preços está constantemente sujeita a alterações. De seguida, encontra-se uma rotina para atualizar o preço de um determinado procedimento clínico.

```
DELIMITER &&  
CREATE PROCEDURE UpdateProcedimento (IN proc INT, IN new_preco DECIMAL(5,2))  
BEGIN  
    UPDATE procedimentos SET preco = new_preco where cod_proc=proc;  
END &&  
DELIMITER;
```

➔ CALL UpdateProcedimento(8, 10.15);

FASE 6: Exploração

➔ PROCEDIMENTOS

Sabemos que um hospital está constantemente sujeito a auditorias e a processos de avaliação dos serviços de saúde prestados. De seguida, encontra-se uma rotina para consultar o número total de consultas num determinado ano.

```
DELIMITER &&  
CREATE PROCEDURE GetConsultasYear (IN ano INT, OUT total_consultas INT)  
BEGIN  
    SELECT count(*) INTO total_consultas FROM consultas WHERE YEAR(dta_ini)=ano;  
END &&  
DELIMITER;
```

➔ `CALL GetConsultasYear(2020, @total_consultas);`
`SELECT @total_consultas AS total_consultas_2020;`

FASE 6: Exploração

➔ PROCEDIMENTOS

No exemplo a seguir, encontra-se uma rotina que aceita um parâmetro IN e outro INOUT. Este procedimento, incrementa o contador de acordo com o valor específico no parâmetro inc.

```
DELIMITER &&  
CREATE PROCEDURE Contador (IN inc INT, INOUT contador INT)  
BEGIN  
    SET contador = contador + inc;  
END &&  
DELIMITER;
```

```
➔ SET @contador = 1;  
   CALL Contador(1, @contador); -- 2  
   SELECT @contador;
```

FASE 6: Exploração

➔ FUNÇÕES

Uma função é um programa armazenado que devolve um único valor. Tipicamente, usam-se funções para encapsular fórmulas ou regras de negócio comuns que são reutilizáveis entre instruções SQL ou programas armazenados.

Síntaxe para criar uma função:

```

DELIMITER &&
CREATE FUNCTION <function_name> (parameter_name parameter_datatype) Tipo de dados e tamanho do parâmetro
RETURNS datatype [NOT] DETERMINISTIC
BEGIN Tipo de dados a retornar
    Body_section → é preciso especificar pelo menos uma instrução RETURN.
    Uma função determinística retorna sempre o mesmo resultado para os mesmos parâmetros de entrada.
END &&
DELIMITER;
```

FASE 6: Exploração

➔ FUNÇÕES

EXEMPLO → Função Idade

```
DELIMITER &&  
CREATE FUNCTION idade (dta DATE)  
RETURNS INT  
NOT DETERMINISTIC  
BEGIN  
    RETURN TIMESTAMPDIFF(YEAR, dta, CURDATE());  
END &&  
DELIMITER ;
```

NOTA: pode precisar de alterar esta variável: SET GLOBAL log_bin_trust_function_creators = 1;

FASE 6: Exploração

➔ TRIGGER

Um trigger é invocado automaticamente quando uma operação de alteração específica (instrução INSERT, UPDATE, ou DELETE) é executada sobre uma determinada tabela. Os triggers são úteis para tarefas como a aplicação de regras comerciais ou até para validação de dados quando inseridos na base de dados.

Síntaxe para criação de trigger:

```

DELIMITER &&
CREATE TRIGGER <trigger_name> {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON <table_name> FOR EACH ROW
[{{FOLLOWS | PRECEDES} existing_trigger_name}]
BEGIN
    Body_section
END &&
DELIMITER;
  
```

Tempo de ação

Operação que ativa o trigger

Caso exista mais do que um trigger na mesma tabela com o mesmo evento e tempo de ação.

FOLLOWS permite que o novo trigger seja ativado após um trigger existente.
PRECEDES permite que o novo trigger seja ativado antes de um trigger existente.

Consegue aceder aos valores das colunas afetadas pela instrução DML. Os modificadores NEW e OLD permitem distinguir entre os valores antes e depois da operação de manipulação.

FASE 6: Exploração

➔ TRIGGER

EXEMPLO → Suponto que existe o atributo nr_consultas na tabela especialidades, crie um trigger para atualizar o nr_consultas, sempre que uma nova consulta é inserida

```
DELIMITER //  
CREATE TRIGGER update_nrconsultas  
AFTER INSERT ON consultas FOR EACH ROW  
BEGIN  
    DECLARE id_esp INT;  
    SELECT e.cod_especialidade INTO id_esp FROM consultas c  
    INNER JOIN medicos m ON m.nr_mec=c.id_medico  
    INNER JOIN especialidades e USING(cod_especialidade)  
    WHERE c.nr_episodio=NEW.nr_episodio;  
    UPDATE especialidades SET nr_consultas=nr_consultas+1 WHERE  
    cod_especialidade=id_esp;  
END //  
DELIMITER ;
```

FASE 6: Exploração

➔ Resolução de Exercícios

Ficha de Exercícios PL10