

Banco de Dados II

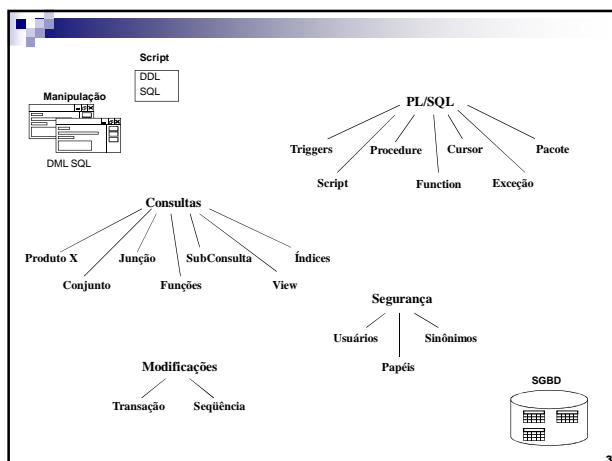
DML – Parte II

Osmar de Oliveira Braz Junior

1

Objetivos

- Construir triggers para replicação e log de tabelas.
- Construir procedimentos armazenados.
- Construir functions para simplificar consultas de tabelas.
- Utilizar cursores para percorrer conjunto de dados.
- Construir tabelas temporárias.
- Tratar exceções em blocos de pl/sql.
- Organizar blocos de pl/sql em pacotes.
- Realizar o gerenciamento de usuários.
- Criar sinônimos para objetos de banco de dados.
- Criar links para outros SGBDs.
- Agendar tarefas no BD.



3

14. Triggers

- Um **gatilho** é um comando que é executado pelo sistema automaticamente, em consequência de uma modificação no banco de dados. Duas exigências devem ser satisfeitas para a projeção de um mecanismo de gatilho:
 - Especificar as condições sob as quais o gatilho deve ser executado.
 - Especificar as ações que serão tomadas quando um gatilho for disparado.
- Os **gatilhos** são mecanismo úteis para avisos a usuários ou para executar automaticamente determinadas tarefas quando as condições para isso são criadas.

4

14. Triggers

14.1 Tipos de Triggers

- Existem **dois tipos** distintos de trigger que podem ser usados em uma tabela.
- **Statement-level- Trigger** Essa trigger é disparado apenas uma vez. Por exemplo, se o comando update atualizar 15 linhas, os comandos contidos na trigger serão executados uma única vez. Também chamado de **trigger em nível de instrução**.
- **Row-level- Trigger** Essa trigger tem os seus comandos executados para todas as linhas que sejam afetadas pelo comando que gerou o acionamento do trigger. Também chamado de **trigger em nível da linha**.

5

14. Triggers

14.2. Componentes de uma Trigger

- **Comando SQL que aciona a trigger.** Uma trigger pode ser ativada pelos comandos **INSERT**, **DELETE** e **UPDATE**. Uma mesma trigger pode ser invocada quando mais de uma ação ocorrer, ou seja, uma trigger pode ser invocada somente quando um comando INSERT for executado, ou então quando um comando UPDATE ou DELETE for executado.
- **Limitador de ação da trigger.** Representado pela cláusula **WHEN(INSERTING,DELETING, UPDATING)**, especifica qual a condição deve ser verdadeira para que a trigger seja disparado.
- **Ação executada pelo trigger.** É o bloco de sql que é executado pela trigger.

6

14. Triggers

14.3. Momento de Disparo

- Um trigger pode ser disparado antes (**BEFORE**), ou depois (**AFTER**) que um dos comandos de ativação (**INSERT**, **UPDATE**, **DELETE**) for executado. Uma tabela pode conter até 12 trigger associados aos comandos de ativação e momento de disparo. São as seis trigger do tipo **row-level** e seis do tipo **statement-level**.

- ☐ BEFORE INSERT
- ☐ AFTER INSERT
- ☐ BEFORE DELETE
- ☐ AFTER DELETE
- ☐ BEFORE UPDATE
- ☐ AFTER UPDATE

7

14. Triggers

14.4. Criação

- Uma trigger pode ser criada através do comando SQL: **CREATE TRIGGER**.

```
CREATE [OR REPLACE] TRIGGER <nome_da_trigger>
[BEFORE/AFTER] Comando_de_disparo
[OF <nome_da_coluna>] ON <nome_da_tabela>
[FOR EACH ROW]
[WHEN (condição)]
[Declare
[Declarações]]
BEGIN
Comandos;
END;
```

- OR REPLACE** Permite alterar a trigger sem que seja necessário apagá-lo previamente.
- Nome_da_trigger** Nome de identificação da trigger.
- Comando_de_disparo** É o nome do comando que ativa a trigger (INSERT, DELETE, UPDATE).
- OF nome_da_coluna** É a lista de nomes dos campos na tabela que podem disparar a trigger.
- ON nome_da_tabela** Indica a tabela ou esquema para qual a trigger está sendo criada.
- FOR EACH ROW** Indica que a trigger é do tipo **row-level** e deve ser executada para todas as linhas selecionadas.
- WHEN (condição)** Especifica a restrição da trigger. Uma condição SQL que precisa ser atendida para disparar a trigger.
- Declarações** - Nessa seção são declaradas constantes, variáveis com o uso de **declare**.
- Comandos** - Nessa seção são colocados os comandos que serão executados pela trigger.

8

14. Triggers

14.5. Limitações de Uso de Trigger

- Uma trigger não pode executar os comando **COMMIT** ou **ROLLBACK** nem tão pouco chamar procedures ou funções que executem essas tarefas.
- Uma trigger do tipo **row-level** não pode ler ou modificar o conteúdo de uma tabela em mutação. Uma tabela mutante é aquela na qual seu conteúdo está sendo alterado por um comando INSERT, DELETE e UPDATE, e o comando não foi terminado, ou seja ainda não foram gravados com **COMMIT**.

9

14. Triggers

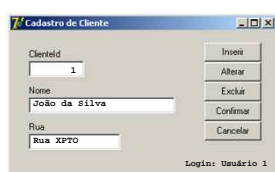
14.6. Referência a Colunas

- Dentro de uma trigger do tipo **row-level** é possível acessar o valor de um campo de uma linha. Dependendo da operação que está sendo executada é necessário preceder o nome da coluna com o sufixo **:new** ou **:old**.
- Para um comando **INSERT**, os valores dos campos que serão gravados devem ser precedidos pelo sufixo **:new**.
- Para um comando **DELETE**, os valores dos campos da linha que está sendo processada devem ser precedidos do sufixo **:old**.
- Para um comando **UPDATE**, o valor original que está sendo gravado é acessado com o sufixo **:old**. Os novos valores que serão gravados devem ser precedidos do sufixo **:new**.

10

14. Triggers

14.6. Referência a Colunas



Formulário de Cadastro de Cliente com campos: Clevelid (valor 1), Nome (João da Silva), Rua (Rua XPTO). Botões: Inserir, Alterar, Excluir, Confirmar, Cancelar. Login: Usuário 1.

Usuário

:old.clienteId
:old.nome
:old.rua

:new.clienteId
:new.nome
:new.rua

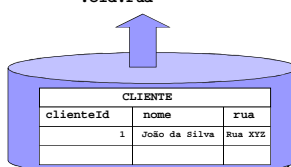


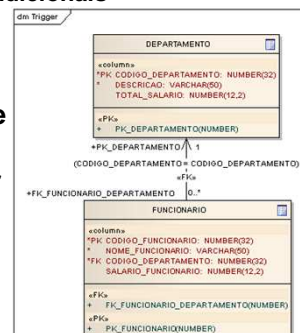
Diagrama de banco de dados com uma tabela CLIENTE. Um usuário está inserindo dados na tabela.

clienteId	nome	rua
1	João da Silva	Rua XYZ

14. Triggers

14.7. Predicados Condicionais

- Considere a necessidade de manter o total de salário pagos a funcionários por departamento.



12

14. Triggers

14.7. Predicados Condicionais

■ INSERTING / UPDATING / DELETING

```
CREATE OR REPLACE TRIGGER TOTAL_SALARIO_POR_DEPARTAMENTO
AFTER DELETE OR INSERT OR UPDATE OF salario_funcionario
ON funcionario FOR EACH ROW
BEGIN
  IF DELETING THEN
    UPDATE departamento
    SET total_salario = total_salario - :OLD.salario_funcionario
    WHERE codigo_departamento = :OLD.codigo_departamento;
  ELSIF INSERTING THEN
    UPDATE departamento
    SET total_salario = total_salario + :NEW.salario_funcionario
    WHERE codigo_departamento = :NEW.codigo_departamento;
  ELSIF UPDATING THEN
    UPDATE departamento
    SET total_salario = total_salario +
      (:NEW.salario_funcionario - :OLD.salario_funcionario)
    WHERE codigo_departamento = :OLD.codigo_departamento;
  END IF;
END;
```

13

14. Triggers

14.8. Manipulação

■ Ativação/Desativação de uma Trigger

`ALTER TRIGGER <nome_da_trigger> DISABLE;`

■ Alteração de uma Trigger

Uma trigger não pode ser alterado diretamente. Na verdade a única opção de alterar uma trigger é sua recriação usando a opção **OR REPLACE** do comando **CREATE**. Se uma trigger teve privilégios cedidos para outros usuários, eles permanecem válidos enquanto o trigger existir.

■ Remoção de uma Trigger

Para apagar uma trigger, o seguinte comando deve ser usado:

`DROP TRIGGER <nome_da_trigger>;`

14

14. Triggers

14.8. Manipulação

■ Visualizando Triggers

Para o usuário corrente a tabela e **USER_TRIGGERS**:

```
SELECT trigger_name
FROM USER_TRIGGERS;
```

Para os usuários system e manager a tabela é **DBA_TRIGGERS**.

■ Mostrando Erros

Se ocorrer algum erro você pode usar o comando **SHOW Errors** para saber o que aconteceu.

```
SHOW Errors;
```

15

14. Triggers

■ Resolver os exercícios de DML Trigger (ExercicioDML_Trigger.pdf)

16

14. Triggers

1) Crie uma TRIGGER para preencher o conteúdo de uma coluna caso ele seja deixado em branco durante uma operação de INSERT. Nesse exercício caso seja deixado em branco o campo data de cadastro durante a inclusão, a TRIGGER automaticamente preenche com a data do sistema. Se for necessário altere a tabela piloto adicionando o campo **DATA_INSERT DATE**.

```
R.
-- Adiciona o campo a tabela
ALTER TABLE piloto ADD data_insert DATE;

CREATE OR REPLACE TRIGGER log_insert_piloto
BEFORE INSERT ON Piloto FOR EACH ROW
BEGIN
  -- Se a aplicacao não preencher
  IF :NEW.data_insert IS NULL THEN
    :NEW.data_insert := SYSDATE;
  END IF;
END;
```

17

14. Triggers

2) Altere a TRIGGER anterior para que armazene também na tupla o nome do usuário que inseriu o registro. Você pega o nome do usuário através da variável **USER**. É necessário alterar a tabela piloto adicionando o campo **USUARIO_INSERT VARCHAR(30)**.

```
R.
-- Adiciona o campo a tabela
ALTER TABLE piloto ADD usuario_insert VARCHAR(30);

CREATE OR REPLACE TRIGGER log_insert_piloto
BEFORE INSERT ON Piloto FOR EACH ROW
BEGIN
  -- Se a aplicacao não preencher
  IF :NEW.data_insert IS NULL THEN
    :NEW.data_insert := SYSDATE;
  END IF;
  -- Se a aplicacao não preencher
  IF :NEW.usuario_insert IS NULL THEN
    :NEW.usuario_insert := USER;
  END IF;
END;
```

18

14. Triggers

- 3) Adicione os campos DATA_UPDATE do tipo date e USUARIO_UPDATE do tipo VARCHAR(30) para a tabela piloto. Em seguida crie uma TRIGGER que armazene a data e o nome do usuário que realizou a última atualização do registro.

```
R.
-- Adiciona os campo a tabela
ALTER TABLE piloto ADD data_update DATE;
ALTER TABLE piloto ADD usuario_update VARCHAR(30);

CREATE OR REPLACE TRIGGER log_update_piloto
BEFORE UPDATE ON piloto FOR EACH ROW
BEGIN
    -- Se a aplicacao não preencher
    IF :NEW.data_update IS NULL THEN
        :NEW.data_update := SYSDATE;
    END IF;
    -- Se a aplicacao não preencher
    IF :NEW.usuario_update IS NULL THEN
        :NEW.usuario_update := USER;
    END IF;
END;
```

19

14. Triggers

- 4) Junte as duas TRIGGER anterior em uma só utilizando predicados condicionais (INSERTING,UPDATING).

```
R.
CREATE OR REPLACE TRIGGER log_piloto
BEFORE INSERT or UPDATE ON piloto FOR EACH ROW
BEGIN
    IF INSERTING THEN
        IF :NEW.data_insert IS NULL THEN
            :NEW.data_insert := SYSDATE;
        END IF;
        IF :NEW.usuario_insert IS NULL THEN
            :NEW.usuario_insert := user;
        END IF;
    ELSIF UPDATING THEN
        IF :NEW.data_update IS NULL THEN
            :NEW.data_update := SYSDATE;
        END IF;
        IF :NEW.usuario_update IS NULL THEN
            :NEW.usuario_update := user;
        END IF;
    END IF;
END;
```

20

14. Triggers

- 4.5 Crie uma TRIGGER para semelhante as anteriores para armazenar a data e o usuário que excluiu o registro.

21

14. Triggers

- 5) A TRIGGER anterior esta auditando a tabela aonde foi criada somente para as operações de inserção e atualização. Mas para que também consiga auditar a operação de exclusão é necessário que uma tabela seja criada e a TRIGGER anterior seja excluída. Uma nova TRIGGER utilizando predicados condicionais (INSERTING,UPDATING e DELETING) deve ser criada desta vez com o DELETING para inserir na tabela de log registros destas operações. A tabela tem os seguintes campos:

Tabela LOG

Log_id	Int	Chave Primária	Utilize uma sequência para gerar o id
Data	Date		Data de inserção do log, utilize SYSDATE
Usuario	Varchar(30)		Nome do usuário utilize USER
Tabela	Varchar(30)		Tabela modificada
Operacao	Char(1)		Operação realizada('I','U','D')
Chave	int		Campo chave modificado

22

14. Triggers

```
CREATE TABLE log(log_id int,
                  data date default sysdate,
                  usuario varchar(30),
                  tabela varchar(30),
                  operacao char(1),
                  chave int,
                  Constraint pk_log primary key(log_id),
                  Constraint ck_log_operacao(
                      operacao in ('I','U','D')));

CREATE SEQUENCE sq_log;

CREATE OR REPLACE TRIGGER log_piloto
BEFORE INSERT or UPDATE or DELETE ON piloto FOR EACH ROW
DECLARE
    NOMETABELA CHAR(32) := 'PILOTO';
BEGIN
    IF INSERTING THEN
        INSERT into log VALUES(seq_log.nextval,SYSDATE,USER,
                                NOMETABELA,:NEW.codigo_piloto);
    ELSIF UPDATING THEN
        INSERT into log VALUES(seq_log.nextval,SYSDATE,USER,
                                NOMETABELA,:NEW.codigo_piloto);
    ELSIF DELETING THEN
        INSERT INTO log VALUES(seq_log.nextval,SYSDATE,USER,
                                NOMETABELA,'D',:OLD.codigo_piloto);
    END IF;
END;
```

23

14. Triggers

6. Vamos estabelecer um valor máximo para o campo salário. Nenhum Piloto poderá ganhar mais do que 10.000 dólares. Além de ativar a TRIGGER na inclusão, evitando que um novo funcionário tenha um salário superior ao limite, as rotinas de atualização(UPDATE) devem ser verificadas para evitar que os Pilotos existentes sejam alterados para valores não permitidos.

```
Utilize a operação abaixo para exibir a mensagem de erro e interromper a inserção ou atualização.
raise_application_error(-20000, 'Error Checking sal/ Salario acima do limite');

R.
CREATE OR REPLACE TRIGGER impede_insercao_salario
BEFORE INSERT ON piloto FOR EACH ROW
BEGIN
    IF :NEW.salario > 10000 THEN
        raise_application_error(-20000, 'Error Checking sal/ Salario acima do limite');
    END IF;
END;

CREATE OR REPLACE TRIGGER impede_atualizacao_salario
BEFORE UPDATE ON piloto FOR EACH ROW
BEGIN
    IF :NEW.salario > 10000 THEN
        raise_application_error(-20000, 'Error Checking sal/ Salario acima do limite');
    END IF;
END;
```

24

15. Stored Procedures

- Uma **Stored Procedure** ou simplesmente **Procedure** é um grupo de comandos SQL, que executa uma determinada tarefa.
- Diferente de uma **Trigger**, que é executado automaticamente, uma **Procedure** precisa ser chamada a partir de um programa ou manualmente pelo usuário.

25

15. Stored Procedures

- O uso de **Stored Procedure** traz uma série de benefícios:
 - **Reusabilidade de código** – Uma procedure pode ser usada por diversos usuários em diversas ocasiões, como um script SQL, dentro de uma trigger ou aplicação.
 - **Portabilidade** – Uma procedure é totalmente portátil dentro de plataformas nas quais o banco roda.
 - **Aumento de Performance** – Guardar a lógica de uma aplicação no próprio banco de dados diminui o tráfego na rede em um ambiente cliente/servidor.
 - **Manutenção centralizada** – Mantendo o código no servidor, uma alteração feita é imediatamente disponibilizada para todos os usuários. Isso evita o controle de versões de programas que são distribuídos pela empresa.

26

15. Stored Procedures

15.1. Sintaxe

- Uma stored procedure possui duas partes. Uma seção de especificação e o corpo da procedure. Vejamos a sintaxe do comando SQL responsável pela criação de procedures.

```
CREATE [OR REPLACE] PROCEDURE <nome_da_procedure>
[ ( lista de parâmetros ) ] IS
[ declarações ]
BEGIN
Comandos;
END [nome_da_procedure];
```

- **OR REPLACE** – Essa opção recria a função mantendo os privilégios previamente concedidos
- **Lista de parâmetros** – Se mais de um parâmetro for usado pela procedure devem ser separados por vírgula. Um parâmetro deve ser definido com a cláusula IN e inicializado como uma variável.
- **Declarações** – Nessa seção são declaradas constantes, variáveis e até mesmo outras procedures e funções locais.
- **Comandos** – Nessa seção são colocados os comandos que serão executados pela procedure.
- **IS** – Pode ser substituído por AS

27

15. Stored Procedures

15.2. Criando Procedure sem Argumento

- Criar uma procedure que aumenta o salário de todos os funcionários por um valor fixo de 10%. Essa stored procedure não exige nenhum parâmetro.

```
CREATE OR REPLACE PROCEDURE Aumenta_Salario
IS
BEGIN
    UPDATE Funcionario
    SET salario_funcionario = salario_funcionario * 1.1;
END;
```

28

15. Stored Procedures

15.2. Executando Procedure sem Argumento

- A execução de uma procedure, é feita através de uma chamada ao seu nome.
- Dentro de um trigger ou outra procedure basta especificar o nome da procedure e seus parâmetros.

```
EXECUTE Aumenta_Salario;
```

Como resposta tem-se:

Procedimento PL/SQL concluído com sucesso

29

15. Stored Procedures

15.3. Criando Procedure com Argumento

- Criar uma procedure para atualizar o salário de um determinado funcionário, o argumento podem ser do tipo básico de dados ou %TYPE.

```
CREATE OR REPLACE PROCEDURE Aumenta_Salario (Argumento
Funcionario.RG_Funcionario%TYPE)
IS
BEGIN
    UPDATE Funcionario
    SET salario_funcionario = salario_funcionario * 1.1
    WHERE rg_funcionario = Argumento;
END;
```

- Se existir mais argumentos estes deve ser separados por vírgula.

30

15. Stored Procedures

15.3. Executando Procedure com Argumento

- A execução de uma **procedure**, é feita através de uma chamada ao seu nome mais o argumentos necessários.
- Dentro de um trigger ou outra procedure basta especificar o nome da procedure e seus parâmetros.

```
EXECUTE Aumenta_Salario(1115);
```

Como resposta tem-se:

Procedimento PL/SQL concluído com sucesso

31

15. Stored Procedures

15.4. Manipulação

■ Mostrando Erros

Se ocorrer algum erro você pode usar o comando SHOW Errors para saber o que aconteceu.

```
SHOW Errors;
```

■ Apagando uma Stored Procedure

Para apagar uma stored procedure deve ser usado o comando Drop Procedure.

```
DROP PROCEDURE <nome_da_procedure>;
```

■ Listando os Procedimentos

```
SELECT name  
FROM USER_SOURCE;
```

32

15. Stored Procedures

15.5. Registros Atualizados

- Para habilitar a saída:

```
set serveroutput on;
```
- Após ao update é exibido uma mensagem de quantos registros foram alterados com o ultima comando.

```
CREATE OR REPLACE PROCEDURE Aumenta_Salario  
    (Argumento Funcionario.RG_funcionario%TYPE)  
IS  
BEGIN  
    UPDATE Funcionario  
    SET salario_funcionario = salario_funcionario * 1.1  
    WHERE rg_funcionario = Argumento;  
    if sql%rowcount > 0 then  
        dbms_output.put_line('Atualizou ' || to_char(sql%rowcount)  
        || ' registros');  
    else  
        dbms_output.put_line('Nenhum Registro Atualizado!');  
    end if;  
END;
```

33

15. Stored Procedures

- Resolver os exercícios de DML Procedure (ExercicioDML_Procedure.pdf)

34

15. Stored Procedures

- 1) Crie uma procedure que elimine um registro de piloto da tabela piloto pelo seu código.

R.

```
CREATE OR REPLACE PROCEDURE ELIMINA_PILOTO  
    (Argumento PILOTO.CODIGO_PILOTO%TYPE)  
IS  
BEGIN  
    DELETE FROM piloto  
    WHERE codigo_piloto = Argumento;  
END;
```

```
EXECUTE ELIMINA_PILOTO(1);
```

35

15. Stored Procedures

- 2) Crie uma procedure que insira um registro de piloto na tabela piloto.

R.

```
CREATE OR REPLACE PROCEDURE INSERE_PILOTO  
    (codigo PILOTO.CODIGO_PILOTO%TYPE,  
    nome PILOTO.NOME_PILOTO%TYPE,  
    salario PILOTO.SALARIO%TYPE,  
    gratificacao PILOTO.GRATIFICACAO%TYPE,  
    companhia PILOTO.COMPANHIA%TYPE,  
    pais PILOTO.PAIS%TYPE)  
IS  
BEGIN  
    INSERT INTO piloto(CODIGO_PILOTO, NOME_PILOTO, SALARIO,  
    GRATIFICACAO, COMPANHIA, PAIS)  
    VALUES(codigo,nome,salario,gratificação,companhia,pais);  
END;
```

```
EXECUTE INSERE_PILOTO(1,'joao',100,10,'Vai','Brasil');
```

36

15. Stored Procedures

- 3) Crie uma procedure que elimine as escalas de um determinado piloto. Exibindo ao final a quantidade registro atualizados, se houver.

```
R.
CREATE OR REPLACE PROCEDURE ELIMINA_ESCALA
  (Argumento PILOTO.CODIGO_PILOTO%TYPE)
IS
BEGIN
  DELETE FROM ESCALA
  WHERE CODIGO_PILOTO = Argumento;
  IF sql%rowcount > 0 THEN
    dbms_output.put_line(' Atualizou ' ||
      to_char(sql%rowcount) || ' registros ');
  END IF;
END;

EXECUTE ELIMINA_ESCALA(1);
```

37

15. Stored Procedures

- 4) Crie uma procedure que elimine todos os registros referentes a um determinado piloto (escala e piloto). Exibindo ao final a quantidade registro atualizados, se houver.

```
R.
CREATE OR REPLACE PROCEDURE ELIMINA_REGISTROS_PILOTO
  (Argumento PILOTO.CODIGO_PILOTO%TYPE)
IS
BEGIN
  ELIMINA_ESCALA(Argumento);
  ELIMINA_PILOTO(Argumento);
END;

EXECUTE ELIMINA_REGISTROS_PILOTO(1);
```

38

16. Funções

- Uma **função** é muito parecida com uma **Stored Procedure**.
- A principal diferença está no fato que uma função retorna um valor e a **Stored Procedure** não.
- Outra diferença é a forma como uma função pode ser chamada.
- Por retorna um valor, ela pode ser chamada através de um comando SELECT e também usada em cálculos como outra função qualquer.

39

16. Funções

16.1. Sintaxe

- A criação é feita através do comando **Create Function**

```
CREATE [OR REPLACE] FUNCTION <nome_da_funcao>
  [ ( lista de parâmetros ) ]
  RETURN Tipo_de_Dado IS
  [ declarações ]
BEGIN
  Comandos;
  RETURN Valor_da_Função;
END [nome_da_funcao];
```

OR REPLACE – Essa opção recria a função mantendo os privilégios previamente concedidos.
Lista de parâmetros – Se mais de um parâmetro for usado pela função devem ser separados por vírgula. Um parâmetro deve ser definido com a cláusula IN e inicializado como uma variável.
Declarações – Nessa seção são declaradas constantes, variáveis e até mesmo outras procedures e funções locais.
Comandos – Nessa seção são colocados os comandos que serão executados pela procedure.
Valor_da_Função – É uma constante ou variável que contém o valor retornado pela função.

40

16. Funções

16.2. Criando

- Vamos criar uma função que retorna o número(quantidade) de funcionários de um determinado departamento. Para isso, ela usará um argumento que recebe o código do departamento.

```
CREATE OR REPLACE FUNCTION QtdeFuncionario(Argumento
Funcionario.codigo_departamento%TYPE) RETURN int IS
  totalFuncionario int;
BEGIN
  SELECT COUNT(*) into totalFuncionario
  FROM Funcionario
  WHERE codigo_departamento = Argumento;
  RETURN totalFuncionario;
END;
```

41

16. Funções

16.2. Executando

- Para exibir o resultado de uma função, deve ser usado o comando SELECT especificando o nome da função e a tabela DUAL deve ser pesquisada. Veja o exemplo.

```
SELECT QtdeFuncionario(3)
FROM DUAL;
```

- Vamos selecionar a quantidade de funcionários por departamento que seja maior que a quantidade do departamento 2;

```
SELECT count(*), codigo_departamento
FROM funcionario
GROUP BY codigo_departamento
HAVING count(*) > QtdeFuncionario(2)
```

42

16. Funções

16.3. Manipulando

■ Apagando uma Função

Para apagar uma função deve ser usado o comando `Drop Function`.

```
DROP FUNCTION <nome_da_function>;
```

■ Listando as Funções criadas

```
SELECT name  
FROM USER_SOURCE;
```

■ Mostrando os erros na Função

```
SHOW Errors;
```

43

16. Funções

- Resolver os exercícios de DML Function (ExercicioDML_Function.pdf)

44

16. Funções

1) Crie uma função que retorna o dia de uma data;

R.

```
CREATE OR REPLACE FUNCTION dia_data(DATA date)  
RETURN number  
is  
BEGIN  
    RETURN TO_CHAR(DATA, 'DD');  
END;
```

```
SELECT dia_data(sysdate)  
FROM DUAL;
```

45

16. Funções

1) Crie uma função que retorna o dia de uma data;

R.

```
CREATE OR REPLACE FUNCTION dia_data(DATA date)  
RETURN number  
is  
BEGIN  
    RETURN TO_CHAR(DATA, 'DD');  
END;
```

```
SELECT dia_data(sysdate)  
FROM DUAL;
```

46

16. Funções

2) Crie uma função que retorna o mês de uma data;

R.

```
CREATE OR REPLACE FUNCTION mes_data(DATA date)  
RETURN number  
is  
BEGIN  
    RETURN TO_CHAR(DATA, 'MM');  
END;
```

```
SELECT mes_data(sysdate)  
FROM DUAL;
```

47

16. Funções

3) Crie uma função que retorna o ano de uma data;

R.

```
CREATE OR REPLACE FUNCTION ano_data(DATA date)  
RETURN number  
is  
BEGIN  
    RETURN TO_CHAR(DATA, 'YYYY');  
END;
```

```
SELECT ano_data(sysdate)  
FROM DUAL;
```

48

16. Funções

4) Crie uma função que retorne o literal 'Florianópolis, 10 de junho de 2009'. Deve ser passado para a função a data e o nome da cidade como argumento.

```
R.
CREATE OR REPLACE FUNCTION data_extenso (data date, cidade varchar) RETURN varchar
IS
    mesextenso varchar(100);
    saida varchar(200);
BEGIN
    saida := '';
    CASE mes_data(data)
        WHEN 1 then mesextenso := 'Janeiro';
        WHEN 2 then mesextenso := 'Fevereiro';
        WHEN 3 then mesextenso := 'Marco';
        WHEN 4 then mesextenso := 'Abril';
        WHEN 5 then mesextenso := 'Maio';
        WHEN 6 then mesextenso := 'Junho';
        WHEN 7 then mesextenso := 'Julho';
        WHEN 8 then mesextenso := 'Agosto';
        WHEN 9 then mesextenso := 'Setembro';
        WHEN 10 then mesextenso := 'Outubro';
        WHEN 11 then mesextenso := 'Novembro';
        WHEN 12 then mesextenso := 'Dezembro';
        ELSE mesextenso := 'mês inválido';
    END CASE;
    saida := cidade || ', ' || dia_data(data) || ' de ' || mesextenso || ' de ' || ano_data (data);
    RETURN saida;
END;

SELECT data_extenso(sysdate, 'São Miguel do Oeste') FROM dual;
```

49

16. Funções

5) Crie uma função que retorne a quantidade de pilotos.

```
R.
CREATE OR REPLACE FUNCTION qtde_piloto
RETURN NUMBER IS qtde NUMBER;
BEGIN
    SELECT COUNT(*) INTO qtde
    FROM piloto;
    RETURN qtde;
END;

SELECT qtde_piloto
FROM DUAL;
```

50

16. Funções

6) Crie uma função que retorne a quantidade de pilotos de uma companhia específica. Depois crie uma consulta que mostre a companhias que possuem mais pilotos que a Gol.

```
R.
CREATE OR REPLACE FUNCTION NumPilotos(argcomp piloto.companhia%type)
RETURN number IS qtde NUMBER;
BEGIN
    SELECT COUNT(*) INTO qtde
    FROM piloto
    WHERE upper(companhia) = upper(argcomp);
    RETURN qtde;
END;

SELECT NumPilotos('Tam')
FROM dual;

SELECT COMPANHIA
FROM PILOTO
GROUP BY COMPANHIA
HAVING COUNT(*) > NumPilotos('GOL');
```

51

17. Cursores

- Se um bloco **PL/SQL**, trigger, function ou stored procedure usa um comando **SELECT** que retorna mais de uma linha, o **SGBD** exibe uma mensagem de erro que invoca a exceção **TOO_MANY_ROWS***.
- Para contornar esse problema o **SGBD** usa um mecanismo chamado cursor.
- Um cursor pode ser visto como um arquivo temporário que armazena e controla as linhas retornadas por um comando **SELECT**.
- O SQL-Plus gera automaticamente cursores para as queries(consultas) executadas.

* Específico para Oracle

52

17. Cursores

17.1 Criando um Cursor

- A criação de um cursor envolve quatro etapas descritas a seguir.
 1. Declaração do cursor. Cria um nome para o cursor e atribui um comando **SELECT** a ele.
 2. Abertura do cursor. Executa a query associada ao cursor e determina quantas linhas serão retornadas.
 3. Fetching. As linhas (conteúdo) encontradas são enviadas para o programa **PL/SQL**.
 4. Fechamento do Cursor. Libera os recursos alocados para o cursor.

53

17. Cursores

17.2. Declaração de um Cursor

- Na declaração do cursor, você deve especificar o nome e a consulta que será executada.
- **Exemplo:**

```
DECLARE
CURSOR CRS2 IS
    SELECT nome_funcionario,
           codigo_departamento,
           salario_funcionario
    FROM empregado
    WHERE codigo_departamento = 2
    ORDER BY nome_funcionario;
```
- Não existe limitações para a quantidade de cursores criados, a não ser pela memória alocada para os cursores.

54

17. Cursores

17.2. Declaração de um Cursor

- Assim como uma **procedure** ou **function**, um cursor pode receber parâmetros.
- **Exemplo:**

```
DECLARE
  CURSOR CRS2(
    argDepto funcionario.codigo_departamento%type)
  IS
    SELECT nome_funcionario,
           codigo_departamento,
           salario_salario
    FROM funcionario
   WHERE codigo_departamento = argDepto
   ORDER BY nome_funcionario;
```

55

17. Cursores

17.3. Abertura e Fechamento

- A abertura do cursor é a operação que executa a query e cria o active set(conjunto ativo), que é o grupo de linhas que satisfaz a condição da query. Um cursor é aberto pelo comando Open.

- **Exemplo:**

```
OPEN CRS2;
```

- ☐ O cursor deixa a primeira linha como linha atual.

- Quando um cursor não estiver sendo utilizado, ele deve ser fechado para que libere os recursos que estavam sendo alocados para ele.

- **Exemplo:**

```
CLOSE CRS2;
```

56

17. Cursores

17.4. Atributos de um Cursor

- **%ISOPEN** – Esse atributo retornará True se o cursor já estiver aberto.
- **%NOTFOUND** – Retorna True quando a última linha do cursor é processada e nenhuma outra está disponível.
- **%FOUND** – Funciona de maneira oposta a %NOTFOUND.
- **%ROWCOUNT** – Retorna o número total de linhas retornadas pelo comando FETCH. Cada vez que o comando FETCH for executado, %ROWCOUNT aumenta em 1.

- Específico para Oracle

57

17. Cursores

17.5. Acessando uma Linha do Cursor

- **Exemplo**

```
FETCH CRS2 INTO temp_nome,
              temp_codigo_depto, temp_salario;
LOOP
  EXIT WHEN CRS2%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' ||
                        TO_CHAR(temp_codigo_depto)
                        || ' ' || TO_CHAR(temp_salario));
END LOOP;
```

58

17. Cursores

17.5. Acessando várias Linhas do Cursor

- **Exemplo;**

```
LOOP
  FETCH CRS2 INTO temp_nome,
                temp_codigo_depto, temp_salario;
  EXIT WHEN CRS2%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' ||
                        TO_CHAR(temp_codigo_depto)
                        || ' ' || TO_CHAR(temp_salario));
END LOOP;
```

59

17. Cursores

17.6. Exemplo sem Argumento

```
CREATE OR REPLACE PROCEDURE LOOPCRS2 IS
  temp_nome funcionario.nome_funcionario%TYPE;
  temp_codigo_depto funcionario.codigo_departamento%TYPE;
  temp_salario funcionario.salario_funcionario%TYPE;
  CURSOR CRS2 IS
    SELECT nome_funcionario,codigo_departamento,salario_funcionario
    FROM funcionario
   WHERE codigo_departamento = 2
   ORDER BY nome_funcionario;
BEGIN
  OPEN CRS2;
  LOOP
    FETCH CRS2 INTO temp_nome, temp_codigo_depto, temp_salario;
    EXIT WHEN CRS2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' ||
                        TO_CHAR(temp_codigo_depto)
                        || ' ' || TO_CHAR(temp_salario));
  END LOOP;
  CLOSE CRS2;
END;
```

60

17. Cursores

17.6. Exemplo com Argumento

```
CREATE OR REPLACE PROCEDURE
LOOPCRS2(argDepto funcionario.codigo_departamento%TYPE) IS
    temp_nome funcionario.nome_funcionario%TYPE;
    temp_codigo_depto funcionario.codigo_departamento%TYPE;
    temp_salario funcionario.salario_funcionario%TYPE;
    CURSOR CRS2(aDepto funcionario.codigo_departamento%TYPE) IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = aDepto
        ORDER BY nome_funcionario;
BEGIN
    OPEN CRS2(argDepto);
    LOOP
        FETCH CRS2 INTO temp_nome, temp_codigo_depto, temp_salario;
        EXIT WHEN CRS2%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' ||
            TO_CHAR(temp_codigo_depto) || ' ' ||
            TO_CHAR(temp_salario));
    END LOOP;
    CLOSE CRS2;
END;
```

61

17. Cursores

17.7. Usando o FOR...LOOP

- Uma alternativa mais simples para exibir as linhas retornadas por um cursor e evitar o trabalho manual de abrir, atribuir e fechar um cursor é o comando **FOR...LOOP**.
- Ao usar esse cursor, o Oracle automaticamente declara uma variável como o mesmo nome da variável usada como contador do comando **FOR**, que é do mesmo tipo de variável criada pelo cursor.
- Basta preceder o nome do campo selecionado com o nome dessa variável para ter acesso ao seu conteúdo.

62

17. Cursores

17.7. Usando o FOR...LOOP sem Argumento

```
CREATE OR REPLACE PROCEDURE FORCRS2 IS
    CURSOR CRS2 IS
        SELECT nome_funcionario, codigo_departamento,
            salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = 2
        ORDER BY nome_funcionario;
BEGIN
    FOR X IN CRS2 LOOP
        DBMS_OUTPUT.PUT_LINE(X.nome_funcionario
            || ' ' || TO_CHAR(X.codigo_departamento)
            || ' ' || TO_CHAR(X.salario_funcionario));
    END LOOP;
END;
```

63

17. Cursores

17.7. Usando o FOR...LOOP com Argumento

```
CREATE OR REPLACE PROCEDURE
FORCRS2(argDepto funcionario.codigo_departamento%TYPE) IS
    CURSOR CRS2(aDepto funcionario.codigo_departamento%TYPE) IS
        SELECT nome_funcionario, codigo_departamento,
            salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = aDepto
        ORDER BY nome_funcionario;
BEGIN
    FOR X IN CRS2(argDepto) LOOP
        DBMS_OUTPUT.PUT_LINE(X.nome_funcionario
            || ' ' || TO_CHAR(X.codigo_departamento)
            || ' ' || TO_CHAR(X.salario_funcionario));
    END LOOP;
END;
```

64

17. Cursores

- Resolver os exercícios de DML Cursores (ExercicioDML_Cursor.pdf)

65

17. Cursores

- 1) Crie um procedimento que utilize um cursor para percorrer todos os pilotos e exibir o nome.
- R.
- Set serveroutput on;

```
CREATE OR REPLACE PROCEDURE nomes_pilotos is
    CURSOR percorre_piloto is
        SELECT codigo_piloto, nome_piloto
        FROM piloto
        ORDER BY nome_piloto;
BEGIN
    dbms_output.enable;
    FOR x in percorre_piloto LOOP
        dbms_output.put_line('Piloto : ' || x.nome_piloto);
    END LOOP;
END;
```

execute nomes_pilotos;

66

17. Cursores

2) Crie um procedimento que utilize um cursor para percorrer todos os pilotos de uma companhia e exibir o seu nome.

R.

```
CREATE OR REPLACE PROCEDURE nomes_pilotos_companhia (ARGUMENTO
piloto.companhia%TYPE) IS
CURSOR CRS (ARG piloto.companhia%TYPE) IS
SELECT nome_piloto
FROM piloto
WHERE companhia = ARG
ORDER BY nome_piloto;
BEGIN
dbms_output.enable;
FOR X IN CRS (ARGUMENTO) LOOP
DBMS_OUTPUT.PUT_LINE (X.nome_piloto);
END LOOP;
END;
```

execute nomes_pilotos_companhia('Tam');

67

17. Cursores

3) Crie um procedimento que utilize um cursor para percorrer todas as escalas e exibir todos os dados.

R.

```
CREATE OR REPLACE PROCEDURE todas_escalas IS
CURSOR CRS IS
SELECT *
FROM escala
ORDER BY codigo_voo;
BEGIN
dbms_output.enable;
FOR X IN CRS LOOP
DBMS_OUTPUT.PUT_LINE (X.codigo_voo||' '||X.data_voo||'
'||X.codigo_piloto||' '||X.AVIAO);
END LOOP;
END;
```

execute todas_escalas;

68

17. Cursores

4) Crie um procedimento que utilize um cursor para percorrer todas as escalas de um piloto.

R.

```
CREATE OR REPLACE PROCEDURE toda_escala_piloto (ARGUMENTO
piloto.codigo_piloto%TYPE) IS
CURSOR CRS (ARG piloto.codigo_piloto%TYPE) IS
SELECT *
FROM escala
WHERE codigo_piloto = ARG
ORDER BY codigo_voo;
BEGIN
dbms_output.enable;
FOR X IN CRS (ARGUMENTO) LOOP
DBMS_OUTPUT.PUT_LINE (X.codigo_voo||' '||X.data_voo||'
'||X.codigo_piloto||' '||X.AVIAO);
END LOOP;
END;
```

execute toda_escala_piloto(5);

69

17. Cursores

5) Crie um procedimento que utilize cursores que mostre para todos os pilotos os nomes dos seus aviões. O nome dos pilotos não podem se repetir

R.

```
CREATE OR REPLACE PROCEDURE piloto_aviao is
CURSOR percorre_piloto is
SELECT codigo_piloto, nome_piloto
FROM piloto;

CURSOR percorre_escala (ARGUMENTO piloto.codigo_piloto%TYPE) is
SELECT aviao
FROM escala
WHERE codigo_piloto = ARGUMENTO;
BEGIN
dbms_output.enable;
FOR x in percorre_piloto LOOP
dbms_output.put_line('Piloto : '||x.nome_piloto);
dbms_output.put_line('Avião: ');
FOR y in percorre_escala(x.codigo_piloto) LOOP
dbms_output.put_line('-----> : '||y.aviao);
END LOOP;
END LOOP;
END;
```

execute piloto_aviao;

70

17. Cursores

11) Crie um procedimento que utilize cursor para atualizar o salário do piloto em 10% se o salário for maior que a gratificação, caso contrario atualize em 5% a gratificação;

R.

```
CREATE OR REPLACE PROCEDURE atualiza_salario IS
CURSOR atualiza_salario IS
SELECT codigo_piloto, salario, gratificacao
FROM piloto;
BEGIN
FOR x IN atualiza_salario LOOP
IF x.salario > x.gratificacao THEN
UPDATE piloto SET salario = (salario*1.1)
WHERE codigo_piloto = x.codigo_piloto;
ELSE
UPDATE piloto SET gratificacao = (gratificacao*1.05)
WHERE codigo_piloto = x.codigo_piloto;
END IF;
END LOOP;
END;
```

Execute atualiza_salario;

71

18. Tabelas Temporárias

- Existe algumas situações em que é necessário criar relatórios ou exibir informações, e para isto resgata-se os dados realiza-se alguma manipulação e apresenta-se o resultado ao usuário.
- Após apresentar o resultado pode-se descartar a "manipulação" realizada uma vez que esta não precisa ser persistida e deve ser refeita a cada solicitação.
- É muito comum nestes casos utilizar uma tabela física para realizar a manipulação. Os desenvolvedores criam tabelas, carregam os dados, fazem as manipulações necessárias, utilizam as informações e depois apagam os dados e ou as tabelas criadas.
- Não há a necessidade de trabalhar com tabelas físicas para trabalhar com dados temporários. Neste caso deve-se criar tabelas temporárias. Com isto você evita que o banco de dados registre informações das operações realizadas nestas tabelas.

72

18. Tabelas Temporárias

- Uma tabela temporária é uma tabela com vida útil de uma sessão ou transação.
- Ela está vazia quando a sessão ou transação começa e descarta os dados ao fim da sessão ou transação.
- Uma tabela temporária é associada à transação. Isto significa que ao término da transação os dados da tabela são perdidos, porém sua descrição permanece gravada no banco de dados mesmo após a mudança de sessão.
- As tabelas temporárias podem ser de dois tipos, **sessão** ou **transação**.
 - Nas tabelas temporárias sessão os dados são visíveis durante a sessão que os criou.
 - A tabela temporária de transação preserva os dados somente durante a transação do usuário, após isto os dados são apagados.

73

18. Tabelas Temporárias

- Criando uma tabela temporária de **sessão** para cliente é necessário alterar para **ON COMMIT PRESERVE ROWS**. Os dados são apagados após a finalização da sessão do usuário (**Log-out**).

```
CREATE GLOBAL TEMPORARY TABLE TempCliente(  
  cpf_cliente VARCHAR(11),  
  saldo numeric(9,2),  
  PRIMARY KEY (cpf_cliente)) ON COMMIT PRESERVE ROWS;  
  
INSERT INTO TempCliente VALUES('1',10);  
  
COMMIT;  
  
SELECT * FROM TempCliente;
```

74

18. Tabelas Temporárias

- Para criar uma tabela temporária para **transação** é necessário alterar para **ON COMMIT DELETE ROWS**. Com isto após o commit os dados da tabela são apagados.

```
CREATE GLOBAL TEMPORARY TABLE TempCliente(  
  cpf_cliente VARCHAR(11),  
  saldo numeric(9,2),  
  PRIMARY KEY (cpf_cliente)) ON COMMIT DELETE ROWS;  
  
INSERT INTO TempCliente VALUES('1',10);  
  
COMMIT;  
  
SELECT * FROM TempCliente;
```

75

19. Exceções

- São usadas no PL/SQL para lidar com quaisquer erros que ocorram durante a execução de um bloco.
 - Dois tipos de exceções
 - definidas internamente pela PL/SQL
 - definidas pelo usuário.
- Quando ocorre um erro no bloco PL/SQL e existe uma exception definida, o controle de execução do bloco passa para a exception que executará os comandos nela contidos, retornando para o fim do bloco onde ocorreu o erro.

76

19. Exceções

19.1. Exceções Definidas Internamente(Oracle)

- Algumas Exceções Internas
 - DUP_VAL_ON_INDEX / Tentativa de gravação de chave duplicada para índice único
 - INVALID_CURSOR / Operação com cursor ilegal. Ex. Fechar cursor não aberto.
 - INVALID_NUMBER / Conversão de caracter para numérico falha.
 - LOGIN_DENIED/ Nome do usuário ou senha inválida
 - NO_DATA_FOUND / Select não retorna nenhuma linha
 - NOT_LOGGED_ON / PL/SQL emite uma chamada ao oracle sem estar conectado
 - OTHERS / Qualquer tipo de erro
 - PROGRAM_ERROR / PL/SQL tem um problema interno
 - STORAGE_ERROR / PL/SQL não tem memória suficiente para rodar ou memória está danificada
 - TIMEOUT_ON_RESOURCE / Decurso de tempo enquanto o oracle espera por um recurso
 - TOO_MANY_ROWS / Select retorna mais de uma linha
 - TRANSACTION_BACKED_OUT / Oracle volta atrás uma transação por causa de erro de processamento.
 - VALUE_ERROR / Ocorrência de erro em expressões aritméticas, conversões e truncamentos
 - ZERO_DIVIDE / Tentativa de divisão por zero

77

19. Exceções

19.1. Exceções Definidas Internamente

Sintaxe:

```
DECLARE  
  <declarações>  
BEGIN  
  <comandos>  
  EXCEPTION  
    WHEN <nome_exception> THEN  
      <comandos><INSERT,UPDATE,DELETE,  
        SELECT,rollback,commit,null>  
      <funções><sqlcode,sqlerrm>  
    WHEN OTHERS THEN  
      <comandos>  
      <funções>  
END;
```

78

19. Exceções

19.1. Exceções Definidas Internamente

Exemplo Exceção Interna em Bloco Anônimo

```
DECLARE
    rg_temp funcionario.rg_funcionario%TYPE;
BEGIN
    SELECT rg_funcionario INTO rg_temp
    FROM funcionario
    WHERE UPPER(nome_funcionario) = UPPER('Joao');
    DBMS_OUTPUT.PUT_LINE('Um João encontrado');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('João não encontrado');
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE('Mais de um João');
        WHEN OTHERS THEN
            ROLLBACK;
END;
```

79

19. Exceções

19.1. Exceções Definidas Internamente

Exemplo Exceção Interna em Procedure

```
CREATE OR REPLACE PROCEDURE TesteExcecao IS
    rg_temp funcionario.rg_funcionario%TYPE;
BEGIN
    SELECT rg_funcionario INTO rg_temp
    FROM funcionario
    WHERE UPPER(nome_funcionario) = UPPER('Joao');
    DBMS_OUTPUT.PUT_LINE('Um João encontrado');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('João não encontrado');
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE('Mais de um João');
        WHEN OTHERS THEN
            ROLLBACK;
END;
```

80

19. Exceções

19.2. Exceções Definidas pelo Usuário

Sintaxe:

```
DECLARE
    <declarações>
    <nome_da_exceção> EXCEPTION;
BEGIN
    <comandos>
    if <condição> then
        RAISE <nome_da_exceção>;
    <comandos>
    EXCEPTION
        WHEN <nome_da_exceção> THEN
            <comandos>
END;
```

81

19. Exceções

19.3. Função RAISE_APPLICATION_ERROR

- É uma função interna e mostra uma mensagem pelo mesmo caminho dos erros do Oracle.
- É um número negativo entre -20000 até -20999
- A mensagem de erro não pode exceder 512 caracteres.

Sintaxe:

```
RAISE_APPLICATION_ERROR (error_number, error_message);
```

82

19. Exceções

19.3. Função RAISE_APPLICATION_ERROR

Exemplo:

```
DECLARE
    nome VARCHAR(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Digite o seu nome');
    nome := '&nome';
    IF length(nome) > 5 THEN
        RAISE_APPLICATION_ERROR(-20000, 'Nome Longo');
    ELSE
        RAISE_APPLICATION_ERROR(-20001, 'Nome Curto');
    END IF;
END;
```

83

19. Exceções

- Resolver os exercícios de DML Exception (ExercicioDML_Exception.pdf)

84

20. Packages

- Uma package (pacote) é uma coleção de objetos de banco de dados como stored procedures, funções, exceptions, variáveis, constantes e cursors.
- Pode conter subprogramas que podem ser chamados a partir de uma trigger, procedure ou função.
- Aliado do desenvolvedor, pois permite organizar melhor os componentes de um sistema em módulos.
- Administração de privilégios facilitada.
- Duas seções,
 - seção de especificação
 - corpo (body) (precisam ser criadas separadamente).

85

20. Packages

20.1 Seção de Especificação

- Funciona como uma espécie de sumário do conteúdo do corpo do package.
- Nessa seção são declarados os nomes de funções e stored procedure juntamente com nomes de variáveis e constantes, incluindo aí sua inicialização.
- A seção de especificação é criada com o comando **CREATE PACKAGE**, cuja a sintaxe é exibida a seguir:

```
CREATE [OR REPLACE] PACKAGE <nome_package> IS
    [seção de declaração]
END;
```

86

20. Packages

20.2 Corpo (Body)

- Contém a definição formal de todos os objetos referenciados na seção de declaração.
- A criação dessa seção é feita através do comando **CREATE PACKAGE BODY**.

```
CREATE [OR REPLACE] PACKAGE BODY <nome_package> IS
    [seção de declaração]
    [definição de procedures]
    [definição de funções]
    [seção de inicialização]
END;
```

87

20. Packages

20.3 Comandos

- Recompilando um Package
 - ALTER PACKAGE <nome_package> COMPILE BODY;
- Apagando um Package
 - DROP PACKAGE [BODY] <nome_package>;
- Listando as Especificações de Package
 - SELECT DISTINCT(NAME) FROM USER_SOURCE WHERE TYPE='PACKAGE';
- Listando os Corpos de Package
 - SELECT DISTINCT(NAME) FROM USER_SOURCE WHERE TYPE='PACKAGE BODY';

88

20. Packages

20.4 Exemplo parte 1

- Criando a Package Especificação

```
CREATE OR REPLACE PACKAGE APPS_RH IS
    PROCEDURE AUMENTA_SALARIO(
        DEPTO funcionario.codigo_departamento%TYPE,
        PERCENTUAL NUMBER);

    FUNCTION QTDE_FUNCIONARIO(
        DEPTO funcionario.codigo_departamento%TYPE)
        RETURN NUMBER;
END;
```

89

20. Packages

20.4 Exemplo parte 2

- Criando a Package Body

```
CREATE OR REPLACE PACKAGE BODY APPS_RH IS
    PROCEDURE AUMENTA_SALARIO(DEPTO funcionario.codigo_departamento%TYPE,
        PERCENTUAL NUMBER) IS
    BEGIN
        UPDATE funcionario
        SET salario.funcionario=salario.funcionario*(1+PERCENTUAL/100)
        WHERE codigo_departamento = DEPTO;
    END;

    FUNCTION QTDE_FUNCIONARIO(DEPTO funcionario.codigo_departamento%TYPE)
        RETURN NUMBER IS
        TOTAL NUMBER;
    BEGIN
        SELECT COUNT(*) INTO TOTAL FROM funcionario
        WHERE codigo_departamento = DEPTO;
        RETURN TOTAL;
    END;
END;
```

90

20. Packages

20.4 Exemplo parte 3

- Para acessar um objeto especificado dentro de um package, basta, precedê-lo com o nome do package.

```
SQL> execute APPS_RH.AUMENTA_SALARIO(20,10);  
PL/SQL Procedure successfully completed;
```

91

20. Packages

- Resolver os exercícios de DML Package (ExercicioDML_Package.pdf)

92

21. Gerenciamento de Usuários

- Para acessar um banco de dados é necessário ser previamente cadastrado.
- Ter estabelecido privilégios com relação às tarefas que poderá executar.
- Controlar o acesso ao banco de dados é uma das principais tarefas de um administrador de banco de dados.
- Cada usuário cadastrado recebe uma senha de acesso que precisa ser fornecida em diversas situações.
- Para cada usuário são atribuídos privilégios individuais ou um papel(role) que consiste de um grupo de privilégios que podem ser atribuídos de uma vez ao usuário que recebe aquele papel.

93

21. Gerenciamento de Usuários

21.1. Privilégio

- Consiste em uma autorização para executar um tipo de operação.
- Características:
 - Disponibilizam e restringem o acesso aos dados;
 - Disponibilizam e restringem a execução de comandos;
 - Podem ser concedidos ou revogados a qualquer tempo.
- Tipos:
 - Privilégio do Sistema
 - Privilégio de Objeto

94

21. Gerenciamento de Usuários

21.1.1. Privilégio do Sistema

- Um privilégio de sistema é o direito ou permissão de executar uma ação no banco de dados em um tipo específico de objeto de banco de dados.
- Existem mais de 70 tipos de privilégios associados a ações de banco de dados.
- O nome do privilégio é praticamente o nome da ação que ele executa

95

21. Gerenciamento de Usuários

21.1.1. Privilégio do Sistema

Privilégios	Comando SQL Permitido
CREATE SESSION	Permite usuário conectar-se ao banco
CREATE TABLE	Permite o usuário criar tabelas
ALTER ANY TABLE	Permite o usuário alterar tabelas em qualquer schema
CREATE VIEW	Permite o usuário criar visões
CREATE SYNONYM	Permite o usuário criar sinônimos
CREATE PROCEDURE	Permite o usuário criar procedures
SELECT ANY TABLE	Permite o usuário executar query em tabelas de qualquer schema
CREATE TRIGGER	Permite o usuário criar trigger

- Outros privilégios:
 - DELETE ANY TABLE, DROP ANY TABLE, DROP TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, CREATE ANY SYNONYM, CREATE PUBLIC SYNONYM, DROP ANY SYNONYM, DROP PUBLIC SYNONYM, etc..

- Exemplo:

```
GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW, CREATE SYNONYM,  
EXECUTE ANY PROCEDURE, CREATE PROCEDURE TO ADMINISTRADOR;
```

96

21. Gerenciamento de Usuários

21.1.2. Privilégio de Objeto

- São autorizações que permitem um usuário manipular dados de uma Tabela ou View, Sequence – alterar suas estruturas, executar Triggers ou Procedures armazenadas – podendo repassar essa autorização a outros usuários.
- O privilégio de objeto é o direito de executar uma determinada ação em um objeto específico, como, por exemplo, o direito de incluir uma linha em uma tabela determinada.
- Os privilégios de objeto não se aplicam a todos os objetos de banco de dados. Triggers, Procedures e Indexes não possuem privilégios de objeto.

97

21. Gerenciamento de Usuários

21.1.2. Privilégio de Objeto

Privilégios	Comando SQL Permitido
ALTER	Alter object(tabela ou sequencia) Create trigger em tabela
EXECUTE	Execute procedure
DELETE	Delete from (tabela ou visão)
INDEX	Create index on tabela
REFERENCES	Create or alter table definindo foreign key
SELECT	Select...
UPDATE	Update...
INSERT	Inserir
ALL	Todos os acima

- Exemplo:
`GRANT SELECT, UPDATE(NOME_CLIENTE),
REFERENCES(NOME_CIDADE) ON CLIENTE TO ADMINISTRADOR;`

98

21. Gerenciamento de Usuários

21.2. Papel(Role)

- Um papel é um grupo de privilégios(Sistema e/ou Objetos), e outros papéis que são associados a um nome que os identifica e podem ser atribuídos a um usuário ou a outro papel.
- Você pode criar um papel que recebeu oito privilégios e, em seguida, atribuir o papel ao usuário.
- Para criar um papel o usuário tem que ter o privilégio **CREATE ROLE**

99

21. Gerenciamento de Usuários

21.2. Papel(Role)

■ Características

- ☐ Não tem proprietário
- ☐ Pode ser dado Grant para qualquer usuário ou para outro papel, exceto para ele mesmo
- ☐ Pode ser ativado ou desativado por usuário

■ Utilização

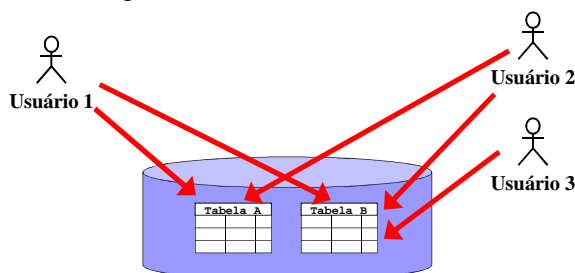
- ☐ Dar segurança a base de dados
- ☐ Controlar usuários por grupo de afinidades

100

21. Gerenciamento de Usuários

21.2. Papel(Role)

- Privilégios a usuários sobre tabelas:

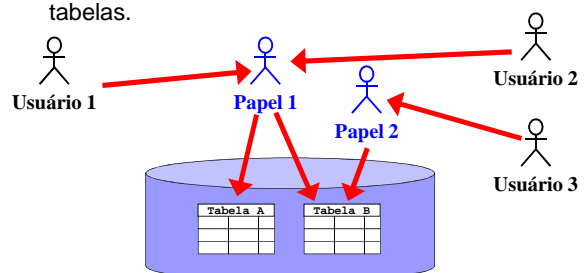


101

21. Gerenciamento de Usuários

21.2. Papel(Role)

- Privilégios usuário sobre papéis, e papéis sobre tabelas.



102

21. Gerenciamento de Usuários

21.3. Criando Usuário

- O comando CREATE USER é o responsável pela criação de novos usuários.
- Sintaxe:

```
CREATE USER <nome_do_usuario> IDENTIFIED BY <senha> /  
EXTERNALLY  
DEFAULT TABLESPACE <nome_do_tablespace>  
TEMPORARY TABLESPACE <nome_do_tablespace>  
QUOTA <numeroK> / <numeroM> UNLIMITED ON TABLESPACE  
<nome_do_tablespace>  
PROFILE <nome_do_profile>
```
- IDENTIFIED BY – Deve ser seguido da senha do usuário ou da palavra EXTERNALLY para indicar que banco de dados deve procurar a senha no sistema operacional.
- DEFAULT TABLESPACE – Identifica a tablespace usada para os objetos do usuário. Se omitido, o banco de dados assume a tablespace system.
- TEMPORARY TABLESPACE – Identifica a tablespace usada para os objetos temporários do usuário.
- QUOTA – Especifica a quantidade máxima de espaço na tablespace em Kbytes ou Mbytes que o usuário terá.
- PROFILE – Atribui os valores armazenados no profile especificado para o usuário. Se omitido, assume o profile chamado Default. Um profile é um arquivo que contém limites de uso do banco de dados para um usuário. Assim como um papel, um profile pode ser utilizado por vários usuários.

103

21. Gerenciamento de Usuários

21.3. Criando Usuário

- Para criar um usuário, você tem que possuir os **privilégios** adequados.
- Quando é criado um usuário no Oracle, ele simultaneamente, cria um **Schema** (de mesmo nome) para este usuário.
 - O Schema é um espaço do usuário para armazenar os seus objetos
- O exemplo mostra o usuário **System** criando um usuário na forma mais simples do comando.

```
SQL> connect  
Enter user-name: system  
Enter password: *****  
Connected.  
SQL> CREATE USER joao IDENTIFIED BY silva;  
User created;
```

104

21. Gerenciamento de Usuários

21.4. Alterando Usuário

- O comando ALTER USER é o responsável pela alteração de usuários.
- Sintaxe:

```
ALTER USER <nome_do_usuario> IDENTIFIED BY <senha> /  
EXTERNALLY  
DEFAULT TABLESPACE <nome_do_tablespace>  
TEMPORARY TABLESPACE <nome_do_tablespace>  
QUOTA <numeroK> / <numeroM> UNLIMITED ON TABLESPACE  
<nome_do_tablespace>  
PROFILE <nome_do_profile>
```
- IDENTIFIED BY – Deve ser seguido da senha do usuário ou da palavra EXTERNALLY para indicar que banco de dados deve procurar a senha no sistema operacional.
- DEFAULT TABLESPACE – Identifica a tablespace usada para os objetos do usuário. Se omitido, o banco de dados assume a tablespace system.
- TEMPORARY TABLESPACE – Identifica a tablespace usada para os objetos temporários do usuário.
- QUOTA – Especifica a quantidade máxima de espaço na tablespace em Kbytes ou Mbytes que o usuário terá.
- PROFILE – Atribui os valores armazenados no profile especificado para o usuário. Se omitido, assume o profile chamado Default. Um profile é um arquivo que contém limites de uso do banco de dados para um usuário. Assim como um papel, um profile pode ser utilizado por vários usuários.

105

21. Gerenciamento de Usuários

21.4. Alterando Usuário

- Para alterar um usuário, você tem que possuir os **privilégios** adequados.
- O exemplo mostra o usuário **System** alterando o usuário joao.

```
SQL> connect  
Enter user-name: system  
Enter password: *****  
Connected.  
SQL> ALTER USER joao IDENTIFIED BY joao;  
User altered;
```

106

21. Gerenciamento de Usuários

21.5. Criando Papel

- O comando SQL usado para criar um papel é o CREATE ROLE.
 - Sintaxe básica:

```
CREATE ROLE <nome_do_papel>  
{NOT IDENTIFIED  
| IDENTIFIED {BY <senha> | EXTERNALLY}}}
```
 - IDENTIFIED BY – Solicita uma senha de verificação para o usuário. Pode ser fornecida a senha do banco de dados ou usar a opção EXTERNALLY para obter a senha no sistema operacional.
- ```
SQL> CREATE ROLE GRUPO;
Role created;
```

107

## 21. Gerenciamento de Usuários

### 21.6. Alterando Papéis

- O comando SQL usado para alterar um papel é o ALTER ROLE.
- Sintaxe básica:

```
ALTER ROLE <nome_do_papel>
{NOT IDENTIFIED
| IDENTIFIED {BY <senha> | EXTERNALLY}}}
```
- IDENTIFIED BY – Solicita uma senha de verificação para o usuário. Pode ser fornecida a senha do banco de dados ou usar a opção EXTERNALLY para obter a senha no sistema operacional.

108

## 21. Gerenciamento de Usuários

### 21.7. Concedendo Privilégios e Papéis a um Papel

#### ■ Sintaxe básica:

```
GRANT <nome_papel>/<nome_do_privilegio> TO
<nome_do_usuario>/<nome_papel>
PUBLIC WITH ADMIN OPTION;
```

- O próximo exemplo concede os papéis **CONNECT** e **RESOURCE** para o papel **GRUPO**.

```
SQL> GRANT CONNECT, RESOURCE TO GRUPO;
Grant succeeded;
```

109

## 21. Gerenciamento de Usuários

### 21.8. Concedendo um Papel a um Usuário

- O usuário João foi criado, mas não possui nenhum privilégio ou papel atribuído a ele. Usa-se o comando **GRANT** para atribuir o papel **GRUPO** a ele.

```
SQL> GRANT GRUPO TO joao;
Grant succeeded;
```

```
SQL> Connect
Enter user-name: joão
Enter password: *****
Connect
SQL>
```

110

## 21. Gerenciamento de Usuários

### 21.9. Concedendo um Privilégio de Objeto para um Usuário

- Uma variação do comando **GRANT** permite atribuir privilégios de objeto para um usuário ou papel.
- Os privilégios concedidos podem ser: **ALTER**, **DELETE**, **EXECUTE**, **INSERT**, **INDEX**, **REFERENCES** e **UPDATE**.
- Os objetos que podem conceder privilégios são tabelas, visões, seqüências e sinônimos

111

## 21. Gerenciamento de Usuários

### 21.9. Concedendo um Privilégio de Objeto para um Usuário

- Sintaxe básica:

```
GRANT {<nome_do_privilegio>|ALL [PRIVILEGES]}
[(coluna[,coluna]...)]
[,{<nome_do_privilegio>|ALL [PRIVILEGES]}
[(coluna[,coluna]...)]...
ON [schema.]objeto
TO {<nome_do_usuario>|<nome_do_papel>|PUBLIC}
[,{<nome_do_usuario>|<nome_do_papel>|PUBLIC}]...
[WITH GRANT OPTION];
```

- **ALL PRIVILEGES** – Atribui todos os privilégios do objeto.
- **(<COLUNAS>)** – Especifica as colunas para as quais o privilégio(somente **INSERT**, **UPDATE** e **REFERENCE**) está sendo concedido.
- **PUBLIC** – Concede o privilégio para todos os usuários do banco de dados.
- **GRANT OPTION** – Permite que o usuário/papel que recebe o privilégio possa concedê-lo a outros usuários.

112

## 21. Gerenciamento de Usuários

### 21.9. Concedendo um Privilégio de Objeto para um Usuário

- O exemplo abaixo mostra o usuário Scott concedendo o privilégio **SELECT** para o usuário João acessar a tabela dept.

```
SQL> Connect
Enter user-name: Scott
Enter password: *****
Connect
SQL> GRANT SELECT ON SCOTT.DEPT TO grupo;
Grant succeeded;
```

```
SQL> Connect
Enter user-name: joão
Enter password: *****
Connect
SQL> SELECT * FROM SCOTT.DEPT;
DEPTO DNAME LOC

10 ACCOUNTING NEW YORK
```

113

## 21. Gerenciamento de Usuários

### 21.10. Visualizando os Papéis de um Usuário

- O banco de dados possui algumas tabelas especiais para controlar os privilégios e papéis.

- **USER\_ROLE\_PRIVS**
  - Privilégios do usuários atual.
- **DBA\_ROLES**
  - Todos os papéis definidos no banco.
- **DBA\_ROLE\_PRIVS**
  - Papéis atribuídos para usuários e papéis.
- **ROLE\_SYS\_PRIVS**
  - Privilégios de sistema dos papéis
- **ROLE\_TAB\_PRIVS**
  - Privilégios de tabela dos papéis.
- **ROLE\_ROLE\_PRIVS**
  - Papéis que são atribuídos a papéis.
- **SESSION\_ROLES**
  - Papéis ativados para o usuário atual.

114

## 21. Gerenciamento de Usuários

### 21.10. Visualizando os Papéis de um Usuário

- A tabela **USER\_ROLE\_PRIVS** exibe os papéis atribuídos ao usuário atual.

```
SQL> SELECT * FROM USER_ROLE_PRIVS;
```

| USERNAME | GRANTED_ROLE | ADM | DEF | OS_ |
|----------|--------------|-----|-----|-----|
| JOAO     | CONNECT      | NO  | YES | NO  |
| JOAO     | RESOURCE     | NO  | YES | NO  |

115

## 21. Gerenciamento de Usuários

### 21.10. Visualizando os Papéis

- A tabela **DBA\_ROLES** exibe todos os papéis definidos no banco de dados.

```
SQL> SELECT * FROM DBA_ROLES;
```

| ROLE     | PASSWORD_REQUIRED |
|----------|-------------------|
| CONNECT  | NO                |
| RESOURCE | NO                |
| DBA      | NO                |
| GRUPO    | NO                |
| ...      |                   |

116

## 21. Gerenciamento de Usuários

### 21.11. Apagando um Usuário

- A remoção de um usuário do banco de dados é feita pelo comando **DROP USER**.
- Com a opção **CASCADE** ele remove tanto o usuário como todos os objetos contidos no esquema do usuário e também remove toda a integridade referencial associado aos objetos do usuário removido.
- **Sintaxe básica:**

```
DROP USER <nome_do_usuario> [CASCADE];
```

117

## 21. Gerenciamento de Usuários

### 21.12. Revogando um Privilégio de Sistema/Papel Concedido

- O comando SQL responsável por essa tarefa é o comando **REVOKE**.
- **Sintaxe básica:**

**REVOKE**

```
{<nome_papel>|<nome_do_privilegio>|ALL [PRIVILEGES]}
FROM {<nome_do_usuario>|<nome_do_papel>|PUBLIC}
[WITH ADMIN OPTION];
```

- **ALL PRIVILEGES** – todos os privilégios do objeto.
- **PUBLIC** – privilégio para todos os usuários do banco de dados.
- **ADMIN OPTION** – para que um privilégio de sistema possa ser revogado.

118

## 21. Gerenciamento de Usuários

### 21.12. Revogando um Privilégio de Sistema/Papel Concedido

- Vamos tentar revogar o privilégio dado ao usuário joao e que não recebeu essa opção.

```
SQL> REVOKE grupo FROM joao;
revoke grupo from joao
*
```

```
ERROR at line 1:
ORA-01932: ADMIN option not grant for role 'GRUPO'
```

119

## 21. Gerenciamento de Usuários

### 21.12. Revogando um Privilégio de Sistema/Papel Concedido

- Ocorreu um erro, pois o papel não recebeu a opção **ADMIN** ao ser concedido. Nós conectamos como o usuário que atribuiu o privilégio ao papel e atribuímos novamente os papéis **CONNECT** e **RESOURCE** para o papel **GRUPO**, agora com a opção **ADMIN**.

```
SQL> connect
Enter user-name: system
Enter password: *****
Connected.
SQL> GRANT CONNECT, RESOURCE to grupo WITH ADMIN OPTION;
Grant succeeded
```

- Agora vamos usar o comando **REVOKE** para retirar o papel grupo do usuário joao.

```
SQL> REVOKE grupo FROM joao;
Revoke succeeded.
```

120

## 21. Gerenciamento de Usuários

### 21.12. Revogando um Privilégio de Objeto de um Usuário

- Para revogar um privilégio, você deve se conectar como um usuário que tenha autoria da concessão. Um usuário não pode conceder/revogar privilégios para si mesmo.

```
SQL> Connect
Enter user-name: Scott
Enter password: *****
Connect
SQL> REVOKE SELECT ON SCOTT.DEPT FROM joao
Revoke succeeded.
```

121

## 21. Gerenciamento de Usuários

### 21.14. Apagando um Papel

- O comando responsável por apagar um papel é **DROP ROLE**. Para que o usuário possa remover um papel, ele precisa ter o privilégio de sistema **DROP ANY ROLE** ou o papel que tenha sido criado com a opção **ADMIN OPTION**.

- **Sintaxe básica:**

```
DROP ROLE <nome_da_rola>;
```

122

## 22. Sinônimos

- Um sinônimo é um apelido dado para um objeto de banco de dados.
- Quando um sinônimo é criado, é feita uma referência ao objeto original.
- Vantagens:
  - Esconder a identidade do objeto que está sendo referenciado. Se o objeto for mudado ou movido, basta atualizar o sinônimo,
  - Uma tabela com o nome extenso e que pertença a um esquema diferente daquele do usuário pode ser abreviado com um sinônimo.

123

## 22. Sinônimos

- Pode ser criado para uma tabela, visualização, seqüência, procedure em função de package ou até mesmo de outro sinônimo.
- Os sinônimos podem ser públicos e visíveis para todos os usuários ou privados e disponíveis apenas para o usuário que o criou.

124

## 22. Sinônimos

### 22.1 Criando sinônimos

- **Sintaxe:**

```
CREATE [PUBLIC] SYNONYM <esquema.nome>
FOR <esquema.objeto>;
```

- **Exemplo:**

```
CREATE SYNONYM departamento
FOR scott.dept;
```

O usuário precisa ter o privilégio **CREATE SYNONYM**

125

## 22. Sinônimos

### 22.2 Renomeando sinônimos

- **Sintaxe:**

```
RENAME <nome_antigo> TO <nome_novo>;
```

- **Exemplo:**

```
RENAME departamento TO depto;
```

126

## 22. Sinônimos

### 22.3 Removendo sinônimos

- **Sintaxe:**

```
DROP [PUBLIC] SYNONYM <nome_sinonimo>;
```

- **Exemplo:**

```
DROP SYNONYM depto;
```

- **Listando os Sinônimos criados**

```
SELECT synonym_name
FROM USER_SYNONYMS;
```

- **Listando os Sinônimos públicos**

```
SELECT synonym_name
FROM ALL_SYNONYMS;
```

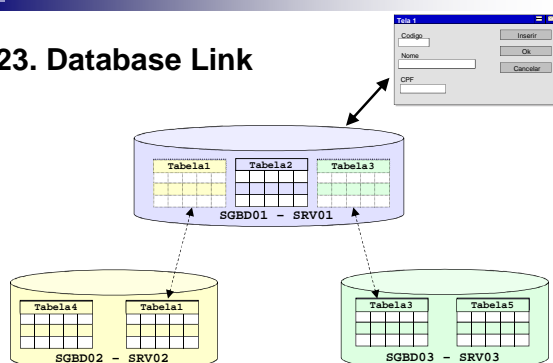
127

## 23. Database Link

- Um database link(**dblink**) é um objeto criado em um esquema de um banco de dados que possibilita o acesso a objetos de outro banco de dados, seja ele Oracle ou não.
- Esse tipo de sistema é conhecido como **Sistema de Banco de Dados Distribuídos**
  - **Homogêneo** – quando acessa outros bancos de dados Oracle
  - **Heterogêneo** – quando acessam outros tipos de bancos de dados
- Nem todo o banco possui este recurso consulte a documentação do seu banco.

128

## 23. Database Link



129

## 23. Database Link

### 20.1 Criando Database Link

- **Sintaxe Básica:**

```
CREATE [PUBLIC] DATABASE LINK <nome_link> CONNECT TO
<usuario> IDENTIFIED BY <senha> USING
'//<endereco>:<porta>/<instancia>';
```

- **nome\_link** – Nome do link para ser utilizado no banco local.
- **usuario** – Nome de um usuário autorizado a se conectar no banco de dados remoto.
- **senha** – Senha do usuário autorizado a se conectar no banco de dados remoto.
- **endereco** – Endereço ip do servidor de banco de dados remoto.
- **porta** – Porta do servidor de banco de dados remoto.
- **instancia** – Nome da instancia do servidor de banco de dados remoto.

130

## 23. Database Link

### 23.1 Criando Database Link

- **Exemplo:**

```
CREATE DATABASE LINK xelink
CONNECT TO scott
IDENTIFIED BY tiger
USING '//192.168.0.1:1521/xe';
```

- Necessário ser um usuário com permissão de conexão no banco de dados remoto.

131

## 23. Database Link

### 23.2 Consultado Database Link Criados

- **Exemplo:**

```
SELECT *
FROM DBA_DB_LINKS;
```

132

## 23. Database Link

### 23.3 Consultando Tabelas via Database Link

#### ■ Exemplo:

```
SELECT *
FROM cliente@xelink;
```

- Para manter-se a transparência no acesso a objetos de outros bancos de dados pode-se criar sinônimos públicos para os objetos acessados através do dblink.

133

## 23. Database Link

### 23.3 Consultando Tabelas via Database Link

#### ■ Criando o sinônimo:

```
CREATE SYNONYM clientepeg
for cliente@xelink;
```

#### ■ Consulta pelo sinônimo:

```
SELECT *
FROM clientepeg;
```

134

## 24. Agendamento de Tarefas(JOB)

- Algumas tarefas ou execuções de processos precisam ser feitas periodicamente, ou simplesmente agendadas para um determinado horário.
- O Oracle possui algumas maneiras de automatizar estas atividades, uma delas é a submissão de atividades como **JOBS**.

135

## 24. Agendamento de Tarefas(JOB)

- JOBS são objetos associados ao schema que permitem agendamento de atividades, para trabalharmos com os jobs precisamos de permissão de execução na package **DBMS\_JOB**.
- Operações deste pacote:
  - DBMS\_JOB.SUBMIT()
  - DBMS\_JOB.REMOVE()
  - DBMS\_JOB.CHANGE()
  - DBMS\_JOB.WHAT()
  - DBMS\_JOB.NEXT\_DATE()
  - DBMS\_JOB.INTERVAL()
  - DBMS\_JOB.RUN()

136

## 24. Agendamento de Tarefas

### 24.1 Criando um Job

- A procedure abaixo irá apagar os clientes que foram inseridos e não possuem conta ou dívida com o banco.

```
create or replace procedure LIMPADADOS is
begin
 delete from cliente
 where cpf_cliente not in
 ((select cpf_cliente from devedor)
 union
 (select cpf_cliente from conta)
);
end;
```

137

## 24. Agendamento de Tarefas

### 24.1 Criando um Job

- Este job deve ser executado diariamente as 02:00.

```
Declare
 job_num number; --Variavel que recebera o id do job
Begin
 DBMS_JOB.SUBMIT(job_num,
 'LIMPADADOS;',
 sysdate,
 'trunc(sysdate + 1) + 2/24');
 COMMIT;
End;
```

- O primeiro parametro de DBMS\_JOB.SUBMIT é o número do job agendado.
- O segundo parametro o nome da procedure que se deseja agendar.
- O terceiro a data da proxima execução do JOB.
- O ultimo o intervalo de execução da JOB, no caso do exemplo no proximo dia as 02:00 horas.

138

## 24. Agendamento de Tarefas

### 24.1 Criando um Job

- Abaixo outros exemplos de intervalo :
  - 'SYSDATE + 7'=>exatamente sete dias da última execução
  - 'SYSDATE + 1/48'=>cada meia hora
  - 'NEXT\_DAY(TRUNC(SYSDATE),'MONDAY')+15/24'=>toda segunda-feira as 15:00
  - 'NEXT\_DAY(ADD\_MONTHS(TRUNC(SYSDATE,'Q'),3),'THURSDAY')'=>primeira quinta-feira de cada trimestre
  - 'TRUNC(SYSDATE + 1)' => todo dia a meia noite
  - 'TRUNC(SYSDATE + 1)+8/24' => todo dia as 08:00
  - 'NEXT\_DAY(TRUNC(SYSDATE),'TUESDAY')+12/24'=>toda terça-feira ao meio dia
  - 'TRUNC(LAST\_DAY(SYSDATE)+1)'=>primeiro dia de cada mês a meia noite
  - 'TRUNC(ADD\_MONTHS(SYSDATE+2/24,3),'Q')-1/24'=>último dia de cada trimestre as 23:00
  - 'NEXT\_DAY(SYSDATE,'FRIDAY'))+9/24'=>cada segunda, quarta e sexta as 09:00

139

## 24. Agendamento de Tarefas

### 24.2 Listando os Jobs Agendados

#### ■ Exemplo:

```
□ SELECT *
□ FROM ALL_JOBS;
```

140

## 24. Agendamento de Tarefas

### 24.3 Listando os Jobs em Execução

#### ■ Exemplo:

```
□ SELECT *
□ FROM DBA_JOBS_RUNNING;
```

- É necessário ter privilégios de DBA.

141

## 24. Agendamento de Tarefas

### 24.4 Removendo um Job

#### ■ Exemplo:

```
□ exec DBMS_JOB.remove(<ID_JOB>);
```

□ ID\_JOB – Id do job que se deseja remover.

142

## Conclusão

- Não precisa nem dizer que Banco de dados são essenciais no desenvolvimento de sistemas.
- Conhecer somente banco de dados ou programação não ajudar para criar bons sistemas.
- Pois o programa depende do banco de dados e o banco de dados depende do programa.
- Se um ou outro for mal projetado todos os dois terão problemas.

143

## Bibliografia

#### ■ Principal

- DATE, C. J. **Introdução a Sistemas de Banco de Dados**. 7. ed. Rio de Janeiro: Campus, 2000. 803 p.
- ELMASRI, S. N.; NAVATHE, B.S.. **Sistemas de Banco de Dados: Fundamentos e Aplicações**. 3. ed. Rio de Janeiro: Livros Técnicos e Científicos, 2002. 837 p.
- SILBERSCHATZ, A. ; KORTH, H.F. ; SUDARSHAN, S. **Sistema de Banco de Dados**. 5. ed. Rio de Janeiro: Elsevier, 2006.

#### ■ Complementar

- FREEMAN, R. **Oracle, referência para o DBA: técnicas essenciais para o dia-a-dia do DBA**. Rio de Janeiro: Elsevier, 2005.
- RAMALHO, J. A. **Oracle: Oracle 10g**, ed. São Paulo: Pioneira Thomsom Learning, 2005.



