Capítulo 4: SQL

- Linguagem de Definição de Dados
- Estrutura básica
- Operações com conjuntos
- Funções de agregação
- Valores nulos
- Subconsultas embutidas
- Relações derivadas
- Junções
- Vistas
- Modificação da Base de Dados
- Embedded SQL
- Dynamic SQL
- Funções e Procedimentos
- SQL Recursivo

Relações Derivadas

- O SQL permite a utilização de sub-consultas na cáusula **from**.
- Determinar o saldo médio das contas em agências cujo saldo médio é superior a $1200.

```
select branch_name, avg_balance
from (select branch_name, avg (balance) as avg_balance
        from account
        group by branch_name
     )
where avg_balance > 1200
```

Repare que neste caso não foi necessário recorrer à cláusula **having**, dado que é calculada a relação temporária *result* na cláusula **from**, e assim os atributos de *result* pode ser utilizado directamente numa cláusula **where**.

Cláusula With

- A cláusula **with** permite a definição temporária de relações, cuja definição está disponível na consulta onde a clausula **with** ocorre. Análogo a procedimentos locais de uma linguagem de programação.
- Encontrar as contas de maior saldo

with *max_balance*(*value*) **as**
    (**select** max (*balance*)
      from *account*)
select *account_number*
from *account, max_balance*
where *account.balance = max_balance.value*

Exemplo com with

- Quais as agências cuja soma de saldos das suas contas é superior à média da soma dos saldos de todas as agências.

```
with branch_total (branch_name, value) as
   ( select branch_name, sum (balance)
   from account
   group by branch_name),
branch_total_avg(value) as
   ( select avg (value)
   from branch_total )
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

Particularidades do With em Oracle

- O Oracle 11g não permite “column aliasing” na instrução WITH. O comando do acetato anterior deve ser escrito:

```
with branch_total (branch_name, value) as
( select branch_name as branch_name, sum (balance) as value
    from account
    group by branch_name
),
branch_total_avg (value) as
( select avg(value) as value
    from branch_total
)
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

Operações de Junção

- As operações de junção retornam uma relação como resultado da combinação de duas outras relações.
- Estas operações adicionais são utilizadas habitualmente em subconsultas na cláusula **from**
- **Condição de junção** – define quais os tuplos que são combinados nas duas relações, assim como quais os atributos que aparecem no resultado da junção.
- **Tipo de junção** – define como tratar os tuplos que não estão relacionados entre si (basedados na condição de junção).

Tipos de Junção
inner join **left outer join** **right outer join** **full outer join**

Condições de Junção
natural **on** <predicate> **using** (A_1, A_2, ..., A_n)

Operações de Junção

Tipos de Junção
inner join
left outer join
right outer join
full outer join

Condições de Junção
natural
on <predicate>
using (A_1, A_2, ..., A_n)

- É obrigatória a utilização de uma condição de junção nos junções de tipo **outer**. É opcional no **inner join** (na ausência, comporta-se como o produto cartesiano)
- A condição **natural** aparece antes do tipo de junção (por exemplo **natural inner join**). As restantes condições aparecem após o tipo de junção.
- Nas junções com condição **natural**, primeiro aparecem os atributos comuns a ambas as relações, na ordem pela qual aparecem na relação do lado esquerdo. Depois, aparecem os restantes atributos da relação do lado esquerdo, seguidos dos restantes atributos da relação do lado direito.
- Nas junções com condição **using** (A_1, A_2, ..., A_n), primeiro aparecem os atributos A_1, A_2, ..., A_n. Depois, aparecem os restantes atributos da relação do lado esquerdo, seguidos dos restantes atributos da relação do lado direito.

Junção versus Produto Cartesiano

- Listar o nome, número de empréstimo e montante de todos os clientes que efectuaram um empréstimo na agência de Perryridge.

  select *borrower.*, amount*
  from *borrower, loan*
  where *borrower.loan_number = loan.loan_number* **and**
  branch_name = 'Perryridge'

  - Versus

    select ***
    from *borrower* **natural inner join** *loan*
    where *branch_name =* 'Perryridge'

- A última separa claramente onde se coloca a origem dos dados de onde se colocam as condições "de filtragem" (selecção)
  - Esta separação não só torna a leitura mais fácil, como pode ser aproveitada para implementações.
- Na última consulta não se pode colocar "*borrower.**" após o **select** pois deixa de ser possível utilizar o nome da tabela nas colunas usadas para fazer a junção natural (isto é, as comuns a ambas as tabelas)

Relações de Exemplo

- Relação *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relação *borrower*

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Exemplos

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

- **select** * **from** *loan* **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*

l.loan_number	branch_name	amount	customer_name	b.loan_number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

select * **from** *loan* **left outer join** *borrower* **on** *loan.loan_number = borrower.loan_number*

l.loan_number	branch_name	amount	customer_name	b.loan_number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Exemplos

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

select * from *loan* natural inner join *borrower*

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

select * from *loan* natural right outer join *borrower*

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Exemplos

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

select * **from** *loan* **full outer join** *borrower* **using** *(loan_number)*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	null	null	Hayes

Utilização típica de USING

- Considerem-se as tabelas **cursos(cod_curso,nome)** e **alunos(num_aluno, nome, cod_curso)**

- A interrogação

  select * **from** *cursos* **natural inner join** *alunos*

  é equivalente a

  select *cursos.cod_curso as cod_curso, cursos.cursos.nome as nome, alunos.num_aluno*
  from *cursos, alunos*
  where cursos.cod_curso = alunos.cod_curso and cursos.nome = alunos.nome

- Enquanto que o comando

  select * **from** *cursos* **inner join** *alunos* **using** *(cod_curso)*

  corresponde a

  select *cursos.cod_curso as cod_curso, cursos.nome, alunos.num_aluno, alunos.nome*
  from *cursos, alunos*
  where cursos.cod_curso = alunos.cod_curso

- Qual é a correcta?

Aspectos a ter cuidado

- Ocasionalmente as junções podem ser ambíguas quando estão envolvidas mais do que 2 tabelas:

  select *
  from a **natural left outer join** b **left outer join** c **on** b.c1 = c.c1

- Pode ser interpretado como:

  select *
  from (a **natural left outer join** b) **left outer join** c **on** b.c1 = c.c1

  ou:

  select *
  from a **natural left outer join** (b **left outer join** c **on** b.c1 = c.c1)

- Os operadores associam à esquerda, ou seja, a primeira hipótese é a executada. Na dúvida usar os parêntesis, em particular quando há muitas condições de junção expressas através de ON.

Exemplos

Listar todos os clientes que têm uma conta e nenhum empréstimo

select *customer_name*
from *depositor* **left outer join** *borrower* **using(**customer_name**)**
where *loan_number* **is null**

Listar todos os clientes que têm uma conta ou um empréstimo no banco (mas não ambos!)

select *customer_name*
from *depositor* **natural full outer join** *borrower*
where *account_number* **is null or** *loan_number* **is null**

Vistas

- Em certas circunstâncias, não é desejável que todos os utilizadores possam aceder a todo o modelo lógico (i.e. a todas as relações armazenadas na base de dados).
- Considere o caso de um empregado que necessita de saber o número de empréstimo de um cliente, mas que não precisa de saber o montante desse empréstimo. Este empregado deverá ver apenas a relação descrita por

  select *costumer_name, loan_number*
  from *borrower* **natural inner join** *loan*

Vistas

- Mesmo que não seja por razões de segurança, pode ser útil apresentar um conjunto de relações personalizada que se adapte melhor às intuições de um dado utilizador do que o modelo lógico.
- Considere o caso de um empregado do departamento de marketing que poderá preferir ver uma relação contendo os clientes que têm uma conta ou um empréstimo no banco, e as respectivas agências. Tal relação seria:

  (**select** *branch_name, customer_name*
  from *depositor* **natural inner join** *account*)

  union

  (**select** *branch_name, customer_name*
  from *borrower* **natural inner join** *loan*)

- Qualquer relação que não pertença ao modelo lógico mas que se torne visível ao utilizador como uma “relação virtual” é designada por *vista*.

Vistas

- Mecanismo que permite apresentar um modelo diferente do modelo lógico, e.g. ocultando (escondendo) informação de certos utilizadores. Para criar uma vista utilizamos o comando:

  create view *v* **as** <query expression>

  em que:

  - <query expression> é qualquer expressão SQL válida
  - O nome da vista é *v*

- Em SQL do Oracle pode-se escrever

  create or replace view *v* **as** <query expression>

  para criar ou substituir uma vista já existente, evitando a utilização do comando **drop view**.

Exemplo

- Uma vista contendo todas as agências e respectivos clientes

create view *all_customer* **as**
(**select** *branch_name, customer_name*
from *depositor* **natural inner join** *account)*
union
(**select** *branch_name, customer_name*
from *borrower* **natural inner join** *loan)*

- Listar todos os clientes da agência de Perryridge

select *customer_name*
from *all_customer*
where *branch_name* = 'Perryridge'

- Uma definição de uma vista não é o mesmo que a criação duma nova relação a partir da avaliação da sua expressão. Em vez disso, a definição da vista permite guardar a expressão que depois é substituída nas consultas que utilizam a vista.

Definição de vistas

- Uma vista pode ser utilizada na expressão de definição de outra vista. Por exemplo:

  create view *perryridge_customer* **as**
  select *customer_name*
  from *all_customer*
  where *branch_name* = 'Perryridge'

- Uma vista v_1 *depende directamente* de uma vista v_2 se v_2 é utilizada na expressão que define v_1
- Uma vista v_1 *depende de* uma vista v_2 se v_1 depende directamente de v_2 ou se existe um caminho de dependências entre v_1 e v_2
- Uma vista *v* diz-se *recursiva* se depender dela própria.
- Em SQL não são permitidas vistas recursivas!

Expansão de vistas

- Forma de atribuir significado a vistas definidas em termos de outras vistas.
- Seja a vista v_1 definida em termos de uma expressão e_1 que pode ela própria conter vistas.
- Para expandir as vistas numa expressão repete-se sucessivamente o seguinte passo:

  repeat
  Encontrar uma vista v_i em e_1
  Substituir a vista v_i pela expressão que a define
  until não ocorram mais vistas em e_1
- Desde que as definições das vistas não sejam recursivas, este ciclo terminará sempre.

Expansão de vistas (exemplo)

- A expressão:

```
select *
from perryridge_customer
where customer_name = 'John'
```

- ... é inicialmente expandida para:

```
select *
from (select customer_name
        from all_customer
        where branch_name = 'Perryridge')
where customer_name = 'John'
```

- ... que por sua vez é expandida para:

```
select *
from (select customer_name
        from (    (select branch_name, customer_name
                   from depositor natural inner join account)
                   union
                   (select branch_name, customer_name
                   from borrower natural inner join loan))
                   where branch_name = 'Perryridge')
        where customer_name = 'John'
```

Modificação da base de Dados – Remoção

- A remoção de tuplos de uma tabela (ou vista) é feita em SQL com a instrução

  delete from *<tabela ou vista>*
  where *<Condição>*

- Apagar todas as contas da agência de Perryridge

  delete from *account*
  where *branch-name* = 'Perryridge'

Exemplo de remoção

- Apagar todas as contas de todas as agências na cidade de Needham.

```
delete from depositor
where account-number in
                (select account-number
                  from branch natural inner join account
                  where branch-city = 'Needham')

delete from account
where branch-name in (select branch-name
                              from branch
                              where branch-city = 'Needham')
```

Exemplo de remoção

- Remover todas as contas com saldos inferiores aos da média do banco.

  delete from *account*
  where *balance* < **some** (**select avg** *(balance)*
  from *account)*

  - Problema:
    - à medida que removemos tuplos de *deposit,* o saldo médio altera-se
  - Solução utilizada no standard SQL:
    1. Primeiro, calcula-se o saldo médio (**avg**) e determinam-se quais os tuplos a apagar
    2. Seguidamente, removem-se todos os tuplos identificados anteriormente (sem recalcular **avg** ou testar novamento os tuplos)

Modificação da base de dados – Inserção

- A inserção de tuplos numa tabela (ou vista) é feita em SQL com a instrução

  insert into *<tabela ou vista>*
  values *<Conjunto de tuplos>*

ou

  insert into *<tabela ou vista>*
  select *...*

- Adicionar um novo tuplo a *account*

  insert into *account*
  values ('A-9732', 'Perryridge',1200)

  ou equivalentemente

  insert into *account (branch-name, balance, account-number)*
  values ('Perryridge', 1200, 'A-9732')

- Adicionar um novo tuplo a *account* em que *balance* é null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Exemplo de Inserção

- Dar como bónus a todos os mutuários da agência de Perryride, uma conta de poupança de $200. O número do empréstimo servirá de número de conta de poupança

```
insert into account
    select loan-number, branch-name, 200
    from loan
    where branch-name = 'Perryridge'
insert into depositor
    select customer-name, loan-number
    from loan natural inner join borrower
    where branch-name = 'Perryridge'
```

- A instrução **select-from-where** é avaliada antes da inserção de tuplos na relação (caso contrário consultas como
  insert into *table*1 **select** * **from** *table*1
  causariam problemas)

Modificação da base de dados – Actualização

- A actualização de tuplos duma tabela (ou vista) é feita em SQL com a instrução

  update *<tabela ou vista>*
  set *<Atributo> = <Expressão>, <Atributo> = <Expressão>, ...*
  where *<Condição>*

- Pagar juros de 1% a todas as contas da agência Perryride.

  update *account*
  set *balance = balance* * 1.01
  where *branch_name* = 'Perryride'

Modificação da base de dados – Actualização

- Pagar juros de 6% a todas as contas com saldos superiores a $10,000, e juros de 5% às restantes contas.
  - Escrever duas instruções de **update**:

update *account*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* * 1.05
where *balance* ≤ 10000

  - A ordem é importante. Porquê?
  - Pode ser efectuado de maneira mais “limpa” recorrendo à instrução **case**

Instrução Case para Actualizações Condicionais

- Pagar juros de 6% a todas as contas com saldos superiores a $10,000, e juros de 5% às restantes contas.

```
update account
set balance =  case
                  when balance <= 10000 then balance *1.05
                  else   balance * 1.06
               end
```

Actualização de uma vista

- Modificações nas bases de dados através de vistas devem ser traduzidas para modificações das verdadeiras relações presentes na base de dados.
  - E.g com uma vista com a informação sobre empréstimos, escondendo o atributo *amount*

    create view *branch-loan* **as**
    select *branch-name, loan-number*
    from *loan*

  - a adição de um novo tuplo em *branch-loan*

    insert into *branch-loan* **values** (‘Perryridge’, ‘L-307’)

  - Causa problemas pois terá que ser traduzido em adições de tuplos em tabelas que existam na base de dados. Duas hipóteses:
    - Rejeitar a inserção e devolver uma mensagem de erro
    - Traduzir na inserção, na relação *loan,* do tuplo

      (‘L-307’, ‘Perryridge’, *null*)

Actualização de uma vista (cont.)

- Outro problema ocorre quando temos, por exemplo, a vista:

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

e pretendemos fazer a seguinte inserção:

insert into *info_empréstimos* **values** *('Johnson',1900)*

- *A única forma seria inserir ('Johnson' ,null) na tabela* borrower *e (null,null,1900) na tabela* loan, não tendo o efeito desejado.

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

loan

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

Actualização de uma vista (cont)

- Outras não têm tradução única, como por exemplo:

  create view *all_costumers* **as**
  (select * **from** *depositor)*

  union
  *(***select** * **from** *borrower)*

- Toda a adição em *all_costumers* não tem tradução única:
  - Deve introduzir-se em *depositor* ou em *borrower*???

Actualização de vistas

- Uma view em SQL é actualizável (updatable) se todas as seguintes condições se verificam:
  - A claúsula **from** só contém uma relação da base de dados;
  - A claúsula **select** apenas contém nomes de atributos da relação, não contendo expressões, agregados, ou especificação de **distinct**;
  - Qualquer atributo que não aparece na claúsula **select** deve poder tomar o valor null;
  - A consulta não contém nenhuma claúsula **group by** nem **having**.
- A view

  create view *downtown_account* **as**
  select *acount_number,branch_name,balance*
  from *account*
  where *branch-name* = 'Downtown'

... é actualizável. No entanto, a inserção

insert into *downtown_account* **values** ('L-307','Perryridge',1000)

apesar de ser efectuada, não produziria efeitos na view.

Actualização de vistas

- Em Oracle 11g é possível impedir as situações anteriores por intermédio da cláusula WITH CHECK OPTION na criação da vista.

  create view *downtown_account* **as**
  select *acount_number,branch_name,balance*
  from *account*
  where *branch-name* = ‘Downtown’
  with check option

- Para impedir a actualização de vistas utiliza-se a cláusula WITH READ ONLY

Capítulo 4: SQL

- Linguagem de Definição de Dados
- Estrutura básica
- Operações com conjuntos
- Funções de agregação
- Valores nulos
- Subconsultas embutidas
- Relações derivadas
- Junções
- Vistas
- Modificação da Base de Dados
- Embedded SQL
- Dynamic SQL
- Funções e Procedimentos
- SQL Recursivo

Embedded SQL

- SQL fornece uma linguagem *declarativa* para manipulação de bases de dados. Facilita a manipulação e permite optimizações muito difíceis se fossem programadas em linguagens imperativas.
- Mas há razões para usar SQL juntamente com linguagens de programação gerais (imperativas):
  - o SQL não tem a expressividade de uma máquina de Turing (há perguntas impossíveis de codificar em SQL – e.g. fechos transitivos)
    - usando SQL juntamente com linguagens gerais é possível suprir esta deficiência
  - nem tudo nas aplicações de bases de dados é declarativo (e.g. acções de afixar resultados, interfaces, etc)
    - Essa parte pode ser programado em linguagens gerais
- O standard SQL define uma série de embeddings, para várias linguagens de programação (e.g. Pascal, PL/I, C, C++, Cobol, etc).
- À linguagem na qual se incluem comandos SQL chama-se linguagem *host*. Às estruturas SQL permitidas na linguagem *host* chama-se SQL embutido (ou *embedded* SQL)

Embedded SQL

- Permite acesso a bases e dados SQL, 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 { .... } ;

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 (eg 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

- 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);
```

JDBC

- JDBC é uma API **Java** para comunicar com sistemas de bases de dados que suportam o SQL.
- JDBC suporta várias formas de consulta e modificação de bases de dados
- O modelo de comunicação com a base de dados:
  - Abre uma ligação
  - Cria um objecto “statement”
  - Executa comandos usando esse objecto para enviar os comandos e obter os resultados
  - Usa mecanismos de excepção para lidar com os erros

Exemplo de código JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn =
          DriverManager.getConnection(   "jdbc:oracle:thin:@aura.bell-
          labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

Exemplo de código JDBC (Cont.)

- Actualização

```
try {
    stmt.executeUpdate(  "insert into account values
                                ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execução de perguntas

```
ResultSet rset = stmt.executeQuery( "select branch_name,
    avg(balance)
                                        from account
                                        group by branch_name");
while (rset.next()) {
     System.out.println(
                rset.getString("branch_name") + "  " + rset.getFloat(2));

}
```

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 (eg 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)
  - Excepções (para tratamento de erros)
- Unidades de programas em PL/SQL podem ser compilados na base de dados Oracle.

Construtores procedimentais

- O SQL:1999 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
```

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 excepção e erros, e declaração de tratamento de excepções

```
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 SQL:1999 suporta funções e procedimentos
  - As funções e procedimentos podem ser escritas directamente 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
- 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:1999
- No Oracle podem-se criar funções e procedimentos através da linguagem PL/SQL, ou directamente 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

- SQL:2003 acrescenta 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 SQL:1999 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 SQL:1999 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 directa 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

Recursão em SQL

- SQL:1999 permite definição recursiva de vistas
- Exemplo: encontrar todos os pares empregado-chefe, onde o empregado responde directa ou indirectamente ao chefe (i.e. ao chefe do chefe do chefe, etc.)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

Esta vista, empl, é o fecho transitivo da relação *manager*

O poder da recursão

- As vistas recursivas permitem a escrita de consultas, tais como as de fecho transitivo, que não podem ser escritas sem recursão (ou iteração).
  - Intuição: Sem recursão, uma um programa não-iterativo e não-recursivo só pode calcular um número fixo de junções de *manager* consigo própria.
    - Isto só pode fornecer um número fixo de níveis de chefias
- Cálculo do fecho transitivo
  - Cada passo do processo iterativo constroi uma versão estendida de *empl a partir da sua definição recursiva*.
  - O resultado final é chamado de *ponto-fixo da definição recursiva da vista*.
- As vistas recursivas têm que ser *monotónicas.* I.e. se acrescentarmos tuplos à relação *manager*, a vista contém todos os tuplos que continha anteriormente, e possivelmente mais.

Exemplo de uma computação ponto-fixo

```
with recursive empl (emp_name, man_name ) as
        select emp_name, man_name
        from manager
    union
        select manager.emp_name, empl.man_name
        from manager, empl
        where manager.man_name = empl.emp_name
select *
from empl
```

1st iteration
2nd iteration
3rd iteration
4th iteration
5th iteration

manager

emp_name	*man_name*
António	Bruno
Bruno	Eduardo
Carla	Duarte
Duarte	João
Eduardo	João
João	Lara
Rita	Lara

empl

emp_name	*man_name*
António	Bruno
Bruno	Eduardo
Carla	Duarte
Duarte	João
Eduardo	João
João	Lara
Rita	Lara
António	Eduardo
Bruno	João
Carla	João
Duarte	Lara
Eduardo	Lara
António	João
Bruno	Lara
Carla	Lara
António	Lara