Apostila do Minicurso de PL/SQL

François Oliveira

PL/SQL é uma extensão para SQL ela adiciona a linguagem procedural ao SQL do Oracle. Ela oferece modernas características como sobrecarga, coleções, tratamento de exceções entre outros.

A linguagem PL/SQL incorpora muitos recursos avançados criados em linguagens de programação projetadas durante as décadas de 70 e 80. Além de aceitar a manipulação de dados, ela também permite que as instruções de consulta da linguagem SQL sejam incluídas em unidades procedurais de código e estruturadas em blocos, tornando a linguagem SQL um linguagem avançada de processamento de transações. Com a linguagem PL/SQL, você pode usar as instruções SQL para refinar os dados do Oracle e as instruções de controle PL/SQL para processar os dados.

Para entender a linguagem PL/SQL vamos dar uma olhada nas principais características da linguagem e suas vantagens. Vamos ver como a linguagem PL/SQL preenche o vazio entre o banco de dados e as linguagens procedurais.

Porque e pra que?

Você pode estar se perguntando por que existe o PL/SQL se já temos o SQL e as linguagens procedurais? Não podemos colocar SQL em nosso código Java ou C#, para que precisamos de mais uma linguagem? Na verdade a linguagem PL/SQL é ideal para **garantir** a integridade dos dados do sistema, mantendo regras do negócio centralizadas no banco de dados. As vantagens dessa centralização são inúmeras, porque não precisamos replicar as regras do negócio já que uma vez no banco de dados todos os programas que fazem parte do sistema como um todo e compartilham o mesmo código, sendo assim, esse código não precisará ser reescrito, será reutilizado e garantirá que as regras sejam cumpridas em diversas camadas. Vamos ver alguns exemplos práticos da utilização das regras em um banco de dados.

Regra 1: O estoque de nossa empresa não pode ficar negativo
Pergunta: Qual seria o melhor lugar pra garantir que esta regra seja cumprida, espalhar essas regras nos diversos sistemas que usam essa base de dados ou no banco de dados, que é único e que armazena os produtos?
Resposta: Você pode até ter essa regra pelos diversos programas, porém você terá alguns problemas, primeiro como esses sistemas estão distribuídos podemos ter problemas de concorrência na atualização desses dados, segundo se esta regra mudar, teremos que atualizar todos os sistemas ficando mais difícil de manter o código, então se esta regra estiver centralizada podemos eliminar esses problemas, simplificando nosso sistema e garantindo essa regra com qualidade e segurança com apenas uma check constraint no banco de dados.

Regra 2: Temos que manter o estoque atualizado sempre que um produto for vendido
Pergunta: Podemos fazer isso através de nosso programa?
Resposta: Sim, mas novamente vamos ter que tratar a concorrência, no banco de dados usamos gatilhos, essa ferramenta lhe permite garantir com rapidez, qualidade e segurança que todo o sistema, no importa de onde vier seu pedido, atualize o estoque toda vez que o produto for vendido.

Regra 3: Temos que manter históricos de nosso estoque mensalmente, no final de cada mês devemos registrar o inventário.
Pergunta: Podemos fazer isso através de nosso programa?
Reposta: Sim, mas em PL/SQL podemos fazer isso com apenas um comando SQL e uma procedure em nosso banco de dados.

Com o PL/SQL conseguimos fazer com que nosso banco de dados “pense”, com ela conseguimos colocar a inteligência das regras dentro do banco de dados, o banco passa não só a armazenar os dados, mas também a gerenciar a qualidade desses dados baseado nas regras que você desenvolver, seus dados terão mais qualidade e dados com qualidade garantirão uma informação com qualidade.

Entendendo a linguagem

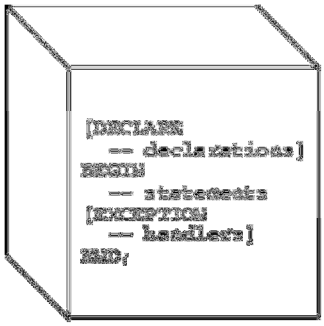
Para entender melhor a linguagem vamos começar por um pequeno trecho de código. Esse pequeno trecho de código processa um pedido de uma raquete de tênis. Primeiro ele declara uma variável do tipo NUMBER para armazenar a quantidade de raquetes de tênis solicitadas. Depois ele pesquisa a quantidade de raquete de tênis que existem no estoque. Se a quantidade for maior que zero, o sistema atualiza o estoque e registra uma raquete de tênis na tabela de pedido, caso contrário, o programa registra a falta do produto na tabela de pedido.

```
DECLARE
   qty_on_hand  NUMBER(5);
BEGIN
   SELECT quantity
   INTO   qty_on_hand
   FROM   inventory
   WHERE  product = 'TENNIS RACKET'
   FOR UPDATE OF quantity;

   IF qty_on_hand > 0 THEN  -- checando a quantidade
      UPDATE inventory
      SET    quantity = quantity - 1
      WHERE  product = 'TENNIS RACKET';

      INSERT
      INTO   purchase_record
      VALUES ('Tennis racket purchased', SYSDATE);
   ELSE
      INSERT
      INTO   purchase_record
      VALUES ('Out of tennis rackets', SYSDATE);
   END IF;

   COMMIT;
END;
```

Com PL/SQL você pode combinar a força da manipulação de dados da linguagem **SQL** com a força do processamento de dados das **linguagens procedurais**, em um único lugar. Principais características da linguagem PL/SQL:

- Blocos de código

  A linguagem PL/SQL é baseada em blocos, como a linguagem Pascal, cada unidade básica de código e composta por blocos que podem conter novos blocos. Um bloco pode conter declarações e comandos. A figura abaixo mostra que um bloco PL/SQL contém três partes, a declaração, execução e exceção. Apenas a parte de execução é obrigatória.



- Declaração de variáveis

  As variáveis podem ser de qualquer tipo SQL, como CHAR, NUMBER, DATE ou de qualquer tipo PL/SQL, como BOOLEAN. Por exemplo, para declarar duas variáveis fazemos o seguinte:

```
part_no     NUMBER(4);
in_stock    BOOLEAN;
```

  Você também pode declarar outros tipos no bloco de declaração, como registros por exemplo.

- Atribuindo valores a variáveis

  Existem três formas de atribuir valores a uma variável em PL/SQL, são elas:

  1. Usando o operador :=

```
valid_id := FALSE;
tax := price * tax_rate;
bonus := current_salary * 0.10;
wages := gross_pay(emp_id, st_hrs, ot_hrs) - deductions;
```

  2. Usando um comando SELECT

```
SELECT sal * 0.10
INTO   bonus
FROM   emp
WHERE  empno = emp_id;
```

  3. Usando a passagem de parâmetros por referência(OUT, IN OUT)

```
DECLARE
   my_sal REAL(7,2);
   PROCEDURE adjust_salary(emp_id INT, salary IN OUT REAL) IS ...
BEGIN
   SELECT AVG(sal)
   INTO   my_sal
   FROM   emp;
   adjust_salary(7788, my_sal); -- atribui um novo valor a my_sal
```

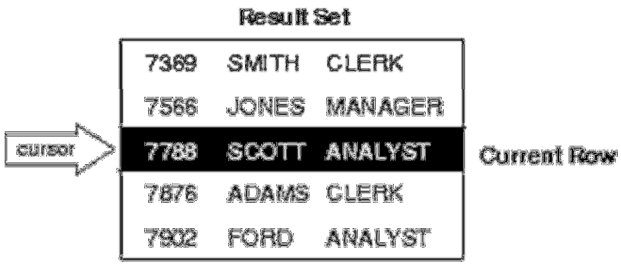
- Declaração de constantes
  A declaração de uma constante é parecida com a declaração de uma variável, basta acrescentar a palavra CONSTANT e o TIPO, exemplo:

```
credit_limit CONSTANT REAL := 5000.00;
```

- Cursor
  O Oracle usa áreas de trabalho para executar comandos SQL e armazenar as informações processadas. Uma construção do PL/SQL chamada de **CURSOR** permite que você de um nome e acesse a informação armazenada. Para qualquer comando SQL o PL/SQL cria um cursor implicitamente, porém se você quiser processar individualmente as linhas que retornam de uma consulta você deve criar explicitamente um cursor.

```
DECLARE
   CURSOR c1 IS
      SELECT empno, ename, job
      FROM emp
      WHERE deptno = 20;
```

  No cursor declarado acima as linhas que retornam desse SELECT poderão ser processadas individualmente, através de um ponteiro você poderá caminhar na tabela resultado do SELECT.



- Cursor FOR Loops

Para navegar no resultado de um cursor a forma mais simples é utilizar o laço FOR LOOP, com ele você navega nos resultados de um cursor até que o ponteiro atinja a última linha da tabela resultado do SELECT. Outra forma de se navegar num Cursor é usar os comando OPEN, FETCH e CLOSE, porém o FOR LOOP é a forma mais simples.

```
DECLARE
   CURSOR c1 IS
      SELECT ename, sal, hiredate, deptno
      FROM   emp;
      ...
BEGIN
   FOR emp_rec IN c1 LOOP
      ...
      salary_total :=  salary_total + emp_rec.sal;
   END LOOP;
```

Note que para referenciar as colunas da tabela resultado do Cursor no FOR criamos um registro que sera usado para acessar a coluna da linha em que o ponteiro esta posicionado. No exemplo acima, o registro **emp_rec** é usado para acessar a coluna **sal** do Cursor.

- Atributos

As variáveis e os cursores do PL/SQL podem usar atributos para definir seus tipos por referência, isso significa que, ao invés de definir um tipo você pode referenciar o tipo de uma coluna ou de toda uma tabela, exemplo:

- %TYPE

Este atributo é muito usado para definir tipos de variáveis que vão receber valores de uma coluna. Vamos assumir que exista um coluna chamada **title** em uma tabela chamada **books**, para declarar uma variável **my_title** do mesmo tipo da coluna title basta usar o atributo %TYPE da seguinte forma no bloco de declaração de variáveis:

```
my_title books.title%TYPE
```

Uma das vantagens dessa notação é que se o tipo da tabela for alterada o tipo da variável também será mudado.

- %ROWTYPE

Se você precisar de um registro que represente uma linha de uma tabela, use o atributo %ROWTYPE, com ele você poderá representar toda uma linha de uma tabela, exemplo:

```
DECLARE
        dept_rec dept%ROWTYPE;  -- declaração de um registro
```

Para referenciar um campo do registro use a notação de pontos, exemplo:

```
my_deptno := dept_rec.deptno;
```

- Estruturas de controle

Vamos agora falar das estruturas de controle da linguagem PL/SQL

- Condições

  - IF - THEN - ELSIF - ELSE - END IF

```
DECLARE
   acct_balance NUMBER(11,2);
   acct         CONSTANT NUMBER(4) := 3;
   debit_amt    CONSTANT NUMBER(5,2) := 500.00;
BEGIN
   SELECT bal
   INTO   acct_balance
   FROM   accounts
   WHERE  account_id = acct
   FOR UPDATE OF bal;

   IF acct_balance >= debit_amt THEN
      UPDATE accounts
      SET    bal = bal - debit_amt
      WHERE  account_id = acct;
   ELSE
      INSERT
      INTO    temp
      VALUES (acct, acct_balance, 'Insufficient funds');
   END IF;

   COMMIT;
END;
```

- CASE - ELSE - END CASE

```
-- This CASE statement performs different actions
           -- based on a set of conditional tests.
           CASE
              WHEN shape = 'square' THEN area := side * side;
              WHEN shape = 'circle' THEN
              BEGIN
                 area := pi * (radius * radius);
                 DBMS_OUTPUT.PUT_LINE('Value is not exact because pi is
                       irrational.');
              END;
              WHEN shape = 'rectangle' THEN area := length * width;
           ELSE
           BEGIN
              DBMS_OUTPUT.PUT_LINE('No formula to calculate area of a' ||
                 shape);
              RAISE PROGRAM_ERROR;
           END;
           END CASE;
```

  - Laços

    - FOR LOOP - END LOOP

```
FOR num IN 1..500 LOOP
   INSERT INTO roots VALUES (num, SQRT(num));
END LOOP;
```

    - WHILE LOOP - END LOOP

```
DECLARE
   salary         emp.sal%TYPE := 0;
   mgr_num        emp.mgr%TYPE;
   last_name      emp.ename%TYPE;
   starting_empno emp.empno%TYPE := 7499;
BEGIN
   SELECT mgr
   INTO   mgr_num FROM emp
   WHERE  empno = starting_empno;

   WHILE salary <= 2500 LOOP
      SELECT sal, mgr, ename
      INTO   salary, mgr_num, last_name
      FROM   emp
      WHERE  empno = mgr_num;
   END LOOP;

   INSERT
   INTO   temp
   VALUES (NULL, salary, last_name);

   COMMIT;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT
      INTO   temp
      VALUES (NULL, NULL, 'Not found');
   COMMIT;
END;
```

- EXIT WHEN

```
LOOP
               ...
               total := total + salary;
               EXIT WHEN total > 25000;  -- sai do loop se condição verdadeira
            END LOOP;
```

- GOTO

```
IF rating > 90 THEN
            GOTO calc_raise;  -- branch to label
         END IF;
         ...
         <<calc_raise>>
         IF job_title = 'SALESMAN' THEN  -- control resumes here
            amount := commission * 0.25;
         ELSE
            amount := salary * 0.10;
         END IF;
```

- Modularização

- Subprogramas

A linguagem PL/SQL possui dois tipos de subprogramas **procedures** e **functions**.

- Procedure

```
PROCEDURE award_bonus (emp_id NUMBER) IS
            bonus REAL;
            comm_missing EXCEPTION;
            BEGIN  -- executable part starts here
               SELECT comm * 0.15
               INTO bonus
               FROM emp
               WHERE empno = emp_id;

               IF bonus IS NULL THEN
                  RAISE comm_missing;
               ELSE
                  UPDATE payroll
                  SET    pay = pay + bonus
                  WHERE  empno = emp_id;
               END IF;
            EXCEPTION  -- exception-handling part starts here
               WHEN comm_missing THEN
                  ...
            END award_bonus;
```

- Function

```
CREATE OR REPLACE FUNCTION area_quadrado(base IN NUMBER,
               altura IN NUMBER)
RETURN NUMBER IS
BEGIN
   RETURN (base * altura) ** 2;
END;
```

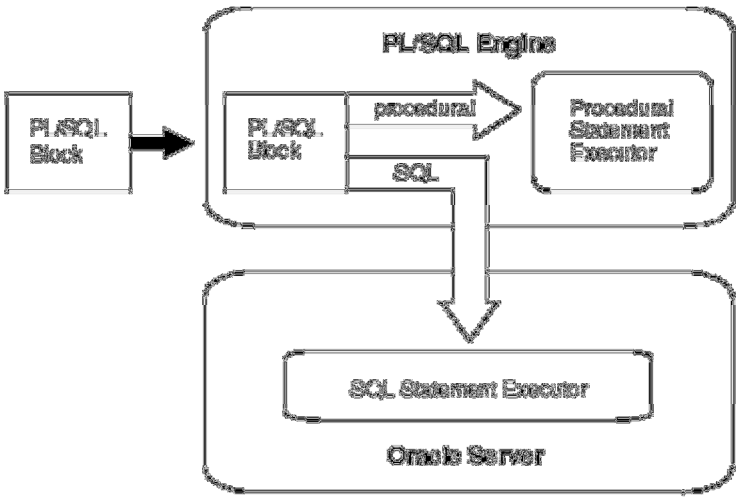
- Pacotes

A linguagem PL/SQL permite a você agrupar tipos, variáveis, cursores e subprogramas em um pacote. Um pacote é dividido em duas partes: especificação e corpo. A especificação é a interface do pacote, nela estão declarados os tipos, constantes, variáveis, exceções, cursores e subprogramas disponíveis pra uso. No corpo temos a implementação das especificações declaradas na interface.

```
CREATE PACKAGE emp_actions AS  -- package specification
           PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
           PROCEDURE fire_employee (emp_id NUMBER);
         END emp_actions;

         CREATE PACKAGE BODY emp_actions AS  -- package body
           PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
           BEGIN
              INSERT INTO emp VALUES (empno, ename, ...);
           END hire_employee;

           PROCEDURE fire_employee (emp_id NUMBER) IS
           BEGIN
              DELETE FROM emp WHERE empno = emp_id;
           END fire_employee;
         END emp_actions;
```

Arquitetura PL/SQL

O PL/SQL em sua versão compilado e run-time é uma tecnologia, não é um produto independente. Essa tecnologia é como um engine que compila e executa os blocos PL/SQL, este engine pode estar instalado num banco de dados Oracle ou em uma ferramenta de desenvolvimento, como Oracle Forms ou Oracle Report.



- Blocos anônimos

Quando programas enviam blocos PL/SQL para Oracle, eles são considerados como bloco anônimos, ao receber esses blocos o Oracle irá compilar e executar os blocos PL/SQL.

- Subprogramas armazenados

Quando um subprograma é armazenado no Oracle ele esta pronto para ser executado. Armazenar um subprograma oferece uma produtividade superior, alta performance, otimização de memória, integridade com aplicação e alta segurança. Armazenando código PL/SQL em bibliotecas você pode compartilhar esse código em diversos sistemas, diminuindo a redundância de código e aumentando a produtividade. Subprogramas são armazenados prontos para serem executados, não havendo a necessidade de se fazer o parse, nem de compilar novamente, sendo assim, quando um subprograma que esta armazenado é chamado ele é executado imediatamente pelo engine PL/SQL, tirando proveito também da memória compartilhada, pois apenas uma cópia do programa sera carregado na memória.

- Gatilhos

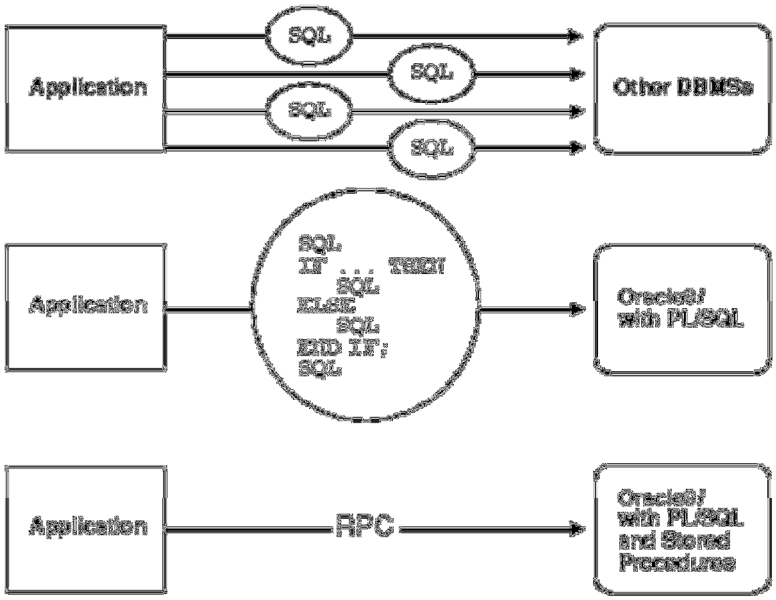
Gatilhos são subprogramas associados a uma tabela, visão ou evento. Em cada instância, você pode ter um gatilho disparado automaticamente antes ou depois de um comando INSERT, UPDATE ou DELETE que tenha afetado uma tabela. Por exemplo, o gatilho abaixo é disparado sempre que o salário da tabela de empregado é alterado.

```
CREATE TRIGGER audit_sal
   AFTER UPDATE OF sal ON emp
   FOR EACH ROW
BEGIN
   INSERT
   INTO emp_audit VALUES ...
END;
```

Vantagens da linguagem PL/SQL

- Suporte a linguagem SQL
- Suporte a linguagem orientada a objetos
- Alta performance
- Alta produtividade
- Portabilidade
- Integração com Oracle
- Segurança

A linguagem SQL se tornou um padrão em linguagem de manipulação de dados por ser uma linguagem flexível, poderosa e fácil de aprender. Com poucas palavras em inglês, SELECT, INSERT, UPDATE e DELETE podemos manipular facilmente os dados em um banco de dados. A problema da linguagem SQL é que ela não é procedural, você tem que executar um comando por vez. A linguagem PL/SQL permite que você use todo o poder da linguagem SQL, de forma procedural num ambiente altamente integrado com o Oracle e seus recursos de forma rápida e segura, reduzindo também o tráfego na rede. Criando um programa em PL/SQL você pode roda-lo em qualquer ambiente que o Oracle rode.



Exercícios - SQL*Plus

O SQL*Plus é um ambiente no qual você pode realizar as seguintes tarefas:

- Executar instruções SQL para manipular dados no banco de dados;
- Formatar, calcular, armazenar e imprimir resultados de consulta em formulários;
- Criar arquivos de script para armazenar instruções SQL para uso repetitivo;

Comando `SQL>`	Descrição do comando
`connect scott/tiger;`	Conectar ao banco de dados
`desc dept;`	Exibir a estrutura de uma tabela
`SELECT * FROM emp;`	Executar um comando SQL
`/`	Repetir o comando SQL que esta no buffer
`save [nome_arquivo] [repl]`	Salvar o comando SQL que esta no buffer
`@[nome_arquivo]`	Executar o arquivo salvo
`ed`	Editar o comando SQL que esta no buffer
`spool [nome_arquivo]`	Armazena a saída de uma consulta no arquivo
`spool off`	Encerra a gravação e fecha o arquivo
`show error`	Mostra o erro com detalhes
`exit`	Sair do SQL*Plus

Diversão com grande SCOTT – Recordar SQL é viver

```
SQL> SELECT *
     FROM    salgrade;

SQL>  SELECT ename, job, sal Salario
      FROM   emp;

SQL>  SELECT ename, mgr
      FROM   emp
      WHERE  mgr = NULL;

SQL>  SELECT ename, job, deptno
      FROM   emp
      WHERE  job = 'CLERK';

SQL>  SELECT ename, job, sal Salario
      FROM   emp
      WHERE  sal BETWEEN 1000 AND 1500;

SQL>  SELECT empno, ename, mgr, deptno
      FROM   emp
      WHERE  ename IN ('FORD', 'ALLEN');

SQL>  SELECT ename
      FROM   emp
      WHERE  ename LIKE 'S%';

SQL>  SELECT e.ename, d.dname
      FROM   emp e, dept d
      WHERE  e.deptno = d.deptno;
```

O sistema de pedidos da **XDK Esportes**

Vamos criar um pequeno sistema de pedidos para uma loja de esportes, vamos implementar as regras de um negócio utilizando a linguagem PL/SQL de forma simples e objetiva. Criaremos um sistema do zero e veremos como o sistema irá funcionar de forma simples. Vocês verão como é fácil implementar as regras no banco de dados e o poder que esta ferramenta proporciona.

Esse sistema irá implementar as três regras que foram citadas no início desta apostila, vamos lembra das regras novamente:

1. O estoque de nossa empresa não pode ficar negativo
2. Temos que manter o estoque atualizado sempre que um produto for vendido
3. Temos que manter históricos de nosso estoque mensalmente, no final de cada mês devemos registrar o inventário.

Criando tabelas – DDL
Solução da regra 1

```
-- se conectando como sys, super usuário do Oracle
CONNECT / AS SYSDBA;

-- criando o usuário minicurso
CREATE USER minicurso IDENTIFIED BY oracle;

-- dando permissões de super usuário para seminfo
GRANT DBA TO minicurso;

-- se conectando como minicurso
CONNECT minicurso/oracle;

CREATE TABLE cliente(
codigo NUMBER(3)        CONSTRAINT cliente_codigo_nn NOT NULL,
nome   VARCHAR(30)      CONSTRAINT cliente_nome_nn NOT NULL,
CONSTRAINT cliente_codigo_pk PRIMARY KEY(codigo)
);

CREATE TABLE produto(
codigo      NUMBER(3)   CONSTRAINT produto_codigo_nn NOT NULL,
descricao   VARCHAR(30) CONSTRAINT produto_descricao_nn NOT NULL,
quantidade  NUMBER(10)  CONSTRAINT produto_quantidade_nn NOT NULL,
CONSTRAINT produto_codigo_pk PRIMARY KEY(codigo),
CONSTRAINT produto_quantidade_ck CHECK(quantidade >= 0)
);

CREATE TABLE pedido(
numero            NUMBER(10)  CONSTRAINT pedido_numero_nn NOT NULL,
data              DATE        CONSTRAINT pedido_data_nn NOT NULL,
codigo_cliente    NUMBER(3)   CONSTRAINT pedido_codigo_cliente_nn NOT NULL,
CONSTRAINT pedido_numero_pk PRIMARY KEY(numero),
CONSTRAINT pedido_codigo_cliente_fk
      FOREIGN KEY(codigo_cliente) REFERENCES cliente(codigo)
);

CREATE TABLE item_pedido(
numero_pedido     NUMBER(10)        CONSTRAINT item_ped_num_pedido_nn NOT NULL,
codigo_produto    NUMBER(3)         CONSTRAINT item_ped_cod_produto_nn NOT NULL,
quantidade        NUMBER(10)        CONSTRAINT item_ped_quantidade_nn NOT NULL,
valor             NUMBER(18,2)      CONSTRAINT item_ped_valor_nn NOT NULL,
CONSTRAINT item_ped_num_ped_cod_prod_pk
      PRIMARY KEY(numero_pedido, codigo_produto),
CONSTRAINT item_ped_num_ped_fk
      FOREIGN KEY(numero_pedido) REFERENCES pedido(numero),
CONSTRAINT item_ped_cod_prod_fk
      FOREIGN KEY(codigo_produto) REFERENCES produto(codigo),
CONSTRAINT item_ped_quantidade_ck CHECK(quantidade > 0),
CONSTRAINT item_ped_valor_ck CHECK(valor > 0)
);
```

Note que na tabela de produto foi criada uma CONSTRAINT para garantir que a regra 1 seja cumprida, neste caso nunca nossos produtos terão o estoque negativo, pois mesmo que um sistema tente vender mais produtos do que existam nosso estoque o banco de dados não irá permitir. Veremos mais adiante como esta constraint junto com um gatilho que será criado posteriormente garantirá minha regra 1.

Povoando o banco de dados

```
INSERT
INTO   cliente
VALUES (1, 'FRANCOIS OLIVEIRA');

INSERT
INTO   cliente
VALUES (2, 'DANIEL OLIVEIRA');

INSERT
INTO   cliente
VALUES (3, 'MIGUEL OLIVEIRA');

INSERT
INTO   produto
VALUES (100, 'RAQUETE DE TENIS WILSON', 10);

INSERT
INTO   produto
VALUES (101, 'TENIS DE CORRIDA OLIMPIKUS', 15);

INSERT
INTO   produto
VALUES (102, 'OCULOS DE NATACAO SPEEDO', 20);
```

Os pedidos – regra 2

Vamos começar a povoar a tabela de pedidos, mas agora com um detalhe vamos fazer um antes e depois, antes do código PL/SQL e depois do código PL/SQL, vamos verificar como funciona o banco de dados sem a inteligência da regra e depois com ele se comporta com a regra implementada. Vamos começar inserindo um pedido no sistema.

```
SELECT *
FROM   produto
WHERE  codigo = 100;

INSERT
INTO   pedido
VALUES (1000, SYSDATE, 1);

INSERT
INTO   item_pedido
VALUES (1000, 100, 1, 150);

SELECT *
FROM   produto
WHERE  codigo = 100;
```

O que você percebeu depois dessa consulta, que o estoque não foi alterado, isso significa que seu banco ainda esta burro, ele recebeu os dados mas não sabe que o estoque tem que ser subtraído quando um item for vendido. Mas para fazer isso temos que colocar essa regra no banco de dados, vou começar a fazer isso traduzindo a regra 2 para PL/SQL, isto é, para garantir que todo produto vendido pela loja seja subtraído da quantidade disponível no estoque eu tenho que dizer ao banco que, toda vez que uma linha for **inserida**, **alterada ou excluída** da tabela de item_pedido o estoque tem que verificado e se necessário atualizado. Note que a tabela de item_pedido é a representação, em forma de dados, da venda de um produto de nossa loja. Note que esta regra é disparada sempre houver mudanças na tabela de item_pedido, isso significa que podemos usar o gatilho par implementa-la. Então vamos começar, vamos apagar as linhas da tabela de pedido e item_pedido e depois de criar o gatilho, vamos repetir os quatro comandos acima.

```
DELETE
FROM   item_pedido;

DELETE
FROM   pedido;

CREATE OR REPLACE TRIGGER atualiza_estoque_produto_tg
AFTER INSERT OR UPDATE OR DELETE ON item_pedido
FOR EACH ROW
BEGIN
   IF DELETING OR UPDATING THEN
      UPDATE produto
      SET    quantidade = quantidade + :old.quantidade
      WHERE  codigo = :old.codigo_produto;
   END IF;

   IF INSERTING OR UPDATING THEN
      UPDATE produto
      SET    quantidade = quantidade - :new.quantidade
      WHERE  codigo = :new.codigo_produto;
   END IF;
END;
```

Em caso de **ERRO** digite **SHOW ERROR** no SQL*Plus.

Depois de repetir os quatro comandos novamente você pode notar que o banco agora sabe que se um item_pedido for inserido, alterado ou excluído ele terá que atualizar o estoque, fazendo com que a quantidade do estoque permanece atualizada.

```
UPDATE item_pedido
SET    quantidade = 3
WHERE  numero_pedido = 1000 AND
       codigo_produto = 100;
```

Lembra da regra 1, vamos tentar quebra-la vendendo mais do que existe no estoque, vamos tentar vender 100 óculos de natação, e só temos 20 no estoque.

```
SELECT *
FROM   produto
WHERE  codigo = 102;

INSERT
INTO   pedido
VALUES (1001, SYSDATE, 2);

INSERT
INTO   item_pedido
VALUES (1001, 102, 100, 30);

INSERT
INTO   item_pedido
VALUES (1001, 102, 10, 30);

SELECT *
FROM   produto
WHERE  codigo = 102;
```

Note que ao tentar vender 100 itens o banco não deixou isso acontecer, porque a regra 1 seria violada, com isso, meus dados permaneceram íntegros e uma possível falha no sistema será descoberta.

Vamos criar agora uma procedure que implemente a regra 3, fechando nosso curso. Vou utilizar uma procedure e um gatilho para implementar a regra três, e isso será feito com bastante simplicidade. Vamos criar mais uma tabela e nela vamos registrar nosso inventário.

```
CREATE SEQUENCE numero_historico;

CREATE TABLE historico(
numero  NUMBER(10)      CONSTRAINT hist_invent_numero_nn NOT NULL,
data    DATE            CONSTRAINT hist_invent_data_nn NOT NULL,
usuario VARCHAR(30)     CONSTRAINT hist_invent_usuario_nn NOT NULL,
CONSTRAINT hist_invent_numero_pk PRIMARY KEY(numero)
);

CREATE TABLE item_historico(
numero_historico  NUMBER(10)  CONSTRAINT item_hist_numero_historico_nn NOT NULL,
codigo_produto    NUMBER(3)   CONSTRAINT item_hist_codigo_produto_nn NOT NULL,
quantidade        NUMBER(10)  CONSTRAINT item_hist_quantidade_nn NOT NULL,
CONSTRAINT item_hist_num_hist_cod_prod_pk
      PRIMARY KEY(numero_historico, codigo_produto),
CONSTRAINT item_hist_numero_historico_fk
      FOREIGN KEY(numero_historico) REFERENCES historico(numero),
CONSTRAINT item_hist_codigo_produto_fk
      FOREIGN KEY(codigo_produto) REFERENCES produto(codigo)
);
```

```
CREATE OR REPLACE PROCEDURE historico_inventario_sp AS
BEGIN
   INSERT
   INTO   historico
   VALUES (numero_historico.NEXTVAL, SYSDATE, USER);
END;

CREATE OR REPLACE TRIGGER historico_inventario_tg
AFTER INSERT ON historico
FOR EACH ROW
BEGIN
   INSERT
   INTO   item_historico
   SELECT :new.numero, p.codigo, p.quantidade
   FROM   produto p;
END;

EXECUTE historico_inventario_sp;
```

Bem pessoal é isso, espero que o material tenha ajudado a enriquecer o conhecimento de vocês e fiquem com essa mensagem para refletir.

Mensagem

As mudanças dependem de suas atitudes,
não espere faça acontecer, e lembre-se,
o conhecimento e a chave para mudança.

François Oliveira