

VISÕES

Sérgio Mergen

Sumário

- Visões
- Visões materializadas
- Aplicações para visões

Visões

- Uma visão é uma consulta que acessa tabelas (ou outras visões) de um banco de dados
- Sintaxe de criação de uma visão:
 - **CREATE VIEW nomeVisao AS (consulta_sql)**

Visões

- Ex. Criação de uma visão (**v1**) com duas colunas (**nome**, **custo**) geradas como resultado de uma consulta

```
CREATE VIEW v1 AS (SELECT nome, custo FROM projeto  
WHERE ano = 2015)
```

Projeto			
Id	nome	Custo	ano
1	ACME	1000	2014
2	XYZ	2000	2014
3	PROJ	5000	2015
4	ABC	2500	2015

V1 <<visao>>	
nome	custo

Visões

- Ex. Criação de uma visão (**v1**) com duas colunas (**nome**, **custo**) geradas como resultado de uma consulta

```
CREATE VIEW v1 AS (SELECT nome, custo FROM projeto  
WHERE ano = 2015)
```

Projeto			
Id	nome	Custo	ano
1	ACME	1000	2014
2	XYZ	2000	2014
3	PROJ	5000	2015
4	ABC	2500	2015

V1 <<visao>>	
nome	custo

Obs. A visão não possui nenhum registro. Ela é apenas a definição de uma consulta

Visões

- Uma visão pode ser utilizada em uma consulta da mesma forma que as tabelas
- Ex.

```
SELECT nome, custo  
FROM v1  
WHERE custo > 3000
```

V1 <<visao>>	
nome	custo

Projeto			
Id	nome	Custo	ano
1	ACME	1000	2014
2	XYZ	4000	2014
3	PROJ	5000	2015
4	ABC	2500	2015

resposta	
nome	custo
PROJ	5000

Visões

- Ao processar uma consulta que tenha visões, o processador **reescreve** a consulta passando a usar as tabelas

```
SELECT nome, custo  
FROM v1  
WHERE custo > 3000
```

Consulta com visões



```
SELECT nome, custo  
FROM projeto  
WHERE custo > 3000 AND  
ano = 2015
```

Consulta sem visões

Visões

- Esse processo é chamado de **desdobramento** de consultas (query unfolding)

```
SELECT nome, custo  
FROM v1  
WHERE custo > 3000
```

Consulta com visões



desdobramento

```
SELECT nome, custo  
FROM projeto  
WHERE custo > 3000 AND  
ano = 2015
```

Consulta sem visões

Visões

- O desdobramento é feito de forma **transparente**. Quem escreve a consulta não precisa se preocupar com o acesso às tabelas originais

```
SELECT nome, custo  
FROM v1  
WHERE custo > 3000
```

Consulta com visões



desdobramento

```
SELECT nome, custo  
FROM projeto  
WHERE custo > 3000 AND  
ano = 2015
```

Consulta sem visões

Visões

- Visões podem ser usadas para diversas finalidades
- Ex.
 - Simplificar consultas complexas
 - Auxiliar em processos de transição
 - Restringir acesso à tabelas
- Mais adiante essas finalidades serão exemplificadas

Sumário

- Visões
- **Visões materializadas**
- Aplicações para visões

Visões Materializadas

- Uma visão materializada é uma consulta que acessa tabelas (ou outras visões) de um banco de dados
- Ao contrário das visões não materializadas, uma visão materializada pode conter registros
 - os registros de resultado da consulta são salvos como registros da visão
- Sintaxe de criação de uma visão materializada(em **Postgres**):
 - CREATE **MATERIALIZED** VIEW nomeVisao AS (consulta_sql)

Visões Materializadas

- Ex. Criação de uma visão materializada (**v1**) com duas colunas (**nome**, **custo**) geradas como resultado de uma consulta

```
CREATE MATERIALIZED VIEW v1 AS (  
    SELECT nome, custo FROM projeto  
    WHERE ano = 2015  
)
```

Projeto			
Id	nome	Custo	ano
1	ACME	1000	2014
2	XYZ	4000	2014
3	PROJ	5000	2015
4	ABC	2500	2015

V1 <<visao>>	
nome	custo
PROJ	5000
ABC	2500

Essa visão tem registros

Visões Materializadas

- Uma visão materializada pode ser utilizada em uma consulta da mesma forma que as tabelas
- Ex.

V1 <<visao>>	
nome	custo
PROJ	5000
ABC	2500

SELECT nome, custo FROM v1 WHERE custo > 3000

resposta	
nome	custo
PROJ	5000

Visões Materializadas

- Quando um registro de alguma das tabelas usadas na visão for atualizado
 - Os registros da visão também serão atualizados pelo próprio SGBD (de forma transparente)

```
CREATE MATERIALIZED VIEW v1 AS (  
    SELECT nome, custo FROM projeto  
    WHERE ano = 2015 )
```

Projeto			
Id	nome	Custo	ano
1	ACME	1000	2014
2	XYZ	4000	2014
3	PROJ	5000	2015
4	ABC	2500	2015

V1 <<visao>>	
nome	custo
PROJ	5000
ABC	2500

Visões Materializadas

- Quando um registro de alguma das tabelas usadas na visão for atualizado
 - Os registros da visão também serão atualizados pelo próprio SGBD (de forma transparente)

```
CREATE MATERIALIZED VIEW v1 AS (  
    SELECT nome, custo FROM projeto  
    WHERE ano = 2015 )
```

Projeto			
Id	nome	Custo	ano
1	ACME	3000	2014
2	XYZ	4000	2014
3	PROJ	8000	2015
4	ABC	2500	2015

V1 <<visao>>	
nome	custo
PROJ	8000
ABC	2500

Visões Materializadas

- Considere um cenário onde a visão é construída para uma consulta complexa
 - Muitas junções e agrupamentos
- Vantagens das visões materializadas
 - Aceleram a recuperação de registros
 - Como os registros de resposta já estão calculados dentro da própria visão, a recuperação da resposta é rápida
- Desvantagens
 - Visões materializadas ocupam espaço
 - Overhead de atualização da visão
 - Alterações nos registros das tabelas levarão à atualização dos registros da visão

Visões Materializadas

- Quando usar visões materializadas?
 - Quando a consulta for complexa e demorada
 - Quando o overhead de espaço não é significativo
 - Quando o tradeoff compensa
 - A redução de tempo pelo uso da visão é maior que o acréscimo de tempo pela atualização da visão
 - Ou seja, as consultas sobre a visão forem mais frequentes que as gravações nas tabelas que a visão usa

Sumário

- Visões
- Visões materializadas
- Aplicações para visões

Aplicações para Visões

- Algumas aplicações para visões são:
 - **Simplificar consultas complexas**
 - Auxiliar em processos de transição
 - Restringir acesso à tabelas
 - Recuperar o resultado mais rapidamente
 - Caso a visão seja materializada

Visões – Simplificar consultas complexas

A consulta abaixo encontre o número, nome e custo de cada projeto

projeto	
idProj*	Nome
1	ACME
2	XYZ

atividade			
idProj*	idAtiv*	custo	Desc
1	1	2000	Ativ 1
1	2	3000	Ativ 2
2	3	3000	Ativ 1
2	2	1000	Ativ 2

```
SELECT idProj, nomeProj, SUM(custo) AS custo
FROM projeto NATURAL JOIN atividade
GROUP BY idProj
```

Resposta		
idProj	nomeProj	Custo
1	ACME	5000
2	XYZ	4000

Visões – Simplificar consultas complexas

A consulta abaixo encontre o número, nome e custo de cada projeto

projeto	
idProj*	Nome
1	ACME
2	XYZ

atividade			
idProj*	idAtiv*	custo	Desc
1	1	2000	Ativ 1
1	2	3000	Ativ 2
2	3	3000	Ativ 1
2	2	1000	Ativ 2

```
SELECT idProj, nomeProj, SUM(custo) AS custo
FROM projeto NATURAL JOIN atividade
GROUP BY idProj
```

Podemos transformar
essa consulta em uma
visão

Resposta		
idProj	nomeProj	Custo
1	ACME	5000
2	XYZ	4000

Visões – Simplificar consultas complexas

```
CREATE VIEW custoProjeto AS (  
  SELECT idProj, nomeProj, SUM(custo) AS custo  
  FROM projeto NATURAL JOIN atividade  
  GROUP BY idProj  
);
```

CustoProjeto <<visão>>		
idProj	nomeProj	Custo

Visão gerada

Obs. Perceba que a visão não possui dado algum.

Visões – Simplificar consultas complexas

A consulta abaixo encontre o número, nome e custo de cada projeto

projeto	
idProj	Nome
1	ACME
2	XYZ

atividade			
idProj	idAtiv	custo	Desc
1	1	2000	Ativ 1
1	2	3000	Ativ 2
2	3	3000	Ativ 1
2	2	1000	Ativ 2

CustoProjeto <<visao>>		
numProj	nomeProj	Custo

```
SELECT *  
FROM custoProjeto
```

Resposta		
idProj	nomeProj	Custo
1	ACME	5000
2	XYZ	4000

Visões – Simplificar consultas complexas

- A consulta abaixo recupera os projetos cujo custo seja superior a média dos custos de todos os projetos
 - **Sem usar** a visão gerada

```
SELECT nomeProj
FROM projeto NATURAL JOIN atividade
GROUP BY idProj
HAVING SUM(custo) >
    (SELECT AVG(soma) FROM
        (SELECT idProj, SUM (custo) AS soma
        FROM projeto NATURAL JOIN atividade
        GROUP BY IdProj
        ) AS tab1
    );
```

Visões – Simplificar consultas complexas

- A consulta abaixo recupera os projetos cujo custo seja superior a média dos custos de todos os projetos
 - **Usando** a visão gerada

```
SELECT nomeProj
FROM custoProjeto
WHERE custo >
    (SELECT AVG (custo)
     FROM custoProjeto
    );
```

Aplicações para Visões

- Algumas aplicações para visões são:
 - Simplificar consultas complexas
 - **Auxiliar em processos de transição**
 - Restringir acesso à tabelas
 - Recuperar o resultado mais rapidamente
 - Caso a visão seja materializada

Visões – Processo de transição

- A tabela “Pes_jur_cont” guarda as pessoas jurídicas contratadas que prestam serviço.
- No entanto, esse nome é deselegante e sujeito a interpretações

PES_JUR_CONT	
*cnpj	char(14)
nomeFantasia	char(30)

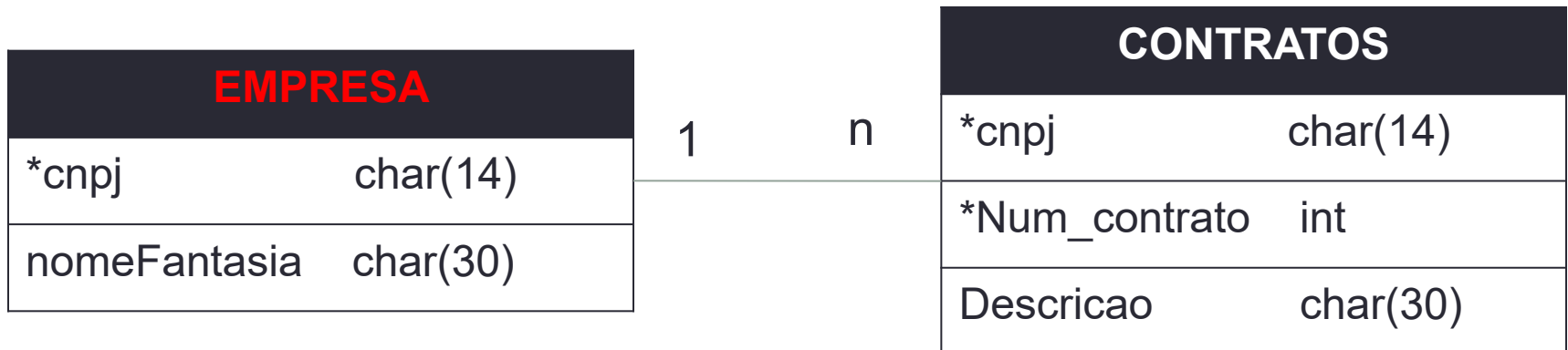
1

n

CONTRATOS	
*cnpj	char(14)
*Num_contrato	int
Descricao	char(30)

Visões – Processo de transição

- A tabela “Pes_jur_cont” guarda as pessoas jurídicas contratadas que prestam serviço.
- No entanto, esse nome é deselegante e sujeito a interpretações
- Desse modo, decidiu-se usar o nome “Empresa”



Visões – Processo de transição

- A tabela “Pes_jur_cont” guarda as pessoas jurídicas contratadas que prestam serviço.
- No entanto, esse nome é deselegante e sujeito a interpretações
- Desse modo, decidiu-se usar o nome “Empresa”
- O que fazer com as aplicações que usam o nome antigo da tabela?
- Como tratar o caso em que algumas aplicações passaram a usar o novo nome e outras continuam usando o nome antigo?

EMPRESA	
*cnpj	char(14)
nomeFantasia	char(30)

1

n

CONTRATOS	
*cnpj	char(14)
*Num_contrato	int
Descricao	char(30)

Visões – Processo de transição

- Ex. visão de transição

```
CREATE VIEW pes_jur_cont AS (  
    SELECT * FROM empresa  
);
```

Empresa	
cnpj	nomeFantasia
1	ACME
2	XYZ

Tabela original

Pes_jur_cont <<visao>>	
cnpj	nomeFantasia
1	ACME
2	XYZ

Visão gerada

- A visão pode ser removida quando houver certeza que todas aplicações já migraram para a nova versão

```
DROP VIEW pes_jur_cont;
```

Aplicações para Visões

- Algumas aplicações para visões são:
 - Simplificar consultas complexas
 - Auxiliar em processos de transição
 - **Restringir acesso à tabelas**
 - Recuperar o resultado mais rapidamente
 - Caso a visão seja materializada

Visões – Restrição de acesso à tabelas

- Ex.
 - `GRANT SELECT(idFunc, nome, idDepto) ON bd1.Func TO joao@localhost;`
 - Com o comando acima, pode-se impedir que o João tenha acesso ao salário de cada funcionário

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões – Restrição de acesso à tabelas

- Ex.
 - E se quiséssemos permitir acesso à folha salarial de cada departamento, sem dar acesso ao salário particular de cada funcionário?
 - Aqui visões podem ajudar

Depto	
*idDepto	int
Nome	char(30)

1

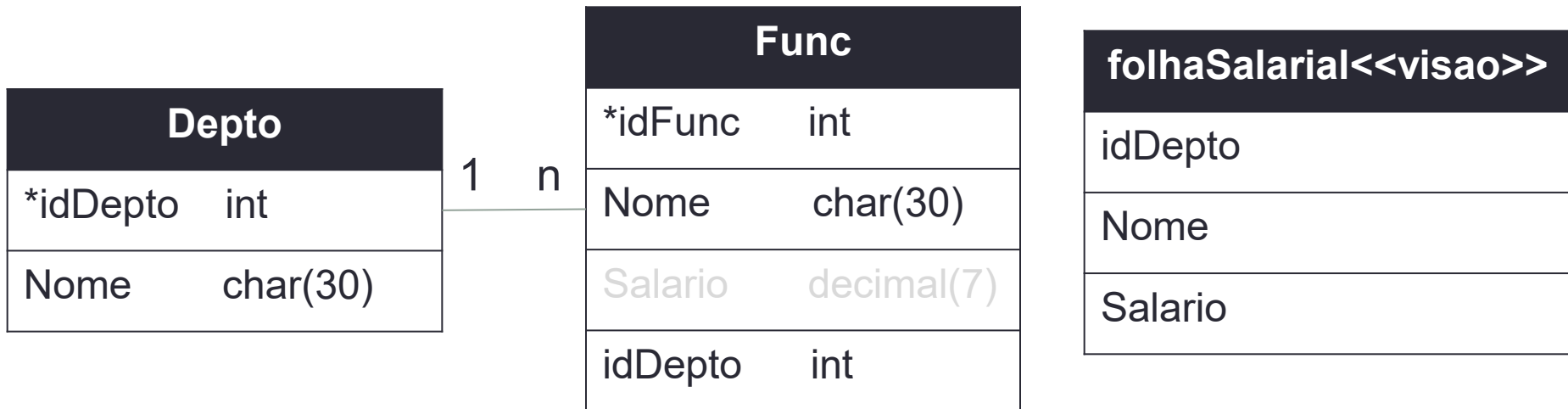
n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões – Restrição de acesso à tabelas

- Ex.

```
CREATE VIEW folhaSalarial AS (  
    SELECT idDepto, nome, SUM (salario) AS salario  
    FROM depto NATURAL JOIN func  
    GROUP BY idDepto  
);  
GRANT SELECT ON bd1.folhaSalarial TO joao@localhost;
```



Aplicações para Visões

- Algumas aplicações para visões são:
 - Simplificar consultas complexas
 - Auxiliar em processos de transição
 - Restringir acesso à tabelas
 - Recuperar o resultado mais rapidamente
 - Caso a visão seja materializada

Visões Materializadas e buscas eficientes

- A folha salarial de cada departamento é uma informação muito requisitada.
- Para acelerar sua busca, decidiu-se adotar uma solução baseada em visões materializadas

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões Materializadas e buscas eficientes

- Assim seria a visão em Postgres

```
CREATE MATERIALIZED VIEW folhaSalarial AS (  
  SELECT idDepto, nome, SUM(salario) FROM func  
  GROUP BY idDepto  
);
```

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões Materializadas e buscas eficientes

- Assim seria a visão em Postgres

```
CREATE MATERIALIZED VIEW folhaSalarial AS (  
  SELECT idDepto, nome, SUM(salario) FROM func  
  GROUP BY idDepto  
);
```

- No entanto, MySQL **não** suporta visões materializadas

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões Materializadas e buscas eficientes

- Assim seria a visão em Postgres

```
CREATE MATERIALIZED VIEW folhaSalarial AS (  
  SELECT idDepto, nome, SUM(salario) FROM func  
  GROUP BY idDepto  
);
```

- Solução MySQL:** Usar uma tabela de resumo

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Visões Materializadas e buscas eficientes

- Uma solução mySQL:
 - **Passo 1:** Criação da tabela de resumo
 - **Passo 2:** Definição da trigger que atualizará a tabela quando um registro de funcionário for inserido
 - **Passo 3:** Definição da trigger que atualizará a tabela quando um registro de funcionário for removido
 - **Passo 4:** Definição da trigger que atualizará a tabela quando um registro de funcionário for alterado

Visões Materializadas e buscas eficientes

- **Passo 1:** Criação da tabela de resumo

```
CREATE TABLE folhaSalarial (  
  idDepto INT,  
  nome CHAR(30),  
  total DECIMAL(7),  
  PRIMARY KEY (idDepto)  
);
```

FolhaSalarial	
*idDepto	int
nome	char(30)
Total	decimal(7)

Visões Materializadas e buscas eficientes

- **Passo 2:** Definição da trigger que atualizará a tabela quando um registro de funcionário for inserido
- Caberá a essa trigger atualizar o registro de folha salarial referente ao departamento do funcionário sendo inserido
- Caso o departamento ainda não esteja registrado em FolhaSalarial
 - Deve-se inserir um novo registro
- Caso já exista
 - Deve-se atualizar esse registro


FolhaSalarial	
*idDepto	int
nome	char(30)
Total	decimal(7)

Visões Materializadas e buscas eficientes

```
CREATE TRIGGER insFunc
AFTER INSERT ON func
FOR EACH ROW
BEGIN
    DECLARE var_idDepto INT;
    SELECT idDepto INTO var_idDepto
        FROM folhaSalarial
        WHERE idDepto = NEW.Iddepto;
    ...
END
```

Visões Materializadas e buscas eficientes


```
CREATE TRIGGER insFunc
AFTER INSERT ON func
FOR EACH ROW
BEGIN
    DECLARE var_idDepto INT;
    SELECT idDepto INTO var_idDepto
        FROM folhaSalarial
        WHERE idDepto = NEW.Iddepto;
    ...
END
```



O valor será diferente de nulo apenas se já existir registro em folhaSalarial com esse id

Visões Materializadas e buscas eficientes

```
CREATE TRIGGER insFunc
AFTER INSERT ON func
FOR EACH ROW
BEGIN
    DECLARE var_idDepto INT;
    SELECT idDepto INTO var_idDepto
    FROM folhaSalarial
    WHERE idDepto = NEW.Iddepto;
    ...
END
```



O próximo slide mostra o conteúdo das reticências

Visões Materializadas e buscas eficientes

CONTINUAÇÃO...

```
IF (var_idDepto IS NULL) THEN
  BEGIN
    DECLARE var_nome CHAR(30);
    SELECT nome INTO var_nome FROM depto
      WHERE depto.idDepto = NEW.Iddepto;
    INSERT INTO folhaSalarial
      VALUES (NEW.IdDepto, var_nome, NEW.salario);
    END;
  ELSE
    UPDATE folhaSalarial SET total = total + NEW.salario
      WHERE idDepto = NEW.idDepto;
  END IF;
```

Visões Materializadas e buscas eficientes

CONTINUAÇÃO...

```
IF (var_idDepto IS NULL) THEN
```

```
  BEGIN
```

```
    DECLARE var_nome CHAR(30);
```

```
    SELECT nome INTO var_nome FROM depto
```

```
      WHERE depto.idDepto = NEW.Iddepto;
```

```
    INSERT INTO folhaSalarial
```

```
      VALUES (NEW.IdDepto, var_nome, NEW.salario);
```

```
  END;
```

```
ELSE
```

```
  UPDATE folhaSalarial SET total = total + NEW.salario
```

```
  WHERE idDepto = NEW.idDepto;
```

```
END IF;
```

Entrará nesse bloco BEGIN-END se esse idDepto ainda não existir em folhaSalarial

Visões Materializadas e buscas eficientes

CONTINUAÇÃO...

```
IF (var_idDepto IS NULL) THEN
```

```
  BEGIN
```

```
    DECLARE var_nome CHAR(30);
```

```
    SELECT nome INTO var_nome FROM depto
```

```
      WHERE depto.idDepto = NEW.Iddepto;
```

```
    INSERT INTO folhaSalarial
```

```
      VALUES (NEW.IdDepto, var_nome, NEW.salarario);
```

```
  END;
```

```
ELSE
```

```
  UPDATE folhaSalarial SET total = total + NEW.salarario
```

```
  WHERE idDepto = NEW.idDepto;
```

```
END IF;
```

Guarda nome do departamento
para usá-lo como valor da coluna
depto de folhaSalarial

Visões Materializadas e buscas eficientes

CONTINUAÇÃO...

```
IF (var_idDepto IS NULL) THEN
```

```
  BEGIN
```

```
    DECLARE var_nome CHAR(30);
```

```
    SELECT nome INTO var_nome FROM depto
```

```
      WHERE depto.idDepto = NEW.Iddepto;
```

```
    INSERT INTO folhaSalarial
```

```
      VALUES (NEW.IdDepto, var_nome, NEW.salario);
```

```
  END;
```

```
ELSE
```

```
  UPDATE folhaSalarial SET total = total + NEW.salario
```

```
  WHERE idDepto = NEW.idDepto;
```

```
END IF;
```

Entrará aqui se o registro já existe. Nesse caso, basta atualizá-lo

Visões Materializadas e buscas eficientes

- **Passo 3:** Definição da trigger que atualizará a tabela quando um registro de funcionário for removido
- Caberá a essa trigger atualizar o registro de folha salarial referente ao departamento do funcionário sendo removido
 - Decrementando o valor total

FolhaSalarial	
*idDepto	int
nome	char(30)
total	decimal(7)

Visões Materializadas e buscas eficientes

```
CREATE TRIGGER dltFunc  
AFTER DELETE ON func  
FOR EACH ROW  
BEGIN  
    UPDATE folhaSalarial SET total = total - OLD.salario  
    WHERE idDeppto = OLD.idDeppto;  
END
```

Exercício

- A folha salarial de cada departamento é uma informação muito requisitada.
- Para acelerar sua busca, decidiu-se adotar uma solução baseada em tabela de resumo em MySQL

Depto	
*idDepto	int
Nome	char(30)

1

n

Func	
*idFunc	int
Nome	char(30)
Salario	decimal(7)
idDepto	int

Exercício

- Para cada departamento, é necessário saber o total da folha e a média dos salários.
- Para atingir o objetivo, o DBA delineou os seguintes passos
 - Passo 1: Criar tabela de resumo para conter os campos de interesse
 - Passo 2: Criar triggers para atualização dessa tabela auxiliar
 - Passo 3: Criar visão que retorne os dados de interesse
 - Passo 4: Criar usuário ana com permissões de acesso
 - Passo 5: Inserir registros para teste
 - Passo 6: obter as médias e totais por departamento através do usuário ana

Exercício

- Passo 1: Criar tabela de resumo para conter os seguintes campos
 - idDepto
 - nomeDepto
 - total_salario
 - quantidade_func

Exercício

- Passo 2: Criar triggers para atualização dessa tabela auxiliar
 - Trigger para inserção de funcionário
 - Trigger para remoção de funcionário
 - Trigger para atualização de funcionário
- **Obs.** Quando o contador de funcionários chegar a zero, o registro deve ser removido da tabela auxiliar

Exercício

- Passo 3: Criar visão que retorne os seguintes dados
 - idDepto
 - nomeDepto
 - Total dos salários
 - Média dos salários
- **Obs.** A visão deverá obter esses valores a partir da tabela auxiliar

Exercício

- Passo 4: Criar usuário ana com permissões de acesso
 - Criar usuário ana
 - Dar ao usuário ana permissão para visualizar todos os esquemas
 - Dar ao usuário ana permissão para buscar dados a partir da visão criada no passo anterior

Exercício

- Passo 5: Inserir os seguintes registros para teste
 - insert into depto values (1,'depto 1'), (2,'depto 2'), (3,'depto 3');
 - insert into func values (1,'joao', 2000,1);
 - insert into func values (2,'ana', 2000,1);
 - insert into func values (3,'jose', 2000,1);
 - insert into func values (4,'marcos', 3000,2);
 - insert into func values (5,'pedro', 1000,2);
 - insert into func values (6,'bia', 4000,2);
 - update func set salario = 2000 where idfunc = 5;
 - insert into func values (7,'paulo', 4000,3);
 - insert into func values (8,'ricardo', 6000,3);
 - delete from func where idfunc >= 7;

Exercício

- Passo 6: obter as médias e totais por departamento através do usuário ana

Atividade Individual

- O objetivo é criar uma consulta que retorne nomes de funcionários que lideraram mais do que 1 projeto em que o número de participantes foi superior a 2 funcionários.
- Resolva de duas formas
 - Sem recorrer à visão
 - Usando uma visão que realiza a soma de participantes de cada projeto

