

Banco de Dados II






*Christiano Cadoná
Edemar Costa Oliveira*

Organizado por Universidade Luterana do Brasil

Banco de Dados II

Universidade Luterana do Brasil – ULBRA
Canoas, RS
2015





Conselho Editorial EAD

Andréa de Azevedo Eick
Astomiro Romais,
Claudiane Ramos Furtado
Dóris Cristina Gedrat
Kauana Rodrigues Amaral
Luiz Carlos Specht Filho
Mara Lúcia Salazar Machado
Maria Cleidia Klein Oliveira
Thomas Heimann

Obra organizada pela Universidade Luterana do Brasil. Informamos que é de inteira responsabilidade dos autores a emissão de conceitos.

Nenhuma parte desta publicação poderá ser reproduzida por qualquer meio ou forma sem prévia autorização da ULBRA.

A violação dos direitos autorais é crime estabelecido na Lei nº 9.610/98 e punido pelo Artigo 184 do Código Penal.

Dados técnicos do livro

Diagramação: Jonatan Souza

Revisão: Igor Campos Dutra

Apresentação

Sejam bem-vindos(as) à disciplina de Banco de Dados II. Nessa disciplina, vocês complementarão seus conhecimentos de banco de dados, permitindo que utilize SGBDs comerciais de maneira eficiente.

Serão apresentadas as principais regras que podem ser implementadas em um banco de dados para garantir a integridade das informações armazenadas, além dos recursos que geralmente os SGBDs disponibilizam como procedimentos, funções e *triggers* para implantar as regras de negócios.

Também serão abordados os conceitos e exemplos de funcionamento das transações em banco de dados, as quais são indispensáveis na correta implementação dos sistemas que utilizam banco de dados como mecanismo de armazenamento.

Vocês conhecerão os principais comandos de PL/SQL, recurso este que possibilita ao administrador de banco de dados autonomia para resolução de problemas relacionados principalmente às regras de negócio de um sistema de informação. Ao estudar este recurso, exercitarão comandos conhecidos da programação, como comandos de seleção e repetição, além de conhecer e exercitar o conceito de cursores em banco de dados.

Será estudado o conceito de concorrência de transações em banco de dados, onde serão explorados os problemas dos processos concorrentes e as alternativas para resolução dos mesmos.

Será apresentada uma visão geral sobre segurança em Sistema de Gerenciamento de Bancos de Dados, com enfoque baseado na arquitetura de bancos de dados relacional, além de algumas arquiteturas em acesso a dados dos Sistemas de Gerenciamento de Bancos de Dados.

De forma ampla, será abordado o uso de banco de dados na aplicação de *Business Intelligence*, apresentando as principais ferramentas utilizadas.

Assim, ao concluir a disciplina, você terá conhecimento necessário para compreender e implementar um projeto de banco de dados.

Prof. Christiano Cadoná
Prof. Edemar Costa Oliveira

Sumário

1	Restrição de Integridade em SGBD	1
2	Transações de Banco de Dados	26
3	Stored Procedure e Function	51
4	Stored Procedure e Function	80
5	Sequence e Trigger	103
6	Cursores.....	130
7	Controle de Concorrência	150
8	Segurança em SGBD	169
9	Arquiteturas de Bancos de Dados: Centralizado, Cliente-servidor, Distribuído e Paralelo.....	186
10	Conceitos e Estratégias de Implantação, Data Warehouse, OLAP e Ferramentas de BI	203

Christiano Cadoná¹

Capítulo **1**

Restrição de Integridade em SGBD

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campus ULBRA Canoas.

Introdução

Uma das principais características de um sistema gerenciador de banco de dados (SGBD) é de manter os dados do banco íntegros. Para isso, eles possuem um conjunto de funcionalidades que podem ser implementadas pelo administrador de banco de dados (DBA), de forma que o mesmo possa criar regras na estrutura de tabelas. Neste capítulo, iremos estudar as principais regras que podem ser implementadas em um banco de dados para garantir a integridade das informações.

Integridade de dados

Uma das principais atividades de um projetista de software e principalmente de um administrador de banco de dados está relacionada à correta definição da modelagem de dados e das tarefas que garantam que os dados armazenados estão íntegros e consistentes.

Dizer que um banco de dados está íntegro é dizer que os dados armazenados estão armazenados de forma correta de acordo com as regras estabelecidas. Por exemplo: garantir que em uma universidade não exista dois alunos com o mesmo código de matrícula; em um cadastro de produto, não permitir que sejam armazenados produtos sem informar o nome do produto; não permitir que seja armazenado em um campo de chave estrangeira um valor que não exista na tabela que este

esteja referenciando; não permitir que em um cadastro de funcionário seja armazenado um valor de salário inferior à zero.

Para isso, os Sistemas Gerenciadores de Banco de Dados (SGBD), possuem regras de integridade que fornecem a garantia de que eventuais mudanças realizadas no banco de dados não resultem em perda da consistência de dados às quais chamamos de restrições de integridade.

Integridade de Vazio ou Restrições sobre valores NULL

A restrição de vazio define se um atributo de uma tabela pode ou não ser vazio (receber NULL). Entende-se como ser vazio quando ao modificar o estado do banco de dados, incluindo ou alterando um registro, o campo que possui essa restrição deve conter ou não uma informação armazenada (se é obrigatório o preenchimento de seu conteúdo). Assim, problemas como a necessidade de informar obrigatoriamente o nome do produto ao incluir um novo produto em uma tabela de produtos está resolvido.

Para definir esse tipo de restrição, basta adicionar a cláusula NOT NULL ao criar ou alterar um atributo de uma tabela. Os Exemplos 1 e 2 exemplificam a definição da restrição NOT NULL nas tabelas. Já o Exemplo 3 apresenta a remoção da cláusula NOT NULL de um atributo.

Exemplo 1: - “Definindo Cláusula NOT NULL ao criar tabela”

```
create table aluno(  
  codigo integer not null,  
  nome varchar(20) not null,  
  idade integer,  
  email varchar (150) not null,  
  constraint alunopk primary key(codigo)  
);
```

Exemplo 2: - “Definindo Cláusula NOT NULL ao alterar tabela”

```
alter table aluno modify idade not null;
```

Exemplo 3: - “Removendo Cláusula NOT NULL de um atributo”

```
alter table aluno modify email null;
```

Observe que, para remover a cláusula NOT NULL de um atributo, basta defini-lo como NULL. Recordando alguns conceitos já trabalhados, em outra disciplina, para que seja possível definir um atributo como de chave primária, seu conteúdo é obrigatório, ou seja, o atributo deve obrigatoriamente possuir a cláusula NOT NULL. É o que ocorre no Exemplo 1 em que o campo “código” está sendo definido como NOT NULL e, em seguida, está sendo criada uma *constraint* de chave primária para o mesmo.

Integridade de chave

No modelo relacional, o conceito de integridade de chave estabelece que existem atributos em uma tabela que são utilizados para identificar um registro entre os demais. Também pode ocorrer a existência de atributos que possam ser utilizados para

estabelecer relações entre tabelas. Essas chaves podem ser classificadas como chaves candidatas, primárias e estrangeiras.

Chaves Candidatas

Uma chave candidata é definida toda vez que existe um determinado atributo ou um conjunto de atributos que possui em seu conteúdo um valor único em relação aos demais registros da mesma tabela, ou seja, o conteúdo armazenado no respectivo atributo não pode se repetir em nenhum outro registro da mesma tabela. Por exemplo, em um cadastro de cliente, não deve existir CPF igual, ou, em um cadastro de produtos, código de barras repetido ou até mesmo, em um cadastro de departamentos, em que não deve ocorrer a existência de departamentos com o mesmo nome. Quando isso ocorrer, é possível defini-lo como um atributo do tipo candidato ou também chamado atributo único. A grande parte do SGBDS implementa esse tipo de atributo definindo uma *constraint* UNIQUE na criação ou alteração do campo. O Exemplo 4 apresenta a definição desse tipo de atributo ao criar e ao alterar um campo de uma tabela.

Exemplo 4: - “Definindo Cláusula UNIQUE ao criar uma tabela”

CREATE TABLE DadosUsuario (

codigo integer not null,

nome varchar(20) **UNIQUE**,

numerofuncionarios integer,

usuario varchar(20),

senha varchar(20));

Em muitos casos, ocorre a necessidade de definir um conjunto de atributos como uma regra de identificação de um registro. Caso ocorra essa condição, é necessário a definição de uma *constraint* contendo como parâmetro na cláusula **UNIQUE** ambos os elementos. No Exemplo 5, é possível identificar a criação desse tipo especial de condição.

Exemplo 5: - “Definindo Cláusula **UNIQUE** com dois ou mais atributos”

```
ALTER TABLE DadosUsuario ADD  
CONSTRAINT DadosUsuario_ususenha UNIQUE(usuario,senha);
```

Caso seja necessário remover uma **UNIQUE**, basta remover a *constraint* que a criou. Observe no Exemplo 6 a linha de comando que remove a **UNIQUE** criada no exemplo anterior. Alguns SGBDs como o **ORACLE**, possibilitam que essas cláusulas possam ser desativadas ao invés de removidas. O Exemplo 7 e 8 apresentam exemplos onde está sendo habilitada e desabilitada a *constraint* **UNIQUE**.

Exemplo 6: - “Removendo Cláusula **UNIQUE**”

```
ALTER TABLE DadosUsuario DROP CONSTRAINT DadosUsuario_  
ususerenha;
```

Exemplo 7: - “Desabilitando a constraint **UNIQUE**”

```
ALTER TABLE DadosUsuario DISABLE CONSTRAINT DadosU-  
suario_ususerenha;
```

Exemplo 8: - “Habilitando a constraint **UNIQUE**”

```
ALTER TABLE DadosUsuario ENABLE CONSTRAINT DadosUsua-  
rio_ususerenha;
```

Quando desabilitada a *constraint* UNIQUE, o banco de dados possibilitará que seja armazenado um registro repetido; porém, ao tentar habilitar novamente essa *constraint*, o SGBD verificará se existe inconsistência e, caso exista, não mais permitirá a operação de habilitar a *constraint* enquanto a inconsistência não seja resolvida, ou seja, antes de habilitar, é necessário que sejam removidos os conteúdos que tornam o campo inconsistente.

Chaves Primárias

Uma chave primária é um atributo que identifica de forma única um registro. De certa forma, é um atributo candidato que foi selecionado como atributo primário. No modelo relacional, toda tabela deve possuir uma chave primária, por mais que a respectiva tabela não se relacione com outra entidade. Diferentemente de uma chave candidata, uma chave primária não deve ser nula, ou seja, o atributo para ser primário deve possuir também integridade de vazio definida. Será através desse atributo que os elementos da tabela serão referenciados por outras tabelas. Da mesma forma que nas chaves candidatas, é possível criar uma chave primária composta, ou seja, em uma tabela, é possível definir que mais de um atributo possa compor a chave primária.

Os Exemplos 9 a 10 apresentam exemplos de como é possível definir uma chave primária em uma tabela. Perceba que os campos que são definidos como chave primária possuem a cláusula NOT NULL também como restrição.

Exemplo 9 - "Criando Chave Primária ao Criar uma Tabela"

```
CREATE TABLE cidade(  
  codcid integer not null,  
  nomecid varchar(30) not null,  
  uf varchar(2),  
  PRIMARY KEY (codcid) );
```

Exemplo 10: - "Criando Chave Primária ao Criar uma Tabela"

```
CREATE TABLE cidade(  
  codcid integer not null,  
  nomecid varchar(30) not null,  
  uf varchar(2),  
  CONSTRAINT cidade_pk PRIMARY KEY (codcid) );
```

Exemplo 11: - "Definindo Chave Primária ao Alterar Tabela"

```
ALTER TABLE cidade ADD CONSTRAINT cidade_pk  
PRI MARY KEY (codcid)
```

Exemplo 12: - "Definindo Chave Primária Composta"


```
CREATE TABLE produto(  
  codigo integer not null,  
  codbarra integer not null,  
  nome varchar(30) not null,  
  estoque integer default 0,  
  CONSTRAINT produto_pk PRIMARY KEY (codigo,codbarra) );
```

Caso seja necessário remover a restrição de chave primária, basta executar uma alteração na estrutura da tabela informando *DROP constraint* e em seguida o nome da *constraint* que foi definida como chave primária (Ex.: ***alter table produto drop constraint produto_pk***). Porém, isso só será possível se a chave primária não fizer referência a uma outra tabela. Caso isso ocorra, primeiro é necessário remover a restrição de integridade referencial para depois sim remover a cláusula de chave primária.

Chaves Estrangeiras

Essa regra determina que um valor ou um grupo de valores possuem correspondência a chaves primárias de uma outra ou da mesma tabela, ou seja, os elementos armazenados na chave estrangeira devem obrigatoriamente existir na chave primária que a mesma está referenciada ou serem nulos (caso isso seja definido). É por meio deste mecanismo que é possível definir as relações entre entidades no modelo relacional.

Na Figura 1, a tabela avaliação está associada à tabela de curso através do atributo “codCurso”. Observe que o conteúdo desta coluna obrigatoriamente deve estar contido na coluna de chave primária da tabela de curso.

Avaliacao			
 codigoAvalia	numMatricula	codCurso	avaliacao
30	1111	5	8,9
31	5555	4	7,5
32	5555	NULL	0
33	2222	5	9,5
34	5555	5	10,0
35	1111	3	8,2


curso	
 codCurso	nomeCurso
5	PHP
4	Java
3	Banco
8	C#

Figura 1

Observe que o registro de avaliação de código “32” não possui nenhum curso vinculado, pois seu conteúdo está vazio. Nesse caso, não está sendo violada a restrição de chave estrangeira, pois possivelmente ao definir o atributo não foi adicionada a restrição de nulo (NOT NULL) no atributo “codCurso” da tabela de avaliação. Outro fator importante é que

a chave primária da tabela de curso não necessita obrigatoriamente estar vinculada a uma avaliação, como é o caso do curso de código 8 que não possui nenhuma avaliação vinculada.

Para criar uma chave estrangeira, basta utilizar a *constraint foreign key* no momento da criação da tabela e ou no comando de alteração de sua estrutura. Os Exemplos 13 e 14 apresentam as duas formas de definição da chave estrangeira “codcurso” da tabela de avaliação. Já o Exemplo 15 apresenta a criação de uma chave estrangeira associada à chave primária de sua mesma tabela (autorrelação).

Exemplo 13: - “Definindo Chave estrangeira ao criar uma tabela”

```
CREATE TABLE avaliacao(  
  codigoAvalia integer not null primary key,  
  numMatricula integer ,  
  codcurso integer,  
  conceito number(15,2),  
  constraint codcurso_fk FOREIGN KEY(codcurso) REFERENCES  
  curso(codcurso)  
);
```

Exemplo 14: - “Definindo Chave estrangeira ao alterar tabela”

```
Alter table avaliação add constraint codcurso_fk foreign  
key(codcurso) references curso(codcurso);
```

Exemplo 15: - “Definindo autorrelação”

```
create table empregado(  
  codemp integer not null primary key,  
  nomeemp varchar(20),  
  salario number(15,2),  
  codresp integer,  
  constraint codresp_fk foreign key(codresp) references  
  empregado(codemp)  
);
```

É importante que fique claro que, para criar uma chave estrangeira, é necessário que a chave primária da tabela cuja chave estrangeira esteja referenciando já tenha sido criada. Outro fator importante a ser observado em um atributo de chave estrangeira é que seu tipo de dado deve ser compatível com o tipo de dado cuja chave primária referencia. Assim, se a chave primária é do tipo inteira, a chave estrangeira também deverá ser do tipo inteira.

Integridade Referencial

Quando definimos uma chave estrangeira, implicitamente definimos uma integridade referencial. A integridade referencial nada mais é do que garantir que um elemento que esteja armazenado na chave estrangeira seja um elemento válido, ou seja, que exista como elemento na chave primária referenciada. Para garantir essa integridade, o SGBD deve se preocupar com ações que essas chaves primárias podem sofrer pelo usuário, como alteração ou até mesmo a remoção do registro. Por exemplo, vamos imaginar que exista a tabela produto e categoria e ambas estão relacionadas em um tipo de relação um para muitos, onde na tabela de produto existe uma chave estrangeira relacionando com a tabela de categoria. A Figura 2 apresenta um exemplo de dados armazenados neste contexto.

categoria		produto				
pkcodCat	nomeCat	pkcodprod	NomeProd	fkcodCat	estoque	valor
50	Fruta	78	Abacaxi	50	200	2,99
51	Açougue	79	Chuleta	51	52	11,80
52	Padaria	21	Uva	50	88	4,99
		585	Banana	50	350	3,50
		25	Picanha	51	10	21,50

Figura 2

Neste contexto, imagine que o usuário necessite remover todo registro de código 50 ("Fruta") da tabela de categoria. Observe que essa categoria faz referência a três registros da tabela de produto. Como ficaria o conteúdo armazenado no campo "fkcodcat" da tabela de produto? Outra situação semelhante ocorre caso seja necessário alterar o código 50 da tabela de categoria para o código 80, como ficariam os elementos relacionados ao respectivo código?

Caso o SGBD permita a remoção ou a alteração do registro da tabela de categoria, não se importando com os elementos relacionados na tabela de produto, tornaria o banco inconsistente, pois este estaria armazenando na tabela de produto um valor que não condiz com a realidade, pois o mesmo não mais existirá.

Para não violar a integridade, os SGBD permitem que seja imposta no momento da definição da chave estrangeira regras que controlam esse tipo de violação. Em geral, são divididas em quatro regras:

- **No action:** desabilita a atualização ou exclusão dos dados da chave primária referenciada. Fazendo uma relação com o exemplo apresentado na Figura 2, não será possível remover o registro ou alterar o código do registro 50 da tabela de categoria, pois esta possui elementos relacionados na tabela de produto. Contudo, se for removido o registro ou alterado o código da categoria 52 da tabela de categoria, o SGBD permitirá, pois não existem registros **vinculados** a ele. Em geral, os SGBD possuem esse **tipo de restrição como a padrão**, ou seja, caso não seja especificado o tipo de relação no momento da criação da chave estrangeira, esta será do tipo NO ACTION.
- **Set to NULL:** quando os dados referenciados (chave primária) são atualizados ou excluídos, todos atributos de chave estrangeira associados a ele são ajustados para NULL. Nesse caso, se a chave estrangeira estiver definida com esse tipo de restrição, e o usuário necessitar remover o registro e ou alterar o código de categoria da tabela de categoria para 50, os três registros associados (Abacaxi, Uva, Banana) terão seu atributo de chave estrangeira fkcodcat alterado para NULL (vazio) e o SGBD realizará a remoção ou alteração da categoria de código 50.
- **Set do Default:** quando os dados referenciados (chave primária) são atualizados ou excluídos, todos os registros associados (chave estrangeira) são ajustados para um valor padrão. Semelhante ao Set to NULL, se for alterado o código (chave primária) ou removido o regis-

tro identificador 50 da tabela de categoria, os registros associados a ele (Abacaxi, Uva, Banana) da tabela de produto, terão o conteúdo alterado para o valor que foi definido como valor *default* do campo *fkcodcat* da tabela produto. Porém, isso só será possível se o mesmo possuir em sua estrutura um valor default (o campo *fkcodcat* contendo um valor default). Dessa forma, o banco de dados realizará a ação proposta pelo usuário no registro da tabela de categoria e alterará o conteúdo das chaves estrangeiras das demais tabelas relacionadas à chave primária de categoria para o valor padrão definido em sua estrutura.

⇒ **Cascade**: o tipo de integridade cascade possui ações diferentes ao alterar a chave primária e ao remover um registro referenciado. Quando o usuário alterar o conteúdo da chave primária, os atributos de outras tabelas que estão referenciados à chave primária alterada, terão seu conteúdo também alterado para o novo valor da chave primária. No nosso exemplo, se a integridade for do tipo cascade e ocorresse a necessidade de alterar o código da categoria da tabela de categoria de 50 para 80, todos os registros relacionados teriam seu conteúdo alterado para o código 80, ou seja, os registros “Abacaxi”, “Uva” e “Banana” da tabela de produtos seriam afetados, alterando o conteúdo do campo *fkcodcat* para 80. Já no caso do usuário remover um registro da tabela de categoria, e esta possuir algum registro em outra tabela referenciado a sua chave primária, o SGBD

fará a remoção também de todos os registros associados (todos os dados do registro associado). Nesse caso, se o usuário remover o registro de código 50 da tabela de categoria, também irá remover automaticamente os registros 78, 21 e 585 da tabela de produtos. A Figura 3 ilustra o uso do tipo cascade, sendo que no Exemplo A está sendo executada, a alteração do código 50 para 80 e no Exemplo B ocorre a remoção do registro de código 50 da tabela de categoria.

A) Exemplo de Alteração do código 50 para 80 da tabela de categoria

categoria		produto				
pkcodCat	nomeCat	pkcodprod	NomeProd	fkcodCat	estoque	valor
80	Fruta	78	Abacaxi	80	200	2,99
51	Açougue	79	Chuleta	51	52	11,80
52	Padaria	21	Uva	80	88	4,99
		585	Banana	80	350	3,50
		25	Picanha	51	10	21,50

B) Exemplo da remoção do registro 50 da tabela de categoria

categoria		produto				
pkcodCat	nomeCat	pkcodprod	NomeProd	fkcodCat	estoque	valor
51	Açougue	79	Chuleta	51	52	11,80
52	Padaria	25	Picanha	51	10	21,50

Figura 3

Agora, imagine com base na Figura 4 o que ocorreria em cada tipo de restrição de integridade caso ocorra a necessidade de alteração do campo “codcidade” ou da remoção de um registro da tabela “tbcidade”.

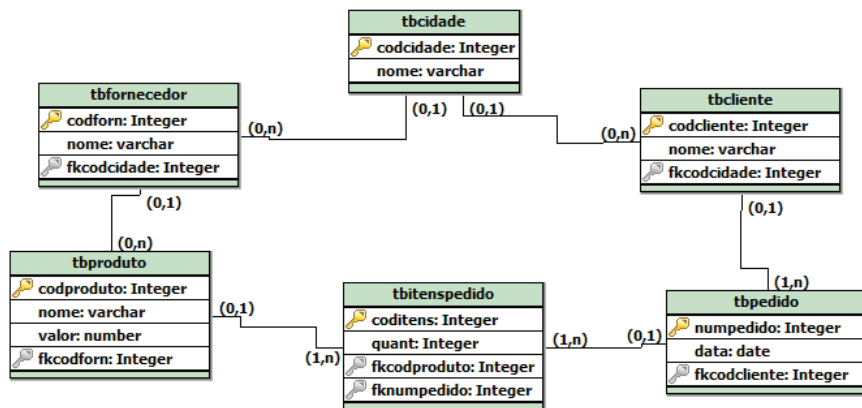


Figura 4

Se o tipo de restrição das chaves estrangeiras estivesse definido como *No Action* e houvesse a necessidade de alterar o conteúdo da chave ou remover um registro da tabela de cidade, somente afetaria a tabela de cidade, e isso, é claro, se não houvesse nenhum registro nas tabelas “*tbfornecedor*” e “*tbcliente*” vinculados ao registro afetado. Caso exista uma ocorrência em algum registro que possua relação com o atributo removido, não será possível removê-lo ou alterá-lo.

Já se o tipo de restrição de integridade referencial seja do tipo “*Set to Default*” ou “*Set to Null*”, o sistema permitiria a execução do comando de alteração e ou remoção do registro na tabela “*tbcidade*”, e todos os registros relacionados ao registro afetado nas tabelas “*tbfornecedor*” e “*tbcliente*” seriam afetados.

O comportamento seria mais impactante se o tipo de restrição estiver definido como do tipo *Cascade*. Nesse caso, se

a ação fosse de alterar a chave primária da tabela “tbcidade”, os registros relacionados nas tabelas “tbfornecedor” e “tbcliente” serão alteradas, ou seja, o conteúdo existente no campo “fkcodcidade” de ambas as tabelas que possua o antigo valor, será alterado automaticamente para o novo valor existente na chave primária da tabela “tbcidade”. Porém, a maior alteração está relacionada se houver a necessidade de remover um registro da tabela “tbcidade”. O SGBD irá remover o registro da tabela “tbcidade” e por sua vez **TODOS** os registros relacionados à chave primária removida, ou seja, os registros das tabelas “tbfornecedor” e “tbcliente” também serão removidos. Mas ao remover esses registros, o banco também removerá os registros relacionados às chaves primárias dos registros afetados das tabelas de fornecedor e cliente. Com isso, poderá ocorrer também remoções de registros nas tabelas “tbprodutos” e “tbpedidos” que possuam relação às chaves primárias removidos em fornecedor e cliente. Se ocorrer alguma remoção na tabela de produtos ou pedidos, a tabela “tbitempedido” também poderá ser afetada se esta possuir algum registro relacionado a essas tabelas. Assim, é possível perceber que uma simples remoção de um registro na tabela de cidade pode acarretar em remoção de vários registros do banco enquanto não for satisfeita a condição de consistência das relações. Perceba nesse exemplo que realmente ocorreu um efeito cascata de remoção em que uma ação poderá acarretar na remoção de registros em todas as tabelas.

Alguns SGBD possibilitam que seja implementado tipos de restrições diferentes de acordo com a ação de alteração ou remoção que os registros sofrerem. Dessa forma, é possível

implementar que ao alterar um registro seja aplicada o tipo *Cascade* e ao remover seja aplicada o tipo *No Action*.

O Exemplo 15 apresenta a implementação de uma *constraint* que determina em ORACLE que o tipo de relação ao realizar um DELETE será “*Cascade*”. Já o Exemplo 16 apresenta a definição de uma *constraint* no SGBD Firebird que, ao realizar um UPDATE, seja implementada o tipo de integridade referencial “*Set to Null*” e, ao realizar um DELETE, seja implementada o tipo “*Cascade*”.

Exemplo 15: - “Aplicação em ORACLE do tipo *Cascade* ao remover”

```
ALTER TABLE aluno ADD CONSTRAINT aluno_codcid  
FOREIGN KEY(fkcodcid)  
REFERENCES cidade(codcid) ON DELETE CASCADE
```

Exemplo 16: - “Aplicação em Firebird de tipos de integridade referencial dependendo da ação”

```
ALTER TABLE aluno ADD CONSTRAINT aluno_codcid FO-  
REIGN KEY (fkcodcid)  
REFERENCES tbcidade(pkcodcid) ON DELETE CASCADE  
ON UPDATE SET NULL;
```

Cada SGBD suporta um conjunto de restrições, alguns suportam todos os tipos outros não. O Oracle não atende todos os tipos de restrições, não sendo possível, por exemplo, executar o Exemplo 6. Já o Firebird, possui todos os tipos de restrições implementadas para ambas as ações (alterar ou remover registros).

Integridade de valor de atributo

Esse tipo de integridade também conhecida como restrição de verificação define que o conteúdo armazenado em um determinado campo da tabela possui uma condição para ser armazenado. Dessa forma, é possível, por exemplo, não permitir que em um atributo que armazene a idade de uma pessoa, seja possível armazenar um número menor do que zero ou superior a 150.

Para isso, utilizamos a *constraint* CHECK que especifica um critério junto a um atributo. O Exemplo 17 apresenta a implementação da CHECK do exemplo relacionado à idade. Já o Exemplo 18 cria uma regra que só permite que sejam armazenados no campo de siglaEstado os estados “RS”, “RO”, e “RJ”.

Exemplo 17: - “Check para não armazenar idade inferior a 0 ou superior a 150”

```
CREATE TABLE aluno(  
    codaluno integer NOT NULL PRIMARY KEY,  
    nomealuno varchar(30) NOT NULL,  
    endereco varchar(40),  
    estado varchar(2),  
    email varchar(100) NOT NULL UNIQUE,  
    idade integer CONSTRAINT check_idade CHECK (idade >=0  
    AND idade <=150)  
)
```

Exemplo 18: - “Check permitindo armazenar apenas os estados RS, RO e RJ”

```
ALTER TABLE aluno ADD CONSTRAINT check_uf  
CHECK (estado IN ('RS','SC','SP'));
```


O Exemplo 19 descreve uma restrição que utiliza o conteúdo de mais de um atributo como regra de armazenamento. Nesse caso, o sistema não está permitindo que seja armazenado um vale superior ao valor do salário do empregado.

Exemplo 19: - “Exemplo de Check validando uma coluna com o resultado de outra”

```
CREATE TABLE empregado(  
    codemp integer not null constraint empr_PK primary key,  
    nome varchar(20),  
    idade integer,  
    salario number(15,2),  
    vale number(15,2),  
    CONSTRAINT valeck CHECK (vale>salario)  
)
```

Asserções

Em banco de dados, uma asserção faz referência a um predicado expressando uma condição que desejamos que o banco de dados sempre satisfaça, ou seja, uma afirmação ou regra. Dessa forma, quando criamos uma asserção, o SGBD somente irá armazenar uma informação se for respeitada a condição da asserção criada.

Como uma asserção é considerada uma regra geral de um banco de dados, sempre que for realizada uma operação no banco, o SGBD testa se a regra criada é mantida, independentemente da tabela envolvida, o que pode ocasionar perda de performance do banco. Alguns SGBD já identificam registros

envolvidos e disparam a asserção somente quando um desses atributos é modificado.

O SQL possui, para criação de uma asserção, a seguinte sintaxe:

```
CREATE ASSERTION <nome da asserção> CHECK <predicado>
```

Onde o nome da asserção representa o nome da regra criada e o predicado identifica a condição para a regra se manter. O Exemplo 20 cria uma asserção que não permite lançamento de uma nota de um aluno negativa.

Exemplo 20: - “Asserção não permitindo notas negativas”

```
CREATE ASSERT nota_negativa CHECK (NOT EXISTS (select * from  
notas where notas < 0) )
```

Por mais que possa ser interessante esse tipo de restrição, os SGBD mais famosos não dão suporte a essa funcionalidade como é o caso do ORACLE. Estes implementam essas regras utilizando outros recursos como *procedures*, *funcion* e *triggers* as quais serão abordadas nos próximos capítulos deste material.

Recapitulando

Como pode ser constatado, a integridade de banco de dados é fundamental para garantir que as informações mantidas no

banco de dados sejam confiáveis. Para isso, os SGBD implementa restrições que auxiliam a manter esses dados confiáveis como a integridade de vazio que permite a obrigatoriedade ou não da informação contida em um atributo, integridade de chave onde foram estudadas as chaves candidatas, primárias e estrangeiras.

Também foi abordado que, na definição de uma chave estrangeira, é possível especificar o tipo de relação com a chave primária a que ela está referenciando, como o tipo No Action, Set to NULL, Cascade e Set do Default. Este capítulo também tratou da possibilidade de criar regras em um atributo específico através da cláusula CHECK e de restrições gerais utilizando as Asserções.

Referências

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. 7. ed. Rio de Janeiro: Campus, 2000.

ELMASRI, Ramez. **Sistema de Banco de Dados**. 6. Ed. São Paulo: Person, 2011.

ROB, Peter. **Sistema de Banco de Dados: Projeto implementação e Gerenciando**. 8 ed. São Paulo: Cengage Learning, 2011.

Atividades

- 1) Assinalar (V) para as assertivas Verdadeiras e (F) para as Falsas.

	V	F	Assertiva
a)	<input type="checkbox"/>	<input type="checkbox"/>	Aplicando regras de integridade, problemas como não permitir que seja informado um valor de um conceito de aluno inferior a zero ou superior a dez serão resolvidos.
b)	<input type="checkbox"/>	<input type="checkbox"/>	Um Banco de dados íntegro é um banco de dados sem informações repetidas como, por exemplo, nome de cidade em um cadastro de cliente.
c)	<input type="checkbox"/>	<input type="checkbox"/>	Integridade de vazio faz referência a obrigatoriedade ou não da existência de dados em um atributo.

- 2) Tendo como base o conteúdo apresentado no Capítulo 1, marque somente a alternativa que está errada.

- a) ☐ No modelo relacional, o conceito de integridade de chave estabelece que existem atributos em uma tabela que são utilizados para identificar um registro entre os demais.
- b) ☐ Uma chave candidata é definida toda vez que existe um determinado atributo ou um conjunto de atributos que possui em seu conteúdo um valor único em relação aos demais registros da mesma tabela.
- c) ☐ Uma chave primária é um atributo que identifica de forma única um registro e, por isso, pode conter somente um elemento nulo armazenado, pois, se existir

mais de um, ela não identifica um elemento de forma única.

- 3) Marque a alternativa correta para remoção somente de uma chave primária já definida em uma tabela.
 - a) ☐ Basta utilizar o comando `alter table` removendo a `constraint` que especifica a chave primária.
 - b) ☐ Utiliza-se o comando `ENABLE PRIMARY KEY` na tabela que possui a chave primária.
 - c) ☐ Somente é possível remover a chave primária se apagar toda tabela.
- 4) A partir dos estudos desenvolvidos nesse capítulo, marque (X) somente nas assertivas verdadeiras (múltipla escolha).
 - a) ☐ O tipo de integridade referencial `No Action` é implementado como `Default` pela maioria dos SGBD.
 - b) ☐ Quando especificado o tipo de Integridade referencial `Set to Null`, não é possível remover um registro que possua referência, pois o elemento referenciado não pode ser nulo.
 - c) ☐ O tipo de integridade referencial `Cascade` não deve ser implementado, pois causa um efeito de remoção de dados em cascata que remove todos os registros do banco.
- 5) Tendo como base o código apresentado, o que a `check` garante.

```
CREATE TABLE produto(  
  codigo integer NOT NULL,  
  nome varchar(20),  
  estoqueAtual integer,  
  valor number(15,2),  
  estoqueMinimo integer DEFAULT 0,  
  CONSTRAINT codigo_PK PRIMARY KEY(codigo),  
  CONSTRAINT estoqueck CHECK (estoqueAtual >= estoqueMinimo)  
)
```

- a) () Determina que o estoque atual deve ser sempre maior ou igual a zero.
- b) () Garante que o estoque atual deve ser sempre maior ou igual ao estoque mínimo.
- c) () Define que o estoque mínimo é maior do que o estoque atual.

Gabarito:

- 1) V, F, V
- 2) c
- 3) a
- 4) a
- 5) b

Christiano Cadoná¹

Capítulo **2**

Transações de Banco de Dados

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campus ULBRA Canoas.

Introdução

Este capítulo tem como objetivo apresentar o conceito e funcionamento das transações em banco de dados. Estas são indispensáveis na correta implementação dos sistemas que utilizam banco de dados como mecanismo de armazenamento. Neste capítulo, serão abordados exemplos práticos em ORACLE, do uso de transações contendo comandos como COMMIT, ROLL-BACK e SAVE POINT.

Conceito de Transações de Banco de dados

Atualmente, não é possível imaginar uma empresa ou um órgão governamental que não opere sem um sistema informatizado. Como já estudado, a grande parte desses sistemas utiliza banco de dados relacional que possui tabelas e relações entre estas tabelas. Cada entidade (tabela) agrupa um conjunto de informações comuns ao domínio que está representando (tabela de produtos, possui informações de produtos; tabelas de clientes, informações de clientes). Algumas funcionalidades desses sistemas necessitam executar rotinas que alteram em vários momentos distintos, a mesma tabela ou um conjunto de tabelas para garantir que atenda a necessidade estabelecida em seu requisito. Por exemplo, em uma transferência bancária, pelo menos devem ocorrer as seguintes ações:

- ➡ Ocorre a diminuição do saldo na conta origem.

- Ocorre o aumento do saldo na conta destino.
- Ocorre o registro da transação bancária.

Nesse caso, para que a funcionalidade de transferência de valores seja alcançada, é necessário que todas as 3 ações (ou instruções SQL) sejam realizadas de forma correta sem erros para garantir que os saldos das contas possuam um valor consistente. Caso uma das operações possua uma falha, mesmo que seja na execução do último comando, deve haver um controle que não permita que as operações já realizadas anteriormente com sucesso passem a valer no banco de dados, tornando o banco de dados inconsistente.

Em situações como essa, em que há necessidade de que, um ou mais comandos SQL sejam executados sem erros para garantir que uma funcionalidade seja executada de forma correta, utilizamos o conceito de transações. Em uma transação, todos os comandos devem ser executados integralmente sem erros ou não serem executados.

Dessa forma, é possível definir que uma transação é uma unidade lógica que compreende uma ou mais declarações SQL executadas por um usuário que devem ser concluídas ou abortadas inteiramente. Não são aceitos estágios intermediários. Uma transação inicia na primeira declaração SQL executável e termina quando ela é explicitamente submetida pelo usuário, ou como ocorre em alguns sistemas gerenciadores de banco de dados quando for encontrado um comando DDL.

Sendo assim, se o exemplo utilizado anteriormente estiver em uma transação e um desses comandos falhar, toda a tran-

sação é desfeita até o estado original de banco de dados que existia antes de seu início. Uma transação bem-sucedida altera o banco de dados de um estado consistente para outro. Geralmente, uma transação está associada diretamente com o conceito de sessão de usuário ao banco de dados a qual será abordada posteriormente neste material.

Cada SGBD possui uma metodologia para identificar o início de uma transação. Alguns SGBD possuem comandos próprios para esse controle como *“Begin Transaction”* ou *“Start Transaction”* e outros, como é o caso do Oracle, iniciam uma transação no momento em que é informado um comando, como uma *“Select”* ou um *“Update”*, por exemplo.

Geralmente, o término de uma transação se dá pela execução do usuário do comando de fim de uma transação (**COMMIT**), onde o banco de dados torna as alterações realizadas pelos comandos permanentes, deixando-as visíveis para outras sessões de usuário. O *Commit* não indica sempre sucesso. Apenas indica explicitamente que o usuário deseja tornar permanente as alterações. No caso do Oracle, todo comando DDL (*create, drop, alter, ...*) ou DCL (*grant* e *revoke*) provoca o fim da transação corrente, promovendo um *commit* implícito.

A qualquer momento antes de confirmar o fim de uma transação (*commit*), alterações feitas por transações através dos comandos de inserção, exclusão ou alteração de registros podem ser desfeitas (revertidas) ou como são chamadas em banco de dados submetidas ao *“rolling back”*. Assim, o *rolling back* consiste em desfazer as alterações já realizadas pelas declarações SQL na transação até o início desta, deixando os

dados afetados inalterados. O comando utilizado para isso é **ROLLBACK**.

Como forma de exemplificar a utilização de transações em banco de dados, o Exemplo 1 apresenta um conjunto de comandos sendo executados. De forma que seja possível ilustrar a aplicação da transação, vamos considerar que antes da execução da transação já exista uma tabela de produtos contendo 3 produtos devidamente cadastrados conforme ilustra a Figura 1.

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	100
2	uva	2,9	41
3	pêra	1,45	230

Figura 1

A transação representada no Exemplo 1 inicia na execução da primeira linha de comando, que lista todos os dados da tabela de produto. O resultado dessa execução é apresentado na Figura 1.

Exemplo 1: - "Exemplo de transação com commit"

Linha	Comandos da Transação
1	select * from produto;
2	insert into produto(codigo,nome,valor,estoque) values(4,'laranja',3.4,60);
3	update produto set estoque=500 where codigo=1;
4	commit;
5	select * from produto;

Observe que a transação possui duas linhas de comando (linha 2 e 3) que modificam o conteúdo da tabela de produto. Após a execução do comando 2, o usuário já poderia verificar o conteúdo armazenado e identificaria que a inserção do produto “laranja” ocorreu, porém a confirmação não foi executada ainda. O mesmo ocorre ao executar o comando da linha 3 que altera o estoque do produto de código 1 (“abacaxi”) para 500 unidades. Contudo, somente após a execução do comando COMMIT da linha 4, os dados realmente serão confirmados no banco de dados. Fazendo uma analogia a outras funcionalidades de outros aplicativos, o commit poderia ser comparado ao ato de salvar de um editor de texto. Quando abrimos um documento e realizamos modificações no seu conteúdo, somente após salvarmos o mesmo ocorre a alteração física do arquivo editado no computador. Observe que após a execução do comando 4, a transação iniciada no comando 1 é concluída. Isso significa que o comando 5 apresentado no Exemplo 1 dá início a uma nova transação. O seu resultado pode ser observado na Figura 2, que apresenta a tabela alterada pela transação anterior.

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	41
3	pêra	1,45	230
4	laranja	3,4	60

Figura 2

Agora, observe o comportamento do trecho de comandos apresentado no Exemplo 2 que tem como base os dados apre-

sentados na Figura 2. No conjunto dos comandos é identificada a existência de duas transações, sendo a primeira que compreende os comandos 1 à 5 e a segunda dos comandos 6 a 8.

Exemplo 2: - “Exemplo de transação utilizando rollback”

Linha	Comandos Executados
1	update produto set estoque=800 where codigo=2;
2	delete from produto where codigo=3;
3	insert into produto(codigo,nome,valor,estoque) values(5,'morango',4.70,50);
4	select * from produto order by codigo;
5	rollback;
6	insert into produto(codigo,nome,valor,estoque) values(6,'pêssego',1.50,70);
7	select * from produto order by codigo;
8	commit;

Na primeira transação, os comandos das linhas 1 a 3 alteram o estoque do produto 2 (“uva”) para 800 unidades, removem o produto 3 (“laranja”) e inserem o produto morango na tabela de produtos. Quando executado o comando 4 que lista todos os registros da tabela de produto, constatamos que as ações dos comandos já realizados foram implementadas no banco. Isso pode ser identificado na Figura 3 que apresenta o resultado do comando no momento de sua execução.

Produto 3 removido 

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	800
4	laranja	3,4	60
5	morango	4,7	50

Figura 3

Contudo, ao executar o comando 5 (ROLLBACK), o banco de dados desfaz todos os comandos já executados na transação iniciada no comando 1 e finaliza a transação. Dessa forma, o banco de dados volta a possuir as mesmas informações existentes da Figura 2. É importante destacar que o comando de desfazer não desfaz parte da transação e sim todos os comandos já executados na transação e a finaliza.

Após executada a segunda transação (linhas 6 a 8 do Exemplo 2) que insere um novo registro na tabela (“pêssego”) e a finalizada com o comando COMMIT, o banco de dados foi novamente alterado e a inserção de dados passa a se confirmar. A Figura 4 apresenta como ficou a tabela de produto após a execução da segunda transação.

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	41
3	pêra	1,45	230
4	laranja	3,4	60
6	pêssego	1,5	70

Figura 4

Observe que os produtos 2 e 3 não foram alterados, e o produto 5 não foi adicionado à tabela. Já o produto pêssego que foi comitado na segunda transação já faz parte de forma definitiva da tabela de produto. Ainda relacionado ao comando de desfazer (rollback), é importante destacar que não é possível desfazer um commit, ou seja, caso após a execução da linha 8 do Exemplo 2 existisse o comando rollback, o banco de dados não iria desfazer o comando da linha 6 que inseriu o produto pêssego na tabela de produto.

Em alguns casos como na existência de transações longas e com muitas declarações SQL é possível criar marcadores intermediários chamados de **savepoints**. Esses marcadores são utilizados para em certas situações realizar um *rolling back* até o devido marcador, sem que a transação seja concluída. Assim, é possível desfazer parte dos comandos realizados e ainda manter a transação ativa. Para criar um savepoint, basta defini-lo dando um nome ao mesmo (**SAVEPOINT nome_do_savepoint;**). Caso queira desfazer os comandos até o savepoint criado, basta digitar o comando **ROLLBACK TO SAVEPOINT** informando o nome do *savepoint* definido (**ROLLBACK TO SAVEPOINT nome_do_savepoint**). Com base no resultado final apresentado na Figura 4, observe o que ocorre na execução da transação do Exemplo 3.

Exemplo 3: - “Exemplo de transação utilizando savepoint”

Linha	Comandos Executados
1	update produto set estoque=400 where codigo=4;
2	update produto set estoque=900 where codigo=6;
3	savepoint ponto1;
4	insert into produto(codigo,nome,valor,estoque) values(7,'melão',6.80,20);
5	delete from produto where codigo=2;
6	savepoint ponto2;
7	delete from produto where codigo=1;
8	rollback to savepoint ponto1;
9	update produto set estoque=700 where codigo=3;
10	commit;


Os dois primeiros comandos alteram o estoque do produto laranja para 400 unidades e do produto pêssego para 900 unidades. O *savepoint* “ponto1” apresentado na linha 3 não realiza nenhuma confirmação, apenas marca um ponto na transação que inicia na linha 1 e termina na linha 10 com o *commit*.

Já os comandos das linhas 4 e 5 inserem o produto “melão” e removem o produto “uva”. A Figura 5 apresenta o conteúdo da tabela de produto após a execução dos *savepoint* ponto1 e ponto2.

Banco de dados após savepoint ponto1

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	41
3	pêra	1,45	230
4	laranja	3,4	400
6	pêssego	1,5	900

Produto 2
removido



Banco de dados após savepoint ponto2

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
3	pêra	1,45	230
4	laranja	3,4	400
6	pêssego	1,5	900
7	melão	6,8	20

Figura 5

O comando 7 realiza a remoção do produto 1 (“abacaxi”) do banco de dados. Já o comando da linha 8 realiza um desfazer até a marca chamada “ponto1”, ou seja, o sistema gerenciador de banco de dados deverá desfazer todos os comandos até a marca “ponto1”. Com isso, os comandos de remoção dos produtos 1 e 2 e a inclusão do produto “melão” serão desfeitas, mantendo os dados idênticos ao apresentado no ponto1 da Figura 5. É importante informar que todos os comandos foram desfeitos inclusive o que cria a marca ponto 2, não podendo mais este ser utilizado.

Quando executado o comando 10 de confirmação, é possível concluir que somente os comandos das linhas 1, 2 e 9 foram executados no banco. A Figura 6 apresenta o resultado da execução completa da transação existente no Exemplo 3.

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	41
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Figura 6

Caso você queira testar os comandos apresentados nos Exemplos 1 a 3 no banco de dados, tenha a garantia de que a ferramenta de conexão com o banco de dados possui esse suporte. Geralmente, essas ferramentas possuem como padrão autocommit, ou seja, após execução de qualquer comando SQL, a ferramenta executa o comando commit.

Mais de uma Transação Acessando o Banco de Dados

Certamente, cada SGBD deve possuir processos muito bem elaborados para possibilitar que o banco de dados possua mais de uma transação simultânea. Alguns SGBD criam uma transação para cada conexão com o banco de dados, e outros, possibilitam que uma única conexão com o banco possua mais de uma transação simultânea.

Uma característica importante dos bancos transacionais é que cada transação somente acessa as informações permanentes do banco de dados, ou seja, caso exista duas transações simultâneas e uma delas altere um registro, e a outra consulte os dados do registro alterado pela primeira transação sem que esta tenha sido concluída (recebido um commit), o banco de dados apresentará o conteúdo anterior à alteração da primeira transação. Isso é necessário, pois, como a primeira transação ainda não foi concluída, pode ocorrer que a mesma venha a ser desfeita (receber um rollback), e a segunda transação mostraria um valor inválido.

Para exemplificar, vamos utilizar como base a tabela de produtos já trabalhada neste capítulo. Vamos definir que o estado permanente do banco de dados antes das transações serem executadas no Exemplo 4 está representado na Figura 6.

○ Exemplo 4 apresenta a execução de duas transações distintas “A” e “B”. Observe que o banco de dados processa uma operação por vez a cada ciclo de tempo, aqui representado pelos tempos do intervalo “t1” à “t9”.

Exemplo 4: - "Transações Simultâneas"

#	Transação "A"	Transação "B"
t1	update produto set valor=2.50 where codigo=1;	:
t2	select * from produto order by codigo;	:
t3	:	select * from produto order by codigo;
t4	:	delete from produto where codigo=2;
t5	:	select * from produto order by codigo;
t6	select * from produto order by codigo;	:
t7		commit;
t8	select * from produto order by codigo;	
t9	Commit;	

Representa o tempo percorrido no banco de dados

No tempo "t1", o banco de dados executou a transação "A" alterando o valor do produto de código 1 para 2,50. No tempo "t2", a mesma transação "A" realiza uma consulta a todos os registros da tabela de produto. Observe que a mesma transação já visualiza os dados do produto de código abacaxi alterado. A Figura 7 apresenta o resultado da consulta.

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	2,5	500
2	uva	2,9	41
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Figura 7

Já no tempo “t3”, o banco de dados inicia a transação “B” listando todos os dados da tabela produto. A Figura 8 apresenta o resultado das consultas dos tempos “t3” e “t5”. Observe que a consulta do tempo “t3”, não mostra a alteração realizada pela transação “A” no tempo “t1”, por mais que a mesma tenha sido realizada anteriormente no tempo “t3”.

Resultado do comando no tempo “t3”

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
2	uva	2,9	41
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Resultado do comando no tempo “t5”

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	4,3	500
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Figura 8

No tempo “t4”, o banco de dados realiza o comando de remoção do produto de código 2 presente na transação “B”. Em seguida, a transação “B” lista todos os dados do banco de dados. Observe que no instante “t5” a transação “B” lista os produtos na Figura 8 sem o código 2, porém ainda sem a alteração do produto abacaxi realizado pela transação “A” no instante “t1”.

No tempo “t6”, o banco de dados executa um comando na transação “A” listando todos os dados do banco. A Figura 9 ilustra o resultado das pesquisas nos tempos “t6” e “t8” realizadas na transação “A”. Observe que no tempo “t6” as informações apresentadas são as mesmas do tempo “t2” ilustrado na Figura 7, pois, até este momento, a transação “B” não foi concluída.

Resultado do comando no tempo “t6”

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	2,5	500
2	uva	2,9	41
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Resultado do comando no tempo “t8”

Produto 2
removido

produto			
CODIGO	NOME	VALOR	ESTOQUE
1	abacaxi	2,5	500
3	pêra	1,45	700
4	laranja	3,4	400
6	pêssego	1,5	900

Figura 9

Já no tempo “t7”, novamente a transação “B” é executada pelo SGBD que recebe o comando para concluir a transação, alterando o estado do banco de dados e sua informação. Quando executado o comando de listar todos os registros no tempo “t8” o banco de dados já apresenta à transação “A” os dados alterados pela transação “B” (produto uva removido), o que pode ser constatado na Figura 9.

Somente após a execução do comando de finalização no tempo “t9” que o valor do produto abacaxi será persistido no banco de dados, e estará disponível para a próxima transação.

Tipos de transação

Alguns bancos de dados possibilitam que se defina o tipo de comando que uma transação pode suportar (leitura/gravação), como, por exemplo, definir que uma transação apenas realize leitura no banco de dados não permitindo que comandos que possam alterar os dados armazenados sejam recuperados. No Oracle, é possível definir esses dois tipos de transação utilizando o comando SET TRANSACTION. Para realizar essa defini-

ção, a primeira linha de comando da transação deve ser SET TRANSACTION. Assim, é possível definir:

- SET TRANSACTION READ ONLY: quando o conjunto de comandos a ser executado em uma transação possui apenas comandos de consulta a dados não possibilitando que seja realizada qualquer alteração aos dados da tabela. Esse tipo de definição é indicada para transações que executem consultas em várias tabelas do banco de dados ao mesmo tempo em que essas tabelas estão sendo alteradas por outros usuários.
- SET TRANSACTION READ WRITE: utilizado quando a transação irá executar tanto comandos de leitura a dados como também de manipulação dos dados armazenados. Caso não informando, esse é o tipo padrão definido pelos SGBD.

Também é possível definir o tipo de isolamento que a transação possui. Essa isolamento especifica como transações que contêm modificações no banco de dados são manipuladas. O Oracle possui dois tipos de isolamento:

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE: quando setado esse tipo de isolamento, a transação somente vê as alterações que foram finalizadas com sucesso no momento em que a transação começou, além das alterações feitas pela própria transação.
- SET TRANSACTION ISOLATION LEVEL READ COMMITTED: quando selecionada essa isolamento, cada consulta executada por uma transação vê apenas dados que fo-

ram finalizados com sucesso antes da consulta e não os dados afetados pela própria transação.

Propriedades desejáveis das transações

Como já abordado, as transações são um recurso importante disponibilizado pelos sistemas gerenciadores de banco de dados. Porém, para que estas funcionem de forma correta, os SGBD devem garantir algumas propriedades às transações chamadas propriedades **ACID** (Atomicidade, Consistência, Isolamento e Durabilidade).

- ➔ Atomicidade: uma transação é uma unidade de processamento atômica, ou seja, ela é indivisível. Nesse caso, as transações devem ser executadas em suas totalidade ou não executar nada. Sendo assim, ao término de uma transação, o banco de dados não pode possuir parte da execução da transação. Em caso de sucesso, este deve efetuar as alterações de todas as declarações SQL, e, em caso de falha, desfazer todas as alterações realizadas. Nesse caso, vamos imaginar que estamos realizando um pagamento de uma conta bancária em um terminal de autoatendimento, e, ao pressionar o botão de confirmação de pagamento, o sistema deva realizar várias rotinas em uma transação, como retirar o valor da conta pagadora, dar baixa do documento e transferir o valor à conta credora. Agora, imagine que assim que executado o primeiro comando de retirar o valor da conta pagadora, ocorra uma queda de energia, ou

uma falha de comunicação com o servidor? O banco de dados não pode concluir a transação sem que este receba o comando de conclusão de transação. Caso o mesmo não receba, o banco (se configurado) pode disparar um comando de desfazer, garantido que a transação não foi realizada. Dessa forma, podemos concluir que a propriedade de atomicidade garante “é tudo ou nada executado”.

- Consistência: uma transação deve preservar a consistência, significando que, se ela for completamente executada do início ao fim sem interferência de outras transações, deve levar o banco de dados de estado consistente para outro. Nesse caso, é possível fazer uma analogia com uma fotografia. É como se tirássemos uma foto do banco antes e após a execução da transação, e comparássemos ambas. Elas não devem possuir diferença em sua consistência. Sendo assim, uma transação não deve tornar o banco inconsistente. O SGBD utiliza as regras de integridade como mecanismo de garantia da integridade dos dados para auxiliar no processo de consistência da transação. Para garantir a consistência, o programador deve tomar cuidado com os comandos que este executa nas transações.
- Isolamento: em sistemas multiusuários como um sistema Web, por exemplo, é comum que ocorra a execução de várias transações simultâneas, porém o SGBD deve garantir que cada uma dessas transações pareça ser executada de forma isolada da outra, ou seja, a execução de uma transação não deve ser interferida por quaisquer

outras transações que acontecem simultaneamente. Por exemplo, imagine que exista duas transações simultâneas onde a primeira está alterando o valor unitário de um produto e a outra está lendo os dados do mesmo produto. O SGBD deve possuir mecanismos que garantam que ambas as transações não se conversem, e somente apresente o valor unitário alterado da segunda transação após a primeira estar concluída. Geralmente, o isolamento é implementado no processo de controle de concorrência imposto pelo SGBD o qual será abordado nos próximos capítulos deste material.

- ➔ Durabilidade: as mudanças aplicadas ao banco de dados pela transação confirmada (que recebeu commit) precisam persistir no banco de dados (fazer parte do banco de dados). Essas mudanças não devem ser perdidas por causa de alguma possível falha. Em caso de commit, o SGBD torna permanente as alterações realizadas por uma transação. Pode parecer estranho, mas a grande parte dos SGBD ao realizar um commit não armazena fisicamente na tabela os dados alterados, e sim de tempos em tempos, para aumentar a performance do banco. Caso ocorra uma falha nesse período de tempo, o banco de dados deve garantir que as informações estejam realmente salvas. O mecanismo que geralmente implementa essa funcionalidade chama-se controle de falhas e também será abordado nos próximos capítulos deste material.

Sendo assim, é possível perceber que as transações estão relacionadas com outras funções dos sistemas gerenciadores de banco de dados e funcionam em conjunto com estes.

Recapitulando

Este capítulo apresentou conceitos relacionados a transações de banco de dados, no qual estudamos a importância de utilizar transações no desenvolvimento de sistemas. Aprendemos o uso do comando de finalização de transação `commit` que realiza a persistência de dados. Também foi abordado e exemplificado o uso de “desfazer” (`rollback`) em uma transação, mantendo os dados anteriores ao início da transação. Foi conceituado e exemplificado o uso de `savepoint` como mecanismo de controle de transações longas.

Este capítulo também abordou o comportamento de transações simultâneas exemplificando o que cada uma enxerga do banco de dados. O capítulo também apresentou os tipos de transações e o nível de isolamento destas.

Por fim, o capítulo abordou as propriedades de uma transação (atomicidade, consistência, isolamento e durabilidade) chamadas propriedades ACID.

Referências

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. 7. ed. Rio de Janeiro: Campus, 2000.

ELMASRI, Ramez. **Sistema de Banco de Dados**. 6. Ed. São Paulo: Person, 2011.

ROB, Peter. **Sistema de Banco de Dado: Projeto implementação e Gerenciando**. 8 ed. São Paulo: Cengage Learning, 2011.

Atividades

Tendo como base o estado inicial do banco apresentado da Figura 10, antes da execução do bloco de comandos, responda as questões que seguem:

ALUNO		
CODIGO	NOME	IDADE
1	Ana	22
2	Carolina	25
3	Maria	19

Figura 10 Estado inicial do banco de dados.

Bloco de comandos executado

Linha	Comandos Executados
1	<code>update aluno set idade=50 where codigo=2;</code>
2	<code>savepoint x1;</code>
3	<code>insert into aluno(codigo, nome, idade) values (4,'Joana',19);</code>
4	<code>delete from aluno where codigo=3;</code>
5	<code>savepoint x3;</code>
6	<code>delete from aluno where codigo=2;</code>
7	<code>rollback to savepoint x1;</code>
8	<code>insert into aluno(codigo, nome, idade) values (5,'Mariana',20);</code>
9	<code>rollback;</code>
10	<code>insert into aluno(codigo, nome, idade) values (6,'Bete',18);</code>
11	<code>update aluno set idade=80 where codigo=1;</code>
12	<code>commit;</code>

- 1) O bloco de comandos apresentado possui quantas transações?
- a) ☐ 1 transação
 - b) ☐ 2 transações
 - c) ☐ 3 transações
 - d) ☐ 5 transações
 - e) ☐ 7 transações
 - f) ☐ 12 transações
- 2) Após a execução da linha 5, quais registros foram afetados na transação?
- a) ☐ O registro de código 2 seria alterado, o registro de código 4 seria inserido no banco, o registro de código 3 seria removido do banco.
 - b) ☐ O registro de código 2 seria alterado e a transação concluída. O registro de código 4 seria inserido no banco, o registro de código 3 seria removido do banco e em seguida a transação seria concluída.
 - c) ☐ Somente o comando da linha 1 seria concluído, pois o *savepoint* desfaz todos os comandos executados.
 - d) ☐ Nenhum comando será executado, pois o *savepoint* aguarda a execução de um *commit* para mostrar o resultado da transação que é concluída na linha 12.

- 3) Após a execução de todos os comandos (12 linhas), quais registros foram removidos.
- a) ☐ Foi removido o registro de código 2.
 - b) ☐ Foi removido o registro de código 3.
 - c) ☐ Foram removidos os registros de código 2 e 3.
 - d) ☐ Nenhum registro foi removido.
- 4) Após a execução da linha de comando 8, quais registros foram afetados na transação?
- a) ☐ O registro de código 2 seria alterado, o registro de código 4 seria inserido no banco, o registro de código 3 seria removido do banco.
 - b) ☐ O registro de código 2 seria alterado, o registro de código 4 seria inserido no banco, os registros de código 3 e 4 serão removidos do banco, será inserido um registro de código 5.
 - c) ☐ O registro de código 2 seria alterado e será inserido um registro de código 5.
 - d) ☐ Apenas será inserido um registro de código 5 no banco.
- 5) Após a execução de todos os comandos (12 linhas), qual é o estado do banco (quais registros foram afetados)?
- a) ☐ Foi incluído o aluno de código 8 e alterado o registro de código 1.

- b) () O registro de código 1 e 2 foram alterados, o registro de código 4, 5 e 6 foram inseridos no banco, os registros de código 2 e 3 foram removidos do banco.
- c) () O registro de código 2 seria alterado e será inserido um registro de código 5.
- d) () Não será realizado nenhum comando, pois existe um rollback.

Gabarito:

- 1) b
- 2) a
- 3) d
- 4) c
- 5) a

Christiano Cadoná¹

Capítulo **3**

Stored Procedure e Function

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campus ULBRA Canoas.

Introdução

Neste capítulo, será apresentado o conceito e exemplos práticos de como funciona os Stored Procedures e Function em um banco de dados. Será utilizada como base a sintaxe do banco Oracle, além do uso do SQL developer como ferramenta de apoio aos testes realizados no banco de dados.

Conceito de Procedimentos e Funções

Quando estudamos alguma linguagem de programação, os autores de bibliografias sempre enfatizam a utilização de funções e procedimentos como forma de organizar e otimizar rotinas. Da mesma forma, isso ocorre dentro de banco de dados. A grande parte do SGBD dá suporte a esse recurso, e com ele é possível que grande parte das regras de negócio, passem a ser implementadas no banco de dados e não mais codificadas no sistema.

Um procedimento pode ser definido como um grupo de comandos em SQL que executam uma determinada tarefa. A execução de um procedimento pode ser por meio da chamada de um programa externo ao SGBD ou explicitamente realizada pelo usuário dentro de uma função, procedimento ou mesmo dentro do banco de dados. Por exemplo, é possível que um procedimento criado no banco de dados possa ser chamado dentro de uma aplicação desenvolvida em linguagem C#, ou Java (desde que esse programa esteja conectado ao banco de dados).

A implementação de uma função é muito semelhante à de um procedimento, porém possui uma característica específica em relação ao procedimento. Uma **função**, além de executar um conjunto de comandos, **sempre devolve uma informação**.

A sintaxe dos procedimentos e funções dependerá do banco de dados em que forem implementadas, pois cada um possui sua linguagem própria, contudo as características são semelhantes, contendo inicialmente uma seção de especificações e em seguida um corpo contendo o conteúdo a ser executado. No Oracle, a sintaxe de procedure e function pode ser especificada como:

PROCEDURE

```
CREATE [OR REPLACE] PROCEDURE [schema.]nome_da_procedure  
[(parâmetro1 [modo1] tipodedado1,  
    parâmetro2 [modo2] tipodedado2,  
    ...)]
```

IS|AS

--declaração de variáveis

Bloco PL/SQL

FUNCTION

```
CREATE [OR REPLACE] FUNCTION [schema.]nome_da_function  
[(parâmetro1 [modo1] tipodedado1,  
    parâmetro2 [modo2] tipodedado2,  
    ...)] RETURN TIPO_DE_DADO
```

IS|AS

--declaração de variáveis

Bloco PL/SQL

Onde:

- **OR REPLACE** – indica que, caso a procedure/function exista, ela será eliminada e substituída pela nova versão criada pelo comando.
- **BLOCO PL/SQL** – inicia com uma cláusula **BEGIN** e termina com **END** ou **END nome_da_procedure/END nome_da_function**. É no bloco de comandos que estão as ações que serão executadas pela procedure ou function.
- **NOME_DA_PROCEDURE** – indica o nome da procedure. Após definido o nome da procedure, esta se torna um termo reservado do banco de dados, ou seja, não será possível criar uma tabela com esse nome, por exemplo.
- **NOME_DA_FUNCTION** – indica o nome da função. Após definido o nome da function, esta se torna um termo reservado do banco de dados, ou seja, não será possível criar uma tabela com esse nome, por exemplo.
- **PARÂMETRO** – indica o nome da variável PL/SQL que é passada na chamada da procedure/function ou o nome da variável que retornará os valores da procedure/function ou ambos. O que irá conter em parâmetro depende de MODO.
- **MODO** – Indica que o parâmetro é de entrada (**IN**), saída (**OUT**) ou ambos (**IN OUT**). É importante notar que IN é o modo default, ou seja, se não dissermos nada, o MODO do nosso parâmetro será, automaticamente IN.

Por exemplo, vamos imaginar que exista a necessidade de informar para uma função um determinado código como parâmetro. Nesse caso, o MODO será IN, pois a função apenas receberá como parâmetro o código. Já no caso de um procedimento, executar uma rotina e o mesmo necessitar retornar uma informação, somente se este possuir um parâmetro do tipo OUT.

- **TIPODE DADO** – indica o tipo de dado do parâmetro. Pode ser qualquer tipo de dado do SQL ou do PL/SQL. Pode usar referências como %TYPE, %ROWTYPE ou qualquer tipo de dado escalar ou composto. Atenção: não é possível fazer qualquer restrição ao tamanho do tipo de dado nesse ponto, como, por exemplo, determinar o tamanho de uma variável do tipo varchar.
- **RETURN TIPO_DE_DADO** – Define o tipo de dado a ser apresentado como retorno da função definida. Toda função possui obrigatoriamente um tipo de dado como retorno.
- **IS|AS** – a sintaxe do comando aceita tanto IS como AS. Por convenção, usamos IS na criação de procedure e function e AS quando estivermos criando pacotes.

Como forma de demonstrar a funcionalidade dos procedimentos e funções, o Exemplo 1 apresenta a criação de uma tabela de funcionário, juntamente com a inserção de 3 registros que serão utilizados como base para exemplificar a aplicabilidade dos procedimentos e funções.

Exemplo 1: - “Criação do cenário para testar dos exemplos”

```
create table tbfuncionario(  
  codfunc integer not null,  
  nomefunc varchar(20),  
  salariofunc number(15,2),  
  constraint funcionario_PK primary key(codfunc)  
);  
insert into tbfuncionario(codfunc,nomefunc,salariofunc) values (1,'Maria',2000);  
insert into tbfuncionario(codfunc,nomefunc,salariofunc) values (2,'Debora',1500);  
insert into tbfuncionario(codfunc,nomefunc,salariofunc) values (3,'Rosana',3500);
```

Inicialmente, vamos criar um procedimento que possui como objetivo aumentar o salário de um funcionário que possui um determinado código. O valor a ser acrescido sobre o atual salário do funcionário será passado como parâmetro e representa o percentual de aumento. Nesse caso, será criado um procedimento chamado **“aumenta_sal”** contendo dois parâmetros de entrada, sendo o primeiro representando o código do funcionário que terá seu salário acrescido, e o segundo o percentual a ser calculado. O Exemplo 2 apresenta uma alternativa de solução do problema proposto.

Exemplo 2: - “Criar procedure aumenta_sal”

```
1  CREATE OR REPLACE PROCEDURE aumenta_sal  
2  (codigo IN tbfuncionario.codfunc%TYPE,  
3  perc IN number)  
4  IS  
5  BEGIN  
    UPDATE tbfuncionario  
6  SET salariofunc = salariofunc + (salariofunc * perc / 100)  
7  WHERE codfunc = codigo;  
7  END aumenta_sal;
```

O Exemplo 2 possui como variáveis de entrada “codigo” e “perc”. A variável “perc” é do tipo number, já a variável “codigo”, possui como tipo uma referência, ou seja, ela possui o mesmo tipo de dado que o atributo “codfunc” da tabela de funcionário possui, que nesse caso é integer. Observe que não foi informado a precisão no tipo number, pois como constatado, o Oracle não permite que seja restringido o tamanho do campo na definição dos parâmetros de um procedimento e ou função.

Observe que a linha 6 apresenta a execução de um comando que altera o salário de um funcionário. Nesse comando, são inseridas as variáveis passadas como parâmetro, e o interpretador do Oracle, ao chamar o procedimento utiliza o conteúdo informado nessas variáveis como valores para executar o comando de forma correta.

O ponto e vírgula é utilizado como identificador de conclusão do comando dentro do bloco de comando. Dessa forma, é possível iniciar um comando em uma linha e continuar na linha subsequente, que o interpretador do Oracle irá concatenar todas as linhas em um único comando de dados até que seja encontrado o ponto e vírgula para execução do comando. Isso é percebido na linha de comando 6 do Exercício 2 que possui duas quebras de linha antes do ponto e vírgula.

Como já abordado, um procedimento criado no SGBD pode ser chamado por uma aplicação externa ao banco como, por exemplo, dentro de um código PHP ou Java, e também dentro do próprio banco de dados. A chamada externa dependerá de cada linguagem, já a chamada interna o ORA-

CLE utiliza o comando EXECUTE seguido do nome do procedimento. Antes de exemplificar a chamada ao procedimento, é importante que fique esclarecido que o suporte ao chamado de procedimento dependerá da interface de comunicação que o usuário está utilizando. Por exemplo, caso o usuário esteja utilizando a versão Oracle 10 g Express Edition, que possui uma interface nativa de comunicação através do uso do navegador, não conseguirá executar corretamente o procedimento utilizando a interface Web nativa, mas sim através do modo console.

Cada interface de comunicação com banco possui suas particularidades e funcionalidades, que auxiliam o usuário como ferramentas importantes de implementação de seus recursos. Existe uma série de ferramentas disponíveis para uso com suas particularidades. Como forma de demonstrar de forma mais completa o funcionamento dos procedimentos e funções do Oracle, vamos trabalhar com uma IDE externa ao Oracle chamada SQL Developer. Assim vamos abordar de forma paralela algumas características e funcionamento dessa ferramenta enquanto exemplificamos os procedimentos e funções.

Quando aberta a interface da ferramenta pela primeira vez, é necessário que seja criada uma conexão com o banco de dados. Dessa forma, realize a conexão informando o usuário e senha de acesso ao banco de dados. Depois de conectado, será aberta uma aba à direita da interface do SQL Developer, onde será possível implementar o código SQL. Caso você ainda não tenha criado o procedure “aumenta_sal” do Exemplo

2, pode executá-lo na interface de edição aberta pelo SQL Developer.

Após criado o procedimento, é possível identificá-lo no grupo de procedimentos que se encontra à esquerda da interface do SQL Developer. O Oracle possui uma característica interessante na criação de um procedimento. Este cria um procedimento, mesmo que o conteúdo existente no corpo do procedimento possua erro em sua sintaxe. Quando isso ocorrer, o procedimento será identificado no SQL Developer por um ícone em forma de um “X” vermelho em seu nome. Caso tenha sido criado sem nenhum erro de sintaxe, o mesmo receberá um ícone verde junto a seu nome. A Figura 1 apresenta um exemplo da criação do procedimento “aumenta_sal”, sem nenhum erro sintático no corpo do procedimento.

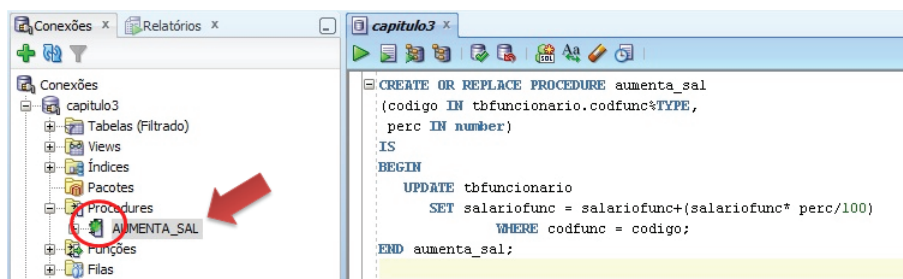


Figura 1

Como forma de provocar um erro, o procedimento “aumenta_sal”, será alterado causando um erro sintático. Para isso, remova o ponto e vírgula do final do SQL que executa a alteração de salário. Ao executar novamente o código que cria a procedure “aumenta_sal”, o SQL Developer apresenta a identificação de erro sintático, mas cria-o em sua estrutura. A

faltando). A figura também identifica o ícone que realiza novamente a compilação do procedimento.

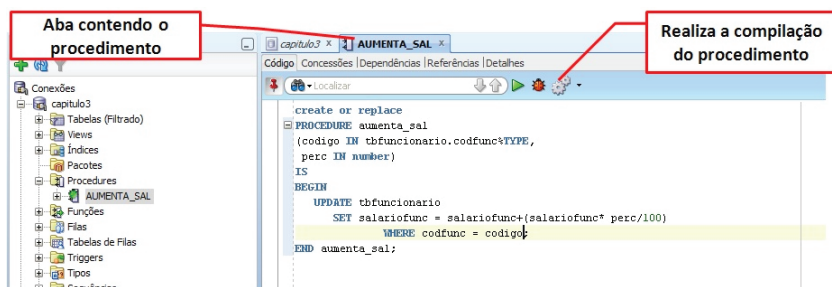


Figura 3

Essa mesma funcionalidade do SQL Developer pode ser aplicada em funções, triggers, pacotes views entre outras operações do banco de dados. Contudo, observe sempre a aba em que está trabalhando para não cometer erros comuns, como a edição de uma procedure de forma equivocada. Depois de compilar, a antiga informação é perdida completamente.

Após uma breve explicação sobre a ferramenta SQL Developer, será realizada a continuidade à exemplificação da criação e execução dos procedimentos e funções.

Como já abordado, dentro do Oracle, um procedimento é executado através da chamada de seu nome precedido da palavra reservada EXECUTE ou EXEC. Quando um procedimento for chamado de dentro de outro procedimento, ou em uma função, não usamos a palavra reservada, apenas o nome do procedimento. O Exemplo 3 apresenta uma sequência de comandos que tem por objetivo apresentar os dados antes e após a chamada do procedimento "aumenta_sal".

Exemplo 3: - "Exemplo de chamada da procedure aumenta_sal"

```

1  Select * from tbfuncionario;
2  EXECUTE aumenta_sal(1,17.5);
3  Select * from tbfuncionario;

```

O procedimento "aument_sal (1,17.5)" chamado na linha 2 do Exemplo 3 executa 17,5% de aumento no salário do funcionário de código 1. A Figura 4 ilustra o resultado da execução da linha 1 e linha 3 do Exemplo 3. Observe que o resultado da execução da linha 3 apresenta a alteração no salário do funcionário de código 1.

Consulta antes da chamada do procedimento "aumenta_sal"

FUNCIONARIO		
CODFUNC	NOMEFUNC	SALARIOFUNC
1	Maria	2000,00
2	Débora	1500,00
3	Rosana	3500,00

Consulta após a chamada do procedimento "aumenta_sal"

FUNCIONARIO		
CODFUNC	NOMEFUNC	SALARIOFUNC
1	Maria	2350,00
2	Débora	1500,00
3	Rosana	3500,00

Figura 4

Caso seja necessário remover um procedimento, basta executar o comando **DROP PROCEDURE** seguido do nome do procedimento. O Exemplo 4 apresenta o SQL necessário para remover o procedimento "aumenta_sal" criado anteriormente.

Exemplo 4: - "Exemplo da remoção da procedure aumenta_sal"

```
DROP PROCEDURE aumenta_sal;
```

O Exemplo 5 apresenta a criação de um procedimento que recebe dois parâmetros, sendo o primeiro parâmetro de entrada (“codigo”) que representa um código de funcionário, e o segundo de entrada e saída (“salario”) que como parâmetro de entrada possuirá um valor a ser acrescido no salário do funcionário passado como parâmetro, e como parâmetro de saída possuirá o novo salário do funcionário. Dessa forma, o objetivo do procedimento chamado “aumenta_salario_valor”, será de buscar antigo salário de um funcionário passado como parâmetro e em seguida alterá-lo acrescentando o valor informado no parâmetro “salario”.

Exemplo 5: - “Exemplo de chamada da procedure aumenta_sal”

```
1 CREATE OR REPLACE PROCEDURE aumenta_salario_valor
2 ( codigo IN tbfuncionario.codfunc%TYPE,
3  salario IN OUT number)
4 IS
5  antigosal tbfuncionario.salariofunc%type;
6 BEGIN
7  select nvl(salariofunc,0) INTO antigosal from tbfuncionario
8  where codfunc=codigo;
9  salario:=salario+antigosal;
10 UPDATE tbfuncionario SET salariofunc = salario WHERE codfunc = codigo;
11 salario:=antigosal;
12 END aumenta_salario_valor;
```

O Exemplo 5 também apresenta uma série de novas informações possíveis no bloco de comandos antes não trabalhadas. A primeira está relacionada à criação de variáveis dentro

o procedimento. Observe que na cláusula IS está sendo criada uma variável chamada “antigosal” definida como do mesmo tipo do atributo “salariofunc” da tabela de funcionário. Também está sendo aplicada a função nvl (<atributo>, <valor_se_nulo>) que é responsável por testar o conteúdo existente no parâmetro atributo. Caso o valor contido nesse atributo retornar uma informação nula, a função retornará o valor definido no parâmetro valor_se_nulo. Nesse caso, se o salário do funcionário pesquisado retornar vazio, a função retornará 0 (zero).

Outro detalhe muito importante no Exemplo 5 faz referência ao comando de busca do salário do funcionário existente na linha 7 e 8 do procedimento. Com já estudado anteriormente, o resultado de uma consulta sempre será uma lista contendo as informações pesquisadas. Quando utilizamos um SELECT dentro de um bloco de comandos de um procedimento, o resultado deve ser armazenado em variáveis para que possamos utilizar o conteúdo retornado pelo SELECT. Para isso, após informar os campos que deseja ter como resultado no comando SQL, utiliza-se a palavra reservada INTO seguida da sequência de variáveis que conterão o resultado da consulta. Na linha 7 isso está ocorrendo com o campo “salarioFunc” que está sendo atribuído à variável “antigosal”.

Quando o resultado da pesquisa não retornar apenas um registro e sim uma lista de valores, apenas a implementação da cláusula INTO não atenderá ao requisito. Para solucionar esse problema, somente com o uso de cursores, os quais serão apresentados em outro capítulo deste material. Sendo assim, sempre que necessário buscar uma determinada informação em uma tabela para ser utilizada em uma determinada rotina,

é necessário que existam variáveis para receber o conjunto de dados de retorno. Observe também, que por ser do tipo entrada e saída, a variável “salario” passada como parâmetro (linha 3), está sendo atribuída no bloco de comandos (linhas 9 e 11), e seu valor poderá ser utilizado pelo processo que chamou o procedimento.

Se executado o procedimento “aumenta_salario_valor” através do comando EXECUTE, não é possível visualizar o retorno da variável “salario”, pois não existe um conjunto de comandos que realize essa ação. Contudo, a ferramenta SQL Developer possui um recurso muito interessante, que possibilita ao usuário a visualização de parâmetros de saída. Para isso, clique com o botão direito do mouse no procedimento e selecione a opção “Executar” conforme Figura 5.

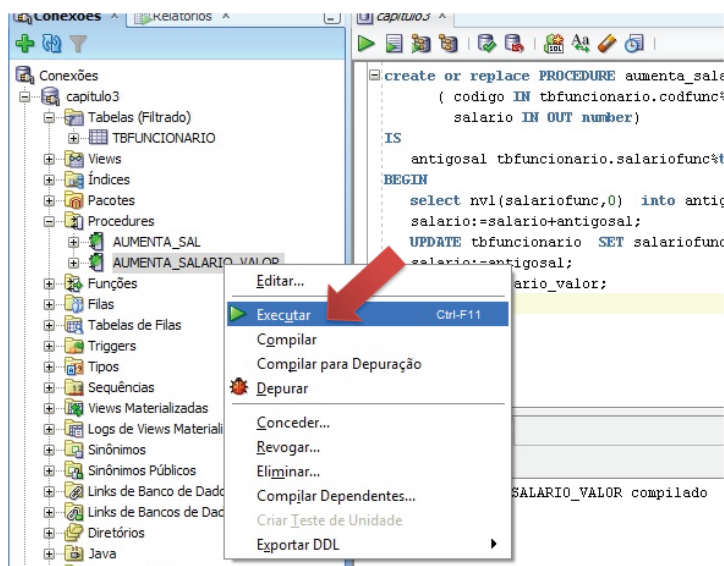
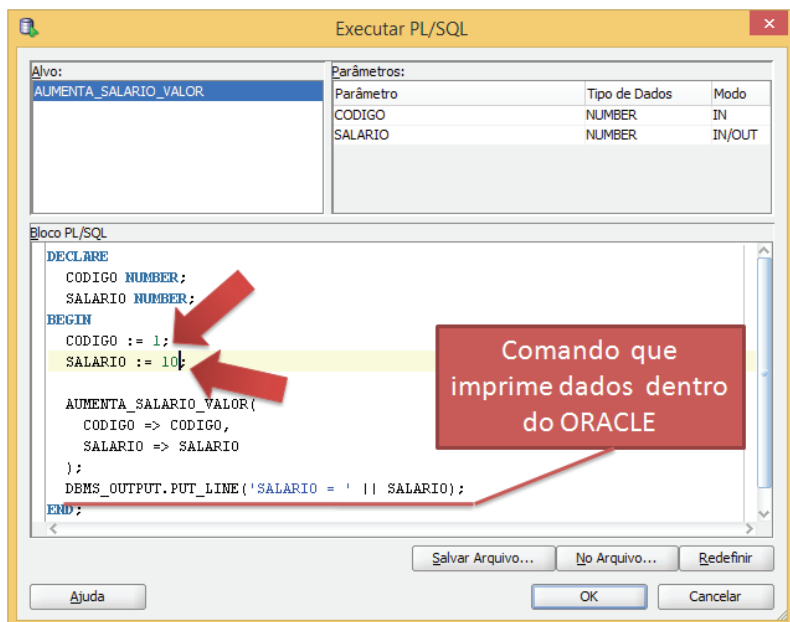


Figura 5

Quando selecionada a opção de Executar, será aberta uma interface que possui algumas linhas de comando em PL/SQL Oracle predeterminadas que chamarão o procedimento, e em caso da existência de um retorno do mesmo, serão responsáveis pela impressão do conteúdo retornado. A partir dessa interface é possível que seja editada quaisquer informações do script para execução do procedimento criado pela ferramenta. A Figura 6 apresenta a interface gerada, onde foi definido antes da chamada do procedimento, o valor inicial das variáveis código e salário. O código do funcionário foi definido como 1 e o valor a ser acrescentado no salário do funcionário foi 10.

**Figura 6**

A Figura 6 apresenta um comando que tem por objetivo imprimir algo dentro da interface Oracle. O comando é `DBMS_OUTPUT.PUT_LINE`, que pode ser utilizado mesclando a impressão de dados literais quanto conteúdo existente em variáveis. Observe que o Oracle utiliza dois papi ("||") para representar a concatenação de dados de um texto. É importante que fique claro que o comando `DBMS_OUTPUT.PUT_LINE` só funciona dentro de IDE Oracle e não em blocos de comandos dentro do PHP ou Java por exemplo. Estamos utilizando ele para exemplificar a funcionalidade dos procedimentos e funções que estamos desenvolvendo.

A Figura 7 apresenta o resultado após execução do procedimento utilizando o recurso de executar ao clicar com o botão direito do mouse sobre o procedimento. Observe que o procedimento retornou 2350, que representa o antigo salário do funcionário de código 1. Caso o usuário realize uma nova consulta na tabela após a execução do procedimento, perceberá que o valor do salário do funcionário de código 1 foi alterado para 2360, ou seja, acrescido 10 sobre seu antigo salário.

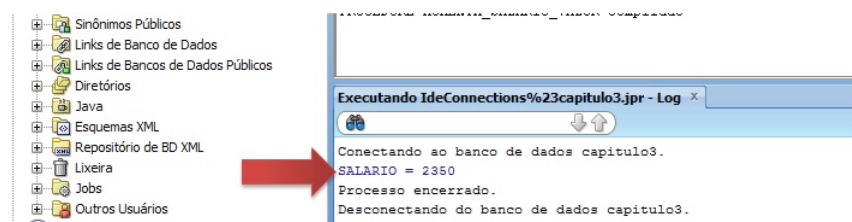


Figura 7

O Exemplo 6 apresenta a implementação de uma função chamada “dobro”, que recebe como parâmetro um número e retorna como resposta o dobro desse número.

Exemplo 6: - “Exemplo de criação de uma função”

1 CREATE OR REPLACE FUNCTION dobro

2 (NUMERO IN NUMBER)

3 RETURN NUMBER

4 IS

5 N NUMBER;

6 BEGIN

7 N:=NUMERO*2;

8 RETURN (N);

9 END dobro;

Toda função necessariamente necessita possuir um tipo de dado como retorno, que no caso do Exemplo 6 é do tipo NUMBER (linha 3). O retorno se dá pelo comando RETURN seguindo do valor de retorno ou a variável que contém o retorno. No exemplo estamos retornando o conteúdo da variável “N” (linha 8).

Uma função pode ser chamada dentro de outra função, em um procedimento ou até mesmo ser chamada utilizando um comando DDL. O Exemplo 7 apresenta uma sequência de comandos que utilizam a função criada anteriormente (com exceção do primeiro comando).

Exemplo 7: - “Exemplo de chamada de função”

```

1 select * from tbfuncionario
2 select nomefunc,salariofunc,dobro(salariofunc)as dobro_s from tbfuncionario
3 select * from tbfuncionario where salariofunc>dobro(1000)
4 update tbfuncionario set salariofunc=dobro(salariofunc) where codfunc=2

```

O primeiro comando apresenta como resposta todo conteúdo existente na tabela de funcionário. Isso foi necessário para ilustrar a execução da função “dobro” que é chamada nos comandos 2 a 4 do Exemplo 7. A Figura 8 apresenta a resposta dos três primeiros comandos do exemplo. Perceba que, no segundo comando, além de listar o nome do funcionário e seu salário, foi criada mais uma coluna contendo o dobro do salário do funcionário.

RESULTADO COMANDO 1			RESULTADO COMANDO 2		
CODFUNC	NOMEFUNC	SALARIOFUNC	NOMEFUNC	SALARIOFUNC	DOBRO_S
1	Maria	2360	Maria	2360	4720
2	Débora	1500	Débora	1500	3000
3	Rosana	3500	Rosana	3500	7000

RESULTADO COMANDO 3		
CODFUNC	NOMEFUNC	SALARIOFUNC
1	Maria	2360
3	Rosana	3500

Figura 8

A função dobro está sendo utilizada como critério de uma condição de um SELECT (linha 3 do Exemplo 7). Neste estão sendo listados os funcionários que possuem seus salários su-

periores ao dobro de 1000. Conforme é possível observar o resultado da Figura 8, somente estão sendo listados os funcionários “Maria” e “Rosana” que atendem ao critério.

Na última linha do Exemplo 7, a função dobro está sendo utilizada com o comando update. Esta altera o salário do funcionário de código 2 para o dobro de seu salário atual.

Caso executássemos novamente o comando 1 após a execução do comando 4 do Exemplo 7, o salário da funcionária “Débora” passaria a ser R\$ 3.000,00.

Já o Exemplo 8 apresenta a criação de outra função denominada “dobro2” que tem como objetivo receber um valor qualquer e retornar sempre um texto concatenando a mensagem “O dobro é” com o dobro do número informado como parâmetro.

Observe na linha 3 do Exemplo 8 que o tipo de retorno da função definida é varchar2, ou seja, a função retorna um texto. A linha 9 executa a concatenação do texto seguido do dobro do número informado como parâmetro. A sequência de dois caracteres pipe (‘|’) no Oracle indica que o conteúdo existente antes e após os caracteres pipe serão concatenados, ou seja, se tornarão um único conteúdo. A concatenação sempre resulta em um dado do tipo texto. No exemplo, foi informado após os dois caracteres pipe a variável “N” do tipo number. O Oracle irá converter o conteúdo da variável “N” em um texto.

Exemplo 8: - “Exemplo de criação de uma função dobro2”

```
1 CREATE OR REPLACE FUNCTION dobro2
2 (NUMERO IN NUMBER)
3 RETURN VARCHAR2
4 IS
5 N NUMBER;
6 NOME VARCHAR2(20);
7 BEGIN
8 N:=NUMERO*2;
9 NOME:=' O dobro é ' || N;
10 RETURN (NOME);
11 END dobro2;
```

Os Exemplos 13 e 14 apresentam a chamada da função definida.

Como forma de ilustrar o funcionamento da nova função, a Figura 9 apresenta dois exemplos da chamada da função “dobro2”. No primeiro exemplo, estamos utilizando um outro recurso bem interessante do Oracle que é a tabela Dual. Essa tabela é muito utilizada para realizar operações de seleção de dados (SELECT) onde não é necessário realizar extração de dados em tabelas. No Exemplo 1 da Figura 9, estamos retornando como resposta o dobro do número 4. Observe que para isso apenas chamamos a função “dobro2” criada com a tabela Dual como referência.

Ex (1):
`select dobro2(4) as resultado from dual`

RESPOSTA EXEMPLO (1)
RESULTADO
O dobro é 8

Ex (2):
`select f.*,dobro2(f.salariofunc) as resultado from tbfuncionario f`

RESPOSTA EXEMPLO (2)			
CODFUNC	NOMEFUNC	SALARIOFUNC	RESULTADO
1	Maria	2360	O dobro é 4720
2	Débora	1500	O dobro é 3000
3	Rosana	3500	O dobro é 7000

Figura 9

Já o Exemplo 2 da Figura 9 utiliza como base o conteúdo existente na tabela de funcionário retornando em tela, o código do funcionário, seu nome, salário e o dobro de seu salário. Observe que o conteúdo retornado pela função “dobro2” será um texto conforme especificado na função “dobro2”.

Por fim, são apresentados os Exemplos 9 e 10 que possuem como objetivo criar uma função e um procedimento, onde a função possui como ação retornar o maior código de funcionário cadastrado, e o procedimento inserir um novo registro na tabela de funcionário.

Exemplo 9: - “Função que retorna maior código de funcionário”

```
1 create or replace FUNCTION maior
2 RETURN integer
3 IS
4 m integer;
5 BEGIN
6 select NVL(max(codfunc),0) into m from tbfuncionario;
7 RETURN (m);
8 END maior;
```

Exemplo 10: - “Procedimento que insere novo registro”

```
1 create or replace PROCEDURE GRAVAFUNC
2 (nome IN VARCHAR2,
3 salario in number)
4 IS
5 proximo integer;
6 BEGIN
7 proximo:=maior()+1;
8 insert into tbfuncionario (codfunc,nomefunc,salariofunc)
9 values(proximo,nome,salario);
9 END GRAVAFUNC;
```

Observe no Exemplo 9 que a função não possui parâmetros, somente retorna o conteúdo da variável “m” (linha 7). O grande segredo da função está na linha 6 que executa um MAX no campo de código de funcionário retornando para a variável “m” o maior código encontrado. Caso não encontre, a função NVL retornará 0 (zero) como maior valor.

No procedimento “gravafunc”, são identificados dois parâmetros de entrada, o nome e o salário (linhas 2 e 3 do Exemplo 10). Na linha 7, a variável próximo recebe o resultado da função criada no Exemplo 9 e acrescenta 1 à resposta. O ob-

jetivo é que a variável próximo contenha o próximo código de funcionário a ser inserido. Em seguida (linha 8), é executado o comando de inserção de dados onde pode ser identificado que os valores informados para serem armazenados são os valores passados como parâmetro e o valor contido na variável próximo.

Assim, para inserir um novo funcionário, basta chamar o procedimento informando como parâmetro o nome e salário do mesmo. O Exemplo 11 apresenta duas chamadas do procedimento “gravafunc”, inserindo a funcionária “Carla” com salário de R\$ 1.250,00 e a funcionária “Adriana” cujo salário é o dobro de R\$ 900,00.

**Exemplo 11: - “Exemplo de chamada do procedimento GRA-
VAFUNC”**

```
1 execute gravafunc('Carla',1250);  
-----  
2 execute gravafunc('Adriana',dobro(900));
```

A Figura 10 apresenta como resposta a uma pesquisa listando todos os funcionários cadastrados em ordem de código de funcionário. Observe que conforme foram executados os procedimentos “gravafunc” no Exemplo 11, os registros foram inseridos com o código do funcionário seguindo uma sequência de números. Observe que, ao cadastrar a “Adriana”, a função dobro foi disparada retornando 1800 ao invés de 900.

SELECT * FROM TBFUNCIONARIO ORDER BY CODFUNC		
CODFUNC	NOMEFUNC	SALARIOFUNC
1	Maria	2360
2	Débora	1500
3	Rosana	3500
4	Carla	1250
5	Adriana	1800

Figura 10

É importante que fique ressaltado que em geral os SGBD gerenciam os procedimentos e funções criados para cada banco de dados existente, ou seja, as funções e procedimentos criados para um banco de dados são específicas do banco de dados em que foram criadas não sendo possível utilizá-las em outros bancos, mas isso não é uma regra e sim o que ocorre na maioria dos SGBD.

Recapitulando

Este capítulo mostrou que alguns SGBD possuem como recurso funções e procedimentos possibilitando que várias regras de negócio que antes eram implementadas na linguagem de programação, possam ser implementadas pelo SGBD. Entendemos que os parâmetros definidos em um procedimento e ou em uma função podem ser de entrada (IN), saída (OUT) ou de entrada e saída (IN OUT). Aprendemos a utilizar a ferramenta SQL developer que auxiliou no entendimento e na execução dos procedimentos e funções.

Neste capítulo, também conhecemos o funcionamento da função NVL, e principalmente entendemos como colocamos o resultado de uma consulta em variáveis para que seja possível manipulá-las. Utilizamos a função DBMS_OUTPUT_LINE para imprimir algo dentro do Oracle. Todos os conceitos abordados foram implementados utilizando exemplos práticos.

Referências

ELMASRI, R. & NAVATHE, S. B. Sistemas de Banco de Dados. Rio de Janeiro: LTC, 2002. 4ª ed

DATE, C. J. Introdução a Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 2003. 9ª ed. .

KORTH, Henry F. e SILBERSCHATZ, Abraham. Sistema de Bancos de Dados. São Paulo: Makron Books, 1999. 3ª edição revisada.

Atividades

- 1) Tendo como base o conteúdo trabalhado, marque a alternativa que representa a correta definição das assertivas que seguem:
 - I – Um procedimento pode conter somente parâmetros de entrada.
 - II – Uma função pode conter somente parâmetros de entrada.

III – Uma função sempre retorna algo.

IV – Um procedimento pode conter comandos de seleção de dados.

- a) ☐ Somente as assertivas I e II estão corretas.
 - b) ☐ Somente as assertivas II e III estão corretas.
 - c) ☐ Somente as assertivas III e IV estão corretas.
 - d) ☐ Somente as assertivas II, III e IV estão corretas.
 - e) ☐ Somente as assertivas I, II e IV estão corretas.
- 2) Marque a(s) alternativa(s) que atendem ao critério solicitado. Uma função pode ser chamada por:
- a) ☐ Outra Função.
 - b) ☐ Um comando de SELECT.
 - c) ☐ Um comando Create Table.
 - d) ☐ Um procedimento.
- 3) Tendo como base que o comando que segue tenha sido implementado no corpo de uma função ou procedimento, está **incorreto** afirmar que:

```
select NVL(idade,18) INTO xidade from tbaluno where codigo=2
```

- a) ☐ Por utilizar a função NVL, caso não exista uma idade cadastrada no aluno de código 2 o SELECT retornará 18.
- b) ☐ O resultado da consulta será alocado na variável xidade.

- c) () A variável xidade deve ter sido definida como parâmetro OUT ou IN OUT ou deve ter sido criada como variável local do procedimento ou função que possui a linha de comando.
 - d) () A variável xidade pode ter sido passada como parâmetro do tipo IN.
- 4) Qual comando que não pode ser executado caso tenhamos uma função denominada teste contendo como parâmetro de entrada um número inteiro que retorne um outro número inteiro qualquer.
- a) () select nome, idade, teste (idade) from pessoa.
 - b) () select nome, idade, teste (30) from pessoa.
 - c) () select nome, idade, teste (idade+20) from pessoa.
 - d) () select nome, idade, teste (teste(idade)) from pessoa.
 - e) () select nome, idade from pessoa where teste (nome)>idade.
- 5) O procedimento a seguir relacionado executa a seguinte rotina:

```
REATE OR REPLACE PROCEDURE YYY
( Q in tbveiculo.codvei%TYPE,
  A in tbveiculo.codvei%TYPE,
  XX in out number
)
IS
BEGIN
  select nvl(valorbase,100) into XX from tbveiculo
  where codvei=Q;
  XX:=XX+A;
END YYY;
```

- a) () Recebe dois parâmetros onde busca o valor base do veículo de código Q adicionando mais 100 na variável XX. Em seguida, o valor de XX é alterado acrescentando o valor informado no parâmetro A.
- b) () Recebe três parâmetros onde busca o valor base do veículo de código Q adicionando mais 100 na variável XX. Em seguida, o valor de XX é alterado acrescentando o valor informado no parâmetro A.
- c) () Recebe dois parâmetros onde busca o valor base do veículo de código Q e adiciona no parâmetro XX. Em seguida, o valor de XX é alterado acrescentando o valor informado no parâmetro A.
- d) () Recebe três parâmetros onde busca o valor base do veículo de código Q e adiciona no parâmetro XX. Em seguida o valor de XX é alterado acrescentando o valor informado no parâmetro A.

Gabarito:

- 1) c
- 2) a, b, d
- 3) d
- 4) e
- 5) d

Christiano Cadoná¹

Capítulo **4**

Stored Procedure e Function

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campus ULBRA Canoas.

Introdução

Este capítulo abordará o uso de PL/SQL como meio de implementar comandos já conhecidos de programação afim de possibilitar que as regras de negócio possam ser implantadas dentro do banco de dados ao invés da aplicação. Para isso será estudado as rotinas de seleção e laço de repetição utilizando exemplos práticos de procedimentos e funções do Oracle.

Contextualização do PL/SQL

O PL/SQL (Procedural Language extensions to SQL) é uma linguagem de programação incorporada a partir da versão 6.0 do Oracle em 1988, tendo como objetivo possibilitar o desenvolvimento de aplicações que são compiladas e executadas dentro do servidor do banco de dados Oracle. Esse conceito não é utilizado apenas pelo Oracle e sim pela maioria dos SGBDs comercializados no mercado. Cada um desses SGBD possuem sua sintaxe de programação e tratamento aos comandos programados. Como até aqui estamos tratando e exemplificando o Banco Oracle, iremos trabalhar com a sintaxe deste, mas seu conceito se aplica a qualquer SGBD existe.

Normalmente, utilizamos essa técnica quando necessitamos de alto desempenho na execução de tarefas do banco de dados, pois estas podem ser otimizadas de acordo com o sistema gerenciador de banco de dados que estamos utilizando, além é claro dessa aplicação rodar no próprio banco de dados, o que otimiza o tráfego de dados, pois não é necessário

trafegar dados na rede entre o bando de dados e uma aplicação para obter o retorno. O uso de PL/SQL possibilita que não seja necessária a alteração de código na programação por troca de versão de sistema, tornando o conteúdo programado mais duradouro. É comum encontrar programações PL/SQL programadas a décadas sem que seja necessário alteração em seu conteúdo, mesmo que tenha sido trocada a versão do próprio banco de dados.

Afinal, qual é a diferença entre SQL e PL/SQL?

Geralmente, as pessoas confundem SQL com PL/SQL. O SQL tem por objetivo oferecer instruções para a recuperação e manipulação de dados em tabelas, controle de transações, definição de objetos e controle de acesso. O SQL possui como limitação ser uma linguagem declarativa. Isso significa que não é possível criar um programa inteiro em SQL, pois a linguagem não possui comandos importantes para uma linguagem de programação como comandos para tomada de decisão ou comandos para execução de laços de repetição.

Já o PL/SQL é uma extensão da linguagem SQL que contém também controle de fluxo, tratamento de exceções, entre outras características que o torna uma linguagem de programação. Utilizando o PL/SQL é possível escrever programas inteiros, desde os mais simples até os mais sofisticados. A linguagem foi criada exatamente com o propósito de oferecer uma

solução de programação que atenda aplicações de missão crítica executadas no SGBD Oracle.

Principais Estruturas PL/SQL

Semelhante a qualquer linguagem de programação, o PL/SQL possui um conjunto de comandos que implementa a estrutura de seleção e de repetição. Como as demais linguagens de programação, o PL/SQL possui uma sintaxe de implementação específica. Sendo assim, nesse tópico, serão abordado os principais controles implementados no PL/SQL.

Atribuição, Operadores Aritméticos, Lógicos e Condicionais e Algumas Funções Importantes

Como toda linguagem de programação, a atribuição e o uso de operadores aritméticos é indispensável para trabalhar com o conteúdo armazenado em variáveis. No PL/SQL para atribuir um valor ou o conteúdo de uma variável para outra variável utilizamos := (dois pontos igual). Os principais operadores aritméticos suportados pelo Oracle estão sendo apresentados na Tabela 1 que segue:

Tabela 1 Operadores Aritméticos

Operador	Símbolo	Exemplo
Soma	+	A:=7+3 (resulta em 10)
Subtração	-	A:=7-3 (resulta em 4)
Multiplicação	*	A:=7*3 (resulta em 21)
Divisão	/	A:=7/2 (resulta em 3 se for inteiro ou 3,5 se for do tipo real)
Exponenciação	**	A:=7**3 (resulta em 343)

Algumas linguagens também possuem operadores que retornam o resto de uma divisão. Em PL/SQL não existe um operador para isso, e sim uma função que executa essa ação chamada MOD. Existem três funções muito utilizadas em PL/SQL relacionadas a números TRUNC, ROUND e MOD.

A função TRUC é utilizada quando ocorre a necessidade de retornar um número com uma quantidade de casas decimais preservadas. Sua sintaxe é:

TRUNC(NUMERO, QUANTIDADE_DE_CASAS_DECIMAIS)

A Tabela 2 apresenta 4 exemplos e seus resultados após o uso da função TRUNC. Observe se informado no parâmetro “quantidade de casas decimais” um valor negativo a função retorna o valor mais significativo do número informado. No terceiro e quarto exemplos, foram informados valores negativos retornado o valor mais significativo do número 129,472. E Exemplo 3 retorna 100, pois a função TRUNC zera os 2 caracteres menos significativos do número, e no Exemplo 4 a função TRUNC retorna 120, pois ele zera o último numero menos representativo.

Tabela 2 Exemplo de função TRUNC

Expressão	Resultado
Ex.: 1) A:=TRUNC(129.472,1)	129.4
Ex.: 2) A:=TRUNC(129.472,0) ou A:=TRUNC(129.472)	129
Ex.: 4) A:=TRUNC(129.472,-2)	100
Ex.: 5) A:=TRUNC(129.472,-1)	120

Observe que, se não informado o parâmetro de quantidade de casas decimais, o TRUNC entende como zero, como é possível perceber no Exemplo 2 da tabela 2.

Já função ROUND tem por objetivo arredondar um valor a partir de uma determinada posição após a vírgula. O parâmetro de posição não é obrigatório. Sua sintaxe é:

ROUND(NUMERO, POSIÇÃO)

Da mesma forma que a função TRUNC caso informado como parâmetro um valor negativo, a função ROUND retorna como resposta um número inteiro arredondado antes da vírgula. A tabela 3 apresenta Exemplos do uso da função ROUND.

Tabela 3 Exemplo de função ROUND

Expressão	Resultado
Ex.: 1) A:= ROUND(1259.572)	1260
Ex.: 2) A:= ROUND(1259.572,1)	1259,6
Ex.: 3) A:= ROUND(1259.572,-1)	1260

A função MOD é utilizada quando existe a necessidade de retornar o resto de uma divisão de dois números. Sua sintaxe é:

MOD(DIVIDENDO, DIVISOR)

A tabela 4 apresenta três exemplos da execução da função MOD. Diferente da maioria das linguagens de programação, a função MOD no PL/SQL pode receber valores reais como parâmetro. Seu resultado sempre será o resto dessa divisão.

Tabela 4 Exemplo da Função MOD

Expressão	Resultado
Ex.: 1) A:=MOD(17,2)	1
Ex.: 2) A:=MOD(17.5,2);	1,5
Ex.: 3) A:=MOD(10,2.5);	0

Na sintaxe dos comandos das estruturas de seleção e repetição, existe uso de condições. Nessas condições, é comum o uso de operadores Lógicos e Relacionais. A tabela 5 apresenta uma relação dos operadores suportados pelo PL/SQL.

Tabela 5 Operadores Lógicos e Relacionais

Operador	Símbolo	Exemplo
Igual	=	If A=B then
	<>	If A<>B then
Diferente	!=	If A!=B then
	~=	If A~=B then
Menor do que	<	If A<B then
Maior do que	>	If A>B then
Menor ou igual a	<=	If A<=B then
Maior ou igual a	>=	If A>=B then
Verifica se um valor está contido em um intervalo especificado de valores	Between	If A between 10 and 50 then

Verifica se um valor está dentro de uma lista especificada de valores	IN	If A in (2,5,3,4,3) then
Testa se uma variável é nula	IS NULL	If A IS NULL then
Testa se duas condições são verdadeiras	AND	If A <= B AND A < 17 then
Testa se pelo menos uma das condições é verdadeira	OR	If A <= B OR A < 17 then
Inverte o resultado de uma condição	NOT	If NOT A < 5 then

Estrutura de Seleção SE – IF

Utilizada para executar um determinado bloco de comandos se uma condição ou um bloco de condições satisfaçam uma ou um conjunto de condições. Em PL/SQL é possível implementar o comando IF de três formas diferentes. Segue a sintaxe do comando IF.

SINTAXE 1: IF condições THEN comandos; END IF;	SINTAXE 3: IF condições THEN comandos; ELSIF condições THEN comandos; ELSE comandos; END IF;
SINTAXE 2: IF condições THEN comandos; ELSE comandos; END IF;	

Os Exemplos 1 e 2 implementam um procedimento e uma função que exemplificam o uso do comando IF. No procedimento, são passados dois parâmetros de entrada e este imprime algumas informações de acordo com testes implementados no comando IF. Já a função chamada “exemplo2_if” recebe um número e retorna se esse número é positivo, negativo ou nulo.

Exemplo 1: - “Exemplo do uso da estrutura IF”

```
1 create or replace
2 procedure exemplo1_if
3 ( n1 in number, n2 in number)
4 is
5 begin
6 if n1=n2 then
7 dbms_output.put_line('n1 é igual a n2');
8 end if;
9 if (n1 > (n2*2)) then
10 dbms_output.put_line('N1 é maior do que o dobro de N2');
11 Else
12 dbms_output.put_line('n1 não é maior do que o dobro de n2 ');
13 end if;
14 end exemplo1_if;
```

Exemplo 2: - “Exemplo do uso da estrutura IF”

```
1 create or replace function exemplo2_if
2 (num in integer) return varchar
3 is
4   resp varchar(10);
5 BEGIN
6   if num >= 0 then
7     resp := 'Positivo';
8   elsif num < 0 then
9     resp := 'Negativo';
10  elsif num = 0 then
11    resp := 'Nulo';
12  end if;
13  return resp;
14 END exemplo2_if;
```

Estrutura de Seleção CASE

Da mesma forma que a estrutura de seleção IF, a estrutura de seleção CASE é utilizada para executar um bloco de comandos se uma condição for válida. Diferente de algumas linguagens, se encontrada uma condição válida em uma das alternativas do comando CASE, o seu conteúdo é executado, e o programa dá um salto para a próxima linha de comando após o término do comando CASE, e não continua testando as demais alternativas, como ocorre na linguagem C por exemplo. Para que isso ocorra em C, é necessário obrigatoriamente utilizar o comando BREAK.

Sintaxe:**CASE****WHEN** *condições _1* **THEN** *comandos;***WHEN** *condições _2* **THEN** *comandos;;*

...

WHEN *condições _n* **THEN** *comandos_n;***ELSE** *comandos;***END CASE;**

O Exemplo 3 apresenta um procedimento que recebe dois valores como parâmetro de entrada e de acordo com as condições existentes no comando CASE imprime algumas informações.

Exemplo 3: - "Exemplo do uso da estrutura CASE"

```
1 create or replace PROCEDURE exemplo_case
2   ( n1 IN number, n2 IN number)
3 IS
4 BEGIN
5   CASE
6     WHEN n1 > n2 THEN
7       DBMS_OUTPUT.PUT_LINE('N1 é MAIOR a N2');
8     WHEN n1 < n2 THEN
9       DBMS_OUTPUT.PUT_LINE('N1 é MENOR a N2');
10    ELSE
11      DBMS_OUTPUT.PUT_LINE('N1 é IGUAL a N2');
12    END CASE;
13 END exemplo_case;
```

Estrutura de Repetição WHILE

A estrutura de repetição While é responsável por executar as rotinas contidas no bloco de comandos do While, enquanto uma determinada condição for verdadeira.

Sintaxe:**WHILE** *condições* **LOOP***Comandos;***END LOOP;**

○ Exemplo 4 apresenta uma função que, passado um número inteiro qualquer, retorna o somatório do número informado utilizando para isso o comando While.

Exemplo 4: - “Exemplo do uso da estrutura de repetição WHILE”

```
1  create or replace FUNCTION somatorio
2  ( num IN integer) return integer
3  IS
4  n integer;
5  r integer;
6  BEGIN
7  n:=1;
8  r:=0;
09  WHILE n<=num LOOP
10  r:=r+n;
11  n:=n+1;
12  END LOOP;
13  return(r);
14  END somatorio;
```


Estrutura de Repetição LOOP

A estrutura de repetição Loop também executa um bloco de comandos enquanto não encontrar o comando EXIT. Diferentemente do comando WHILE que sempre testa antes de executar o bloco de repetição, o comando LOOP somente será abortado quando em seu bloco encontrar o comando EXIT. Normalmente, ocorre um teste utilizando o comando condicional ou quando é executado o comando EXIT WHEN seguido de uma determinada condição. Sua Sintaxe é:

Sintaxe 1:

LOOP

-- Comandos que devem ser executados

IF condições THEN

EXIT;

END IF;

-- Comandos que devem ser executados

END LOOP;

Sintaxe 2:

LOOP

-- Comandos que devem ser executados

EXIT WHEN condições;

-- Comandos que devem ser executados

END LOOP;

○ Exemplo 5 apresenta a criação de um procedimento que recebe um número qualquer e lista em tela duas listas de valores iniciando em 1 e indo até o valor informado como parâmetro.

Exemplo 5: - “Exemplo do uso da estrutura de repetição LOOP”

```
1 create or replace PROCEDURE exemplo_loop
2   ( fim IN integer)
3 IS
4   n integer;
5 BEGIN
6   n:=1;
7   LOOP
8     DBMS_OUTPUT.PUT_LINE('Número é '|| n);
9     IF n=fim THEN
10      EXIT;
11    END IF;
12    n:=n+1;
13  END LOOP;
14  n:=1;
15  LOOP
16    DBMS_OUTPUT.PUT_LINE('Número é '|| n);
17    EXIT WHEN n=fim ;
18    n:=n+1;
19  END LOOP;
20 END exemplo_loop;
```

Estrutura de Repetição FOR

A estrutura de repetição FOR executa o bloco de comandos no intervalo de valores estabelecidos entre o valor inicial e valor final. Esses valores serão armazenados na variável de controle e incrementados a cada ciclo do laço de repetição. Caso seja necessário executar o controle de forma invertida, ou seja, no maior para o menor valor, é necessário utilizar a cláusula REVERSE.

Sintaxe:

```
FOR variável_de_controle IN [REVERSE] valor_inicial..valor_final  
LOOP  
    Comandos;  
END LOOP
```

O Exemplo 6 apresenta a definição de uma função que recebe como parâmetro dois números e retorna a soma de todos os números do intervalo do primeiro para o segundo valor.

Exemplo 6: - "Exemplo do uso da estrutura de repetição FOR"

```
1 create or replace function somaintervalo  
2 ( v1 in number,  
3 v2 in number) return number  
4 IS  
5 soma number;  
6 cont number;  
7 begin  
8 soma:=0;  
9 for cont in v1..v2 loop  
10 soma:=soma+cont;  
11 end loop;  
12 return soma;  
13 end somaintervalo;
```

Após a construção da função, se o usuário chamá-la informando como parâmetro 2 e 5 por exemplo (select somaintervalo(2,5) from dual) o Oracle retornará o valor 14 como resposta, pois a soma de 2+3+4+5 é igual a 14.

Tratamento de exceção em PL/SQL

Como qualquer linguagem de alto nível, o PL/SQL também possui tratamento de exceção. Exceção geralmente existe quando ocorre algum tipo de erro ao executar um comando. No PL/SQL existem dois tipos de exceção, sendo que o primeiro tipo (mais comum), faz referência às exceções que são disparadas automaticamente pelo Oracle quando ocorre algum tipo de erro, como, por exemplo, a duplicidade de chave primária ou uma divisão por zero. Esse tipo de exceção é chamada de exceção de sistema. O segundo tipo de exceções são exceções criadas pelo próprio programador que possibilitam o tratamento de eventuais rotinas definidas por este, chamadas exceções programadas.

Para tratar uma exceção do sistema, adicionamos o comando `EXCEPTION` no final de um bloco de comandos de uma procedure ou de uma function e tratamos o mesmo. Quando ocorre um erro em qualquer linha do programa, o mesmo para de ser executado e salta diretamente para o controle de exceção, ignorando qualquer outro comando que esteja programado para ser executado posteriormente na linha que causou o problema.

O Exemplo 7 apresenta um procedimento que tem como objetivo executar a divisão de dois números passados como parâmetro. Se em alguma circunstância ocorrer a chamada do procedimento “testeexception1” com o conteúdo passado no parâmetro V2 0 (zero), o sistema terá um erro matemático na linha 7 do procedimento. Caso isso ocorra, o Oracle identifi-

cará que existiu um erro e executará a primeira linha existente no bloco de exceção (linha 10 do procedimento).

Exemplo 7: - “Exemplo do uso de exception”

```

1  create or replace procedure testeexception1
2  (v1 in number,
3   v2 in number)
4  IS
5   resp number;
6  BEGIN
7   resp:=v1/v2;
8   DBMS_OUTPUT.PUT_LINE('A divisao é' || resp);
9  EXCEPTION
10 WHEN ZERO_DIVIDE THEN
11   DBMS_OUTPUT.PUT_LINE('ERRO ==== divisão por ZERO ====');
12   DBMS_OUTPUT.PUT_LINE('Não é possível dividir ' || v1 || ' por ' || v2);
13 WHEN OTHERS THEN
14   DBMS_OUTPUT.PUT_LINE('ERRO == Ocorreu um erro na operação ==');
15 END testeexception1;
```

No bloco de exception, é possível identificar o tipo de erro de acordo com uma tabela de erros prevista pelo Oracle. Para isso, basta testar o tipo de erro encontrado. O Exemplo 7 possui dois testes de tipos de erros (linhas 10 e 13), sendo que o primeiro teste verifica se ocorreu um erro de divisão por zero (linha 10) e o segundo (linha 13) testa a existência de qualquer tipo de erro diferente dos já especificados anteriormente. Assim, se utilizada a cláusula WHEN OTHERS, ela é automaticamente usada para tratar qualquer tipo de erro que não esteja sendo tratado dentro do bloco de exceções já definida. O Oracle possui duas funções muito utilizadas na em Exception. A primeira é SQLCODE que retorna o código do erro causa-

do na exceção, e a SQLERRM que retorna uma mensagem associada ao código de erro. Dessa forma, seria possível, por exemplo, realizar a impressão desse erro ao usuário.

O Exemplo 8 apresenta um trecho de um procedimento que apresenta apenas a cláusula EXCEPTION, onde é possível identificar o uso das funções que retornam informações sobre o erro.

Exemplo 8: - “Exemplo do uso de SQLCODE e SQLERRM”

```
:  
EXCEPTION  
WHEN OTHERS THEN  
  DBMS_OUTPUT.PUT_LINE('Erro ao executar o procedimento');  
  DBMS_OUTPUT.PUT_LINE('Código do erro : ' || SQLCODE);  
  DBMS_OUTPUT.PUT_LINE('Erro : ' || SQLERRM);  
:
```

Já uma exceção programada é definida e disparada somente por ação programada pelo desenvolvedor. Dessa forma, esta não é disparada automaticamente como as exceções do sistema. Para criar uma exceção programada, é necessário que exista uma variável do tipo EXCEPTION que será chamada dentro do bloco da programação através do comando RAISE que força uma chamada de uma exceção. Essa exceção deve ser programada no bloco EXCEPTION de um procedimento ou função.

O Exemplo 9 apresenta a criação de uma exceção programada dentro de um procedimento que tem por objetivo, receber dois valores nos parâmetros “A” e “B”, e imprimir todos os números no intervalo de “A” a “B”. A exceção programada

testa se o valor informado na variável “B” é inferior ao valor existente na variável “A” (linha 8). Caso isso ocorra, é disparada uma exceção que informa ao usuário um alerta que não conseguiu executar o procedimento (linhas 15 e 16).

Foi criada a variável “excecaoprog” do tipo EXCEPTION (linha 6) que é manipulada como exceção do procedimento. Esta é chamada na linha 9 através do comando RAISE. Caso for chamada, a exceção é tratada na cláusula EXCEPTION linha 15.

Exemplo 9: - “Exemplo do uso de exceção programada”

```

1  CREATE OR REPLACE PROCEDURE TESTEEXCEPTION2
2  (v1 in number,
3   v2 in number)
4  IS
5   num number;
6   excecaoprog exception;
7  BEGIN
8   if v2<v1 then
9    RAISE excecaoprog;
10  end if;
11  for num in v1..v2 loop
12   DBMS_OUTPUT.PUT_LINE('Número :'| num);
13  end loop;
14  EXCEPTION
15  WHEN excecaoprog THEN
16   DBMS_OUTPUT.PUT_LINE('O valor informado como segundo
    parâmetro é inferior ao primeiro');
17  WHEN OTHERS THEN
18   DBMS_OUTPUT.PUT_LINE('ERRO = Ocorreu um erro na
    operação =');
19  END TESTEEXCEPTION2;
```

Recapitulando

Este capítulo abordou conceitos relacionados a comandos do PL/SQL como alternativa para implementar rotinas dentro do banco de dados. Foram estudados os operadores aritméticos, lógicos e condicionais. Em seguida, foram apresentadas algumas funções importantes como a função MOD. Também foi abordado e exemplificado o uso do comando IF e o comando CASE como alternativa condicional do bloco de um comando. Ainda foram apresentados exemplos de procedimentos e funções utilizando a estrutura de repetição, onde foram apresentados os comandos WHILE, LOOP e FOR.

Por fim, foi estudado o tratamento de exceção, onde foi compreendido que existem dois tipos de exceção: a exceção de sistema, que é disparada automaticamente pelo Oracle; e a exceção programada, que é criada e programada pelo programador e disparada utilizando o comando RAISE.

Referências

- ELMASRI, R. & NAVATHE, S. B. Sistemas de Banco de Dados. Rio de Janeiro: LTC, 2002. 4ª ed
- DATE, C. J. Introdução a Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 2003. 9ª ed. .
- KORTH, Henry F. e SILBERSCHATZ, Abraham. Sistema de Bancos de Dados. São Paulo: Makron Books, 1999. 3ª edição revisada.

Atividades

- 1) Tendo em vista o que foi apresentado neste capítulo, está correto afirmar que:
 - a) ☐ A Linguagem PL/SQL vem com o SQL.
 - b) ☐ É possível compilar e criar um executável utilizando a IDE PL/SQL.
 - c) ☐ O único problema da linguagem PL/SQL é que ela diferencia comandos SQL dos comandos estruturados como IF e FOR.
 - d) ☐ A Linguagem PL/SQL é utilizada dentro do Oracle e possibilita a implementação de rotinas que possuem laço de repetição, por exemplo.
 - e) ☐ O PL/SQL pode ser utilizado em qualquer banco de dados comercial.
- 2) Marque V na(s) alternativa(s) verdadeiras e F na(s) alternativa(s) falsas:
 - a) ☐ MOD é um operador nativo da linguagem PL/SQL.
 - b) ☐ ** representa um operador de exponenciação entre dois números.
 - c) ☐ / pode ser utilizado somente para números reais e não inteiros.
 - d) ☐ A função ROUND retorna um número com uma quantidade de casas decimais preservadas.

- 3) Marque a(s) alternativa(s) que possibilitam utilizar na sintaxe de sua estrutura a cláusula ELSE segundo a sintaxe do PL/SQL.
- a) ☐ IF
 - b) ☐ CASE
 - c) ☐ WHILE
 - d) ☐ LOOP
- 4) Marque a alternativa correta em relação ao uso de estrutura de repetição.
- a) ☐ No comando While, sempre executamos pelo menos uma vez o conteúdo existente em seu bloco de comandos para depois sim testar a sua condição.
 - b) ☐ Diferentemente do comando While no comando LOOP, é possível testar a condição de parada em qualquer parte do bloco LOOP.
 - c) ☐ O uso de IF possibilita executar um conteúdo somente se uma determinada condição estiver correta.
 - d) ☐ Quando utilizamos “for” não é possível percorrer os elementos do maior valor para o menor valor.
- 5) Marque V na(s) alternativa(s) verdadeiras e F na(s) alternativa(s) falsas relacionadas a tratamento de exceções:
- a) ☐ As exceções programadas são disparadas automaticamente pelo Oracle.

- b) () Nas exceções do sistema, é possível identificar o tipo de erro através de uma lista de identificação prevista no Oracle.
- c) () Em uma exceção programada, é necessário que exista uma variável definida do tipo EXCEPTION.
- d) () A chamada de uma exceção do sistema é feita utilizando o comando RAISE.

Gabarito:

1) d

2) **F, V, F, F**

3) a e b

4) b

5) **F, V, V, F**

Christiano Cadoná¹

Capítulo **5**

Sequence e Trigger

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campos ULBRA Canoas.

Introdução

Neste capítulo, será abordado um dos principais recursos de um Sistema Gerenciador de Banco de Dados chamado Trigger. As triggers são utilizadas como alternativa na implementação de regras de negócio as quais possuem como principal característica ser disparada automaticamente pelo SGBD assim que ocorrer alguma ação na manipulação dos dados armazenados. Para melhor compreensão, será utilizado exemplos práticos dirigidos de rotinas de Triggers como subsídio a especificações de regras de negócio. Também será abordado o uso de sequence, como alternativa à geração de números sequenciais dentro do SGBD Oracle.

Sequence

Uma Sequence é um objeto de banco de dados que tem por objetivo gerar números inteiros únicos de forma sequencial. São excelentes por possuírem alta performance devido ao SGBD armazenar um range desses números em seu cache. Sua principal utilização está relacionada ao uso desses números gerados na definição de elementos de chaves primárias das tabelas, pois, como já estudado, esses atributos também têm a característica de possuírem valores únicos. Em sua sintaxe, existe uma série de parâmetros, mas, para este material, vamos nos deter em alguns.

Sintaxe:

```
CREATE SEQUENCE nome_da_sequence  
[INCREMENTY BY n]  
[START WITH n]  
[MINVALUE n]  
[MAXVALUE n]  
[CACHE n];
```

Onde:

- ➔ **Nome_da_Sequence**: especifica o nome que a sequence terá. O Nome de uma sequence não poderá ser o mesmo de uma tabela ou de uma view.
- ➔ **INCREMENTY BY**: define como deve ocorrer o incremento ou decremento de um valor. Caso for informado um número positivo, a sequence será crescente e incrementará sempre o valor informado. Caso o valor seja negativo a sequence será decrescente diminuindo o valor de acordo no o valor informado nesse parâmetro. Caso essa cláusula seja omitida, o Oracle entenderá que a sequence será crescente e incrementada pelo valor 1.
- ➔ **START WITH**: especifica ao SGBD o primeiro número a ser gerado na sequence. Caso essa cláusula não seja especificada na criação da sequence, será iniciada por 1.
- ➔ **MINVALUE**: define o menor valor que a sequence poderá possuir. Normalmente, o valor dessa propriedade é o mesmo da propriedade Start With.

- **MAXVALUE**: define o maior valor que a sequence poderá possuir. Caso não informado esse parâmetro, poderá chegar até o número 10^{27} em uma sequence crescente e 1.
- **CACHE**: define o número de valores da sequence que ficarão armazenados no cache do Oracle. Ou seja, só haverá processamento para a recuperação de valores, quando o range de valores armazenados no cache acabar. Quando não informado, o servidor Oracle armazenará 20 valores em seu cache.

Caso seja necessário remover uma sequence do banco, basta executar o comando **DROP SEQUENCE** seguido do nome da sequence. Caso seja necessário alterar a configuração de uma sequence, basta utilizar o comando **ALTER SEQUENCE** seguido do nome da sequence e das alterações nos parâmetros necessários. Os Exemplos 1 e 2 criam duas sequences com configurações distintas. No Exemplo 1, a sequence “teste_sequence1” está sendo criada com o valor inicial de 120 e incremento de 10 para cada. Nesse caso, o próximo número a ser gerado pela sequence será 130. Também está sendo especificado que a sequence irá ser percorrida até que seja atingido o número 9990.

Exemplo 1:

```
create sequence teste_sequence1  
  increment by 10  
  start with 120  
  maxvalue 9990;
```

Exemplo 2:

```
create sequence teste_sequence2;
```

Já no Exemplo 2 está sendo criada uma sequence sem nenhum parâmetro de configuração. Nesse caso, a sequence será configurada com os valores padrões, ou seja, iniciará com o valor 1, será incrementada de um em um elemento, seu maior valor gerado será 10^{27} e possuirá 20 elementos em cache junto ao Oracle. O Exemplo 2 representa a maioria dos sequences que normalmente são criados no SGBD Oracle.

Os Exemplos 3 e 4 ilustram uma alteração e uma remoção das sequences geradas anteriormente. No Exemplo 3, a "teste_sequence1" é removida, e, no Exemplo 4, a sequence "teste_sequence2" foi alterada modificando o valor final da mesma para reger no máximo 1000 números, e o seu incremento que passou a ser de 2 em 2 números.

Exemplo 3:

```
drop sequence teste_sequence1;
```

Exemplo 4:

```
alter sequence teste_sequence2  
increment by 2  
maxvalue 1000;
```

Após criada uma sequence, é necessário conhecer alguns recursos necessários para saber manipulá-la de forma correta. Para realizar referência aos valores da sequence utiliza-se as pseudocolunas **NEXTVAL** e **CURRVAL**, após o nome de uma sequence. A CURRVAL retorna o valor corrente de uma sequence, e a NEXTVAL gera um novo número para a sequence e retorna-o.

O Exemplo 5 apresenta uma sequência de 4 comandos que tem por objetivo exemplificar o funcionamento das sequence. Na primeira linha do exemplo está sendo criada a sequence chamada "exemplo_sequence". Em seguida, estamos executando o NEXTVAL na sequence criada para que esta retorne o valor que acaba de gerar. Nesse caso, o valor retornado será 1, pois a sequence criou o primeiro elemento assim que disparou a pseudocoluna NEXTVAL. É importante que fique esclarecido que, ao criar uma sequence, esta não possui nenhum valor gerado, e, com isso, se executássemos CURRVAL como primeiro comando após a criação de uma sequence, esta retornaria um erro, pois não haveria nenhum valor criado até aquele momento.

Exemplo 5: - "Exemplo de NEXTVAL e CURRVAL"

```
1 create sequence exemplo_sequence
2 select exemplo_sequence.NEXTVAL from dual
3 select exemplo_sequence.CURRVAL from dual
4 select exemplo_sequence.NEXTVAL from dual
```

O comando 3 lista o valor contido na sequence, que neste caso será também 1, pois o comando anterior (linha 2) acabou de ser executado. Já no comando da linha 4, gera novamente um novo valor à sequence e retorna como resposta o valor 2, por ser o próximo número de acordo com a configuração estabelecida na criação da sequence (linha 1), que nesse caso é a configuração padrão.

O Exemplo 6 apresenta uma forma de utilização da sequence criada no Exemplo 5 como alternativa na criação de números de uma chave primária. Para isso, vamos imaginar que já tenha sido criada uma tabela chamada `tbAluno`, com os campos `pkcodalu`, `nomealu` e `mediaalu`. Observe que no exemplo, são inseridos 3 novos alunos, e no local onde estabelecemos o código do aluno, que é uma chave primária, chamamos a sequence com a pseudocoluna `NEXTVAL`, que gerou um novo número para cada uma das linhas de `INSERT` executadas.

Exemplo 6: - “Chamando sequence em INSERT”

```
1 insert into tbAluno(pkcodalu, nomealu, mediaalu)
  values (exemplo_sequence.NEXTVAL, 'Ana',8.5);
2 insert into tbAluno(pkcodalu, nomealu, mediaalu)
  values (exemplo_sequence.NEXTVAL, 'Maria',9.2);
3 insert into tbAluno(pkcodalu, nomealu, mediaalu)
  values (exemplo_sequence.NEXTVAL, 'Carla',5.4);
```

Assim, o controle dos números de chave primária da tabela de alunos não fica mais com o programador, o que sem dúvida daria maior trabalho para ser controlado.

Trigger

Uma trigger, ou gatilho, como também é conhecida, é um tipo especial de procedimento, que é executado automaticamente sempre que há uma tentativa de modificar os dados de uma

tabela, ou seja, uma trigger é disparada automaticamente quando uma instrução INSERT, UPDATE ou DELETE é executada em uma tabela cuja a trigger esteja vinculada. Sua sintaxe:

Sintaxe:

```
CREATE OR REPLACE TRIGGER nome_da_trigger  
[MOMENTO] [EVENTO ]ON nome_da_tabela_vinculada  
[bloco PL/SQL];
```

Onde:

- **Nome_da_trigger**: especifica o nome que a trigger possuirá. O nome não pode ser igual a uma tabela, procedimento ou sequence.
- **Momento**: especifica em que momento de um determinado evento deve ocorrer a chamada da trigger. O momento pode ser:
 - o **BEFORE**: define que a rotina prevista no bloco PL/SQL será executada antes que ocorra a alteração de dados na tabela.
 - o **AFTER**: define que o conteúdo programado no bloco PL/SQL somente será executado depois que os dados de uma tabela foram alterados.
 - o **INSTEAD OF**: indica que o conteúdo existente na trigger irá ser executado no lugar da instrução que disparou a trigger. INSTEAD OF somente pode ser utilizado se a trigger estiver associada a uma VIEW. Trigger

desse tipo são usadas, por exemplo, quando em uma view não temos acesso a todos os dados de uma tabela e necessitamos, por exemplo, inserir um novo registro que necessitará de informações obrigatórias em um de seus atributos. Em seguida, será exemplificado uma trigger com essa característica.

- ➔ **Evento:** o evento especifica qual é a instrução que dispara uma determinada trigger (INSERT, UPDATE ou DELETE). Diferente dos eventos INSERT e DELETE, quando o evento for definido como UPDATE, é possível informar quais colunas que, ao serem alteradas, irão disparar a trigger. Caso não informado, o evento será disparado assim que o comando UPDATE é executado para qualquer coluna que esteja sendo atualizada. O Exemplo 7 apresenta a criação de duas triggers. Na primeira trigger será disparado antes que seja alterado um campo “idadealu” ou “nomealu” da tabela tbaluno. Já a segunda trigger será disparada após o somando insert ou update ou delete serem executados.

Exemplo 7: - “Exemplo de criação parcial de duas trigges”

TRIGGER 1	TRIGGER 2
<pre>CREATE TRIGGER TRIGGER_ TESTE1 BEFORE UPDATE OF IDADEALU,NOMEALU ON TBALUNO BEGIN NULL; END;</pre>	<pre>CREATE TRIGGER TRIGGER_ TESTE2 AFTER INSERT OR DELETE OR UPDATE ON TBALUNO BEGIN NULL; END;</pre>

- **Tipo:** define quantas vezes uma trigger será executada. A trigger pode ser executada apenas uma vez (STATEMENT) quando a instrução a que ela foi criada foi disparada (insert, update ou delete), ou executar quantas foram as linhas em que ela causou modificações (ROW) na tabela a que estava vinculada. Nesse caso, devemos identificar o tipo de trigger que desejamos. Ela pode ser:
 - o Por Instrução (STATEMENT): quando a trigger for do tipo instrução, ela será disparada apenas uma vez para cada evento de trigger, mesmo que nenhum registro tenha sido modificado. Caso não for informado um tipo na especificação da trigger, ela assumirá ser do tipo instrução.
 - o Por Linha (ROW): quando a trigger for do tipo por linha, ela será executada quantas vezes forem os registros que ela afetou. Por exemplo, se existir uma trigger relacionada ao evento UPDATE, e for realizado um update que altera o salário de 200 funcionários, a trigger será disparada 200 vezes, uma para cada linha modificada na tabela. O Exemplo 8 apresenta um esboço de duas triggers, sendo a primeira do tipo instrução, por não ser especificado o seu tipo, e a segunda do tipo linha onde é utilizado a cláusula FOR EACH ROW em sua especificação.

Exemplo 8: - “Exemplo definição de tipo de trigger”

TRIGGER 3	TRIGGER 4
<pre>CREATE TRIGGER TRIGGER_TESTE3 ALTER UPDATE OR DELETE ON TBALUNO BEGIN NULL; END;</pre>	<pre>CREATE TRIGGER TRIGGER_TESTE4 BEFORE INSERT OR DELETE ON TBALUNO FOR EACH ROW BEGIN NULL; END;</pre>

➤ **Corpo:** no bloco corpo, são programadas as ações que a trigger deve executar quando esta for disparada. O bloco inicia por DECLARE ou BEGIN e termina por END.

Todos os conceitos já abordados nos procedimentos e funções já trabalhados podem ser aplicados dentro de uma trigger exceto a trigger que possuir finalização de transações através dos comandos COMMIT ou ROLLBACK. Sendo assim, é possível chamar funções e ou procedimentos dentro de uma trigger desde que estes não façam controle de transação.

Outro recurso muito utilizado em triggers é a possibilidade de realizar referências a valores existentes nas colunas. Para isso, utilizamos antes do atributo a referência :OLD (antigo valor) ou :NEW (novo valor). A referência :OLD indica que estamos usando os valores antes de uma alteração. Caso quiséssemos usar o valor atualizado, a referência seria :NEW. Ambas podem ser usadas na mesma trigger, contudo a aplicação dessas referências só será possível se for utilizado o tipo ROW (FOR EACH ROW) na definição da trigger, ou a trigger possua o momento INSTEAD OF.

Para exemplificar o conceito de trigger, vamos criar uma nova tabela chamada “TBALUNO”, uma sequence chamada “tbaluno_seq” iniciando em 20 além de popular a tabela com 3 registros fixos. Também criaremos uma View de nome “alunoview” que lista apenas o nome do aluno e sua média. O Exemplo 9 apresenta as instruções dessa estrutura base para os testes com trigger.

Exemplo 9: - “Estrutura base para testar os exemplos de trigger”

```
1 CREATE TABLE TBALUNO(  
2  codigo integer NOT NULL,  
3  nome VARCHAR2(30),  
4  idade integer NOT NULL,  
5  nota1 number(15,2),  
6  nota2 number(15,2),  
7  media number(15,2),  
8  constraint tbaluno_pk primary key(codigo));  
9 insert into TBALUNO(codigo,nome,idade,nota1,nota2,media) values (1, 'Marta',19, 9.5, 8.5, 9);  
10 insert into TBALUNO(codigo,nome,idade,nota1,nota2,media) values (2, 'Ana', 17, 7.3, 8.5, 7.9);  
11 insert into TBALUNO(codigo,nome,idade,nota1,nota2,media) values (3, 'Carla', 22, 5.8, 4.8, 5.3);  
12 create sequence tbAluno_seq START WITH 20;  
13 create or replace view alunoview as (select nome,media from tbaluno);  
14 select * from tbaluno;  
15 Select * from alunoview;
```

A Figura 1 apresenta os resultados da execução dos comandos das linhas 14 e 15 do Exemplo 9. Observe que, por enquanto, a sequence “tbAluno_seq” criada não foi utilizada.

SELECT * FROM TBALUNO						SELECT * FROM ALUNOVIEW	
CODIGO	NOME	IDADE	NOTA1	NOTA2	MEDIA	NOME	MEDIA
1	Marta	19	9,5	8,5	9	Marta	9
2	Ana	17	7,3	8,5	7,9	Ana	7,9
3	Carla	22	5,8	4,8	5,3	Carla	5,3

Figura 1 Resultado do comando das linhas 14 e 15 do Exemplo 9.

Inicialmente, vamos exemplificar a criação de uma trigger que possua o momento `INSTEAD OF`. Lembrando que esse tipo de trigger só pode ser executado se estiver vinculada a uma `VIEW`. Nesse caso, vamos criar uma trigger que será disparada quando incluir (executar um `insert`) na `VIEW` criada. Devemos considerar que a `VIEW` só possui dois campos, o nome e a média, e pelo menos devemos considerar que não será possível inserir um novo registro sem informar o código e a idade do aluno, pois ambos são campos obrigatórios.

Nesse caso, vamos criar uma trigger que execute um `INSERT` na tabela de aluno onde o código será gerado através da sequence especificada, e, para o campo idade, será atribuída 0 (zero). Os campos `Nota1` e `Nota2` também serão setados com o mesmo valor que será armazenada a média. Assim será mantida uma consistência na informação armazenada. O Exemplo 10 apresenta a criação da trigger e uma instrução de inserção de dados na `VIEW`.

Exemplo 10: - “Trigger de momento INSTEAD OF”

```
1 create or replace
2 trigger alunoview_IOINS INSTEAD OF insert on alunoview
3 BEGIN
4 insert into TBALUNO(codigo,nome,idade,nota1,nota2,media)
5 values (tbaluno_seq.NEXTVAL, :NEW.NOME, 0, :NEW.MEDIA, :NEW.
6 MEDIA, :NEW.MEDIA);
7 END alunoview_IOINS;

7 insert into alunoview(nome,media) values('Bruna',6.7);
8 Select * from tbaluno;
```

O primeiro conceito a ser exemplificado é o de substituir o comando original pelo conteúdo existente na trigger. Nesse caso, observe que o comando que dispara essa trigger é o da linha 7 do Exemplo 10. Nesse comando de INSERT, apenas podemos informar os campos de nome e média, pois são os únicos campos que a VIEW chamada “alunoview” possui. Assim que executado o comando, a trigger é disparada e automaticamente ignora a linha de comando original (linha 7). Dentro do bloco de comando da trigger, é criado um novo comando de inserção de dados onde são informados todos os parâmetros da tabela “tbaluno”. Observe que o campo código que é a chave primária da tabela de aluno e está sendo populado com o conteúdo da sequence, que neste caso deverá ser 20, por ser o primeiro elemento criado na sequence “tbaluno_seq”. O campo idade que era obrigatório está sendo preenchido com valor igual a 0. Mas a maior consideração relacionada ao exemplo da trigger criada no Exemplo 10 está relacionada ao uso das referências ao dados da tabela. Na

linha 5 está sendo utilizado :NEW seguido dos campos que faziam parte da VIEW original. Com isso, é possível saber qual eram os valores informados no comando INSERT original existente na linha 7 (nesse caso, o nome valerá "Bruna" e a média será igual á "6,7"). Assim, ao utilizar :NEW.nome e :NEW.media, está sendo informado ao comando INSERT os valores que foram informados na linha de comando 7. A Figura 2 apresenta o resultado do comando 8 que manda listar todos os dados da tabela aluno.

SELECT * FROM TBALUNO					
CODIGO	NOME	IDADE	NOTA1	NOTA2	MEDIA
1	Marta	19	9,5	8,5	9
2	Ana	17	7,3	8,5	7,9
3	Carla	22	5,8	4,8	5,3
20	Bruna	0	6,7	6,7	6,7

Figura 2 Resultado do comando da linha 8 do Exemplo 10

O Exemplo 11 apresenta a criação de uma trigger chamada **TBALUNO_BI**, que será disparada antes da inserção de dados e possui como objetivo, colocar um código de aluno baseado na sequence criada no Exemplo 9. A trigger também irá calcular a média aritmética das notas 1 e 2 e armazenará no campo média. Por fim, se for informado uma idade menor do que zero ou nula, a trigger definirá a idade como sendo zero (0). Assim, se informada uma média diferente, a média aritmética, a trigger irá corrigir, além de manter pelo menos um valor válido na coluna de idade. Observe que da mesma forma como o momento **INSTEAD OF**, triggers do tipo por li-

na (FOR EACH ROW) podem acessar o conteúdo das colunas da tabela associada à trigger como é o caso do exemplo que segue através de :NEW.

Exemplo 11: - “Exemplo de Trigger chamada TBALUNO_BI”

```
1 create or replace TRIGGER tbaluno_bi
2 BEFORE INSERT ON tbaluno
3 FOR EACH ROW
4 declare
5   xcod tbaluno.codigo%type;
6 BEGIN
7   select tbaluno_seq.NEXTVAL into xcod from dual;
8   :new.codigo :=xcod; -- atribui para a coluna codalu o conteúdo de xcod
9   :new.media:=(:new.nota1 + :new.nota2)/2; --calcula a média e atribui o valor
10  if (:new.idade is null) or (:new.idade<0) then
11    :new.idade:=0;
12  end if;
13 END;
```

Como forma de identificar a funcionalidade da trigger criada, o Exemplo 12 apresenta a inclusão de 3 novos alunos na tabela de alunos, contendo em suas colunas valores que tornam os dados inconsistentes.

Exemplo 12: - “Inserindo novos registros”

```
1 insert into TBALUNO(codigo,nome,idade,nota1,nota2,media)
   values (5, 'Anderson',26, 8.5, 9.5, 6);
2 insert into TBALUNO(nome,idade,nota1,nota2) values ('Sergio', -25,
   6.1, 8.3);
3 insert into TBALUNO(nome ,nota1,nota2) values ('Juliano', 4.5, 5.7);
```

Já a Figura 3 mostra como ficaram os dados após a inclusão dos três novos registros no banco de dados. Observe que o aluno “Anderson” que no exemplo foi informado como contendo o código 5, foi armazenado na verdade com o código 20, pois a trigger ignorou a informação passada pelo comando insert e executou a rotina que atribui o valor da sequence ao código. No mesmo registro, havia sido informada que a média do aluno “Anderson” seria 6, porém a trigger recalculou e armazenou como 9 (a média dos valores 8,5 e 9,5).

SELECT * FROM TBALUNO					
CODIGO	NOME	IDADE	NOTA1	NOTA2	MEDIA
1	Marta	19	9,5	8,5	9
2	Ana	17	7,3	8,5	7,9
3	Carla	22	5,8	4,8	5,3
20	Bruna	0	6,7	6,7	6,7
21	Anderson	26	8,5	9,5	9
22	Sergio	0	6,1	8,3	7,2
23	Juliano	0	4,5	5,7	5,1

Figura 3 Resultado do banco após a execução do Exemplo 12

No INSERT que armazenava o registro do aluno “Sergio” (linha 2 do Exemplo 12), a idade foi informada como negativa, e a trigger tratou a informação colocando sua idade para 0. O mesmo ocorreu com a idade do aluno “Juliano” que não foi informada e a trigger, adicionou a idade 0 para o aluno, além, é claro, de que em ambos os casos a média foi calculada e atribuída a cada registro.

É importante ressaltar que caso ocorra um comando de alteração de algum registro (UPDATE), não ocorrerá essa validação, pois essa trigger do Exemplo 11 está programada somente para ações do tipo INSERT.

Outro ponto que deve ser registrado é a possibilidade de atribuir um novo valor à coluna através de :NEW, como foi abordado no Exemplo 11, onde foram alterados os campos de código do aluno, idade e média do mesmo. Isso somente é possível no momento BEFORE (antes), pois a tabela ainda não foi alterada. Caso tente utilizar essa ação no momento AFTER (depois) ocorrerá um erro, pois essa ação só é disparada depois de realizar a alteração, impossibilitando edição do registro modificado.

O Exemplo 13 cria uma outra tabela chamada TBALUNOANT que contém praticamente a mesma estrutura da tabela TBALUNO. Essa tabela servirá para armazenar todas a informação que existiam na tabela TBALUNO, antes da execução do comando UPDATE ou DELETE executadas na tabela TBALUNO. Dessa forma, teremos um histórico dos dados armazenados anteriormente na tabela de aluno.

Exemplo 13: - "Criar tabela que armazenará dados antigos de aluno"

```
1 CREATE TABLE TBALUNOANT(  
2  codigo integer NOT NULL,  
3  nome VARCHAR2(30),  
4  idade integer,  
5  nota1 number(15,2),  
6  nota2 number(15,2),  
7  media number(15,2),  
8  operacao varchar2(15));
```

A trigger criada no Exemplo 14 implementa a rotina proposta de popular uma tabela que contenha os dados anteriores à alteração e ou remoção de registros da tabela “tbaluno”.

Exemplo 14: - “Exemplo de Trigger chamada TBALUNO_AUD”

```
1  create or replace trigger TBALUNO_AUD
2  AFTER update or delete on tbaluno
3  FOR EACH ROW
4  DECLARE
5  woperacao varchar2(15);
6  BEGIN
7  IF updating THEN
8  woperacao:='Alterou';
9  END IF;
10 IF deleting THEN
11 woperacao:='Removeu';
12 END IF;
13 insert into TBALUNOANT(codigo,nome,idade,nota1,nota2,media,operacao)
   VALUES(:old.codigo,:old.nome,:old.idade,:old.nota1,:old.nota2,
   :old.media,woperacao);
14 END;
```

Observe que os testes realizados pelo IF da trigger implementada no Exemplo 14, verificam se a trigger foi disparada no estado de alteração (updating) ou de Exclusão (deleting – linhas 7 e 10). Caso um dos tipos corresponda, ela retornará verdadeira, que possibilitará, nesse caso, que a variável woperacao, possua um valor distinto. Caso seja necessário testar o estado de inclusão, o comando é inserting. Foi utilizado o método :OLD para recuperar as informações antigas do aluno, as quais foram utilizadas para popular a tabela TBALUNOANT

(linha 13). Dessa forma, após alterar ou remover um registro na tabela de aluno, será incluído um novo registro na tabela de “TBALUNOANT”.

Como forma de ilustrar a ação da trigger criada no Exemplo 14, o Exemplo 15 apresenta uma sequência de comandos que executam duas alterações na tabela (linhas 1 e 2) e a remoção de dados de um registro (linha 3).

Exemplo 15: - “Exemplos para chamada da trigger TBALUNO_AUD”

1	update tbaluno set idade=40 where codigo=23;
2	update tbaluno set idade=idade+1 where codigo<=3;
3	delete tbaluno where codigo=21;
4	select * from tbaluno;
5	select * from tbalunoant

Tendo como base que o banco de dados possui como conteúdo os dados apresentados na Figura 3, na primeira linha do exemplo, a idade do aluno de código 23 passará de 0 (zero) para 40 anos. Já na segunda linha de comando, a idade de “Marta”, “Ana” e “Carla” será incrementada em mais um ano. Na terceira linha de comando do Exemplo 15, será removido o registro que contém os dados do aluno “Anderson”.

As Figuras 4 e 5 apresentam o resultado das consultas realizadas nos comandos 4 e 5 do Exemplo 15, onde pode ser identificado a execução correta da trigger criada no Exemplo 14. Observe que os registros contidos na tabela TBALUNOANT, são exatamente os registros anteriores à execução dos comandos do Exemplo 15.

SELECT * FROM TBALUNO					
CODIGO	NOME	IDADE	NOTA1	NOTA2	MEDIA
1	Marta	20	9,5	8,5	9
2	Ana	18	7,3	8,5	7,9
3	Carla	23	5,8	4,8	5,3
20	Bruna	0	6,7	6,7	6,7
22	Sergio	0	6,1	8,3	7,2
23	Juliano	40	4,5	5,7	5,1

Figura 4 Resultado do banco após a execução da linha 4 do Exemplo 15

SELECT * FROM TBALUNOANT						
CODIGO	NOME	IDADE	NOTA1	NOTA2	MEDIA	OPERACAO
23	Juliano	0	4,5	5,7	5,1	Alterou
1	Marta	19	9,5	8,5	9	Alterou
2	Ana	17	7,3	8,5	7,9	Alterou
3	Carla	22	5,8	4,8	5,3	Alterou
21	Anderson	26	8,5	9,5	9	Removeu

Figura 6 Resultado do banco após a execução da linha 5 do Exemplo 15

Como já mencionado anteriormente, o uso de :NEW e :OLD somente é possível em triggers do tipo linha FOR EACH ROW, e em triggers cujo momento seja INSTEAD OF. Porém, é importante ressaltar que, dependendo do evento, alguns valores não estarão disponíveis. Por exemplo, caso tenhamos uma trigger de evento INSERT, esta não possuirá nenhuma informação em :OLD, pois não existem dados anteriores ao comando de inserção armazenados. O mesmo ocorre em um evento de DELETE, onde não será possível utilizar :NEW, pois o comando de DELETE remove dados e não adiciona um novo elemento à

tabela. Mas no evento de UPDATE é possível utilizar ambas as ações, inclusive utilizar as duas em conjunto.

Como forma de exemplificar o uso de :OLD e :NEW dentro da mesma trigger, o Exemplo 16 apresenta uma alteração na trigger criada no Exemplo 14. Nesse exemplo, foi alterado o modo como os dados são inseridos na tabela TBALUNOANT na ocorrência de um evento de UPDATE na tabela TBALUNO. Essa inserção somente será permitida se algum campo da tabela TBALUNO foi alterado. O código de inserção quando excluído um aluno não foi modificado.

Exemplo 16: - “Uso de :NEW e :OLD juntos”

```
1  create or replace trigger TBALUNO_AUD
2  AFTER update or delete on tbaluno
3  FOR EACH ROW
4  declare
5  woperacao varchar2(15);
6  begin
7  IF (updating) AND
8  (:old.codigo<>:new.codigo or :old.nome<>:new.nome
9  or :old.idade<>:new.idade or :old.nota1<>:new.nota1
10 or :old.nota2<>:new.nota2 or :old.media<>:new.media) THEN
11  woperacao:='Alterou';
12  insert into TBALUNOANT(codigo,nome,idade,nota1,nota2,media,
13 operacao) VALUES(:old.codigo,:old.nome,:old.idade,:old.nota1,
14 :old.nota2,:old.media,woperacao);
15 END IF;
16 IF deleting THEN
17  woperacao:='Removeu';
18  insert into TBALUNOANT(codigo,nome,idade,nota1,nota2,media,
19 operacao) VALUES(:old.codigo,:old.nome,:old.idade,:old.nota1,
20 :old.nota2,:old.media,woperacao);
21 END IF;
22 END;
```

Observe que na linha 8 do Exemplo 16 está sendo testado se o antigo e o novo valor contido em cada uma das colunas da tabela de aluno é diferente. Somente se alguma informação é diferente é executada a linha de inserção de dados na tabela TBALUNOANT.

Recapitulando

Este capítulo contextualizou e exemplificou os conceitos de sequence e triggers. Inicialmente, entendemos que a sequence são objetos do banco de dados que geram números inteiros únicos e geralmente são utilizados para criação de chave primária. Ela possui pseudocolunas que realizam o incremento de valores e retornam estes para serem utilizados.

Em seguida, foi abordada a sintaxe das triggers onde foram estudadas cada uma de suas partes, iniciando pelo seu nome, seu momento (antes, depois ou INSTEAD OF), o evento no qual a trigger está relacionada (INSERT, UPDATE ou DELETE), seu tipo se for por linha ou por instrução, e por fim o corpo que contém todo o conteúdo a ser executado na trigger. Foram realizados exemplos práticos mostrando o conteúdo dos dados armazenados após um evento que possuísse uma trigger associada a ele. Nesses exemplos, foram explorados o uso de :NEW e :OLD como referência aos dados contidos na tabela além de compreender como é possível testar dentro de uma trigger que possui mais de um evento relacionado qual evento disparou a trigger (updating, inserting e deleting).

Referências

ELMASRI, R. & NAVATHE, S. B. Sistemas de Banco de Dados. Rio de Janeiro: LTC, 2002. 4ª ed

DATE, C. J. Introdução a Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 2003. 9ª ed. .

KORTH, Henry F. e SILBERSCHATZ, Abraham. Sistema de Bancos de Dados. São Paulo: Makron Books, 1999. 3ª edição revisada.

Atividades

1) Assinale a alternativa correta para a assertiva:

I – A cláusula :NEW e :OLD podem ser utilizadas apenas em:

- a) () Triggers cujo momento seja INSTEAD OF e triggers do tipo por linha.
- b) () Triggers do tipo instrução e possuam BEFORE.
- c) () Em Sequences que geram valores únicos.
- d) () Em Sequences que possuam o tipo por linha.

2) Marque “V” para a assertiva verdadeira e “F” para a assertiva falsa, relaciona à sequence:

- a) () Uma sequence pode iniciar por qualquer valor.

- b) () Em uma sequence, é possível gerar números em ordem decrescente.
 - c) () É possível em uma sequence criar uma sequência de caracteres.
 - d) () Na criação de uma sequence, é possível determinar o tamanho do cache que armazena os números no Oracle.
- 3) Com apenas parte do cabeçalho da trigger que segue, é possível definir uma série de características que a trigger irá possuir, exceto duas das alternativas que são:

```
create or replace trigger TBTESTETRIGGER  
BEFORE INSERT or delete on TBTESTE  
BEGIN  
:
```

- a) () A Trigger será disparada antes de inserir ou remover um registro.
 - b) () A Trigger será disparada após inserir ou remover um registro.
 - c) () A Trigger é do tipo instrução.
 - d) () A Trigger é do tipo linha.
- 4) Tendo como base o esboço da trigger que segue, qual das opções representa o correto preenchimento das lacunas de forma que a trigger funcione corretamente.

```
1 create or replace trigger TBALUNO_BUD
2 _____ update or delete on tbaluno
3 FOR EACH ROW
4 begin
5 IF ( _____ ) AND (:old.nota1 <> :new.nota1
6 or :old.nota2 <> :new.nota2 ) THEN
7 :new.media:=(:new.nota1 + :new.nota2)/2;
8 END IF;
9 IF _____ THEN
10 insert into TBALUNOANT(codigo, nome, idade, nota1, nota2, media)
11 VALUES(:old.codigo, :old.nome, :old.idade, :old.nota1,
12 :old.nota2, :old.media);
13 END IF;
14 END;
```

- a) () AFTER – UPDATING – DELETING
 - b) () AFTER – DELETING – UPDATING
 - c) () BEFORE – UPDATING – DELETING
 - d) () BEFORE – DELETING – UPDATING
- 5) Tendo como base parte de uma trigger, é possível definir que o comando faltante na lacuna pode ser preenchido pelo(s) comando(s).

```
1  create or replace trigger TBALUNO_BIUD
2  BEFORE insert or update or delete on tbaluno
3  FOR EACH ROW
4  Begin
5  IF ( _____ ) AND (:old.nota1 <> :new.nota1
6  or :old.nota2 <> :new.nota2 ) THEN
7  :new.media:=(:new.nota1 + :new.nota2)/2;

  :
```

- a) () INSERTING
- b) () UPDATING
- c) () DELETING
- d) () INSTEAD

Gabarito:

1) a 2) V, V, F, V 3) b e d 4) c 5) a

Christiano Cadoná¹

Capítulo 6

Cursores

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campos ULBRA Canoas.

Introdução

Este capítulo tem como objetivo apresentar o conceito e funcionamento de cursores relacionados à PL/SQL, os quais subsidiarão ao programador do banco resolver problemas de retorno de consultas com mais de um registro dentro de um bloco PL/SQL. Para melhor compreensão, todos os conceitos apresentados neste capítulo possuirão exemplos práticos de uso tendo como base o SGBD Oracle.

Conceito e exemplo de cursores

Quando foi abordado o conceito de procedimentos, funções e trigger, foram realizados alguns exemplos que executavam o comando `SELECT` no bloco PL/SQL. Contudo, foi observado que a aplicação da cláusula `INTO` no comando `SELECT` poderia não funcionar de forma correta dependendo de como o comando `SELECT` fosse especificado.

O Exemplo 1 ilustra a definição de uma função que possui em seu corpo um exemplo da ocorrência desse tipo de problema. A função tem como objetivo buscar o valor de venda de um produto, cujo produto possua uma categoria informada como parâmetro na definição da função.

Exemplo 1: - "Exemplo de problema na execução do SELECT"

```
1 create or replace
2 FUNCTION funcaocomerro (codcategoria IN NUMBER)
3 RETURN number
4 IS
5 wvalor NUMBER;
6 BEGIN
7 wvalor:=0;
8 select nvl(valorproduto,0) INTO wvalor from tbproduto
   where fkcodprod=codcategoria;
9 RETURN (wvalor);
10 END funcaocomerro;
```

Observe que sintaticamente esse procedimento não possui erros, porém existe um erro lógico na Linha 8 da função. A linha 8, como já mencionado anteriormente, retorna o valor de um produto de uma determinada categoria. Caso a categoria passada como parâmetro possua nenhum ou apenas um produto, o comando funcionará sem nenhum problema. Contudo, se o resultado retornar mais de um valor de produto (uma lista de valores de produtos) ocorrerá um erro, pois não é possível que uma variável (neste caso a variável **wvalor**), possua mais do que um valor de forma simultânea. Esse tipo de erro seria muito previsível, pois a probabilidade de existir mais de um produto com valores diferentes de uma determinada categoria é muito grande. Em casos como este, onde o resultado de uma consulta possa retornar mais de um valor (lista de valores), devemos utilizar o conceito de cursores.

Cursores são utilizados então quando ocorre a necessidade de tratar uma lista de informação resultada de um SELECT

dentro de um bloco de código PL/SQL. Sendo assim, inicialmente, deve-se obedecer ao conceito de que não é possível acessar ao mesmo tempo mais de um registro de retorno. Partindo dessa premissa, o cursor nada mais é do que uma forma de apontar para um determinado registro de resposta de uma lista de respostas.

Sendo assim, para que seja possível utilizar um cursor, este deve ser declarado, e o resultado da linha de onde este está apontando deve ser atribuído a variáveis.

Geralmente, utilizamos os cursores em quatro etapas:

- **Declaração**: na declaração de um cursor, especificamos o que ele irá executar, ou seja, qual será o comando SQL que este irá executar (opcionalmente). A declaração do cursor é realizada normalmente na declaração de uma variável e esta deve ser do tipo CURSOR. Quando um cursor é declarado, ele não é executado; neste momento, apenas está sendo definido que a variável será um cursor.
- **Abertura**: para que o cursor funcione, é necessário abri-lo. Somente quando abrimos um cursor é que a consulta a qual ele faz referência será carregada para a memória.
- **Atribuição de valores às variáveis**: assim que aberto o cursor, é possível atribuir os valores do registro que o cursor está apontando a variáveis que podem ser manipuladas no PL/SQL. O ato de atribuir elementos do

cursor a variáveis possibilita o avanço do cursor para o próximo registro da pesquisa SQL.

➔ **Fechamento:** após a manipulação do cursor, é importante que este seja fechado para que libere memória para outros processos do banco.

○ Exemplo 2 apresenta um trecho de um procedure que cria um cursor de nome “C1” que irá apontar para uma consulta que lista o código, nome e salário de todos os funcionários do estado de São Paulo.

Exemplo 2: - “Exemplo de declaração de Cursor”

```
1 create or replace procedure declaracursor
2 AS
3   CURSOR C1 IS SELECT cod,nome,salario FROM
   tbfuncionario where estado='SP';
4 BEGIN
5   :
6   :
```

Observe que nesse método de criação a forma de como se declara um cursor é diferente da declaração de uma variável. Primeiramente, colocamos a referência CURSOR seguida do nome do cursor (nesse caso “C1”). Depois informamos a palavra reservada “IS” seguida da consulta que será realizada.

A Tabela 1 apresenta os comandos mais utilizados quando trabalhamos com cursores.

Comando	Funcionalidade
OPEN <i>nome_do_cursor</i> [for seleção_de_dados];	O comando OPEN abre um cursor. Caso o cursor não possua a especificação da consulta que irá realizar, é possível, através do parâmetro FOR, definir qual será a <u>consulta a ser realizada</u> .
FETCH <i>nome_do_cursor</i> INTO <i>variáveis</i> ;	Atribui o conteúdo do registro em que o cursor referencia as variáveis informadas após a cláusula INTO. Todas as vezes que esta operação é realiza, o cursor avança para o próximo registro.
CLOSE <i>nome_do_cursor</i> ;	Fecha um cursor.
<i>nome_do_cursor</i> %ISOPEN	Atributo que retorna TRUE se o cursor está aberto.
<i>nome_do_cursor</i> %ROWCOUNT	Atributo retorna o número da linha em que o cursor está.
<i>nome_do_cursor</i> %NOTFOUND	Atributo do cursor que retorna um booleano (seu valor default é TRUE) indicando se NÃO existem itens dentro do cursor.
<i>nome_do_cursor</i> %FOUND	Atributo do cursor que retorna um booleano (seu valor default é TRUE) indicando se existem itens dentro do cursor.

Para exemplificar a utilização de cursores, o Exemplo 3 apresenta a criação de uma tabela de produtos contendo a inserção de quatro produtos quaisquer.

Exemplo 3: - "Script para criação da estrutura modelo dos exemplos"

1	create table tbproduto (
2	codigo integer not null,
3	nome varchar(20) not null,
4	estoque integer,
5	valor decimal(15,2),
6	primary key(codigo));
7	insert into tbproduto(codigo, nome, estoque, valor) values (1,'banana',200,2.99);
8	insert into tbproduto(codigo, nome, estoque, valor) values (2,'uva',150,5.25);
9	insert into tbproduto(codigo, nome, estoque, valor) values (3,'abacaxi',75,2.75);
10	insert into tbproduto(codigo, nome, estoque, valor) values (4,'pera',48,4.35);

O Exemplo 5 apresenta a criação de um procedimento que possui como finalidade realizar a impressão do código e nome de todos os produtos cadastrados em ordem de código de produto. Com o propósito de exemplificar os comandos apresentados na Tabela 1, o Exemplo 4 implementa esses comandos. Utilize o SQLDeveloper ou outra IDE de conexão ao banco de dados Oracle que consiga implementar o comando "dbms_output" para que consiga ver o resultado da execução do procedimento criado.

Observe que na linha 5 do Exemplo 4 o cursor C1 é definido como uma consulta listando o código(s) e nome(s) do(s) produtos(s) cadastrados. Somente na execução da linha 7 será aberto o cursor e seu conteúdo será carregado para a memória. Nesse momento, o cursor apenas está aberto, mas não

possui nenhum registro vinculado a ele. O comando `FETCH`, existente na linha 15 do exemplo, é responsável por pegar o conteúdo existente na posição que o cursor está posicionando. Observe que na linha 9 está sendo testado se o cursor possui algum valor. Nesse caso, a primeira vez que essa linha é executada dentro do laço irá retornar falso, pois o cursor ainda não possui dados associados a ele, nas demais vezes retornará verdadeiro.

Como já mencionado, na linha 15, o cursor é preenchido com o conteúdo do registro no qual o cursor está apontando e atribui esses dados às variáveis relacionadas a este. Observe que a sequência das colunas executadas como retorno do `SELECT` deve ser a mesma da atribuição dos valores que as variáveis são apresentadas no comando `FETCH`. Nesse caso, a primeira variável deve conter um código e a segunda o nome do produto. Na primeira vez que o comando `FETCH` for executado, as variáveis `wcodigo` e `wnome` possuirão o código "1" o nome "banana". E execução do comando `FETCH` também avança o cursor para o próximo registro do resultado da consulta `SELECT` (registro de código "2", produto "uva") definida na linha 5.

Na linha 18, o comando `%ROWCOUNT` é executado e retornará na primeira vez que for executado o valor 1 como resposta.

Exemplo 4: - "Exemplo 1 de utilização de cursores"

```

1  create or replace procedure exemploCur1
2  As
3  wcodigo tbproduto.codigo%type;
4  wnome tbproduto.nome%type;
5  CURSOR C1 IS SELECT codigo, nome FROM tbproduto order by codigo;
6  begin
7  OPEN C1; --Abrindo o cursor
8  loop --instrução de início do loop
9  if (C1%FOUND) then
10 dbms_output.put_line('Existe registros no cursor');
11 else
12 dbms_output.put_line('Não existe ninguém no cursor');
13 end if;
14 dbms_output.put_line('=====');
15 FETCH C1 INTO wcodigo, wnome;
16 dbms_output.put_line('Código: ' || wcodigo);
17 dbms_output.put_line('Nome: ' || wnome);
18 dbms_output.put_line('Estou "varrendo" a linha ' || C1% ROWCOUNT);
19 exit when C1%NOTFOUND=true; --condição de saída do laço loop
20 end loop;
21 CLOSE C1;
22 if (C1%ISOPEN) then
23 dbms_output.put_line('O cursor está aberto!');
24 else
25 dbms_output.put_line('O cursor foi fechado com sucesso!');
26 end if;
27 end ;

```

Na linha 19, temos o teste de permanência para o laço de repetição. Nessa linha está sendo testado se o cursor não possui dados. Caso não exista o laço, é interrompido. Já na linha 21, o cursor está sendo fechado através do comando CLOSE. A linha 22 faz um teste se o cursor está fechado ou não. Nesse caso, ele deve sempre tornar falso, pois na linha 21 fechamos o cursor. Esse teste está sendo apresentado apenas para ilustrar o funcionamento do comando %ISOPEN.

O Exemplo 5 implementa o procedimento chamado exemploCur2 que lista o código, nome e estoque de todos os produtos cadastrados em ordem de nome de produto. Na linha 6, o cursor "C2" está sendo definido e na linha 8 está sendo aberto. A principal diferença desse procedimento em relação ao anterior é o tratamento da parada do loop, o qual está utilizando o comando %FOUND que testa se existe conteúdo no cursor (linha 12). Essa linha verifica se não existe um registro no cursor. Caso não exista, o LOOP será abordado (finalizado). Observe que, nesse exemplo, já foram omitidos uma série de testes que foram utilizados no exemplo anterior a fim de demonstrar o funcionamento de alguns comandos de cursores.

Exemplo 5: - "Exemplo 2 de utilização de cursores"

```
1 create or replace procedure exemploCur2
2 As
3 wcodigo tbproduto.codigo%type;
4 wnome tbproduto.nome%type;
5 westoque tbproduto.estoque%type;
6 CURSOR C2 IS select codigo, nome, estoque from tbproduto
order by estoque;
7 begin
8 OPEN C2; --Abrindo o cursor
9 loop --instrução de início do loop
10 dbms_output.put_line('*****');
11 FETCH C2 INTO wcodigo, wnome,westoque;
12 if (not C2%FOUND) then --se não possuir dados sai do loop
13 exit;
14 end if;
15 dbms_output.put_line('Estou "varrendo" a linha ' ||
C2%rowcount );
16 dbms_output.put_line('Código do produto: ' || wcodigo);
17 dbms_output.put_line('Nome no produto: ' || wnome);
18 dbms_output.put_line('Estoque: ' || westoque);
19 end loop;
20 CLOSE C2; --fechando o cursor
21 end exemploCur2;
```

No Exemplo 6, o controle relacionado ao cursor utiliza o laço de repetição WHILE. Nesse caso, foi necessário que seja realizado a recuperação dos dados utilizando FETCH em dois momentos (linha 8 e 14). Nesse exemplo, o cursor percorre uma consulta que lista o nome e o valor de todos os produtos cadastrados em ordem de valor do produto.

Exemplo 6: - “Exemplo 3 de utilização de cursores e laço While”

```
1 create or replace procedure exemploCur3
2 As
3 wnome tbproduto.nome%type;
4 wvalor tbproduto.valor%type;
5 CURSOR C3 IS SELECT nome, valor FROM tbproduto ORDER BY valor;
6 begin
7 OPEN C3;
8 FETCH C3 INTO wnome, wvalor;
9 while (C3%found) loop
10 dbms_output.put_line('*****');
11 dbms_output.put_line('Estou “varrendo” a linha ’ || C3%rowcount );
12 dbms_output.put_line('Nome no produto: ’ || wnome);
13 dbms_output.put_line('Valor: ’ || wvalor);
14 FETCH C3 INTO wnome, wvalor;
15 end loop;
16 CLOSE C3;
17 end exemploCur3;
```

O Exemplo 7 descreve a criação de um procedimento cujo resultado do cursor criado dependerá do valor informado na variável WINICIAL (linha 2), que é um parâmetro de entrada do procedimento. Assim, o procedimento só listará os dados dos registros cujo nome do produto inicie com o conteúdo informado na variável WINICIAL. A linha 6 e 7 apresentam a consulta SQL relacionada ao cursor, onde é identificado na linha 7 que a variável WINICIAL participa da regra estabelecida no retorno do cursor.

O restante do procedimento é semelhante aos procedimentos já discutidos neste capítulo, como a regra de permanência no laço LOOP, a abertura e fechamento do cursor e a captura do conteúdo da linha de onde o cursor aponta para as variáveis relacionadas.

Exemplo 7: - “Exemplo 4 de utilização de cursores com parâmetro”

```
1 select * from tbproduto create or replace procedure exemploCur4
2 ( WINICIAL IN varchar2)
3 As
4 wcodigo tbproduto.codigo%type;
5 wnome tbproduto.nome%type;
6 CURSOR C4 IS SELECT codigo,nome FROM tbproduto
7 where upper(nome) like upper(WINICIAL || '%') ORDER BY nome;
8 begin
9 OPEN C4;
10 loop
11 dbms_output.put_line('*****');
12 FETCH C4 INTO wcodigo, wnome;
13 if (not C4%found) then
14 exit;
15 end if;
16 dbms_output.put_line('Código do produto: ' || wcodigo);
17 dbms_output.put_line('Nome no produto: ' || wnome);
18 end loop;
19 CLOSE C4;
20 end exemploCur4;
```

Por fim, o Exemplo 8 ilustra a criação de um cursor dinâmico, cujo conteúdo de pesquisa pode ser alterado dinamicamente dentro do bloco PL/SQL. Para que um cursor possua qualquer operação (qualquer SELECT), sua declaração deve ser alterada. Em vez de declarar no cursor a consulta desejada, este apenas deverá ser definido como uma referência de um dado do tipo cursor. Para isso, observe que está sendo criado na linha 3 um novo tipo de dado chamado tipoCursor que é uma referência ao tipo de dados Cursor. Em seguida (linha 4), o nosso cursor dinâmico é declarado com o nome de cursorDinamico do tipo de dado tipoCursor. Como não existe um comando para ser executado quando o cursor é aberto, torna-se necessário que, ao abrir o cursor, seja especificado o comando de seleção que o mesmo irá controlar. Para isso, após o comando OPEN cursorDinamico, é especificado a consulta desejada, como pode ser identificado nas linhas 10 e 12.

Esse tipo de cursor é muito utilizado, porém sua implementação deverá ser sempre planejada para que no momento em que recuperarmos uma informação, esta não retorne em uma variável cujo tipo de dado não seja suportado, pois se isso ocorrer o procedimento retornará um erro.

Exemplo 8: - “Exemplo 5 de utilização de cursores Dinâmicos”

```

1  create or replace procedure exemploCur5 (opcao IN integer)
2  As
3  TYPE tipoCursor IS REF CURSOR;
4  cursorDinamico tipoCursor;
5  wcodigo tbproduto.codigo%type;
6  wnome tbproduto.nome%type;
7  westvalor number;
8  begin
9  if (opcao=1) then
10   OPEN cursorDinamico FOR 'SELECT codigo,nome,estoque FROM tbproduto
   ORDER BY nome';
11  Else
12   OPEN cursorDinamico FOR 'SELECT codigo,nome,valor FROM tbproduto
   ORDER BY nome';
13  end if;
14  loop
15   dbms_output.put_line('*****');
16   fetch cursorDinamico into wcodigo, wnome,westvalor;
17   if (not cursorDinamico%found) then
18    exit;
19   end if;
20   dbms_output.put_line('Estou “varrendo” a linha ' || cursorDinamico%rowcount);
21   dbms_output.put_line('Código do produto: ' || wcodigo);
22   dbms_output.put_line('Nome no produto: ' || wnome);
23   if (opcao=1) then
24    dbms_output.put_line('Estoque do produto: ' || westvalor);
25   else
26    dbms_output.put_line('Valor do produto: ' || westvalor);
27   end if;
28  end loop;
29  close cursorDinamico;
30  end exemploCur5;

```

Dependendo da funcionalidade que seja necessário implementar, é comum utilizar mais de um cursor para atingir um determinado objetivo. Em muitos casos, o resultado de um cursor implica na execução de outro cursor. No Exemplo 9, é possível identificar a utilização de dois cursores dentro do mesmo procedimento, onde o resultado de um cursor ("ccat") implica na execução do outro cursor ("cursorDinamico"). Para que o procedimento funcione, a estrutura definida na tabela "tbproduto" implementada no Exemplo 3 teve que ser alterada adicionando mais um atributo "fkcodcat", que referencia como chave estrangeira a tabela "tbcategoria", que também foi implementada, possuindo apenas dois campos, o "pkcodcat" que representa um código de categoria o qual foi definido como chave primária e o campo "nomecat" que armazena o nome de uma categoria.

Após essas alterações, é possível executar o procedimento do Exemplo 9, o qual possui dois cursores, sendo um fixo ("ccat") que lista o código e nome de todas as categoria cadastradas e a quantidade de produtos vinculados a essa categoria. Para cada categoria impressa, o cursor "cursorDinamico" é modificado (linha 20) o qual é responsável por listar o nome de todos os produtos vinculados à categoria impressa.

Exemplo 9: - "Exemplo 6 de utilização de cursores Dinâmicos"

```

1  create or replace procedure exemploCur6
2  As
3  TYPE tipoCursor IS REF CURSOR;
4  cursorDinamico tipoCursor;
5  wnome tbproduto.nome%type;
6  wnomecat tbcategoria.nomecat%type;
7  wcodcat tbcategoria.pkcodcat%type;
8  wquant number;
9  cursor ccat is select c.pkcodcat,c.nomecat,count(p.codigo) from tbcategoria c
10 left join tbproduto p on c.pkcodcat=p.fkcodcat
11 group by c.pkcodcat, c.nomecat order by c.nomecat;
12 begin
13 open ccat;
14 loop
15 dbms_output.put_line('*****');
16 fetch ccat into wcodcat, wnomecat,wquant;
17 exit when ccat%NOTFOUND=true;
18 dbms_output.put_line('Categoria ' || wnomecat);
19 dbms_output.put_line('Quantidade de produtos vinculados ' || wquant);
20 OPEN cursorDinamico FOR 'SELECT nome FROM tbproduto p
    where p.fkcodcat=' || wcodcat || ' ORDER BY nome';
21 Loop
22 fetch cursorDinamico into wnome;
23 exit when cursorDinamico%NOTFOUND=true;
24 dbms_output.put_line('Produto vinculado a categoria ' || wnome);
25 end loop;
26 close cursorDinamico;
27 end loop;
28 close ccat;
29 end exemploCur6;

```

Observe que, para cada cursor, é necessário que ocorra um laço que percorra o mesmo (linhas 14 e 21), com controle

de permanência no laço para cada cursor (linhas 17 e 23). No caso do cursor “cursorDinamico”, observe que, antes de ser executado novamente, ele é fechado (linha 26) para que não ocorra erro ao abri-lo novamente dentro do laço que controla o cursor “ccat”.

Recapitulando

Neste capítulo, foi possível conhecer um pouco sobre o conceito de cursores. Os cursores são indispensáveis quando ocorre a necessidade de retornar mais de um registro como resposta em uma consulta SELECT dentro de um bloco PL/SQL, seja em um procedimento, função ou até em uma trigger. Dessa forma, foram apresentados os principais comandos de controle de um cursor, como, por exemplo, os comandos OPEN, CLOSE e FETCH, os quais foram exemplificados através de 6 exemplos práticos de procedimentos que listavam conteúdo de tabelas como forma de justificar o uso de cursores.

Nesse contexto, foi exemplificado a diferença de cursores simples e cursores dinâmicos, sendo que o simples é especificado no momento da declaração de uma variáveis do tipo cursor, e o dinâmico pode ser definido dentro do Bloco PL/SQL.

Referências

ELMASRI, R. & NAVATHE, S. B. Sistemas de Banco de Dados. Rio de Janeiro: LTC, 2002. 4ª ed

DATE, C. J. Introdução a Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 2003. 9ª ed. .

KORTH, Henry F. e SILBERSCHATZ, Abraham. Sistema de Bancos de Dados. São Paulo: Makron Books, 1999. 3ª edição revisada.

Atividades

Marque V para as assertivas verdadeiras e F para as assertivas falsas:

- 1) () Para extrair dados de um registro que possua um cursor referenciando sua linha, utilizamos o comando FETCH.
- 2) () O comando Close seguido do nome do cursor avança para o próximo registro.
- 3) () Após o comando FETCH, o cursor avança para o último registro do cursor.

- 4) () O comando %ISOPEN precedido do nome do cursor retorna falso se o curso está fechado.
- 5) () Em Oracle, um cursor dinâmico necessita possuir um tipo de dados que seja referência de um dado do tipo CURSOR. Para isso, utiliza-se **TYPE nome_do_tipo_que_faz_referencia IS REF CURSOR**.

Gabarito:

1) V 2) F 3) F 4) V 5) V

Christiano Cadoná¹

Capítulo **7**

Controle de Concorrência

¹ Especialista em Desenvolvimento de Software para Web, professor das disciplinas presenciais de banco de dados I e II dos cursos de computação do Campos ULBRA Canoas.

Introdução

Este capítulo apresenta o conceito e exemplos ilustrativos de como ocorre o controle de concorrência, funcionalidade esta indispensável para qualquer banco de dados que possibilite o compartilhamento de dados. O processo de controle de concorrência garante que os dados possam ser acessados por mais de um usuário no bando de dados.

Controle de Concorrência

Concorrência de banco de dados está relacionado ao fato dos sistemas gerenciadores de banco de dados permitirem que mais de uma transação acesse os dados ao mesmo tempo no banco de dados. Essa funcionalidade é indispensável para qualquer banco de dados que tenha interesse de ser utilizado como fonte de dados que compartilhe informações entre mais de um usuário. Agora, imagine a complexidade e a quantidade de controles que o bando de dados deve possuir para que seja possível implementar essa funcionalidade.

Segundo Date (2003), existem três problemas que o mecanismo de controle de concorrência deve tratar, pois mesmo que a transação seja concluída com sucesso, poderá levar o banco de dados a um estado inconsistente. São eles:

- ➡ O problema da atualização perdida (lost update).
- ➡ O problema da dependência sem COMMIT.

➡ O problema da análise inconsistente.

Como forma de exemplificar os três problemas, vamos considerar que a Figura 1 possui os dados de uma tabela que sofrerão alteração de acordo com cada um dos problemas a serem tratados pelo SGBD relacionados ao controle de concorrência.


Tbproduto				
	pkcodprod	NomeProd	Estoque	valor
	78	Abacaxi	200	2,99
	21	Milho	88	4,99
	585	Banana	40	3,50
	25	Uva	10	6,70

Figura 1 Tabela base para exemplificar os problemas de processos concorrentes

Atualização Perdida (Lost update)

O problema de atualização perdida ocorre quando duas transações que estão sendo executadas de forma concorrente alteram o mesmo registro do banco de dados. Nesse caso, uma transação irá substituir o conteúdo alterado pela outra transação. O Exemplo 1 apresenta um exemplo em alto nível do problema de atualização perdida. Nesse exemplo, estão sendo executadas duas transações de forma concorrente, a transação “A” e a transação “B”. Observe que cada linha de comando é executada em um determinado tempo no SGBD.

No tempo chamado tempo “T1”, a transação “A” busca os dados cadastrados do registro denominado registro de código 21, que no exemplo da Figura 1 faz referência ao produto “Milho” de estoque 88, com valor unitário de 4,99. No tempo “T2”, o SGBD processa uma linha de comando da transação “B” que também busca os mesmos dados que a transação “A” buscou; nesse caso, o produto “Milho” de estoque 88, com valor unitário de 4,99.

Exemplo 1: - “Problema de Atualização Perdida”		
Tempo	Transação “A”	Transação “B”
T1	Busca dados do Registro de código 21	
T2		Busca dados do Registro de código 21
T3	Altera o valor do produto do registro de código 21 para 5.75	
T4		Altera o valor do produto do registro de código 21 para 2,66

Já no tempo “T3”, a transação “A” altera o valor do produto de código 21 para 5,75. Porém, no tempo “T4”, o valor do mesmo produto também foi alterado para 2,66. Nesse caso, dizemos que a atualização da transação “A” foi perdida no tempo “T4”, pois a transação “B” sobrescreve os dados atualizados pela transação “A”. O problema seria ainda maior

se além do valor do produto ambas as transações alterassem todos os dados do registro de código 21, pois a execução do comando de alteração da transação “B” substitui completamente as alterações da alteração da transação “A”.

Dependência sem COMMIT

O problema da dependência sem COMMIT ocorre quando uma transação busca dados de um registro ou atualiza um registro que já foi modificado por outra transação qualquer, mas que essa transação qualquer não tenha realizado um commit. Relembrando do conceito de transação, uma transação pode ser concluída com a confirmação dos comandos executados através do comando COMMIT, ou com o cancelamento dos comandos executados utilizando o comando ROLLBACK.

Observe o Exemplo 2 que também executa de forma concorrente duas transações que utilizam o mesmo registro em suas operações. Nesse exemplo, no tempo “T1”, a transação “A” busca os dados do registro de código 25, que neste caso é o produto “Uva” que possui 10 unidades em estoque e possui como valor de venda 6,70. No tempo “T2”, a mesma transação “A” altera no estoque o produto de código 25 para 7 unidades.

Exemplo 2: - “Problema de dependência sem commit”

Tempo	Transação “A”	Transação “B”
	:	:
T1	Busca dados do Registro de código 25	:
	:	:
T2	Altera o estoque do produto do registro de código 25 para 7 unidades	:
	:	:
T3	:	Busca dados do Registro de código 25
	:	:
T4	ROLLBACK	:
	:	:

No tempo “T3”, a transação “B” busca os dados do mesmo registro alterado na transação “A” (“Uva”) de valor 6,70 e estoque alterado na transação “A” para 7 unidades. Porém, no tempo “T4”, a transação “A” executa um comando ROLLBACK que desfaz todos os comandos que foram executados nessa transação. Sendo assim, o valor de estoque do produto “Uva” volta a ser 10 unidades, mas a transação “B” possui como informação que o produto “Uva” possui 7 unidades em estoque. Assim, a transação “B” possui uma informação inconsistente, pois a informação foi perdida tendo em vista que, para que ela seja uma informação válida, ela dependeria de um COMMIT na transação “A”.

Análise Inconsistente

O problema de análise inconsistente faz referência a transações concorrentes que manipulam e utilizam os dados para um determinado resultado. Vamos considerar que exista duas transações concorrentes “A” e “B”, onde uma das transações (transação “A”) tenha como objetivo, buscar o estoque de cada produto e somá-lo em uma variável chamada total, e a transação “B” altera o valor de estoque de dois dos produtos cadastrados. Dependendo de como estas forem executadas, o resultado poderá ficar inconsistente.

O Exemplo 3 apresenta exatamente um erro causado pelos processos concorrentes de análise de inconsistente. Nesse exemplo, a transação “A” busca o estoque de cada produto e sumariza na variável “TOTAL”, e a transação “B” altera o estoque dos produtos de código 585 (“Banana”) e 21 (“Milho”).

Antes de detalhar o Exemplo 3, vamos considerar, para esse exemplo, que o estoque dos produtos envolvidos está definido como o mesmo da Figura 1, ou seja, o produto “Abacaxi” possui 200 unidades, o produto “Milho” possui 88 unidades, o produto “Banana” possui 40 unidades e, por fim, o produto “Uva” possui 10 unidades.

No tempo “T1”, a transação “A” inicializa uma variável chamada “Total” com valor zero. No tempo “T2”, a mesma transação “A” busca o valor de estoque o produto de código 78 (“Abacaxi”) que é 200 unidades, e acrescenta ao valor já contido na variável “Total”, fazendo com que esta contenha o valor 200. Em seguida, no tempo “T3”, a transação “A” busca o valor de estoque do produto de código 21 (“Milho”) que é

88 unidades, e o acrescenta ao valor já contido na variável "Total" alterando o valor desta para 288 unidades.

No tempo "T4", a transação "B" inicia e busca todos os dados do produto de código 585 ("Banana", estoque 40 unidades, valor de venda 3,50). A mesma transação no tempo "T5", altera do estoque do produto de código 585 para 100 unidades no lugar de 40 unidades. Até este momento, o resultado da variável "Total" da transação "A" não sofreu nenhuma interferência, pois o produto, até o presente tempo, não foi selecionado pela transação. O grande problema ocorre nos tempos "T6" e "T7" que busca os dados e altera o conteúdo de um produto ("Milho") que já foi capturado pela transação "A".

Quando o tempo "T6" é processado, a transação "B" busca todos os dados do produto de código 78 ("Abacaxi", estoque 200 unidades, valor de venda 2,99). No tempo "T7", o estoque do produto "Abacaxi" é alterado para 150 unidades, ou seja, foi retirado 50 unidades do produto. Neste momento, o conteúdo da variável "Total" da transação concorrente "A" possui um valor inconsistente, pois deveria possuir 50 unidades a menos. No tempo "T8", a transação "B" é comitada, ou seja, as alterações são confirmadas de forma definitiva no banco de dados.

Quando o processamento retorna a transação "A" no tempo "T9", a transação busca o estoque do produto de código 585 ("Banana"), que foi alterado pela transação "B" passando a valer 100 unidades e o acrescenta à variável "Total", alterando o valor da mesma para 388. Por fim, no tempo "T10" a transação "A" busca o valor de estoque do produto de código 25 ("Uva") que é 10 unidades e acrescenta à variável "Total", fazendo ela passar a valer 398 unidades.

Exemplo 3: - “Problema de Análise Inconsistente”

Tempo	Transação “A”	Transação “B”
T01	<div> <div>:</div> <div>Total=0</div> <div>Total=0</div> </div>	:
T02	<div> <div>Busca estoque do produto 78 (200 unidades); Total=Total+200;</div> <div>Total=200</div> </div>	:
T03	<div> <div>Busca estoque do produto 21 (88 unidades); Total=Total+88;</div> <div>Total=288;</div> </div>	:
T04	:	Busca dados do produto 585
T05	:	Altera o estoque do produto de código 585 para 100 unidades
T06	:	Busca dados do produto 78
T07	:	Altera o estoque do produto de código 78 para 150 unidades
T08	:	COMMIT
T09	<div> <div>Busca estoque do produto 585 (100 unidades); Total=Total+100;</div> <div>Total=388</div> </div>	:
T10	<div> <div>Busca estoque do produto 25 (10 unidades); Total=Total+10;</div> <div>Total=398</div> </div>	:
	:	:

Erro, pois o conteúdo da variável “Total” deveria ser 348 unidades.

Ou seja, ao final do processamento, o valor da variável “Total” não corresponde à soma do estoque de todos os produtos, o que chamamos de análise inconsistente, pois a mesma deveria conter como resposta 348, tendo em vista que o produto “Abacaxi” foi alterado para menos 50 unidades pela transação “B” e a transação “A” lê o mesmo como possuindo 200 unidades.

Bloqueio

Como alternativa para resolver os problemas apresentados no processo de controle de concorrência, a grande parte dos SGBDs implementa o conceito de bloqueios. Segundo C. J. Date, o conceito de bloqueio é bem simples.

Quando uma transação precisa de uma garantia de que um objeto no qual está interessada – em geral, uma tupla de banco de dados – não mudará de algum modo enquanto ela estiver ativa, a transação adquire um bloqueio sobre esse objeto. O efeito do bloqueio é “impedir que outras transações atuem” sobre o objeto em questão e portanto, em particular, impedir que elas alterem o objeto. Desse modo, a transação A é capaz de executar seu processamento tendo a certeza de que o objeto em questão permanecerá em um estado estável durante o tempo que a transação A desejar. (C. J. Date. Página 402)

Sendo assim, o conceito de bloqueio compreende que sempre que um determinado registro seja utilizado em uma transação o mesmo deverá ser bloqueado para evitar problemas, como os já relatados anteriormente. A maioria dos bancos de dados implementa dois tipos de bloqueio, o **bloqueio exclusivo** e o **bloqueio compartilhado**. Quando um registro possui um bloqueio exclusivo, somente a transação que definiu o registro com esse bloqueio poderá operacionalizar os dados do registro bloqueado (alterar suas informações). Dessa forma, se o mesmo já estiver bloqueado por uma transação qualquer, outra transação que necessitar utilizar o registro não terá acesso às informações do mesmo, ou seja, o acesso será negado.

Já o bloqueio compartilhado compreende permitir que mais de uma transação acesse um mesmo registro, desde que a transação não realize alterações sobre os dados. Nesse caso, se uma determinada transação realizar um bloqueio compartilhado sobre um registro específico e outra transação qualquer necessitar realizar um bloqueio compartilhado sobre a transação, esta será concedida, ou seja, duas transações poderão possuir bloqueio compartilhado sobre o mesmo registro. Agora, caso um determinado registro possua um bloqueio compartilhado em uma transação qualquer, e outra transação necessitar de um bloqueio exclusivo para esse mesmo registro, que já está bloqueado de forma compartilhada, não conseguirá, ou seja, a transação que necessitar modificar um registro que está sendo lido em uma outra transação não conseguirá, pois o acesso ao registro será negado.

O bloqueio compartilhado é utilizado geralmente em leitura de dados (quando realizamos uma busca de informação

no bando de dados) e o bloqueio exclusivo é implementado quando ocorre a modificação de um registro no banco de dados (incluído, alterado ou removido).

Segundo Date, um grande número de SGBD implementa o bloqueio exclusivo e compartilhado utilizando um protocolo de acesso aos dados conhecido como bloqueio escrito em duas fases. Esse protocolo possui as seguintes características (C. J. Date, p. 403):

- Uma transação que deseja ler uma tupla primeiro tem que adquirir um bloqueio compartilhado sobre essa tupla.
- Uma transação que deseja alterar um registro primeiro deve adquirir um bloqueio exclusivo sobre o registro. Como alternativa, se a transação já possuir um bloqueio compartilhado no registro, ou seja, se o registro estiver sendo lido pela transação, o bloqueio passa a ser exclusivo no lugar de compartilhado.
- Os bloqueios exclusivos são mantidos até o fim da transação, ou seja, quando a transação executa um COMMIT ou ROLLBACK. Normalmente, os bloqueios compartilhados também são mantidos até esse momento, mas depende de cada SGBD.
- Se uma requisição de bloqueio da transação B não puder ser atendida imediatamente devido a um conflito com um bloqueio já mantido pela transição A, a transação B entrará em estado de espera. B esperará pelo menos até que bloqueio possa ser concedido, o que não ocorrerá antes que o bloqueio A seja liberado. Natural-

mente, o sistema precisará garantir que não esperará para sempre. Um modo simples de fornecer essa garantia é atender a todas as requisições de bloqueio na ordem “primeiro a chegar, primeiro a ser atendido”.

O Exemplo 4 apresenta uma sequência de passos executados por duas transações concorrentes “A” e “B”, onde o SGBD implementa o protocolo de bloqueio escrito em duas fases. No exemplo, a transação “A” no tempo “T1”, solicita bloqueio compartilhado e o mesmo é concedido para o produto de código 21. No mesmo tempo “T1”, a transação “A” busca os dados do produto de código 21 bloqueado de forma compartilhada. No tempo “T2”, a mesma transação “A” promove o bloqueio compartilhado para bloqueio exclusivo, pois irá modificar o registro. Em seguida, a transação “A” altera o valor do produto de código 21 para 5,75.

No tempo “T3”, a transação “B” é executada onde solicita bloqueio compartilhado para o registro de código 21 que já possui bloqueio exclusivo. Como o registro já possui esse tipo de bloqueio, a transação ficará em estado de espera, não sendo mais executada até que o registro de código 21 seja liberado.

No tempo “T4”, novamente a transação “A” será executada onde a mesma é concluída através do comando COMMIT, que libera todos os registros bloqueados. Assim, no tempo “T5”, a transação que estava em estado de espera é liberada e a mesma pode criar um bloqueio compartilhado no registro de código 21.

Exemplo 4: - “Exemplo de bloqueio de duas faces”

Tempo	Transação “A”	Transação “B”
	:	:
T1	<ul style="list-style-type: none"> Solicita Bloqueio Compartilhado no registro de código 21; Cria Bloqueio Compartilhado no registro de código 21 Busca dados do Registro de código 21 	:
	:	:
T2	<ul style="list-style-type: none"> Migra para bloqueio Exclusivo o registro de código 21 Altera o valor do produto do registro de código 21 para 5.75 	:
	:	:
T3	:	Solicita Bloqueio Compartilhado no registro de código 21;
	:	Entra do estado de espera
	:	Espera
	:	Espera
	:	Espera
T4	COMMIT Desbloqueia o registro de código 21	Espera
	:	Espera
T5	:	<ul style="list-style-type: none"> Sai do estado de espera Cria Bloqueio Compartilhado no registro de código 21 Busca dados do Registro de código 21
	:	:

A grande diferença é que no tempo “T5”, quando o registro é recuperado pela transação “B”, o mesmo já terá o valor do produto alterado para 5,75, o que resolve o problema de inconsistência.

Impasses (DEADLOCK)

Porém o bloqueio de duas faces também pode ocasionar eventualmente um grande problema em duas ou mais transações concorrentes. Como já foi apresentado, quando um registro está bloqueado de forma exclusiva por uma transação “A”, e outra transação “B” deseja utilizá-la, esta fica em estado de espera. Agora, imagine se, antes dessa transação “B”, entrar em estado de espera, esta bloqueou um outro registro qualquer também de modo exclusivo. Se a transação “A” requisitar o registro bloqueado pela transação “B”, a transação “A” ficará também bloqueada, ou seja, ambas as transações estão aguardando que a outra libere o registro bloqueado para prosseguir, o que fará com que ambas fiquem em estado de espera eterno. Quando isso ocorre, dizemos que as transações estão em um impasse ou, como são conhecidas, estão em DEADLOCK.

Observe o Exemplo 5 que ilustra o processamento de duas transações “A” e “B” até que estas entrem em impasse em sua execução.

Exemplo 5: - “Exemplo de impasse (DEADLOCK)”

Tempo	Transação “A”	Transação “B”
	:	:
T1	<ul style="list-style-type: none"> Solicita Bloqueio Exclusivo no registro de código 21; Cria Bloqueio Compartilhado no registro de código 21 Altera o valor do produto do registro de código 21 para 5,75 	:
	:	:
T2	:	<ul style="list-style-type: none"> Solicita Bloqueio Exclusivo no registro de código 78; Cria Bloqueio Compartilhado no registro de código 78 Altera o valor do produto do registro de código 78 para 8,99
	:	:
T3	Solicita Bloqueio Compartilhado no registro de código 78; Entra do estado de espera	:
	Espera	:
	Espera	:
	Espera	:
T4	Espera	Solicita Bloqueio Compartilhado no registro de código 21;
	Espera	Entra do estado de espera
	Espera	Espera
	Espera	Espera
	Espera	Espera

Nesse exemplo, a transação “A” no tempo “T1” solicita e atribui bloqueio exclusivo no registro de código 21. Em seguida, altera o valor do produto para 5,75. No tempo “T2”, o processamento passa para a transação “B” que solicita e executa bloqueio no registro de código 78. Nesse mesmo tempo “T2”, a transação “B” altera o valor do produto de código 78

para 8,99. No tempo “T3”, a transação “A” solicita bloqueio exclusivo no registro de código 78 que possui bloqueio exclusivo pela transação “B”. O bloqueio é negado e a transação “A” entra em estado de espera até que o registro seja liberado. No tempo “T4”, a transação “B” solicita bloqueio exclusivo no registro de código 21, mas não consegue permissão, pois o mesmo possui bloqueio exclusivo pela transação “A”. Dessa forma, a transação “B” também entra em estado de espera, criando uma situação de impasse, onde uma transação aguarda que outra transação libere o registro para prosseguir.

Quando ocorrer um impasse, o ideal é que o SGBD o detecte e o interrompa. Interromper o impasse envolve escolher uma das transações envolvidas e a desfazer, ou seja, realizar um rollblack na transação. A escolha de qual transação deve ser desfeita depende de cada SGBD. Alguns inclusive desfazer todas as transações envolvidas. Alguns SGBD assim que selecionam uma transação vítima e a desfazem executam a mesma novamente em uma tentativa de manter o processo o mais automatizado possível, e não causando erro ao usuário do sistema.

Recapitulando

Este capítulo tratou de um dos recursos mais interessantes de um SGBD que é a possibilidade de utilizar transações concorrentes a um mesmo registro no banco de dados. Aprendemos que o processo concorrente pode ocasionar pelo menos três tipos de inconsistência, a atualização perdida, a depen-

dência sem COMMIT e a análise inconsistente. Após verificar de forma prática cada um dos problemas, foi apresentada e exemplificada como solução o conceito de bloqueio. A grande parte dos bancos de dados implementa dois tipos de bloqueio, o bloqueio exclusivo e o bloqueio compartilhado. A implementação de bloqueio resolve os três problemas de inconsistência, porém cria outros como é o caso dos impasses. Geralmente, os impasses são solucionados selecionando uma transação e desfazendo-a.

Referências

ELMASRI, R. & NAVATHE, S. B. **Sistemas de Banco de Dados**. Rio de Janeiro: LTC, 2002. 4. ed.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro: Campus, 2003. 9. ed.

KORTH, Henry F. e SILBERSCHATZ, Abraham. **Sistema de Bancos de Dados**. São Paulo: Makron Books, 1999. 3. edição revisada.

Atividades

Marque V para as assertivas verdadeiras e F para as assertivas falsas:

- 1) () Dizemos que possuímos um problema de atualização perdida quando uma transação "A" qualquer altera dados de um registro que está bloqueado por uma transação "B".
- 2) () O problema de Dependência sem Commit ocorre quando uma transação "A" busca dados de um registro que foi alterado por outra transação "B" concorrente, mas ainda não recebeu commit pela transação "B". Nesse caso, corre-se o risco da transação que alterou o registro (transação "B") ser desfeita, o que torna o conteúdo retornado para transação "A" inconsistente.
- 3) () A grande parte dos SGBD implementa dois tipos de bloqueios, o bloqueio exclusivo e o bloqueio inconsistente.
- 4) () Quando um registro estiver bloqueado de forma exclusiva e outra transação que não seja a que o bloqueou deseja realizar um novo bloqueio sobre esse registro, este será negado, e a transação ficará em estado de espera.
- 5) () Quando uma transação "A" está bloqueada aguardando que a transação "B" o libere, e a transação "B" está em estado de espera aguardando que a transação "A" a libere, identificamos o problema de DEADLOCK.

Gabarito:

1) F 2) V 3) F 4) V 5) V

Edemar Costa Oliveira¹

Capítulo 8

Segurança em SGBD

¹ Mestre em Engenharia, Pós-graduado em Tecnologias para Negócios na Internet, Bacharel em Matemática Computacional. Analista de Sistemas e Professor na Ulbra.

Introdução

Apresentamos neste capítulo uma visão geral sobre segurança em Sistema de Gerenciamento de Bancos de Dados (SGBD). Nosso enfoque baseia-se na arquitetura de bancos de dados relacionais.

Devido à abrangência, limitações de escopo e cada sistema apresentar características próprias de segurança, não podemos abordar a segurança em todas as plataformas existentes. Devido a isso, vamos abordar a segurança em relação a um dos líderes de mercado: o ORACLE.

O objetivo deste capítulo é apresentar um padrão de gerenciamento de contas de usuários e determinação de limites de acesso aos objetos de um banco de dados. Vamos abordar desde a criação, alteração, exclusão, configuração e acesso a dados de usuários na plataforma Oracle.

Não pretendemos esgotar o assunto, mas somente mostrar um formato que difere em pouco das muitas plataformas relacionais e passar uma base de conhecimento a respeito desse tópico.

1 A segurança no Oracle

Cada banco de dados Oracle tem uma lista de usuários válidos. Para acessar um banco de dados, um usuário deve executar um aplicativo de banco de dados e se conectar à ins-

tância de banco de dados usando um nome de usuário válido definido no banco de dados.

Um banco de dados Oracle permite que você configure a segurança para os usuários de diversas formas.

Quando você criar uma conta de usuário, você pode especificar limites para a mesma. Você também pode definir limites na quantidade de vários recursos de sistema disponíveis para cada usuário como parte do domínio de segurança desse usuário.

Um banco de dados Oracle também fornece um conjunto de visões de banco de dados que você pode consultar para encontrar informações de sessão e informações como recurso.

Este capítulo também descreve os perfis. Um perfil é a coleção de atributos que se aplicam a um usuário. Ele permite um único ponto de referência para qualquer um dos vários usuários que compartilham esses atributos.

2 Criações de novas contas de usuário

2.1 Criando a nova conta de usuário

Você pode criar um usuário para um banco de dados Oracle com o comando `CREATE USER`. Para criar um usuário, você deve ter o privilégio de `CREATE USER system`. Este é um poderoso privilégio, um administrador de banco de dados ou administrador de segurança geralmente é o único usuário que tem o privilégio de sistema `CREATE USER`.

O exemplo abaixo cria um usuário e especifica a senha. Ele também concede ao usuário o privilégio mínimo, `CREATE SESSION`, para entrar para uma sessão de banco de dados.

Exemplo: Criando uma conta de usuário com o privilégio de `CREATE SESSION`

```
CREATE USER Ulbra  
IDENTIFIED BY password  
GRANT CREATE SESSION TO Ulbra;
```

Substitua *password* por uma senha que seja segura. Um usuário recém-criado não pode se conectar ao banco de dados até que você conceda ao mesmo os privilégios `CREATE SESSION`. Então, imediatamente depois de criar a conta de usuário, use a instrução de `GRANT` para conceder ao usuário esses privilégios. Se o usuário deve acessar o Oracle Enterprise Manager, você deve também dar ao usuário o privilégio de `SELECT ANY DICTIONARY`.

Como um administrador de segurança, você deve criar suas próprias funções e atribuir somente os privilégios que são necessários.

2.2 Especificando um nome de usuário

Dentro de cada banco de dados, um nome de usuário deve ser exclusivo em relação a outros nomes de usuário e funções. Um perfil e o usuário não podem ter o mesmo nome. O nome é armazenado em letras maiúsculas.

No entanto, se você colocar o nome de usuário entre aspas duplas no momento de criar, então o nome é armazenado usando as maiúsculas e minúsculas que você usou para o nome (*Case sensitive*). Por exemplo:

```
CREATE USER "Ulbra" IDENTIFIED BY password;
```

Então, quando você consulta a exibição de informações dos usuários utilizando o comando `ALL_USERS`, você encontrará que a conta de usuário é armazenada usando o caso que você usou para criá-lo.

```
SELECT USERNAME FROM ALL_USERS;
```

```
USERNAME
```

```
-----
```

```
Ulbra
```

Usuário `ULBRA` e usuário `ulbra` são ambos armazenados no banco de dados como contas de usuário separadas. Mais adiante, deve-se modificar ou deixar o usuário que você criou usando aspas duplas, então coloque o nome de usuário entre aspas duplas.

Por exemplo:

```
DROP USER "ulbra";
```

2.3 Atribuindo ao usuário uma senha

O novo usuário deve ser autenticado usando o banco de dados. Nesse caso, o usuário deve fornecer a senha correta para

o banco de dados para se conectar com êxito. Para especificar uma senha para o usuário, use a cláusula `IDENTIFIED BY` identificado na instrução `CREATE USER`:

```
CREATE USER ulbra  
  
IDENTIFIED BY password
```

2.4 Especificações de um perfil para o usuário

Quando você criar um usuário, você pode especificar um perfil. Um perfil é um conjunto de limites de recursos de banco de dados e a senha de acesso ao banco de dados. Então, se você não especificar um perfil, o banco de dados Oracle atribui ao usuário um perfil padrão.

O exemplo a seguir demonstra como atribuir um usuário a um perfil.

```
CREATE USER ulbra  
  
IDENTIFIED BY password  
  
DEFAULT TABLESPACE data_ts  
  
QUOTA 100M ON test_ts  
  
QUOTA 500K ON data_ts  
  
TEMPORARY TABLESPACE temp_ts  
  
PROFILE professor
```

3 Alterando uma conta de usuário

Os usuários podem alterar suas próprias senhas. No entanto, para alterar qualquer outra opção de um domínio de segurança de usuário, você deve ter o privilégio ALTER USER.

Você pode alterar as configurações de segurança do usuário com a instrução ALTER USER. Alterando as configurações de segurança de usuário, afeta o usuário nas futuras sessões. A sessão atual não é afetada.

O exemplo abaixo mostra como usar o comando ALTER USER para alterar as configurações de segurança para o usuário ulbra:

```
ALTER USER ulbra  
IDENTIFIED EXTERNALLY  
DEFAULT  
TABLESPACE data_ts  
TEMPORARY  
TABLESPACE temp_ts  
QUOTA 100M ON data_ts  
QUOTA 0 ON test_ts  
PROFILE professor;
```

No exemplo anterior, o comando ALTER USER altera as configurações de segurança para o usuário ulbra como segue:

- Autenticação é alterada para usar a conta de sistema operacional do usuário.

O padrão e tablespaces temporários são explicitamente definidos para o usuário ulbra.

O usuário ulbra recebe uma cota de 100M para a DATA_TS tablespace.

A cota sobre a tabela test_ts é revogada para o usuário ulbra.

O usuário ulbra recebe perfil de professor.

Os usuários podem alterar suas próprias senhas com o comando PASSWORD, como segue:

```
PASSWORD ulbra
```

```
Alterando a senha para ulbra
```

```
Nova senha: senha
```

```
Redigite a nova senha: senha
```

Nenhum privilégio especial (exceto para conectar ao banco de dados e criar uma sessão) é necessário para um usuário alterar sua própria senha. Incentive os usuários a alterar suas senhas com frequência. “Diretrizes para proteger senhas” oferecem conselhos sobre as melhores maneiras de proteger senhas. Você pode encontrar os usuários existentes para a instância atual do banco de dados consultando com o comando `SELECT USERNAME FROM ALL_USERS`.

Os usuários também podem usar a instrução ALTER USER para alterar suas senhas. Por exemplo:

```
ALTER USER ulbra IDENTIFIED BY password
```

No entanto, para maior segurança, use o comando PASSWORD para alterar a senha da conta.

O ALTER USER exibe a senha nova na tela, onde pode ser visto por quaisquer colegas de trabalho excessivamente curiosos. O comando PASSWORD não exibe a senha nova, assim que é conhecido somente para você, não para seus colegas de trabalho. Em ambos os casos, a senha é criptografada na rede.

Os usuários devem ter o privilégio PASSWORD e ALTER USER para alternar entre os métodos de autenticação. Geralmente, somente um administrador tem esse privilégio.

Se você precisar alterar a senha do usuário SYS, em seguida, você deve usar o utilitário de linha comando ORAPWD para criar um novo arquivo de senha que contém a senha que você deseja usar.

Não use o comando ALTER USER ou o comando PASSWORD para alterar a senha do usuário SYS.

Observe o seguinte:

A conta SYS é usada pela maioria dos comandos SQL internos. Portanto, se você tentar usar o ALTER USER para alterar essa senha, enquanto o banco de dados estiver aberto, então há uma chance de ocorrer um deadlocks.

Se você tentar usar `ALTER USER` para alterar a senha do usuário de sistema `SYS`, e se o parâmetro de inicialização de instância `REMOTE_LOGIN_PASSWORDFILE` foi definido como `SHARED`, então você não pode mudar a senha de `SYS`. O comando `USER ALTER` falhará.

O exemplo, abaixo, mostra como usar o comando `ORAPWD` para criar um arquivo de senha que tem a nova senha do usuário de sistema `SYS`. Nesse exemplo, a nova senha vai ser armazenada no arquivo de senhas chamado `useroracle`. Se o arquivo já existir, será apresentada a seguinte mensagem de erro: "OPW-00005: File with same name exists - please delete or rename", Então escolha outro nome.

Exemplo: Usando `ORAPWD` para alterar a senha do usuário de sistema `SYS`

```
orapwd file= 'useroracle'
```

```
Enter password for SYS: new_password
```

4 Configurando limites de recursos do usuário

Você pode definir limites na quantidade de recursos de sistema disponíveis para cada usuário. Ao fazer isso, você pode impedir o consumo descontrolado de recursos valiosos do sistema, tais como tempo de CPU. Para definir os limites de recursos, você usa o Gerenciador de recursos de banco de dados, que é descrita no guia do administrador do banco de dados Oracle.

Esse expediente de limite de atividades é muito útil em sistemas multiusuários, grandes, onde os recursos do sistema são muito caros. Consumo excessivo desses recursos por um ou mais usuários prejudicialmente pode afetar os outros usuários do banco de dados. Em sistemas de banco de dados monousoário ou em pequena escala, o uso de recursos de sistema não é tão preocupante, pois diminui a probabilidade de um impacto negativo.

Você pode gerenciar os limites de recursos de usuário usando o Gerenciador de recursos de banco de dados. Você pode definir preferências de gerenciamento de senha usando perfis. Defina individualmente ou usando um perfil padrão para muitos usuários. Cada banco de dados Oracle pode ter um número ilimitado de perfis. Banco de dados Oracle permite ao administrador de segurança habilitar ou desabilitar a imposição de limites de recursos de perfil universalmente.

4.1 Limitar o nível de sessão de usuário

Cada vez que um usuário se conecta a um banco de dados, uma sessão é criada. Cada sessão usa o tempo de CPU e memória no computador que executa o banco de dados Oracle. Você pode definir vários limites de recursos no nível da sessão.

Se um usuário excede um limite de recurso de nível de sessão, a instrução atual retorna uma mensagem indicando que o limite de sessão foi atingido. Nesse caso, todas as declarações anteriores na transação atual estarão intactas, e as únicas operações que o usuário poderá executar são COMMIT, ROLLBACK, ou desligar (nesse caso, a transação

atual é confirmada). Todas as outras operações produzem um erro. Mesmo depois que a transação é confirmada ou revertida, o usuário não pode realizar qualquer trabalho durante a sessão atual.

4.2 Limitar os níveis de chamada de banco de dados

Cada vez que um usuário executa uma instrução SQL, o banco de dados Oracle realiza diversas etapas para processar a instrução. Durante essa transformação, várias chamadas são feitas para o banco de dados como uma parte das fases diferentes da execução. Para evitar qualquer uma chamada usando recursos excessivamente, banco de dados Oracle permite que você defina vários limites de recursos no nível de chamada.

Se um usuário exceder um limite de recurso de nível de chamada, então banco de dados Oracle interrompe o processamento da instrução, a instrução e retornará um erro. No entanto, todas as declarações anteriores da transação atual permanecem intactas, e a sessão do usuário permanece conectada.

4.3 Limitar o tempo de CPU

Quando instruções SQL e outros tipos de chamadas são feitos para o banco de dados Oracle, uma certa quantidade de tempo de CPU é necessária para processar a chamada. Chamadas de média complexidade exigem uma pequena quantidade de tempo de CPU. No entanto, uma instrução SQL que envolve uma grande quantidade de dados ou uma consulta

em fuga potencialmente pode usar uma grande quantidade de tempo de CPU, reduzindo o tempo de CPU disponível para outro processamento.

Para evitar a utilização descontrolada de tempo de CPU, você pode definir limites de fixos ou dinâmicos no tempo da CPU para cada chamada e a quantidade total de tempo de CPU usado para chamadas de banco de dados Oracle durante uma sessão. Os limites são definidos e medidos em CPU centésimo de segundo (0,01 segundos) usados por uma chamada ou uma sessão.

5 Exclusão de contas de usuário

Quando você exclui uma conta de usuário, o banco de dados Oracle remove a conta de usuário e o esquema associado do dicionário de dados. Também imediatamente são removidos todos os objetos de esquema contidos no esquema do usuário, se houver.

Um usuário que está conectado a um banco de dados não pode ser excluído. Para remover um usuário conectado, você primeiro deve encerrar as sessões de usuário usando a instrução SQL `ALTER SYSTEM` com a cláusula `SESSION` de encerrar. Você pode encontrar a sessão ID (SID) consultando a view interna `V$SESSION`.

Exemplo: Como usar `V$SESSION` para mostrar informações do usuário ulbra.

```
SELECT SID, SERIAL#, USERNAME FROM V$SESSION;
```

SID	SERIAL#	USERNAME
-----	---------	----------

127	55234	ULBRA
-----	-------	-------

Para remover a sessão do usuário, você deve usar o seguinte comando:

```
ALTER SYSTEM KILL SESSION '127, 55234';
```

Você pode excluir um usuário de um banco de dados usando o comando `USER`. Para descartar um usuário e todos os esquemas e objetos de usuário (se houver), você deve ter o privilégio `DROP USER`. Porque o privilégio `DROP USER` é poderoso, um administrador de segurança normalmente é o único usuário que tem esse privilégio.

Se o esquema do usuário contém quaisquer objetos dependentes, você deve usar a opção `CASCADE` para descartar o usuário e todos os objetos associados, inclusive chaves estrangeiras. Se você não especificar `CASCADE` e o esquema de usuário contém objetos dependentes, então uma mensagem de erro é retornada e o usuário não é Descartado.

Antes de descartar um usuário cujo esquema contém objetos, investigue quais objetos que contém dependência e as implicações de excluí-los. Você pode encontrar os objetos pertencentes a um determinado usuário consultando a View `DBA_OBJECTS`.

Exemplo: Encontrar objetos dependentes do usuário ULbra.

```
SELECT OWNER, OBJECT_NAME FROM DBA_OBJECTS  
WHERE OWNER LIKE 'ULBRA';
```

O exemplo a seguir exclui do usuário ULBRA todos os objetos associados e chaves estrangeiras que dependem as tabelas pertencentes a ULBRA.

```
DROP USER ULBRA CASCADE;
```

Recapitulando

Cada banco de dados Oracle tem uma lista de usuários válidos. Para acessar um banco de dados, um usuário deve executar um aplicativo de banco de dados e se conectar à instância de banco de dados usando um nome de usuário válido definido no banco de dados.

Você pode criar um usuário para um banco de dados Oracle com o comando `CREATE USER`. Para criar um usuário, você deve ter o privilégio de `CREATE USER system`.

Os usuários podem alterar suas próprias senhas. No entanto, para alterar qualquer outra opção de um domínio de segurança de usuário, você deve ter o privilégio `ALTER USER`.

Se você precisar alterar a senha do usuário `SYS`, em seguida, você deve usar o utilitário de linha comando `ORAPWD`

para criar um novo arquivo de senha que contém a senha que você deseja usar.

Não use o comando `ALTER USER` ou o comando `PASSWORD` para alterar a senha do usuário SYS.

Você pode gerenciar os limites de recursos de usuário usando o Gerenciador de recursos de banco de dados. Você pode definir preferências de gerenciamento de senha usando perfis. Defina individualmente ou usando um perfil padrão para muitos usuários.

Antes de descartar um usuário cujo esquema contém objetos, investigue quais objetos que contém dependência e as implicações de excluí-los.

Referências

ORACLE CORPORATION. **IDC: Database Security**. Disponível em: <<http://www.oracle.com/technetwork/database/security/index.html>>.

ELMASRI, R. & NAVATHE, S. B. **Sistemas de Banco de Dados**. Rio de Janeiro: Person, 2005. 4. ed.

MACHADO, Felipe N. R. **Banco de Dados: Projeto e Implementação**. São Paulo: Érica, 2004. 1. ed.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro: Campus, 2000. 7. ed.

Atividades

A partir dos estudos desenvolvidos nesse capítulo, Assinale (V) para as assertivas Verdadeiras e (F) para as Falsas.

- a) () Se você colocar o nome de usuário entre aspas duplas no momento de criar, então o nome é armazenado usando a maiúsculas e minúsculas que você usou para o nome (Case sensitive).
- b) () Recomenda-se usar o comando ALTER USER para alterar a senha do usuário SYS.
- c) () Você pode excluir um usuário de um banco de dados usando o comando USER. Para descartar um usuário e todo os esquemas e objetos de usuário (se houver), você deve ter o privilégio DROP USER.
- d) () Se o esquema do usuário contém quaisquer objetos dependentes, você deve usar a opção CASCADE para descartar o usuário e todos os objetos associados, inclusive chaves estrangeiras.
- e) () Em um banco de dados Oracle, cada usuário deve ter seu próprio perfil.

Gabarito:

- a) V b) F c) V d) V e) F

Edemar Costa Oliveira¹

Capítulo 9

Arquiteturas de Bancos de Dados: Centralizado, Cliente-servidor, Distribuído e Paralelo

¹ Mestre em Engenharia, Pós-graduado em Tecnologias para Negócios na Internet, Bacharel em Matemática Computacional. Analista de Sistemas e Professor na Ulbra.

Introdução

Apresentamos neste capítulo algumas arquiteturas em acesso a dados dos Sistemas de Gerenciamento de Bancos de Dados (SGBD's). Desde os tempos de uso exclusivo de mainframes até os sistemas baseados em microcomputadores. A evolução e tendências, muitas vezes, segerem a disponibilidade de tecnologia, situação econômica e outros fatores.

As arquiteturas dos SGBDs, em geral, têm seguido as mesmas tendências de arquiteturas de sistemas de computadores. As primeiras arquiteturas utilizavam os grandes computadores centrais (*mainframes*) para processar todas as funções do sistema, incluindo os programas de aplicação e os de interface com os usuários, bem como todas as funcionalidades do SGBD.

Com a evolução das tecnologias, entrada de novos componentes e queda dos preços, várias empresas mudaram seus terminais para os computadores pessoais (PCs) e estações de trabalho (*workstations*). Primeiro, os sistemas de banco de dados usavam esses computadores da mesma maneira que os terminais, então o SGBD ainda era um SGBD centralizado, no qual as funcionalidades, execuções de programas e processamento das interfaces com o usuário eram executados em uma única máquina. Gradualmente, os sistemas SGBD começaram a explorar o poder de processamento disponível do lado do usuário, que direcionou para as arquiteturas SGBD cliente-servidor.

Iremos mostrar exemplos de cada uma dessas arquiteturas e o paradigma de acessos a dados diferenciado em cada uma delas.

1 As arquiteturas

O primeiro modelo de arquitetura foi a centralizada devido à disponibilidade de tecnologia. Os computadores eram mainframes que centralizavam as atividades não só de armazenamento de dados como das aplicações disponíveis. Eram os computadores propriamente ditos e todos os usuários conectados ao computador central faziam através de terminais. Esses terminais não possuíam poder de processamento, apenas capacidade de visualização. Todos os processamentos eram feitos remotamente, apenas as informações a serem visualizadas e os controles eram enviados do *mainframe* para os terminais de visualização, conectados a ele por redes de comunicação.

Com a evolução das tecnologias, entrada de novos componentes e queda dos preços, várias empresas mudaram seus terminais para os computadores pessoais (PCs) e estações de trabalho (*workstations*). Primeiro, os sistemas de banco de dados usavam esses computadores da mesma maneira que os terminais, então o SGBD ainda era um SGBD centralizado, no qual as funcionalidades, execuções de programas e processamento das interfaces com o usuário eram executados em uma única máquina. Gradualmente, os sistemas SGBD começaram a explorar o poder de processamento disponível do lado do usuário, que direcionou para as arquiteturas SGBD cliente-servidor (VASKEVITCH, 2008).

A arquitetura cliente-servidor foi desenvolvida para dividir ambientes de computação onde um grande número de PCs, estações de trabalho, servidores de arquivos, impressoras, servidores de banco de dados e outros equipamentos são co-

nectados juntos por uma rede. A ideia é definir servidores especializados, tais como servidor de arquivos, que mantém os arquivos de máquinas clientes, ou servidores de impressão que podem estar conectados a várias impressoras; assim, quando se desejar imprimir algo, todas as requisições de impressão são enviadas a esse servidor. As máquinas clientes disponibilizam para o usuário as interfaces apropriadas para utilizar esses servidores, bem como poder de processamento para executar aplicações locais.

Esta arquitetura se tornou muito popular por algumas razões: primeiro, a facilidade de implementação dada a clara separação das funcionalidades e dos servidores. Segundo, um servidor é inteligentemente utilizado porque as tarefas mais simples são delegadas às máquinas clientes mais baratas. Terceiro, o usuário pode executar uma interface gráfica que lhe é familiar, ao invés de usar a interface do servidor. Dessa maneira, a arquitetura cliente-servidor foi incorporada aos SGBDs comerciais. Diferentes técnicas foram propostas para se implementar essa arquitetura, sendo que a mais adotada pelos Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDRs) comerciais é a inclusão da funcionalidade de um SGBD centralizado no lado do servidor. As consultas e a funcionalidade transacional permanecem no servidor, sendo que este é chamado de servidor de consulta ou servidor de transação. É assim que um servidor SQL é fornecido aos clientes. Cada cliente tem que formular suas consultas SQL, prover a interface do usuário e as funções de interface usando uma linguagem de programação (BATTISTI, 2010).

As arquiteturas atuais de bancos de dados dividem-se em várias tendências e que se justificam pelas diferentes necessidades de armazenamento e distribuição dos dados.

2 Centralizado

O primeiro modelo computacional utilizado exclusivamente durante anos foi o chamado sistema centralizado. Nesse modelo, vários “terminais” compostos apenas pelo monitor e o teclado e sem poder de processamento ficam conectados a um grande computador “mainframe” que possui a capacidade de processar e armazenar os dados. Esse tipo de arquitetura tem como característica a necessidade de grandes investimentos tanto na aquisição de equipamentos como na manutenção dos mesmos.



Figura 1 Modelo centralizado (ELMASRI, R. & NAVATHE, 2005).

Considerando a utilização de um sistema de gerenciamento de banco de dados, ele e todos os seus aplicativos são hospedados no computador central e são acessados pelos terminais “burros” que estão conectados a esse computador central onde todo o processamento dos dados é realizado.

De acordo com MACHADO (2004), as principais vantagens e desvantagens dessa arquitetura podem ser resumidas da seguinte forma:

Principais Vantagens:

Um único host fornece alto grau de segurança, concorrência e controle de cópias de segurança e recuperação.

Não há necessidade de um diretório distribuído, já que todos os dados estão localizados em um único host.

Não existe a necessidade de junções distribuídas, já que todos os dados estão em um único host.

Principais desvantagens:

Todos os acessos aos dados realizados por outro que não seja o host onde o banco de dados está, gera alto custo de comunicação.

O host em que o banco de dados está localizado pode criar um “gargalo”, dependendo da quantidade de acessos simultâneos.

Podem acontecer problemas de disponibilidade dos dados, se o host onde os dados estão armazenados sair do ar.

3 Cliente-servidor

Com o advento das redes de computadores, surgiu a necessidade de compartilhar recursos, principalmente com o avanço dos microcomputadores. A partir daí o conceito de rede cliente/servidor ganhou força. A necessidade do compartilhamento de recursos impulsionou para que tivéssemos mais e mais servidores disponibilizando serviços a várias estações clientes.

Segundo BATTISTI, (2001), a arquitetura cliente/servidor É uma arquitetura onde o processamento da informação é dividido em módulos ou processos distintos. Um processo é responsável pela manutenção da informação (Servidor) e o outro é responsável pela obtenção dos dados (Cliente)".

Outro conceito seria de que é uma abordagem da computação que separa os processos em plataformas independentes que interagem, permitindo que os recursos sejam compartilhados enquanto se obtém o máximo de benefício de cada dispositivo diferente, ou seja, Cliente/Servidor é um modelo lógico". VASKEVITCH, (2008).

Arquitetura desenvolvida para trabalhar com ambientes computacionais, nos quais um grande número de PCs, estações de trabalho, servidores de arquivo, impressoras, servidores de banco de dados, servidores Web e outros equipamentos estão conectados via rede. A ideia é definir os servidores especializados com funcionalidades específicas. Por exemplo, é possível conectar PCs ou pequenas estações de trabalho como clientes a um servidor de arquivos que mantém os arquivos das máquinas clientes.

A Figura 2 ilustra a arquitetura de cliente/servidor no nível lógico, e a Figura 3 é um diagrama simplificado que mostra a arquitetura física. Algumas máquinas poderiam ser apenas sites clientes (por exemplo, estações de trabalho sem disco, estações de trabalho ou estações de trabalho/PCs com discos que tenham somente um *software* cliente instalado). Outras poderiam ser servidores dedicados. Outras máquinas poderiam, ainda, ter as funcionalidades de cliente e de servidor.

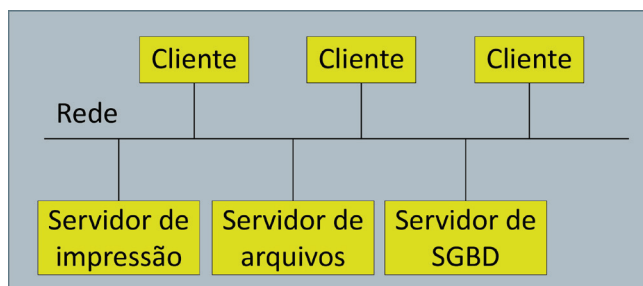


FIGURA 2 A arquitetura lógica de duas camadas cliente/servidor (DATE, 2000).

Um **cliente**, nessa estrutura, é em geral uma máquina de usuário que tem as funcionalidades de interface com o usuário e processamento local. Quando um cliente precisa de uma funcionalidade adicional, como acesso ao banco de dados, inexistente naquela máquina, ela se conecta a um servidor que disponibiliza a funcionalidade. Um **servidor** é uma máquina que pode fornecer serviços para as máquinas clientes, como acesso a arquivos, impressão, arquivamento ou acesso a um banco de dados. Em geral, algumas máquinas instalam apenas o *software* cliente; outras, somente o *software* servidor; e algumas podem incluir ambos, como pode ser visto na Figura 3. Entretanto, é mais comum que os *softwares* cliente e servidor

normalmente sejam executados em máquinas separadas. Dois tipos principais de arquiteturas de SGBD foram criados utilizando-se os fundamentos da estrutura cliente/servidor: duas e três camadas (DATE, 2000).

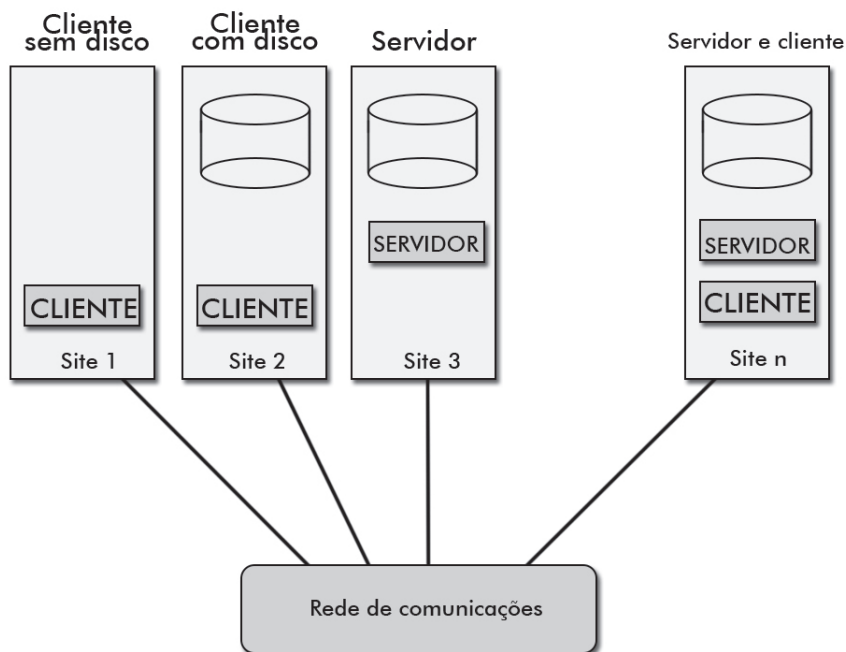


FIGURA 3 Arquitetura física cliente/servidor de duas camadas (DATE, 2000).

3.1 Arquiteturas Cliente/Servidor de Três Camadas para Aplicações Web

Muitas aplicações para a Web usam uma arquitetura chamada arquitetura de três camadas, que possui uma camada interme-

diária entre o cliente e o servidor de banco de dados, como ilustrado na Figura 4. Essa camada intermediária, ou **camada do meio**, é, algumas vezes, chamada **servidor de aplicações** ou **servidor Web**, dependendo da aplicação. Esse servidor desempenha um papel intermediário armazenando as regras de negócio (procedimentos ou restrições) que são usadas para acessar os dados do servidor de banco de dados. Também pode incrementar a segurança do banco de dados checando as credenciais do cliente antes de enviar uma solicitação ao servidor de banco de dados. Os clientes possuem interfaces GUI e algumas regras de negócio adicionais específicas para a aplicação. O servidor intermediário aceita as solicitações do cliente, processa-as e envia comandos de banco de dados ao servidor de banco de dados e então atua como um conduíte por passar (parcialmente) os dados processados do servidor de banco de dados para os clientes – dados que podem ser processados novamente e filtrados para a apresentação aos usuários em um formato GUI. Desse modo, a *interface com o usuário*, as *regras de aplicação* e o *acesso aos dados* atuam como três camadas (ELMASRI, R. & NAVATHE, 2005).

Os avanços na tecnologia de criptografia tornam mais segura a transferência dos dados sensíveis, criptografados, do servidor para o cliente, os quais serão decriptografados. A decriptografia pode ser feita por *hardware* ou por *software* avançados. Essa tecnologia oferece níveis elevados de segurança dos dados, mas as questões relativas à segurança da rede continuam preocupando os especialistas. Várias tecnologias para a compressão dos dados também estão auxiliando na transferência de grandes quantidades de dados dos servidores para os clientes nas redes com ou sem fios.

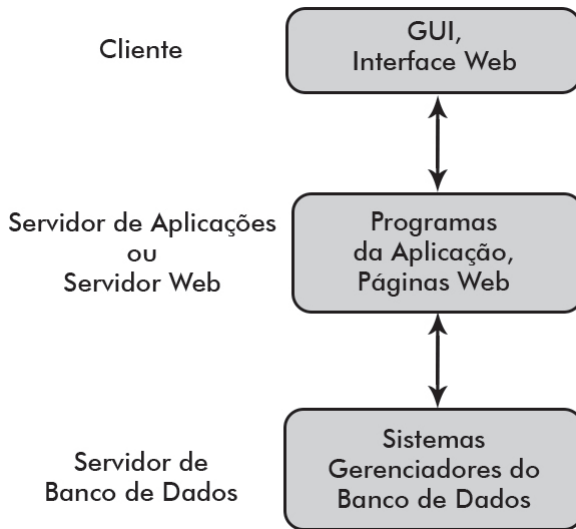


FIGURA 4 Arquitetura lógica cliente/servidor de três camadas (DATE, 2000).

4 Distribuído

A arquitetura de banco de dados distribuída tem como característica a subdivisão dos dados e o seu compartilhamento entre vários computadores, e a sua atualização é feita através de um sincronismo entre os computadores pertencentes ao circuito de distribuição. A arquitetura distribuída utiliza vários SGBD's que são espalhados pelos hosts e contendo dados replicados em vários lugares em que bancos de dados diferentes são consultados ao mesmo tempo por um mesmo usuário. Os dados são distribuídos pela rede, e, quando é feita uma consulta pelo usuário, essa distribuição é transparente, fazendo com que o usuário pense

que ele está consultando os dados de apenas um computador e não vários computadores (DATE, 2000). O modelo é exemplificado na Figura 5 a seguir:

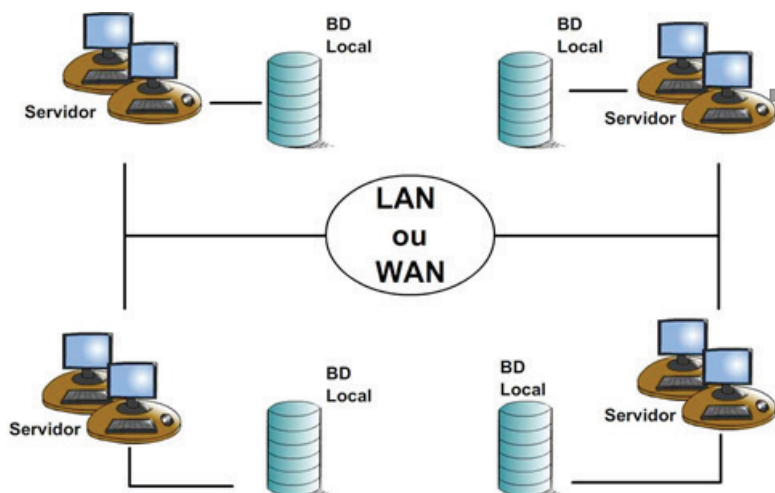


Figura 5 Modelo distribuído (ELMASRI, R. & NAVATHE, 2005).

Principais vantagens:

- Utilização de diferentes SGBD's;
- Utilização de diferentes tipos de computadores;
- Possibilidade de acessar os dados de vários computadores como se fosse de apenas um só.

Principais desvantagem:

- Esforço computacional para que seja mantida a consistência dos dados e a sua segurança.

5 Paralelo

Na arquitetura de banco de dados paralela, é utilizado o paralelismo de vários servidores que trabalham simultaneamente sobre os seus respectivos bancos de dados para a obtenção de um rápido resultado. No paralelismo, os servidores são alinhados e cada um é responsável pelas sub tarefas de um banco de dados (**ELMASRI, R. & NAVATHE, 2005**). O modelo paralelo exemplificado na Figura 6.

Toda unificação e distribuição é realizada por um *hardware* que tem o papel de roteamento, fazendo com que o particionamento seja de dois tipos:

- **HASH:** no particionamento HASH, as tabelas são particionadas igualmente segundo uma chave de hashing. A sua utilização é ideal para aplicativos que enfatizam o acesso associado.
- **RANGER:** no particionamento RANGER, as tabelas são divididas por vários valores de chaves. A sua utilização é ideal para aplicações que enfatizam a busca pela chave de particionamento.
- **ESQUEMA:** as tabelas são distribuídas entre vários servidores e a sua utilização é ideal quando se possui grandes quantidades de tabelas pequenas.

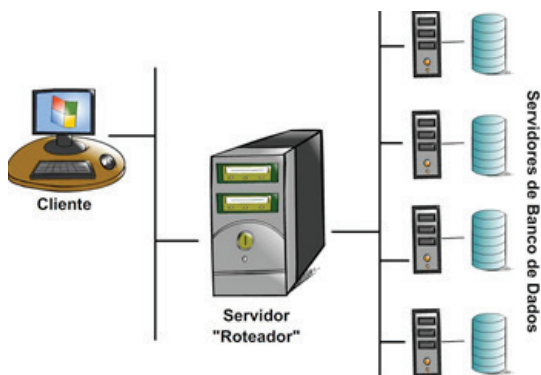


Figura 6 Modelo paralelo (ELMASRI, R. & NAVATHE, 2005).

Recapitulando

Na arquitetura centralizada, existe um computador com grande capacidade de processamento, o qual é o hospedeiro do SGBD e emuladores para os vários aplicativos. Essa arquitetura tem como principal vantagem a de permitir que muitos usuários manipulem grande volume de dados. Sua principal desvantagem está no seu alto custo, pois exige ambiente especial para *mainframes* e soluções centralizadas.

- **Sistemas de Computador Pessoal – PC.** Os computadores pessoais trabalham em sistema *stand-alone*, ou seja, fazem seus processamentos sozinhos. No começo, esse processamento era bastante limitado, porém, com a evolução do *hardware*, tem-se hoje PCs com grande capacidade de processamento. A principal vantagem dessa arquitetura é a simplicidade.

- Banco de Dados Cliente-Servidor. Na arquitetura Cliente-Servidor, o cliente (*front_end*) executa as tarefas do aplicativo, ou seja, fornece a interface do usuário (tela e processamento de entrada e saída). O servidor (*back_end*) executa as consultas no DBMS e retorna os resultados ao cliente. Apesar de ser uma arquitetura bastante popular, são necessárias soluções sofisticadas de software que possibilitem: o tratamento de transações, as confirmações de transações (*commits*), desfazer transações (*rollbacks*), linguagens de consultas (*stored procedures*) e gatilhos (*triggers*). A principal vantagem dessa arquitetura é a divisão do processamento entre dois sistemas, o que reduz o tráfego de dados na rede.
- Banco de Dados Distribuídos (N camadas). Nessa arquitetura, a informação está distribuída em diversos servidores. Cada servidor atua como no sistema cliente-servidor, porém as consultas oriundas dos aplicativos são feitas para qualquer servidor indistintamente. Caso a informação solicitada seja mantida por outro servidor ou servidores, o sistema encarrega-se de obter a informação necessária, de maneira transparente para o aplicativo, que passa a atuar consultando a rede, independente de conhecer seus servidores. Exemplos típicos são as bases de dados corporativas, em que o volume de informação é muito grande e, por isso, deve ser distribuído em diversos servidores. Porém, não é dependente de aspectos lógicos de carga de acesso aos dados, ou base de dados fracamente acopladas, em que uma informação solicitada vai sendo coletada em uma propagação da consulta em uma cadeia de servidores. A

característica básica é a existência de diversos programas aplicativos consultando a rede para acessar os dados necessários, porém, sem o conhecimento explícito de quais servidores dispõem desses dados.

Referências

ELMASRI, R. & NAVATHE, S. B. **Sistemas de Banco de Dados**. Rio de Janeiro: Person, 2005. 4. ed.

MACHADO, Felipe N. R. **Banco de Dados: Projeto e Implementação**. São Paulo: Érica, 2004. 1. ed.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro: Campus, 2000. 7. ed.

BATTISTI, Júlio. **SQL Server 2000: Administração e Desenvolvimento – Curso Completo**. 2. ed. Rio de Janeiro: Axccl Books, 2001.

VASKEVITCH, David. **Estratégia Cliente/Servidor: um guia para a reengenharia da empresa**. São Paulo: Berkeley, 1995.

Atividades

A partir dos estudos desenvolvidos nesse capítulo, assinale (V) para as assertivas Verdadeiras e (F) para as Falsas.

- a) () O primeiro modelo computacional utilizado exclusivamente durante anos foi o chamado sistema centralizado composto por um “host” e vários “terminais” onde os terminais possuem pouco poder de processamento e são compostos apenas pelo monitor e o teclado.
- b) () Uma das vantagens da arquitetura cliente/servidor é possibilidade de acessar os dados de vários computadores como se fosse de apenas um só.
- c) () A arquitetura cliente/servidor foi desenvolvida para trabalhar com ambientes computacionais, nos quais um grande número de PCs, estações de trabalho, servidores de arquivo, impressoras, servidores de banco de dados, servidores Web e outros equipamentos estão conectados via rede.
- d) () Na arquitetura paralela, temos um mainframe e vários terminais “burros”.
- e) () A arquitetura de banco de dados distribuída tem como característica a subdivisão dos dados e o seu compartilhamento entre vários computadores e a sua atualização é feita através de um sincronismo entre os computadores pertencentes ao circuito de distribuição.

Gabarito

a) V b) F c) V d) F e) F

Edemar Costa Oliveira¹

Capítulo **10**

Conceitos e Estratégias de Implantação, Data Warehouse, OLAP e Ferramentas de BI

¹ Mestre em Engenharia, Pós-graduado em Tecnologias para Negócios na Internet, Bacharel em Matemática Computacional. Analista de Sistemas e Professor na Ulbra.

Apresentação

Neste capítulo, vamos apresentar uma breve descrição dos conceitos de Business Intelligence (BI) e suas principais ferramentas. Essas técnicas e tecnologias começaram a ganhar terreno na década de 1990. Nessa época, novas tecnologias começaram a despontar, bancos de dados relacionais começaram a tornar-se padrão de modelo comercial e a possibilidade de ler grandes volumes de dados analiticamente favoreceram a inteligência de negócios e uma visão mais abrangente para os sistemas de informações executivas.

No início dos anos 1980, surgiu o conceito de *sistemas de informações executivas (EIS)*, esse conceito expandiu o suporte computadorizado aos gerentes e executivos de nível superior. Alguns dos recursos introduzidos foram sistemas de geração de relatórios dinâmicos multidimensionais (*ad hoc* ou sob demanda), prognósticos e previsões, análise de tendências, detalhamento, e acesso a *status* e fatores críticos de sucesso. Esses recursos apareceram em dezenas de produtos comerciais até o meio da década de 1990. Depois, os mesmos recursos e alguns recursos novos apareceram sob o nome BI. Assim, o conceito original de Sistemas de Informações Executivas foi transformado em BI (TURBAN, 2008).

Apresentaremos, a seguir, a relação de ferramentas que caracterizam o BI moderno e permitem às empresas aplicar esses conceitos a grandes volumes de dados buscando vantagem competitiva nos seus ramos de negócio.

Introdução

Estudar BI ou *Business Intelligence* é estar em contato com um processo fundamental para as empresas modernas. No mundo atual globalizado e competitivo, a disputa pela busca de informação privilegiada e de qualidade é acirrada. Este pode ser apontado como fator determinante pelo qual essa ciência evolui tanto nos últimos anos.

O termo *Business Intelligence* ou BI foi utilizado pela primeira vez pelo Gartner Group nos anos 1990. Segundo BARBIERI (2001), BI ou, ainda, inteligência de negócio, é o conhecimento e previsão dos ambientes interno e externo à empresa, orientando as ações gerenciais, tendo em vista a obtenção das vantagens competitivas.

O conceito de *Business Intelligence* surgiu para resolver questões da ampliação da ação dos sistemas informatizados do nível operacional para estratégico e da criação de condições para a extração e visualização de informações gerenciais dos acervos de dados, criando um ambiente de conhecimento, onde há produção sistemática de informação gerencial, veloz e consistente (TURBAN, SHARDA, ARONSON, KING, 2008).

A evolução das tecnologias tem permitido que as empresas possam buscar informações de qualidade e aumentar seu destaque no mercado de atuação. A busca de diferenças e de informações sobre concorrência e sobre seus próprios produtos intensifica-se e movimenta o uso das ferramentas de BI.

Para apoiar a tomada de decisão, a nível estratégico, são utilizadas ferramentas e métodos de inteligência de negócios

através da aplicação de técnicas de BI em que as empresas podem transformar os dados recolhidos pelos diversos sistemas de informação em conjuntos de informações qualitativas que irão apoiar para a tomada de decisão inteligente (BARBIE-RI-2001).

1 Conceitos básicos

Sistemas de *BI* são os que permitem aos altos executivos de uma empresa a análise de informações agrupadas de forma relevante possibilitando a tomada de decisões.

As ferramentas de BI podem fornecer uma visão sistêmica do negócio e ajudar na distribuição uniforme dos dados entre os usuários, sendo seu objetivo principal transformar grandes quantidades de dados em informações de qualidade para a tomada de decisões. Através delas, é possível cruzar dados, visualizar informações em várias dimensões e analisar os principais indicadores de desempenho empresarial. Essas características das ferramentas de BI podem contribuir diretamente nas funções pelos responsáveis das decisões estratégicas da empresa Partner Sistemas, na obtenção, análise e monitoramento das atividades da empresa de modo geral (BATISTA, 2004).

O objetivo da inteligência de negócios é converter o volume de dados em informações relevantes ao negócio, através de relatórios analíticos. A Figura 1 mostra a visão conceitual do objetivo de BI ao reunir dados das mais diversas fontes e converter em informações relevantes à tomada de decisão.

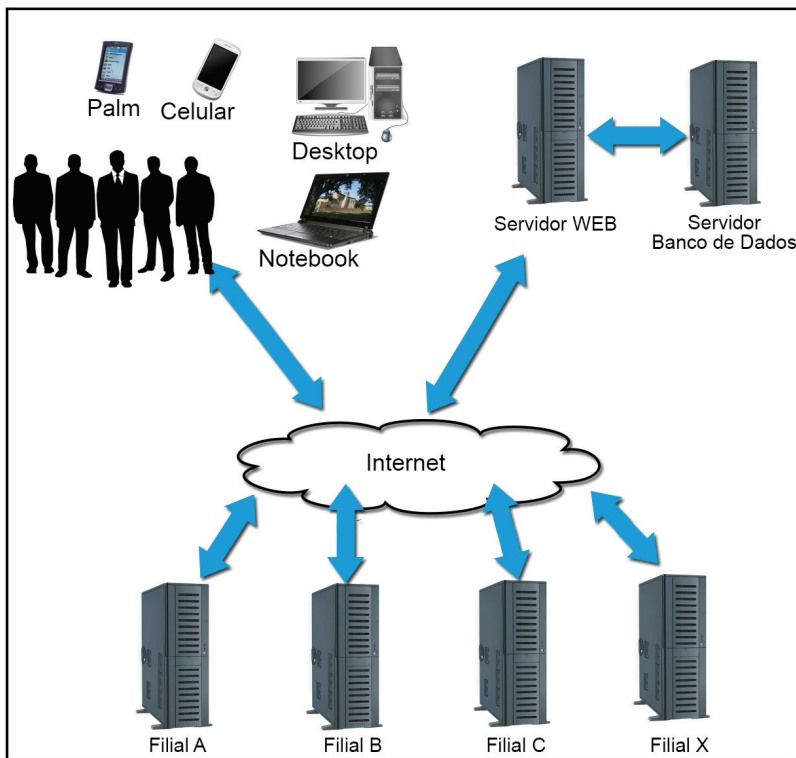


Figura 1 Visão conceitual de *Business Intelligence* (TEOREY; LIGHTSTONE; NADEAU, 2007).

Segundo PRIMAK (2008), os sistemas de *Business Intelligence* têm como principais características:

- A capacidade de extrair e integrar dados de múltiplas fontes.
- A transformação dos registros obtidos em informação útil para o conhecimento empresarial.
- A valorização da experiência adquirida.

- A análise de dados contextualizados.
- Trabalhar com hipóteses.
- A procura de relações de causa e efeito, trabalhando com hipóteses e desenvolvendo estratégias e ações competitivas.

Alguns benefícios dos sistemas de *Business Intelligence*:

- Desenvolver e analisar novas opções de futuro, frente às mudanças de ambiente externo e concorrência.
- Antecipar mudanças externas e identificar oportunidades e ameaças.
- Reduzir incertezas do ambiente de negócio.
- Apoiar as decisões estratégicas e táticas da organização através de informações analisadas do ambiente externo e avaliações de impacto das tendências tecnológicas nas áreas de negócio da empresa.

2 Data Warehouse

O conceito de armazém de dados representa um repositório de múltiplas fontes de dados heterogêneas, organizado em um mesmo gerenciador sob um esquema unificado, com o objetivo de facilitar tomadas de decisões gerenciais.

Os *Data Warehouses* são projetados para suportar altas demandas de processamento, uma vez que manipulam quantidades elevadas de dados oriundos de vários bancos de dados, que podem inclusive pertencer a plataformas diferentes ou possuem estruturas de dados distintas, a Figura 2 sintetiza a ideia.

Segundo BARBIERI (2001), *Data Warehouse*, cuja tradução literal é armazém de dados, é um banco de dados, destinado a sistemas de apoio à decisão e cujos dados foram armazenados em estruturas lógicas dimensionais, possibilitando o seu processamento analítico por ferramentas especiais (OLAP e Mining).

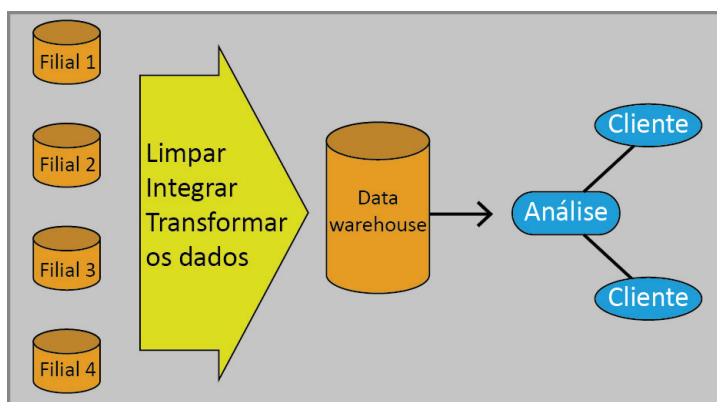


Figura 2 Arquitetura básica do Data Warehouse (TEOREY; LIGHTSTONE; NADEAU, 2007).

3 DATAMART

O Data Warehouse, ou armazém de dados, agrega informações de várias áreas funcionais da empresa. Cada área funcional tem seus dados representados dentro do Data Warehouse em conjuntos de informações. Esses conjuntos de informações são os DataMart.

Barbieri (2001) caracteriza um Datamart como um subconjunto de informações existentes em um *Data Warehouse*. O desenho é elaborado de tal forma a entender a um segmento ou unidade de uma organização, vide Figura 3. São considerados como Data Warehouses departamentais, nos quais os dados são ajustados aos requisitos de cada área ou departamento.

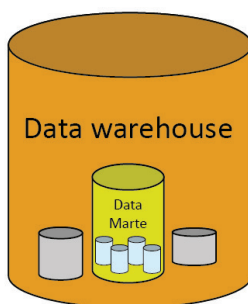


Figura 3 Data Mart (NERY,2007).

Pequenas e médias empresas que adotam sistemas de BI adotam como forma de reduzir a complexidade de um *Data Warehouse*. Segundo Willian Pereira (2004), é costume dividir essa arquitetura em três camadas, assim distribuídas:

1. Camada do banco de dados transacional, em que os dados da empresa são propriamente armazenados.
2. Camada do *Data Warehouse*, um repositório de dados históricos com informações detalhadas.
3. Camada do *Data Mart*, que são conjuntos de tabelas estruturadas, alimentadas pela segunda camada.

Veja o diagrama apresentado na Figura 4.

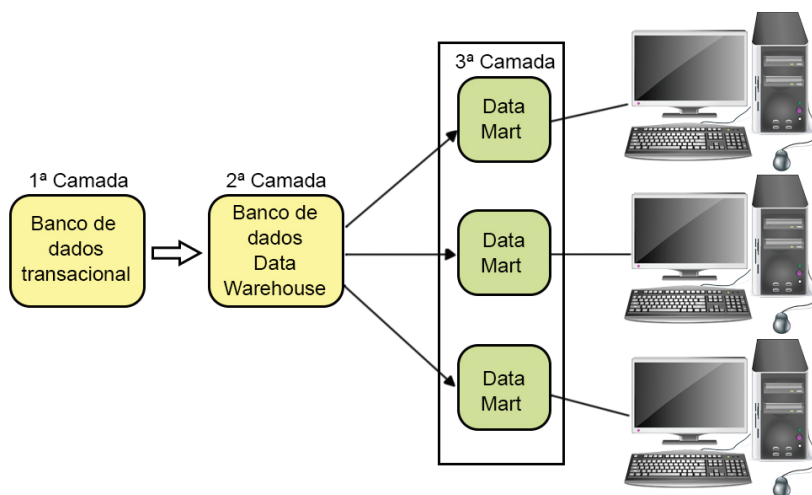


Figura 4 Diagrama de um Data Mart (WILLIAN PEREIRA, 2004).

Segundo PRIMAK (2008), "Algumas vezes, projetos que começam como data warehouses se transformam em *Data Marts*. Quando as organizações acumulam grandes volumes de dados históricos para suporte à decisão que se mostram pouco ou nunca utilizados, elas podem reduzir o armazenamento ou arquivamento de informações e con-

trair o seu Data Warehouse em um *Data Mart* mais focado. Ou elas podem dividir o DW em vários *Data Marts*, oferecendo tempos de resposta mais rápidos, acesso mais fácil e menos complexidade para os usuários finais”.

4 OLAP (On-line-analytical processing)

Olap ou processamento analítico on-line pode ser vista como uma base de dados multidimensional. A base Olap agrega as informações das áreas funcionais da empresa com a finalidade de gerar “cubos” de dados. Esses cubos são exatamente os objetos que caracterizam esse aspecto multidimensional dessa tecnologia. As informações geradas a partir de cubos extrapolam a dimensão linha, coluna oferecendo ao usuário dados mais complexos. Nos bancos de dados multidimensionais (multidimensional data base (MDB)) todos os dados são armazenados em vetores com mais de uma dimensão, eliminando a sobrecarga de processamento associada aos comandos de combinação de linhas de tabelas (joins, comuns em banco de dados relacionais).

Assim, o armazenamento de informações é mais eficiente, consumindo menos espaço em disco e consequentemente menos *hardware*.

De acordo com SIQUEIRA (2005), OLAP é considerado uma categoria de *software* que permite a analistas, gerentes e executivos obterem respostas dentro dos dados, através de uma rápida, consistente e interativa forma de acesso a uma

ampla variedade de possíveis visões. As ferramentas OLAP permitem que o negócio da empresa possa ser visualizado e manipulado de forma multidimensional, isto é, agrupando as informações em várias dimensões como: produtos, fornecedores, departamentos, localização, clientes e recursos.

A necessidade de receber um grande número de dados de um grande banco de dados (centenas de Giga ou até mais) são os motivos de existir o OLAP (não é um aplicativo, é uma arquitetura de aplicação). Quando temos a necessidade de um sistema multidimensional, precisamos de um OLAP.

Uma outra vantagem do OLAP é que ele é interativo. O analista pode jogar um valor para simular algo. Assim, pode inclusive descobrir padrões escondidos. Felipe Neri (2007).

Eu posso acrescentar ou tirar uma dimensão do cubo, conforme eu necessitar. O tempo de resposta de uma consulta multidimensional depende de quantas células são requeridas.

Para resolver o tamanho do problema do cubo, que cresce exponencialmente, a saída é consolidar todos os sub totais lógicos e os totais por todas as dimensões. Essa consolidação faz sentido quando as dimensões fazem parte de uma mesma hierarquia (anos, semestres, meses, dias).

Linhas Guia do OLAP (MILLER, 2002)

- Visão conceitual multidimensional: enfatiza a forma como o usuário “vê” dados sem impor que os dados sejam armazenados em formato multidimensional.

- **Transparência:** localização da funcionalidade OLAP deve ser transparente para o usuário, assim como a localização e a forma dos dados.
- **Facilidade de Acesso:** acesso a fontes de dados homogêneas e heterogêneas deve ser transparente.
- **Desempenho de consultas consistente:** não deve ser dependente do número de dimensões.
- **Arquitetura cliente/servidor:** produtos devem ser capazes de operar em arquiteturas cliente/servidor.
- **Dimensionalidade genérica:** todas as dimensões são iguais.
- **Manipulação dinâmica de matrizes esparsas:** produtos devem lidar com matrizes esparsas eficientemente.
- **Suporte multiusuário.**
- **Operações entre dimensões sem restrições.**
- **Manipulação de dados intuitiva.**
- **Relatórios/consultas flexíveis.**
- **Níveis de agregação e dimensões ilimitados:** ferramentas devem ser capazes de acomodar 15 a 20 dimensões.

Categorias de ferramentas OLAP (MILLER, 2002)

MOLAP: é utilizado, tradicionalmente para organizar, navegar e analisar dados.

ROLAP: permite que múltiplas consultas multidimensionais de tabelas bidimensionais relacionais sejam criadas sem a necessidade de estrutura de dados normalmente requerida nesse tipo de consulta.

MQE: possui a capacidade de oferecer análise “datacube” e “slice and dice”. Isso é feito primeiro desenvolvendo uma consulta para selecionar dados de um DBMS que entrega o dado requisitado para o *desktop*, que é o local onde está o datacube. Uma vez que os dados estão no datacube, usuários podem requisitar a análise multidimensional.

Produtos no mercado: (PRIMAK, 2007)

Cognus Power Play: é um *software* maduro e popular que é caracterizado como um MQE. Ele pode aproveitar o investimento feito na tecnologia de banco de dados relacional para oferecer acesso multidimensional para a corporação, com a mesma robustez, escalabilidade e controle administrativo.

IBI Focus Fusion: é um banco de dados com tecnologia multidimensional para OLAP e data warehouse. É desenhado para endereçar aplicações de negócios que precisem de análise dimensional dos dados dos produtos.

Sua aplicação mais específica é para a formação de aplicações de inteligência de negócios em um ambiente de data warehouse.

Pilot Software: é uma suíte de ferramentas que incluem: um banco de dados multidimensional de alta velocidade (MO-LAP), integração com data warehouse (ROLAP), data mining e

várias aplicações de negócio customizáveis focando pós-venta e profissionais de *marketing*

Ferramentas OLAP e Internet

A Web é um perfeito meio para suporte de decisão:

- A Internet é um recurso virtualmente livre que permite conectividade com e entre as empresas.
- A Web permite a companhias guardar e gerenciar dados e aplicações que podem ser gerenciados centralmente, mantidos e atualizados, eliminando problemas com *software* e dados financeiros.
- A Web facilita as tarefas administrativas complexas de ambiente de gerenciamento distribuído.

5 Data Mining

Data mining ou mineração de dados pode ser descrito como a busca de padrões e informações relevantes dentro de um universo de dados. A partir do momento em que os dados estão organizados em Data Warehouse e DataMarts, o cruzamento das informações em várias dimensões possibilita que alguns padrões surjam e “insights” bastante úteis sejam descobertos para auxiliar na tomada de decisões.

Data mining, em sua definição mais simples, automatiza a detecção de padrões relevantes em um banco de dados (MILLER, 2002). Ou ainda: é o processo de descobrir conhecimen-

tos interessantes a partir de grandes conjuntos de dados, os quais podem estar armazenados em bases de dados, data warehouses ou em outros repositórios de dados (SIQUEIRA, 2005).

O *data mining* usa técnicas de estatística e de inteligência artificial bem estabelecidas para construir modelos que predizem o comportamento do cliente. Hoje, a tecnologia automatiza os processos de busca e integra-os com os data warehouses comerciais, apresentando os resultados de uma maneira interessante aos usuários do negócio.

A ideia principal é usar técnicas e estatísticas e alguns recursos de inteligência artificial que permitem construir modelos cujo objeto é prever o comportamento do cliente. Essas técnicas aliadas aos Data Warehouse proporcionam resultados interessantes e bastante válidos para o mundo dos negócios. Diariamente, somos alvos de campanhas de *marketing* das mais interessantes oriundas de dados passados pelo Data Mining.

Importância para os negócios

Para o *data mining* impactar um negócio, é necessário que ele considere o processo de negócio fundamental. O *data mining* é parte de um conjunto de ações maior que está entre a empresa e seus clientes. O caminho pelo qual o *data mining* impacta o negócio depende do processo de negócio, não do processo do *data mining*. Pegue, por exemplo, *marketing* de produto. O trabalho de um gerente de *marketing* é entender seu mercado. Com esse entendimento vem a habilidade de

interagir com o cliente nesse mercado, usando diversos canais. Isso envolve uma quantidade de áreas, incluindo *marketing* direto, propaganda impressa, *telemarketing* e propaganda em rádio/televisão, entre outros (JACOBSON, 2005).

Nery (2007) Salienta que o ponto que deve ser destacado é que o resultado do *data mining* é diferente de outros processos de negócio baseados em dados. Na maior parte das interações padrões com os dados do cliente, quase todos os resultados apresentados ao usuário são coisas que ele já sabia existir nos bancos de dados. Um relatório mostrando a diminuição das vendas de certa linha de produtos em uma região é direto para o usuário porque ele intuitivamente sabe que esse tipo de informação já existe no banco de dados. Se a empresa vende diferentes produtos em diferentes regiões do país, não há problema em transformar uma tela dessa informação em entendimento relevante do processo de negócio.

O mais interessante no processo de Data Mining é a propriedade de trazer informações que o usuário desconhecia e que está no banco de dados. O comportamento do cliente está descrito nos dados, mas nem sempre de forma clara e explícita. Tornar esses dados em informações claras e que podem ser utilizadas de forma competitiva é um dos objetivos do Data Mining.

Recapitulando

○ BI é um conjunto de:

- Ferramentas
- Técnicas
- Métodos

Permite que informações em estado bruto, sejam:

- Organizadas
- Analisadas
- Comparadas
- Divulgadas

Tornando-se uma importante ferramenta no apoio à tomada de decisões.

O BI faz parte do processo de inteligência de uma empresa e deve auxiliar e permitir a tomada das melhores e mais rápidas decisões.

Através do BI:

Os dados presentes na empresa, em suas diversas bases de informação, sejam elas informatizadas ou não:

- ERP, CRM
- Sistemas transacionais
- Sistemas Legados
- Bancos de Dados Públicos
- Internet

- ➡ Planilhas
- ➡ Documentos
- ➡ Cultura e história da empresa

Podem ser filtrados e disponibilizados para diversas pessoas da Organização.

Vantagens:

- ➡ Redução de custos no processo de inteligência
- ➡ Rapidez na tomada de decisões
- ➡ Dados confiáveis
- ➡ Rotinas gerenciais intercambiáveis
- ➡ Visão mais consistente e flexível
- ➡ Fluxo de informação padronizada
- ➡ Organização das informações

Aplicações:

As áreas de aplicação dependem da área de negócios da empresa.

O BI pode ser aplicado em qualquer área que haja informação armazenada com demanda de estudo e análise.

- ➡ Vendas
- ➡ Marketing
- ➡ Clientes
- ➡ RH

- ➡ Produção
- ➡ Contabilidade
- ➡ Estoque

Análises e consultas:

Uma das grandes vantagens da organização das informações e uma cultura de Business Intelligence na empresa é a facilidade para a geração de análises e consultas.

Processos que demoravam dias para serem concluídos devido ao levantamento de grandes volumes de informações, podem ser elaborados em alguns minutos.

Etapas:

Estudo da cultura e necessidade de informações e decisões da empresa.

Levantamento das bases de dados e processos de armazenamento de informações.

- ➡ Organização das informações
- ➡ Criação de cubos e dimensões
- ➡ Implantação das ferramentas de análise e geração de consultas
- ➡ Customização de relatórios e indicadores
- ➡ Treinamento, acompanhamento
- ➡ Suporte

Referências

- BARBIERI, Carlos. **BI – business intelligence**: modelagem & tecnologia. Rio de Janeiro: Axel Books do Brasil, 2001.
- BATISTA, Emerson de Oliveira. **SISTEMA DE INFORMAÇÃO**: o uso consciente da tecnologia para o gerenciamento, Ed. Saraiva, 2004.
- JACOBSON, Reed, MISNER, Satcia, CONSULTING, Hitachi. **SQL SERVER 2005 ANALYSIS SERVICES**. Ed. Bookman, 2008.
- MILLER, Jerry P. **O milênio da inteligência competitiva**. Porto Alegre: Bookmam, 2002.
- MISNER, Stacia, CONSULTING, Hitachi. **SQL SERVER 2005 REPORTING SERVICES**, Ed. Bookman, 2007.
- NERY, Felipe. **TECNOLOGIA E PROJETO DE DATA WAREHOUSE**, Ed. Érica, 3. Ed. 2007.
- PRIMAK, Fábio Vinícius. **Decisões Com B.I. – Business Intelligence**. 2008.
- SIQUEIRA, M.C. **Gestão Estratégica da Informação**. 2005.
- TURBAN, Efraim/SHARDA, Ramesh/ARONSON, Jay E./KING, David. **Business Intelligence – Um enfoque gerencial para a inteligência do negócio**, 2008.

Atividades

A partir dos estudos desenvolvidos nesse capítulo, Assinale (V) para as assertivas Verdadeiras e (F) para as Falsas.

- a) () OLAP é um método de extração de dados, consistente e interativa com acesso a uma ampla variedade de possíveis visões.
- b) () O termo BI teve suas raízes nos sistemas de geração de relatórios SIG dos anos 1970.
- c) () O Datamart é um subconjunto do OLAP, cujo desenho é elaborado de tal forma a entender a um segmento ou unidade de uma organização.
- d) () As ferramentas de BI podem fornecer uma visão sistêmica do negócio e ajudar na distribuição uniforme dos dados entre os usuários, sendo seu objetivo principal transformar grandes quantidades de dados em informações de qualidade para a tomada de decisões.
- e) () Data Warehouse, cuja tradução literal é armazém de dados, é um banco de dados, destinado a sistemas de apoio a decisão e cujos dados foram armazenados em estruturas lógicas dimensionais, possibilitando o seu processamento analítico por ferramentas especiais.

Gabarito

- a) F b) V c) F d) V e) V