Integração de SQL com outras linguagens de programação

- Tópicos:
  - JDBC
  - Embedded e Dynamic SQL
  - Linguagens proprietárias
- Bibliografia:
  - Secções 5.1 e 5.2 do livro recomendado

Embedded SQL

- JDBC é demasiado dinâmico, erros não podem ser detetados pelo compilador
- O SQL embutido permite acesso a bases de dados, via outra linguagens de programação.
  - Toda a parte de acesso e manipulação da base de dados é feito através de código embutido. Todo o processamento associado é feito pelo sistema de bases de dados. A linguagem *host* recebe os resultados e manipula-os.
  - O código tem que ser pré-processado. A parte SQL é transformada em código da linguagem *host,* mais chamadas a run-time do servidor.
- A expressão EXEC SQL é usado para identificar código SQL embutido

  EXEC SQL <embedded SQL statement > END-EXEC

  Nota: Este formato varia de linguagem para linguagem. E.g. em C usa-se ';' em vez do END-EXEC.

  Em Java usa-se # SQL { …. } ;

SQLJ

- SQLJ: SQL embutido em Java
  - #sql iterator deptInfoIter (String dept name, int avgSal);

```
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary) from instructor
                group by dept name };
while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

Cursores

- Para executar um comando SQL numa linguagem *host* é necessário começar por declarar um cursor para esse comando.
- O comando pode conter variáveis da linguagem *host,* precedidas de :
- E.g. Encontrar os nome e cidades de clientes cujo saldo seja superior a *amount*

EXEC SQL

declare *c* **cursor for**
select *customer-name, customer-city*
from *account* **natural inner join** *depositor*
natural inner join *customer*
where *account.balance > :amount*

END-EXEC

Embedded SQL (Cont.)

- O comando **open** inicia a avaliação da consulta no cursor

  EXEC SQL **open** *c* END-EXEC

- O comando **fetch** coloca o valor de um tuplo em variáveis da linguagem *host*.

  EXEC SQL **fetch** *c* **into** :*cn, :cc* END-EXEC

- Chamadas sucessivas a **fetch** obtêm tuplos sucessivos
- Uma variável chamada SQLSTATE na *SQL communication area* (SQLCA) toma o valor '02000' quando não há mais dados.
- O comando **close** apaga a relação temporária, criada pelo **open**, que contem os resultados da avaliação do SQL.

  EXEC SQL **close** *c* END-EXEC

Modificações com Cursores

- Como não devolvem resultado, o tratamento de modificações dentro doutras linguagens é mais fácil.
- Basta chamar qualquer comando válido SQL de **insert, delete,** ou **update** entre EXEC SQL e END SQL

- Em geral, as váriaveis da linguagem *host* só podem ser usadas em locais onde se poderiam colocar variáveis SQL.
- Não é possível *construir* comandos (ou parte deles) manipulando strings da linguagem *host*

Dynamic SQL

- Permite construir e (mandar) executar comandos SQL, em run-time.
- E.g. (chamando dynamic SQL, dentro de um programa em C)

```
char *  sqlprog = "update account
                   set balance = balance * 1.05
                   where account-number = ?";
EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- A string contém um ?, que indica o local onde colocar o valor a ser passado no momento da chamada para execução.

ODBC

- Standard Open DataBase Connectivity(ODBC)
  - Standard para comunicação entre programas e servidores de bases de dados
  - application program interface (API) para
    - Abrir uma ligação a uma base de dados
    - Enviar consultas e pedidos de modificações
    - Obter os resultados
- Aplicações diversas (e.g. GUI, spreadsheets, etc) podem usar ODBC

ODBC (Cont.)

- Um sistema de bases de dados que suporte ODBC tem uma "driver library" que tem que ser ligada com o programa cliente.
- Quando o cliente faz uma chamada à API ODBC, o código da library comunica com o servidor, que por sua vez executa a chamada e devolve os resultados.
- Um programa ODBC começa por alocar um ambiente SQL, e um *connection handle*.
- Para abrir uma ligação a uma BD, usa-se SQLConnect(). Os parâmetros são:
  - connection handle,
  - servidor onde ligar
  - username,
  - password

Exemplo de código ODBC 2.0

```
int ODBCexample()
{
    RETCODE error;
    HENV    env;     /* environment */
    HDBC    conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS,
      "avipasswd", SQL_NTS);
    { …. Manipulação propriamente dita … }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

ODBC (Cont.)

- Os programas enviam comandos SQL à base de dados usando SQLExecDirect
- Os tuplos resultado são obtidos via SQLFetch()
- SQLBindCol() liga variáveis da linguagem a atributos do resultado do SQL
  - Quando um tuplo é obtido com um *fetch*, os valores dos seus atributos são automaticamente guardados nas ditas variáveis.

Exemplo de código ODBC

```
char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;

    SQLAllocStmt(conn, &stmt);
    char * sqlquery = "select branch_name, sum (balance)
                       from account
                       group by branch_name";

    error = SQLExecDirect(stmt, sqlquery, SQL_NTS);

    if (error == SQL_SUCCESS) {
      SQLBindCol(stmt, 1, SQL_C_CHAR,   branchname , 80, &lenOut1);
      SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,        0 , &lenOut2);

      while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf (" %s  %g\n", branchname, balance);
      }
    }
    SQLFreeStmt(stmt, SQL_DROP);
```

Linguagens proprietárias

- A maior parte dos sistemas comerciais incluem linguagens proprietárias que, para além do embedded SQL, têm primitivas próprias para (entre outras) criar interfaces no ecrã (forms) e para formatar dados para apresentação de relatórios (reports).
- Algumas destas linguagens têm ainda construtores de mais alto nível, para trabalhar sobre cursores.
- Tipicamente os programas nestas linguagens, compilam para outras linguagens (e.g. C) embedded SQL.
- Os sistemas comerciais costumam ainda ter aplicações de geração fácil de programas na linguagem proprietária
- No Oracle a linguagem proprietária é o *PLSQL*. O *Forms,* o *Reports e o APEX* são aplicações que geram *PLSQL*.

PL/SQL

- Extensão procedimental ao SQL, do Oracle.
- Suporta:
  - Variáveis (mesmos tipos do Oracle)
  - Condições (IF-THEN-ELSE e CASE)
  - Ciclos (LOOP, FOR)
  - Exceções (para tratamento de erros)
- Unidades de programas em PL/SQL podem ser compilados na base de dados Oracle.

Construtores procedimentais

- O standard SQL suporta uma grande variedade de construtores procedimentais
  - O Oracle suporta aqueles que existem no PL/SQL
- Expressões com whiles e repeats

```
declare n integer default 0;
while n < 10 do
   set n = n+1;
end while;

repeat
    set n = n  – 1;
until n = 0;
end repeat
```

- Em Oracle, em vez de **set** *var* =… usa-se *var* **:=**…

Construtores procedimentais (Cont.)

- Ciclos
  - Iterações sobre o resultado de perguntas
  - E.g. soma de todos os saldos da agência Perryridge

```
declare n integer default 0;
for r as
    select balance from account
    where branch-name = 'Perryridge'
do
    set n = n + r.balance;
end for
```

- Claro que isto não se deve fazer assim!
  - O que se deve fazer para obter isto é:

```
select sum(balance) from account
where branch-name = 'Perryridge'
```

Construtores procedimentais (cont.)

- Expressões condicionais (if-then-else)
  E.g. Soma dos saldos por categorias de contas (com saldo <1000, entre 1000 e 5000, > 5000)

```
if r.balance < 1000
    then set l = l + r.balance
elseif r.balance =< 5000
    then set m = m + r.balance
else set h = h + r.balance
end if
```

- Assinalar condições de exceção e erros, e declaração de tratamento de exceções

```
declare out_of_stock condition;
declare exit handler for out_of_stock ;
begin
…
..  signal out-of-stock;
end
```

  - Neste exemplo o tratamento da excepção é **exit** – sai do bloco **begin…end**
- No Oracle em vez de **signal** usa-se **raise**

Funções e Procedimentos

- O standard SQL suporta funções e procedimentos
  - As funções e procedimentos podem ser escritas diretamente em SQL, ou em linguagens de programação externas (e.g. PL/SQL).
  - Alguns sistemas de bases de dados (entre eles o Oracle) permitem definir funções que devolvem tabelas
  - As funções e procedimentos são armazenados na própria base de dados
    - Definem funcionalidades disponíveis a vários utilizadores
- Grande parte dos sistemas de bases de dados têm linguagens proprietárias onde se podem definir funções e procedimentos, e que diferem bastante do standard SQL
- No Oracle podem-se criar funções e procedimentos através da linguagem PL/SQL, ou diretamente na base de dados.

Funções SQL

- Definir uma função que, dado o nome de um cliente, devolva o número de contas de que ele é titular.

```
create function account_count (customer_name varchar(20))
returns integer
begin
    declare a_count integer;
    select count (*) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Encontrar o nome e morada dos clientes com mais do que uma conta.

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```

Funções que retornam Tabelas

- O standard SQL também inclui funções que devolvem uma relação como resultado.
- Examplo: Devolver todas as contas de um dado cliente

```
create function accounts_of (customer_name char(20)
        returns table (     account_number char(10),
                            branch_name char(15)
                            balance numeric(12,2)))
return table
        (select account_number, branch_name, balance
         from account A
         where exists (
            select *
            from depositor D
            where D.customer_name = accounts_of.customer_name
                  and D.account_number = A.account_number ))
```

- Utilização

```
select *
from table (accounts_of ( 'Smith' ))
```

Funções e procedimentos SQL

- A função *account_count* pode ser escrita como procedimento:

```
create procedure account_count_proc (in customer_name varchar(20),
                                     out a_count integer)
begin
  select count(*) into a_count
  from depositor
  where depositor.customer_name = account_count_proc.customer_name
end
```

- Os procedimentos podem ser chamados dentro de outros procedimentos SQL, ou de linguagens SQL embedded ou proprietárias.
  - E.g. num procedimento SQL

```
declare a_count integer;
call account_count_proc( 'Smith' , a_count);
```

  - O standard SQL permite que haja mais que uma função ou procedimento com o mesmo nome, desde que o número de argumentos (ou, pelo menos, os seus tipos) sejam diferentes

Funções e procedimentos externos

- O standard SQL permita o uso de funções e procedimentos escritos noutras linguagens (e.g. C ou C++)
- A declaração de funções e procedimentos externos faz-se da seguinte forma:

create procedure account_count_proc(**in** *customer_name* **varchar**(20),**out** count **integer**)
language C
external name ’ /usr/avi/bin/account_count_proc’

create function account_count(*customer_name* **varchar**(20))
returns integer
language C
external name ‘/usr/avi/bin/author_count’

Funções e procedimentos externos (Cont.)

- Vantagens:
  - Mais eficiente para muitas operações
  - Mais poder expressivo
- Desvantagens
  - O código que implementa as rotinas externas pode ter que ser carregado no sistema de bases de dados e executado no espaço de endereços deste
    - risco de corromper acidentalmente a estrutura da base de dados
    - risco de segurança dos dados
  - Há alternativas que garante segurança (à custa, por vezes, da deterioração da performance)
  - A execução direta no sistema de bases de dados só é feita se a eficiência for bem mais importante que a segurança

Segurança para rotinas externas

- Para lidar com estes problemas de segurança
  - Usar técnicas de **sandbox**
    - i.e. usar linguagem segura como o Java, que não permite o acesso a outras parte do código da base de dados
  - Ou executar rotinas externas em processo separado, sem acesso à memória usada por outros processos do sistema de bases de dados
    - Os parâmetro e resultados são passados via comunicação entre processos
- Ambas as alternativas têm custos de performance

Integridade de Bases de Dados em SQL

- Tópicos:
  - Restrições ao domínio e chaves
  - Asserções
  - Triggers
- Bibliografia:
  - Secções 4.4 e 5.3 do livro recomendado

Restrições ao domínio

- Já vimos que a DDL do SQL permite definir restrições ao domínio:
  - **not null** elimina o valor *null* do domínio de atributos
  - **check (***Cond***)** restringe o domínio apenas aos valores que tornam a condição *Cond* verdadeira
    - As condições têm que poder ser verificadas tuplo a tuplo, e só podem referir atributos da tabela em causa
- Por exemplo:

```
create table products (
        product_no integer not null,
        name varchar2(50) not null,
        price number check (price > 0),
        discounted_price number check (discounted_price > 0),
        check (price > discounted_price)
        );
```

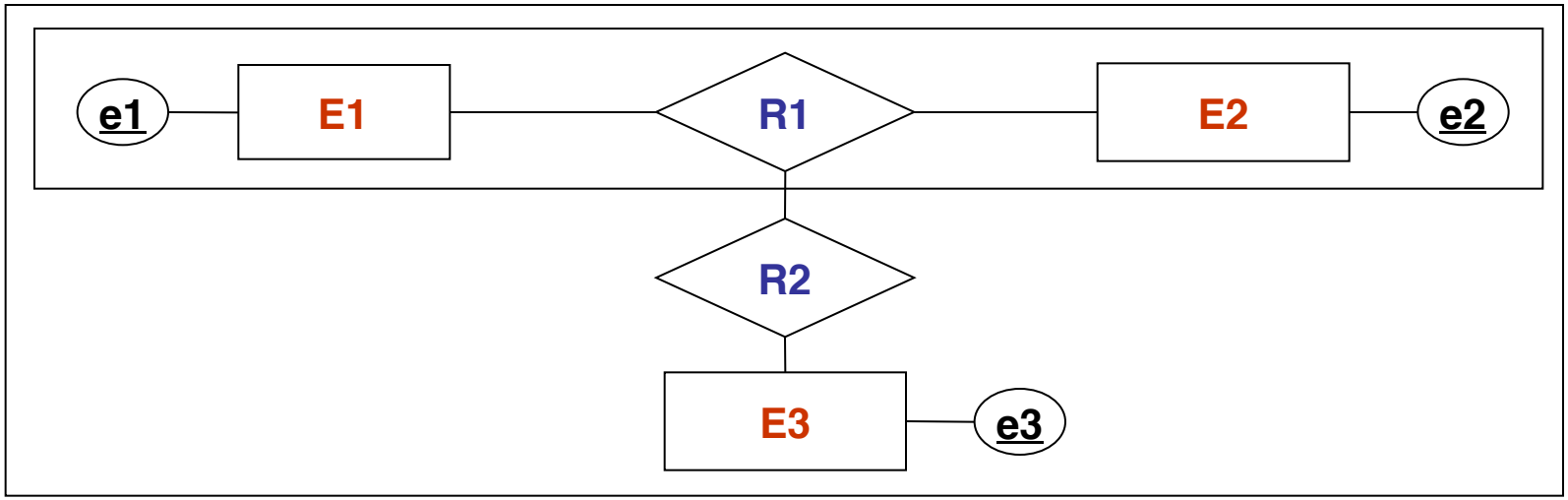
Chaves primárias e candidatas

- Também vimos na DDL que se pode usar:
  - **primary key** denotando que o(s) atributo(s) são chave primária
    - Só pode haver uma declaração de **primary key** por tabela
    - Nos atributos da **primary key** não são permitidos valores *null*
  - **unique(***A1, ..., An***)** impondo que o conjunto de atributos *A1, ..., An* é chave candidata
    - I.e. não podem haver 2 tuplos com os mesmos valores em todos os atributos *A1, ..., An*
- Ao contrário da **primary key**
  - podem haver vários **unique** numa tabela
  - os atributos em **unique** podem ter valores *null*
    - Nestes casos, os *null* são assumidos como todos diferentes uns dos outros
    - I.e. pode haver 2 tuplos com *null* num atributo de uma chave candidata – não podem é haver 2 tuplos com um mesmo valor diferente de *null* num atributo de uma chave candidata

Chaves estrangeiras

- A DDL também permite definir chaves estrangeiras em tabela *R*
  - **foreign key (***A1,...,An***) references** *S***(***B1,...,Bn***)**
  - Impõe que $\Pi_{A1,...,An}(R) \subseteq \Pi_{B1,...,Bn}(S)$
  - Pode-se omitir os atributos *B1,...,Bn*; nesse caso assume-se que esses atributos são os que formam a chave primária de *S*
- Nas chaves estrangeiras, os valor *null* são ignorados
  - I.e. a restrição acima só é imposta se os atributos tiverem valores diferentes de *null*
  - Dito de outra forma, a chave estrangeira acima impõe que, para cada tuplo de *R*
    - ou os valores nos atributos *A1,...,An* são *null*
    - ou então tem que existir algum tuplo em *S* com esses valores nos atributos nos atributos *B1,...,Bn*

Integridade Referencial em SQL – Exemplo



create table E1(e1 number(3) **not null primary key**);

create table E2(e2 number(3) **not null primary key**);

create table E3(e3 number(3) **not null primary key**);

create table R1(e1 number(3) **not null**,
e2 number(3) **not null**,
primary key (e1, e2),
foreign key (e1) **references** E1,
foreign key (e2) **references** E2);

create table R2(e1 number(3) **not null**,
e2 number(3) **not null**,
e3 number(3) **not null**,
primary key (e1, e2, e3),
foreign key (e1,e2) **references** R1,
foreign key (e3) **references** E3);

Acções em Cascata em SQL

create table *account*

. . .
foreign key*(branch_name)* **references** *branch*
on delete cascade
on update cascade
. . .)

- Com as cláusulas **on delete cascade**, se a remoção de um tuplo na relação *branch* resulta na violação da restrição da integridade referencial, a remoção propaga-se em "cascata" para a relação *account*, removendo o tuplo que referia a agência que tinha sido eliminada.
- Actualizações em cascata são semelhantes. Não estão implementadas pelo Oracle!

Acções em cascata em SQL (cont.)

- Se existe uma cadeia de dependências de chaves externas através de várias relações, com um **on delete cascade** especificado em cada dependência, uma remoção ou actualização num dos extremos pode-se propagar através de toda a cadeia.
- Se uma remoção ou actualização em cascata origina uma violação de uma restrição que não pode ser tratada por uma outra operação em cascata, o sistema aborta a transacção. Como resultado, todas as alterações provocadas pela transacção e respectivas acções em cascata serão anuladas.
- Alternativas às operações em cascata:
  - **on delete set null**
  - **on delete set default**

Asserções

- O SQL permite ainda impor restrições de integridade mais gerais:
  - Uma *asserção* é um predicado que exprime uma condição que gostaríamos de ver sempre satisfeita na base de dados.
- Em SQL as asserções têm a forma:

  create assertion <nome> **check** <predicado>
- Quando se define uma asserção, o sistema testa-a, e volta a testá-la, sempre que há modificações na base de dados (que a possam violar)
  - Estes testes podem introduzir um overhead significativo; logo as asserções são para usar com cuidado e de forma comedida.
  - Por isso, embora o standard SQL preveja a existência de asserções, a maior parte dos sistemas não o implementa.
    - O Oracle não permite definir asserções!

Exemplo de Asserção

- Em cada balcão, a soma dos montantes de todos os seus empréstimos tem que ser sempre inferior à soma de todos os seus depósitos.

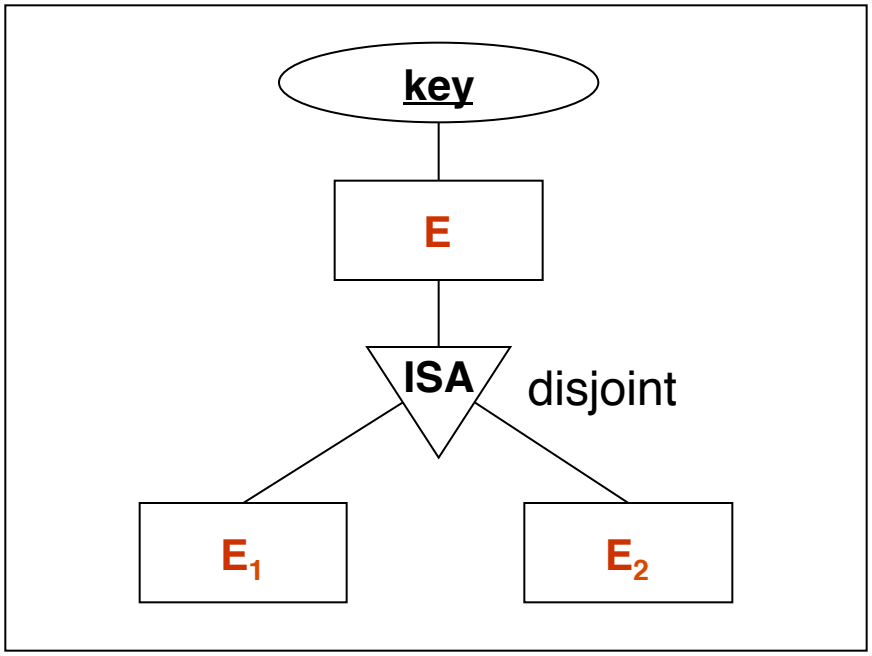
```
create assertion sum_constraint check
    (not exists (select * from branch
                 where
                     (select sum(amount) from loan
                        where loan.branch_name =
                              branch.branch_name
                     )
                    >= some
                     (select sum(balance) from account
                        where account.branch_name =
                              branch.branch_name
                     )
                 )
    )
```

Outro Exemplo

- Todo o empréstimo tem que estar sempre ligado a pelo menos um cliente de uma conta (de depósito) cujo saldo é não inferior a metade do valor do empréstimo

```
create assertion balance_constraint check
   (not exists (
       select * from loan
       where not exists (
           select *
           from borrower natural inner join depositor
                                natural inner join account
           where loan.loan_number = borrower.loan_number
                 and account.balance >= 0,5 * loan.ammount
                                )
                  )
    )
```

Exemplo de Asserção



- Uma especialização duma entidade geral E (com chave key) em E_1 e E_2 é disjunta.

create assertion *disjE1E2* **check**
(not exists ((select *key* **from** E_1**) intersect (select** *key* **from** E_2**)))**

Triggers

- Um **trigger** é um "comando" que é executado automaticamente pelo sistema, como side-effect duma modificação à base de dados dum determinado tipo pré-definido.
- Para definir um trigger, há que:
  - Especificar que *evento* faz disparar o trigger
  - Especificar em que *condições* o trigger deve ser executado.
  - Especificar que *ação* fazer quando o trigger é executado.
- São conhecidos como **event-condition-action rules**
- Os triggers são armazenados na base de dados, e executados para todos as interações com esta.
- Triggers foram introduzidos no standard SQL:1999, mas eram suportados anteriormente por sintaxe não standard, logo:
  - O Oracle suporta triggers, embora com uma sintaxe ligeiramente diferente da do SQL.
  - Em geral, cada sistema tem a sua sintaxe e pecularidades

Exemplo de Trigger

- Imagine uma situação em que o banco aceita que haja saldos negativos e, nesses casos:
  - coloca o saldo a 0
  - cria um empréstimo com o valor em dívida
  - Atribui a este empréstimo um número idêntico ao da conta de depósito
- O trigger deve ser executado sempre que há uma atualização na relação *account* que faz com que o saldo passe a negativo.

Codificação do Exemplo em SQL

```
create trigger overdraft_trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
        insert into borrower
           (select customer_name, account_number
            from depositor
            where nrow.account_number =
                        depositor.account_number);
        insert into loan values
           (nrow.account_number, nrow.branch_name,
                                                – nrow.balance);
        update account set balance = 0
        where account.account_number = nrow.account_number
end
```

Eventos e Acções de Triggers em SQL

- Os eventos que podem fazer disparar um trigger são **insert**, **delete** ou **update**
- No Oracle, também podem disparar triggers eventos de **servererror**, **logon**, **logoff**, **startup** e **shutdown**.
- Triggers sobre **update** podem-se restringuir só a alguns atributos
  - **E.g. create trigger** *overdraft_trigger* **after update of** *balance* **on** *account*
- Pode-se referenciar o valor dos atributos antes e depois da modificação
  - **referencing old row as** : para **delete**s e **update**s
  - **referencing new row as** **:** para **insert**s e **update**s
- Pode-se fazer disparar um trigger antes do evento, para codificar restrições. E.g. converter espaços em **null**.

```
create trigger setnull_trigger before update on r
referencing new row as nrow
for each row
    when nrow.phone_number = ' '
    set nrow.phone_number = null
```

- Para além do **before** e do **after** no Oracle existe também o **instead of.**

Ações Externas

- Por vezes podemos querer que um dado evento faça disparar uma ação para o exterior.
  - Por exemplo, numa base de dados de uma armazém, sempre que a quantidade de um produto desce abaixo (devido a um **update)** de um determinado valor podemos querer encomendar esse produto, ou disparar algum alarme.
- Os triggers não podem ser usados para implementar ações sobre o exterior, mas...
  - podem ser usados para guardar numa tabela separada ações-a-levar-a-cabo. Podem depois haver procedimentos que, periodicamente verificam essa tabela separada.
- E.g. Uma base de um armazém com as tabelas
  - *inventario(item, quant):* Que quantidade há de cada produto
  - *quantMin(item, quant) :* Qual a quantidade mínima de cada produto
  - *reposicoes(item, quant):* Quanto encomendar sempre que está em falta
  - *aencomendar(item, quant) :* Coisas a encomendar (lido por procedimento)

Exemplo de Ações Externas

```
create trigger aenc_trigger after update of quant on inventario
referencing old row as orow, new row as nrow
for each row
        when nrow.quant <= some (select quant
                                 from quantMin
                                 where quantMin.item = orow.item)
                and orow.quant > some (select quant
                                       from quantMin
                                       where quantMin.item = orow.item)
  begin
          insert into aencomendar
                (select item, quant
                  from reposicoes
                  where reposicoes.item = orow.item)
  end
```

Sintaxe de Triggers em Oracle

create [**or replace**] **trigger** *<nome_trigger>*
{**before** | **after** [**instead of**} *<evento>*
[**referencing old as** *<nome_antes>*]
[**referencing new as** *<nome_depois>*]
for each row
when *<condição>*
begin
<Sequencia de comandos, terminados por ;>
end;
/

- *Evento* pode ser:
  - **delete on** *<tabela ou view>*
  - **insert on** *<tabela ou view>*
  - **update on** *<tabela ou view>*
  - **update of** *<atributos separados por ,>* **on** *<tabela ou view>*
  - **servererror**, **logon**, **logoff**, **startup** ou **shutdown**
- Os comandos são PL/SQL o que inclui os comandos SQL, mais WHILEs, IFs, etc (ver manuais)
- Dentro da condição os *nome_antes* e *nome_depois* podem ser usados sem mais. Mas nos comandos têm que ter o símbolo ‘:’ antes!!!

Statement Triggers

- São executados após (antes, ou em vez de) uma instrução completa vs. os anteriores que são executadas após alterações em cada linha
- Sintaxe:
  create [**or replace**] **trigger** *<nome_trigger>*
  {**before** | **after** | **instead of**} *<evento>*
  begin
  <Sequencia de comandos, terminados por ;>
  end;

- Para ser usado quando as condições são para testar globalmente e não linha a linha.

Uso de triggers

- Os triggers permitem uma grande generalidade na imposição de restrições e, também por isso mesmo, devem ser usados com grande cuidado.
- Podem se usar para implementar assertions, fazendo raise_application_error quando as condições não se verificam.
- Não usar triggers:
  - Quando as restrições podem ser impostas doutra forma (com a exceção das asserções)!!
    - Os triggers são mais difíceis de manter e são menos eficientes.
  - Quando se querem manter sumários
    - Para tal usem-se **view**s e se eficiência for importante usem-se **materialized view**s
  - Replicar a base de dados
    - Os sistemas modernos têm mecanismos muito eficientes e de baixo nível para replicação das bases de dados

Outros problemas com triggers (Cont.)

- Podem ser executados quando não se pretende:
  - Ler dados de uma cópia de backup
  - Replicar atualizações num site remoto
  - A execução dos triggers pode ser desligada antes de executar as acções anteriores
- Outros riscos com triggers:
  - Erros podem levar à falha de transações críticas que disparam o trigger
  - Execução em cascata
  - Tabelas em mutação

Triggers para actualização de vistas

- Podemos utilizar triggers para efectuar modificações através de vistas.
- Para tal, criamos triggers para todas as operações permitidas, como por exemplo:
  - para a inserção (do tipo instead of insert on),
  - para a remoção (do tipo instead of delete on)
  - para a actualização (do tipo instead of update on).
- Consideremos a vista:

create view *info_empréstimos* **as**
select *loan_number, customer_name, amount*
from *borrower* **natural inner join** *loan*

Triggers para actualização de vistas

- Se quisermos permitir a remoção de empréstimos através da vista, criamos o trigger:

```
create trigger remove_empréstimos
        instead of delete on info_empréstimos
        referencing old row as orow
        for each row
begin
        delete from loan
                where loan_number = orow. loan_number ;
        delete from borrower
                where loan_number = orow. loan_number ;
end
```

Triggers para actualização de vistas

- Se quisermos permitir a inserção de empréstimos através da vista, criamos o trigger:

```
create trigger insere_empréstimos
        instead of insert on info_empréstimos
        referencing new row as nrow
        for each row
begin
        insert into loan
                values (nrow.loan_number, NULL, amount);
        insert into borrower
                values (nrow.customer_name, nrow.loan_number)
end
```

Triggers para actualização de vistas

- Se quisermos permitir a actualização do valor do empréstimo através da vista, criamos o trigger:

```
create trigger actualiza_empréstimos
        instead of update of amount on info_empréstimos
        referencing new row as nrow
        referencing old row as orow
        for each row
begin
        update loan
        set amount = nrow. amount
                where loan_number = orow. loan_number ;
end
```

Triggers para inserção de chaves

- Se quisermos preencher automaticamente a chave de um tuplo, quando da sua inserção, recorrendo a uma sequência:

```
create trigger chave_aluno
before insert on alunos
for each row
declare
        aluno_id number;
begin
        select seq_aluno.nextval into aluno_id
        from dual;
        :new.num_aluno := aluno_id;
end
```