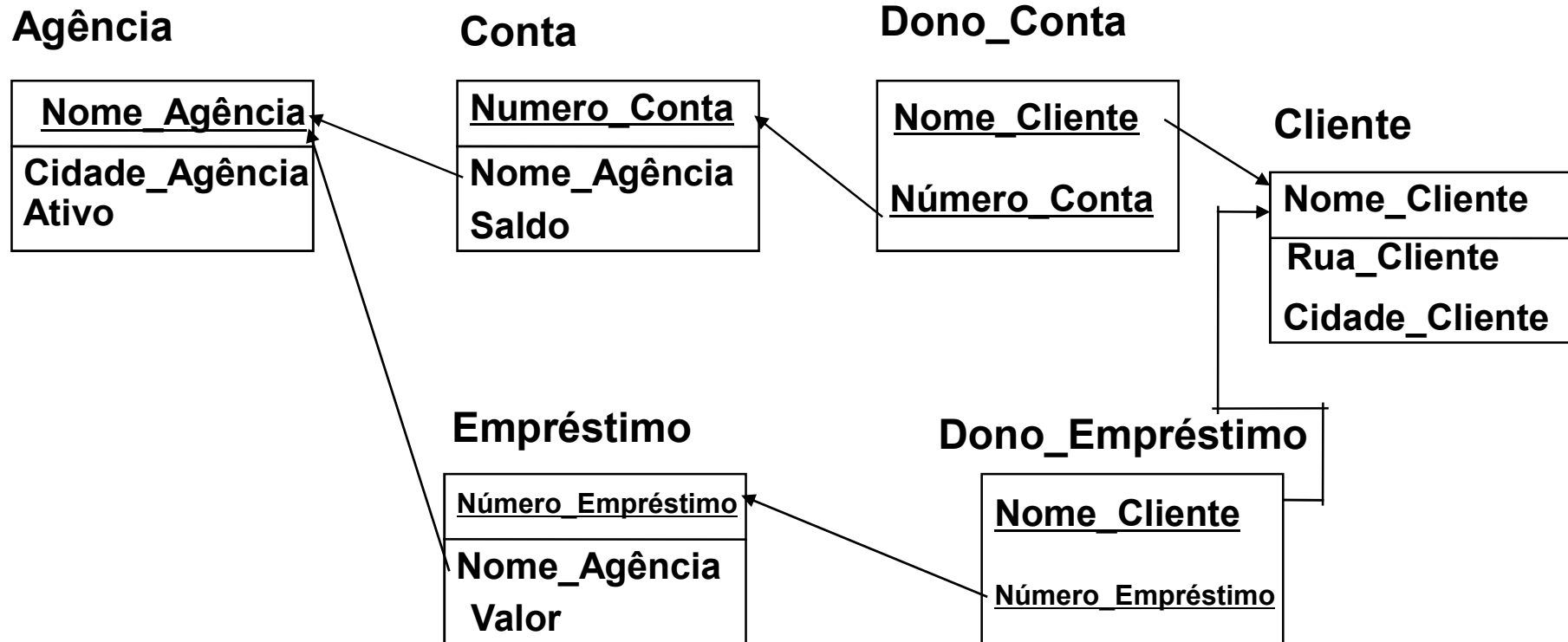


# Novo Esquema Exemplo



# Valores Nulos

- É possível que as tuplas tenham um valor nulo, descrito por *null*, para alguns de seus atributos
- *null* significa um valor desconhecido ou um valor que não existe.
- O predicado **is null** pode ser usado para verificar valores nulos.
  - E.g. Ache todos os números de empréstimos que aparecem na relação empréstimo com valores nulos para valor

```
select Numero_Empréstimo  
      from Empréstimo  
      where valor is null
```

- O resultado de qualquer expressão aritmética que considera *null* é *null*
  - E.g. 5 + null retorna null
- Entretanto, as funções agregadas simplesmente ignoram nulls

# Valores Nulos e Lógica de três valores

- Qualquer comparação com *null* retorna *unknown*
  - *E.g. 5 < null ou null <> null ou null = null*
- Lógica tri-valorada que usa o valor de verdade *unknown*:
  - **OR:** *(unknown or true) = true, (unknown or false) = unknown, (unknown or unknown) = unknown*
  - **AND:** *(true and unknown) = unknown, (false and unknown) = false, (unknown and unknown) = unknown*
  - **NOT:** *(not unknown) = unknown*
  - “*P is unknown*” avalia se o predicado *P* é *unknown*
- Os resultados do predicado da cláusula *where* é tratado como *false* se é *unknown*

---

# Valores Nulos e Agregados

- **Quantidade Total de todos os Empréstimos**  
**`select sum (valor)`**  
**`from Empréstimo`**
    - ❑ O comando acima ignora as quantidades nulas
    - ❑ O resultado é null se não há quantidades não null
  - **Todas as operações de agregação exceto `count(*)` ignoram as tuplas com valores null sobre os atributos agregados.**
-

---

# Teste da Ausência de Tuplas Duplicatas

- A cláusula **unique** testa se uma subconsulta tem algumas tuplas duplicatas em seu resultado.
- Ache todos os clientes que tem no máximo uma conta na agência 'Perryridge'.

```
select T.Nome_Cliente  
from Dono_Conta as T  
where unique (  
    select R.Nome_cliente  
        from Conta, Dono_Conta as R  
        where T.Nome_Cliente = R.Nome_cliente and  
            R.Número_Conta = Conta.Número_Conta and  
            Conta.Nome_Agência = 'Perryridge')
```

- (Esquema usado en este ejemplo)
-

# Exemplo

- **Ache todos os clientes que tem no mínimo duas contas na agência Perryridge.**

```
select distinct T.Nome_Cliente  
from Dono_Conta T  
where not unique (  
  select R.Nome_Cliente  
  from Conta, Dono_Conta as R  
  where T.Nome_Cliente = R.Nome_Cliente  
  and  
    R.Número_Conta = Conta.Número_Conta  
  and  
    Conta.Nome_Agência = 'Perryridge'  
)
```

■ (Esquema usado en este ejemplo)

# Junção de Relações

- **Operações de junção pegam duas relações e retornam como resultado outra relação.**
- **Estas operações adicionais são tipicamente usadas como expressões de subconsulta na cláusula from**
- **Condição de junção – define que tuplas nas duas relações se concatenam, e que atributos estão presentes no resultado da junção.**
- **Tipo de junção – define como as tuplas de cada relação que não combinam com nenhuma tupla da outra relação são tratadas (baseado na condição de junção).**

Tipo de junção
<b>inner join</b> <b>left outer join</b> <b>right outer join</b> <b>full outer join</b>

Condição de Junção
<b>natural</b> <b>on</b> <predicado> <b>using</b> ( $A_1, A_2, \dots, A_n$ )

# Em Oracle

- Você pode usar três abordagens para criar um equi-join:
- NATURAL JOIN – cria uma junção automaticamente entre duas tabelas, baseada em colunas com nomes iguais.
- USING – cria junções baseada numa coluna que tem o mesmo nome e definição em ambas as tabelas.
- Quando as tabelas a serem juntadas na cláusula USING não têm um campo do mesmo nome e do mesmo tipo, deve-se adicionar a cláusula ON.



# Junção de Relações – Dados para os Exemplos

## ■ Relação *Empréstimo*

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

## ■ Relação *Dono\_Empréstimo*

<i>Nome_Cliente</i>	<i>Número_Empréstimo</i>
Jones	L-170
Smith	L-230
Hayes	L-155

- 
- **Nota: Dono\_Empréstimo não tem a informação de L-260 e Empréstimo não tem a informação de L-155**

## *Junção de Relações – Exemplos*

- ***Empréstimo inner join*** ***Dono\_Empréstimo on***  
***Empréstimo.Número\_Empréstimo= Dono\_Empréstimo.Número\_Empréstimo***

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>	<i>Nome_Cliente</i>	<i>Número_Empréstimo</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- ***Empréstimo left outer join*** ***Dono\_Empréstimo on***  
***Empréstimo.Número\_Empréstimo= Dono\_Empréstimo.Número\_Empréstimo***

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>	<i>Nome_Cliente</i>	<i>Número_Empréstimo</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>

# Junção de Relações – Exemplos

## ■ *Empréstimo* natural inner join *Dono*\_*Empréstimo*

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>	<i>Nome_Cliente</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

## ■ *Empréstimo* natural right outer join *Dono*\_*Empréstimo*

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>	<i>Nome_Cliente</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

# Junção de Relações – Exemplos

- ***Empréstimo* full outer join *Dono\_Empréstimo* using (*Número\_Empréstimo*)**

<i>Número_Empréstimo</i>	<i>Nome_Agência</i>	<i>Valor</i>	<i>Nome_Cliente</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes

- **Achar todos os clientes que têm ou uma conta ou um empréstimo (mas não os dois) no banco.**

```
select Nome_Cliente  
from (Dono_Conta natural full outer join Dono_Empréstimo)  
where Número_Conta is null or Número_Empréstimo is null
```

---

# Visões

- **Fornecer um mecanismo para esconder certos dados da visão de certos usuários. Para criar uma visão nos usamos o comando:**

**create view *v* as <expresión de consulta>**

**onde:**

👉 **<expresão de consulta> é qualquer expressão legal**

👉 **O nome da visão é representado por *v***

---

## Duas visões especificadas para o esquema do banco de dados da Empresa.

```
V1: CREATE VIEW  TRABALHA_EM1
      AS  SELECT  PNAME, UNOME, PJNAME, HORAS
            FROM    EMPREGADO, PROJETO, TRABALHA_EM
            WHERE   SSN=ESSN AND PNO=PNUMERO;
V2: CREATE VIEW  DEPT_INFO(DEPT_NOME,NO_EMPS,TOTAL_SAL)
      AS  SELECT  DNAME, COUNT (*), SUM (SALARIO)
            FROM    DEPARTAMENTO, EMPREGADO
            WHERE   DNUMERO=DNO
            GROUP BY DNAME;
```

TRABALHA\_EM1

PNAME	UNOME	PJNAME	HORAS
-------	-------	--------	-------

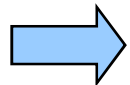
DEPT\_INFO

DEPT_NOME	NO_EMPS	TOTAL_SAL
-----------	---------	-----------

## O exemplo do livro

```
SELECT PNOME, UNOME  
FROM TRABALHA_EM1  
WHERE PJNOME = 'ProjetoX'
```

Modificando a consulta:



```
SELECT PNOME, UNOME  
FROM EMPREGADO, PROJETO, TRABALHA_EM  
WHERE SSN=ESSN AND PNO=PNUMERO  
AND PJNOME = 'ProjetoX'
```

---

# Exemplos

- Uma visão que consiste de agências e seus clientes

```
create view todo-cliente as  
(select Nome_Agencia, Nome_Cliente  
from Dono_Conta, Conta  
where Dono_Conta.Número_Conta=Conta.Numero_Conta)  
union  
(select Nome_Agencia, Nome_Cliente  
from Dono_Emprestimo, Empréstimo  
where  
Dono_Empréstimo.Numero_Emrestimo=Emréstimo.Número_Emprestimo)
```

- Ache todos os clientes da agência Perryridge

```
select Nome_Cliente  
from todo-cliente  
where Nome_Agência ='Perryridge'
```

---



---

# Relações Derivadas

- **Ache o saldo médio das contas das agências onde seu saldo médio das contas é maior de \$1200.**

```
select Nome_Agência, Avg-Saldo  
from (select Nome_Agência, avg (saldo)  
from Conta  
group by Nome_Agência)  
as result (Nome_Agência, Avg-Saldo)  
where Avg-Saldo > 1200
```

**Veja que nos não precisamos usar a cláusula having, já que calculamos a relação temporal (visão) *result* na cláusula from, e os atributos de *result* podem ser usados diretamente na cláusula where.**

---

---

# Cláusula With

- A cláusula With permite que as visões possam ser definidas localmente a uma consulta, no lugar de globalmente. Análogo a procedimentos numa linguagem de programação.
- Ache todas as contas com o máximo saldo

```
with max-saldo(valor) as  
  select max (saldo)  
  from Conta  
select Número_Conta  
from Conta, max-saldo  
where Conta.saldo = max-saldo.valor
```

---

# Consulta complexa usando a cláusula With

- Ache todas as agências onde o saldo total de suas contas é maior que a média dos saldos totais de todas as agências.

```
with Total_Agência (Nome_Agência, valor) as  
  select Nome_Agência, sum (saldo)  
  from Conta  
  group by Nome_Agência  
with Média_Total_Agência(valor) as  
  select avg (valor)  
  from Total_Agência  
select Nome_Agência  
from Total_Agência, Média_Total_Agência  
where Total_Agência.valor >= Média_Total_Agência.valor
```

# Atualização de uma Visão

- Criar uma visão de todos os dados de empréstimo na relação *Empréstimo*, escondendo o atributo *Valor*

**create view Agência-Empréstimo as  
select Nome\_Agência, Número\_Empréstimo  
from *Empréstimo***

- Adicione uma nova tupla a *Agência-Empréstimo*

**insert into Agência-Empréstimo  
values ('Perryridge', 'L-307')**

**Esta inclusão deve ser representada pela inclusão da tupla**

**('L-307', 'Perryridge', *null*)**

**na relação *Empréstimo***

- Atualizações de visões mais complexas são difíceis ou impossíveis de mapear, e portanto não são permitidas.
- Muitas implementações SQL permitem atualizações unicamente sobre visões simples (sem agregados) definidas sobre uma só relação

---

# Implementação e operações sobre visões

```
CREATE VIEW PEÇAS_VERMELHAS ( P#, PNOME,PS, CIDADE) AS  
      SELECT    P#, PNOME, PESO, CIDADE  
      FROM      P  
      WHERE     COR = 'Vermelha';
```

## REMOÇÃO DE VISÕES

**Sintaxe**      **DROP VIEW** <Nome visão>

**Exemplo:**

```
DROP VIEW PEÇAS_VERMELHAS
```

**A visão; Peças\_Vermelhas é removida do catálogo. Toda visão definida em termos desta visão também são removidas automaticamente.**

---

---

# Implementação e operações sobre visões

- Visões são implementadas para consultas como:
    - Modificações de consultas
    - Materialização da visão
  - No caso de atualização, elas podem ser complicadas ou impossíveis. Exs:
-

# Views Materializadas

## **Listing 7-8.** *Example of a Materialized View*

```
CREATE MATERIALIZED VIEW Orders_LineItems
REFRESH ON COMMIT
ENABLE QUERY REWRITE
AS SELECT
    -- data items from the Orders table
    Orders.PONo,
    Orders.Custno,
    Orders.OrderDate,
    Orders.ShipDate,
    Orders.ToStreet,
    Orders.ToCity,
    Orders.ToState,
    Orders.ToZip,
    -- data items from the LineItems table
    LineItems.LineNo,
    LineItems.StockNo,
    LineItems.Quantity,
    LineItems.Discount
FROM Orders, LineItems
WHERE LineItems.PONo = Orders.PONo;
```

---

# Atualizações

**Considere as seguintes visões**

```
CREATE VIEW F#_CIDADE  
AS SELECT F#, CIDAD E  
FROM F;
```

```
CREATE VIEW STATUS_CIDADE  
AS SELECT STATUS , CIDADE  
FROM F;
```

**F#\_CIDADE**      **teóricamente é atualizável, STATUS\_CIDADE não**  
**¿Por que ?**

---



---

**Seja: F#\_CIDADE**

**1) Incluir um novo registro ('F6', 'MANAUS')**

**2) Remover o registro ('F1', 'SP')**

**3) Atualizar a cidade de 'F3' de 'BH' para 'SP'**

**no caso de STATUS\_CIDADE**

**1) Incluir um novo registro (40, 'SP')**

**2) Remover o registro (20, 'BH')**

**3) Atualiza o registro (20, 'BH') para (20, 'SP')**

**A diferença é que F#\_CIDADE é atualizável, teoricamente, sse esta preserva a CHAVE primária da tabela básica.**

---

```
CREATE VIEW PQ ( P#, QUANTOTAL)  
AS SELECT P#, SUM (QUANTIDADE)  
FROM FP  
GROUP BY P#;
```

**ESTA VISÃO NÃO SUPORTA OPERAÇÕES INSERT NEM OPERAÇÕES UPDATE sobre o campo QUANTOTAL**

**Operações DELETE e operações UPDATE no campo P# poderiam teóricamente serem definidas.**

**Caso “junção” → Nenhum produto permite sua atualização**

**OUTROS CASOS:**

```
CREATE VIEW BONS_FORNECEDORES  
AS SELECT F#, STATUS, CIDADE  
FROM F  
WHERE STATUS > 15
```

---

**Ela é atualizável, porém:**

**a) Com a tabela da figura, F. 'F2' não seria visível.**

**O que acontece se fosse incluída uma tupla com valor 'F2' ou actualiza uma tupla existente com o valor 'F2' ?**

**b) Considere a seguinte actualização.**

```
UPDATE    BONS_FORNECEDORES  
SET       STATUS = 5  
WHERE     F # = 'F1';
```

**Deveria ser aceiteada ?**

---

---

```
INSERT  
INTO    BONS_FORNECEDORES (F#, STATUS, CIDADE)  
VALUES  ['F13', 5, 'Porto Alegre']
```

**CHECK OPTION** trata estas situações:

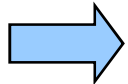
```
CREATE VIEW    BONS_FORNECEDORES  
AS SELECT     F#, STATUS, CIDADE  
FROM          F  
WHERE         STATUS > 15  
WITH CHECK OPTION;
```

---

## O exemplo do livro

```
SELECT PNOME, UNOME  
FROM TRABALHA_EM1  
WHERE PJNOME = 'ProjetoX'
```

Modificando a consulta:



```
SELECT PNOME, UNOME  
FROM EMPREGADO, PROJETO, TRABALHA_EM  
WHERE SSN=ESSN AND PNO=PNUMERO  
AND PJNOME = 'ProjetoX'
```

---

## O que acontece com atualização

```
UPDATE TRABALHA_EM1  
SET PJNOME = 'ProdutoY'  
WHERE UNOME = 'Smith' AND PNOME = 'John'  
      AND PJNOME = 'ProdutoX'
```

Pode ser mapeada em diversas atualizações:

a) 

```
UPDATE PROJETO SET PJNOME = 'ProdutoY'  
WHERE PJNOME = 'ProdutoY'
```

---

# O que acontece com atualização?

```
b) UPDATE TRABALHA_EM
    SET PNO = (SELECT PNUMERO
                FROM PROJETO
                WHERE PJNOME = 'ProdutoY')
    WHERE ESSN IN (SELECT SSN
                   FROM EMPREGADO
                   WHERE UNOME='Smith' AND PNOME = 'John')
    AND
    PNO = (SELECT PNUMERO
            FROM PROJETO
            WHERE PJNOME = 'ProdutoX')
```

---

## Então, quando são possíveis as atualizações de visões?

- Em geral ela é viável quando apenas uma atualização é possível, nas relações básicas.
  - Resumindo:
    - Uma visão de uma única tabela básica é atualizável se ela contiver nos seus atributos, a chave primária da tabela básica, bem como todos os atributos com restrição NOT NULL que não tiverem valores default especificados.
    - As visões definidas a partir de diversas tabelas utilizando-se as junções, em geral, não são atualizáveis.
    - As visões definidas usando-se as funções de agrupamento não são atualizáveis.
-



---

# **Independência lógica de dados**

**Para que são exatamente as visões ?**

**Um sistema oferece independência lógica de dados se os programas dos usuários são independentes da estrutura lógica do banco de dados.**

**Dois conceitos para este tipo de independência:**

**Crescimento**

**Reestruturação.**

---

---

# **CRESCIMENTO**

**Um SGBD deve permitir a evolução no esquema. Facilita a manutenção do sistema.**

**Existem duas formas de crescimento:**

**1. A expansão de uma tabela básica existente, com um novo campo.**

**2. A inclusão de uma nova tabela básica.**

**Nenhum destes dois tipos de mudanças deveria ter qualquer efeito sobre os programas de usuários (outras visões).**

---

---

# **REESTRUTURAÇÃO**

**As vezes é necessário a reestruturação do banco de dados.**

**Por exemplo:**

**Por questões de eficiência é necessário dividir a tabela F em duas :**

**FX ( F#, FNOME, CIDADE )**

**FY ( F#, STATUS )**

**Importante : a relação F é a “junção” das relações FX e FY.**

---

# REESTRUTURAÇÃO

**Assím deve ser criada a visão :**

<b>CREATE</b>	<b>VIEW</b>	<b>F ( F#, F NOME, STATUS, CIDADE )</b>
<b>AS</b>	<b>SELECT</b>	<b>FX.F#, FX. F NOME, FY.STATUS, FY.CIDADE</b>
	<b>FROM</b>	<b>FX, FY</b>
	<b>WHERE</b>	<b>FX.F# = FY.F# ;</b>

**Desta maneira qualquer programa que referenciava F não precisa ser atualizado.**

**As operações de SELECT sobre F terão um “overhead” de tempo de execução.**

**F é uma visão teóricamente atualizável.**

**¿ Por qué ?**

---

# **VANTAGENS DAS VISÕES**

- **Oferecem uma certa independência lógica.**
  - **Permitem que o mesmo dado possa ser visto por usuários de diferentes formas.**
  - **A percepção do usuário é simplificada.**
  - **O usuário preocupa-se apenas com os dados que lhe são interessantes e esquece o resto.**
  - **Segurança automática é oferecida por dados escondidos ( discutido mais na frente ).**
-

# VIEWS EM POSTGRESQL

```
CREATE VIEW minha_visao AS
    SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
```

```
SELECT * FROM minha_visao;
```

```
CREATE [ OR REPLACE ] VIEW nome [ ( nome_da_coluna [, ...] ) ] AS consulta
```

As visões são somente de leitura, na versão 8. É possível obter o efeito de uma visão atualizável, criando regras que reescrevem as inserções, etc. na visão como ações apropriadas em outras tabelas.

```
CREATE VIEW minha_visão AS SELECT * FROM minha_tabela;
```

quando se compara com os dois comandos

```
CREATE TABLE minha_visão (mesma lista de colunas de minha_tabela);
CREATE RULE "_RETURN" AS ON SELECT TO minha_visão DO INSTEAD
    SELECT * FROM minha_tabela;
```

# Transações

- **Uma transação é uma sequência de consultas e comandos de atualização executados como uma só unidade**
  - **As transações são inicializadas implicitamente e terminadas por um dos seguintes comandos:**
    - **commit work: Faça todas as atualizações da transação permanentes no banco de dados**
    - **rollback work: desfça todas as atualizações desenvolvidas pela transação.**
- **Exemplo de motivação**
  - **Transfira dinheiro de uma conta a outra implica dois passos:**
    - **tirar de uma conta e depositar em outra**
  - **Se um dos passos tem sucesso e o outro falha, o BD cai num estado inconsistente**
  - **Portanto, os dois devem ter sucesso o nenhum deles.**
- **Se qualquer passo de uma transação falha, todo o trabalho feito pela transação pode ser desfeito pelo rollback work.**
- **O Rollback de transações incompletas é feito automaticamente, no caso de falhas do sistema**

---

# Transações (Cont.)

- **Muitos SGBDs, cada comando SQL que se executa com sucesso é automaticamente confirmado (committed).**
    - **Cada transação então consistiria de unicamente um só comando**
    - **Confirmação automática pode usualmente ser “desligada” (turned off), permitindo transações multi-comandos, mas como fazer isto depende do SGBD**
    - **Outra opção em SQL:1999: colocar os comandos dentro do  
begin atomic  
...  
end**
-



---

# Transações em PostgreSQL

```
BEGIN;  
UPDATE conta_corrente SET saldo = saldo - 100.00  
    WHERE nome = 'Alice';  
-- etc etc  
COMMIT;
```



Pode ser  
ROLLBACK

---

# Programação com o Banco de Dados

- Uma interface interativa é muito conveniente para a criação de esquemas e restrições ou para consultas ad hoc eventuais.
  - Mas para construir aplicações mais elaboradas: existem diversas técnicas de interação com o BD.
-


---

# Programação com o Banco de Dados

- Embutindo os comandos de BDs numa linguagem de programação de propósito geral. Precompiladores.
  - Usando uma biblioteca de funções para o BD. APIs  $\Rightarrow$  JDBC, CLI
  - Projetando uma nova linguagem. Linguagem de programação de um banco de dados  $\Rightarrow$  PL/SQL, Transact SQL, PL/pgSQL
-

---

## Impedância de Correspondência – “Impedance Mismatch”

- Termo usado para se referir aos problemas que ocorrem em decorrência das diferenças entre modelos de dados e LPs.
  - Problemas:
    - Tipos de dados diferentes. Necessário estabelecer uma forma de correspondência para cada LP.
    - O resultado da maioria das consultas é um conjunto de tuplas. Não existe uma estrutura equivalente nos LPs  Cursor ou variável iterativa.
  - Este problema é solucionado totalmente com as LPBDs
-

## Seqüência Típica de Interação em Programação com o BD

- Quando um programador cria um programa que acessa um BD. É muito comum que o programa rode em um computador enquanto o BD está em outro  $\implies$  C/S
- Sequência de interação:
  - Estabelecer uma conexão com o servidor de BD (url servidor, login, senha)
  - Interagir com o BD, através de consultas, atualizações, etc. A maioria das declarações SQL podem ser inseridas.
  - Fechar a conexão.
- Um programa pode acessar diversos BDs

---

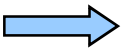
# SQL Incorporado ou Embutida

- **O SQL padrão define a incorporação de SQL numa variedade de linguagens de programação tais como Pascal, PL/I, Fortran, C, Java e Cobol.**
- **Uma linguagem para qual os comandos SQL são incorporados é denominada uma linguagem hospedeira (host). Ela e as estruturas SQL permitidas na linguagem hospedeira formam o SQL embutida.**
- **A forma básica destas linguagens segue as regras definidas no Sistema R incorporando SQL em PL/I.**
- **O comando EXEC SQL é usado para identificar os comandos SQL incorporados dentro do programa pelo pré-processador ou pré-compilador**  
**EXEC SQL <embedded SQL statement > END-EXEC**

**PS: isto muda de acordo ao linguagem. E.x: Java preprocessador usa**  
**# SQL { .... };**

---

# SQL Embutida

- Para ilustrar os conceitos, vamos usar C como linguagem hospedeira. Dentro dos comandos SQL embutidos, podemos nos referir especificamente a variáveis declaradas do programa C  variáveis compartilhadas.
- Elas são usadas dentro dos comandos SQL com um prefixo “:”.
- Suponhamos o BD da EMPRESA.
- As variáveis compartilhadas são declaradas dentro de uma seção de declaração.

**EMPREGADO**

PNOME	MINICIAL	UNOME	<u>SSN</u>	DATANASC	ENDEREÇO	SEXO	SALARIO	SUPERSSN	DNO
-------	----------	-------	------------	----------	----------	------	---------	----------	-----

**DEPARTAMENTO**

DNOME	<u>DNUMERO</u>	GERSSN	GERDATAINICIO
-------	----------------	--------	---------------

**DEPTO\_LOCALIZACOES**

<u>DNUMERO</u>	<u>DLOCALIZACAO</u>
----------------	---------------------

**PROJETO**

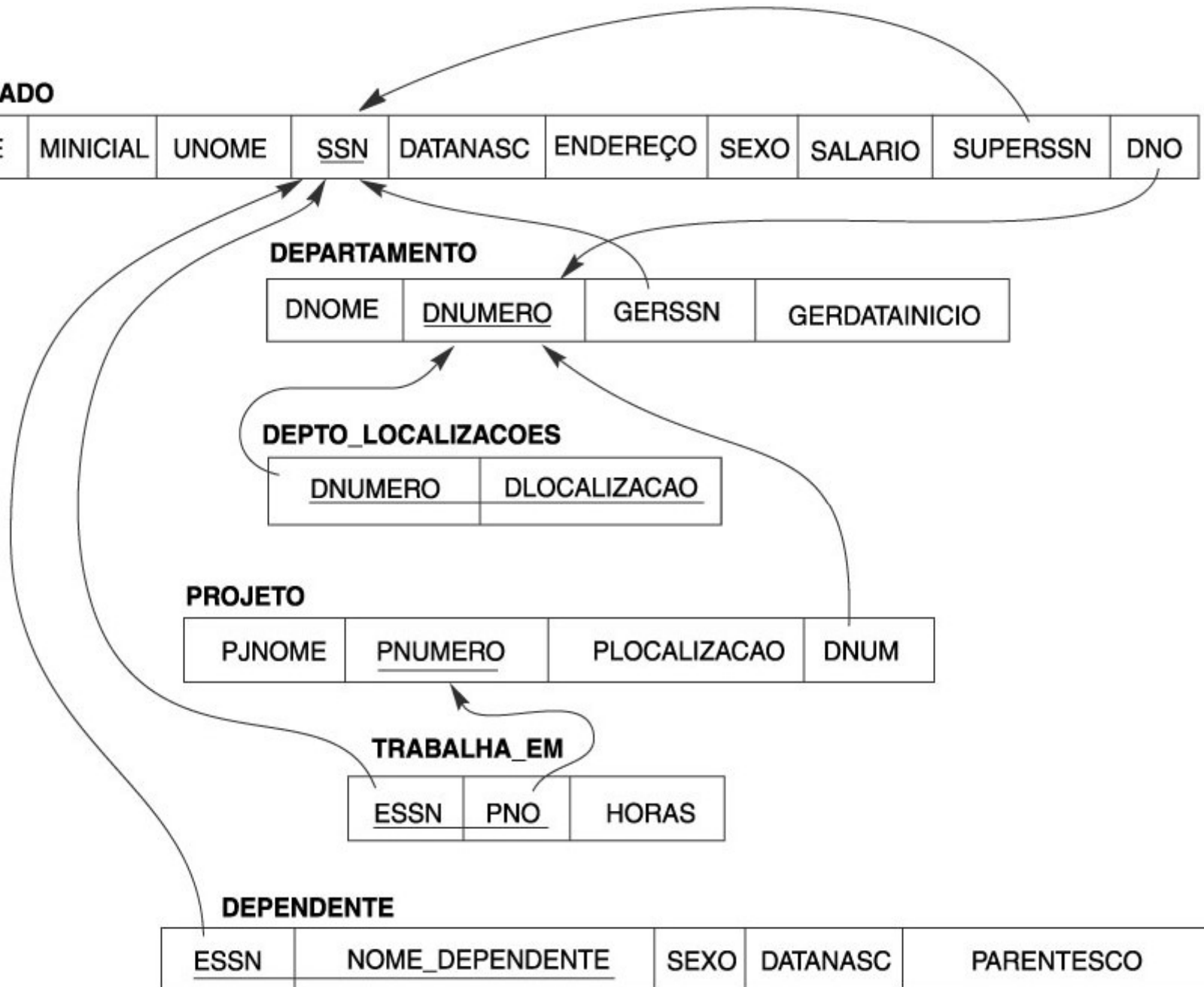
PJNOME	<u>PNUMERO</u>	PLOCALIZACAO	DNUM
--------	----------------	--------------	------

**TRABALHA\_EM**

<u>ESSN</u>	<u>PNO</u>	HORAS
-------------	------------	-------

**DEPENDENTE**

<u>ESSN</u>	<u>NOME_DEPENDENTE</u>	SEXO	DATANASC	PARENTESCO
-------------	------------------------	------	----------	------------





## Variáveis do programa C usadas nos exemplos de SQL embutida E1 e E2.

```
0) int loop;  
1) EXEC SQL BEGIN DECLARE SECTION ;  
2) varchar dnome [16], pnome [16], unome [16], endereço [31] ;  
3) char ssn [10], datnasc [11], sexo [2], inicial [2] ;  
4) float salario, aumento;  
5) int dno, dnumero ;  
6) int SQLCODE ; char SQLSTATE [6] ;  
7) EXEC SQL END DECLARE SECTION ;
```

As linhas 2 a 5 correspondem aos atributos das tabelas EMPREGADO e DEPARTAMENTO. As variáveis da linha 6 são usadas para a comunicação de erros e condições de exceção entre o SBD e o programa.

# Conectando o BD

CONNECT TO <nome do servidor> AS <nome da conexão>  
AUTHORIZATION <nome e senha do usuário da conta>;

- Um programa, em geral, pode acessar diversos servidores de BDs, várias conexões podem ser estabelecidas, mas somente uma conexão poderá ser ativa por vez. Para trocar a conexão ativa:

SET CONNECTION <nome da conexão>;

- Quando não for mais necessária:

DISCONNECT <nome da conexão>;

- Nos exemplos pressupomos uma conexão prévia.

---

# Comunicação entre o Programa e o SGBD.

- Depois da execução de cada comando do BD, o SGBD devolve um valor para o SQLCODE.
  - SQLCODE = 0; execução com sucesso.
  - SQLCODE > 0 (especif. = 100), indica que não há mais dados disponíveis no resultado.
  - SQLCODE < 0, ocorreu um erro.
  - Em ORACLE, por exemplo, SQLCODE é um campo de um registro chamado SQLCA.
  - Precisa incluir o seguinte comando no programa C:
    - ❑ EXEC SQL include SQLCA;
-

# Comunicação entre o Programa e o SGBD.

- Últimas versões de SQL, foi adicionada a variável SQLSTATE, cadeia de 5 caracteres.
- O valor “00000” indica que não houve erro ou exceção. Outros valores apontam por erros.  
Ex:
  - “02000” indica que não há mais dados.
- Muitos dos códigos de erro de SQLSTATE estão padronizados pelas diversas plataformas SQL. SQLCODE não.
- Melhor usar SQLSTATE. Uso independente do SGBD.

---

# Exemplo de Programação com a SQL Embutida

```
//Segmento de Programa E1:
0)   loop = 1 ;
1)   while (loop) {
2)       prompt ("Entre com o Numero do Seguro Social: ", ssn) ;
3)       EXEC SQL
4)           select PNAME, MINICIAL, UNOME, ENDEREÇO, SALARIO
5)           into :pname, :minicial, :unome, :endereco, :salario
6)           from EMPREGADO where SSN = :ssn ;
7)       if (SQLCODE == 0) printf (pname, minit, unome, endereco, salario)
8)       else printf ("Numero do Seguro Social nao existe: ", ssn) ;
9)       prompt("Mais Numeros de Seguro Social (entre 1 para Sim, 0 para Nao): ", loop) ;
10)  }
```

Mas, quando retornar mais de uma tupla?

---

---

## Recuperando várias tuplas com a SQL Embutida Usando Cursores

- Podemos imaginar o cursor como um ponteiro que aponta a uma única tupla do resultado de uma consulta que retorna diversas consultas.
  - Ex: Desde dentro de uma linguagem hospedeira, achar os nomes e cidades dos clientes com um saldo maior que a variável *amount* dólares em alguma conta.
-

---

# Exemplo de Consulta

- Especifique a consulta em SQL e declare um *cursor* para isto  
EXEC SQL

```
declare c cursor for  
  select Nome_Cliente, Cidade_Cliente  
  from Dono_Conta, Cliente, Conta  
  where Dono_Conta.Nome_Cliente = Cliente.Nome_Cliente  
    and Dono_Conta. Número_Conta = Conta.Número_Conta  
    and Conta.Saldo > :amount  
END-EXEC
```

---

---

# Uso de Cursores (Cont.)

- O comando **open** causa que a consulta seja avaliada  
EXEC SQL **open** *c* END-EXEC (;
  - O comando **fetch** causa que os valores de uma tupla no resultado da consulta sejam colocadas nas variáveis da linguagem hospedeira.  
EXEC SQL **fetch** *c into* *:cn, :cc* END-EXEC (;  
Chamadas repetidas a **fetch** conseguem tuplas sucessivas do resultado da consulta
  - Uma variável chamada SQLSTATE na área de comunicação do SQL (SQLCA) vira '02000' para indicar que não existe mais dados disponíveis
  - O comando **close** faz com que o SMBD elimine a relação temporal que contem o resultado da consulta.  
EXEC SQL **close** *c* END-EXEC (;
-



---

# Atualizações por meio de Cursores

- Podem-se atualizar tuplas recuperadas por cursores declarando que o cursor é para ser atualizado.

```
declare c cursor for  
  select *  
  from Conta  
  where Nome_Agência = 'Perryridge'  
for update
```

- Para atualizar uma tupla na posição atual do cursor

```
update Conta  
set saldo = saldo + 100  
where current of c
```

---

---

//Segmento de Programa E2:

```
0)  prompt ("Entre com o Nome do Departamento: ", dnome) ;
1)  EXEC SQL
2)      select DNUMERO into :dnumero
3)      from DEPARTAMENTO where DNAME = :dnome ;
4)  EXEC SQL DECLARE EMP CURSOR FOR
5)      select SSN, PNAME, MINICIAL, UNOME, SALARIO
6)      from EMPREGADO where DNO = :dnumero
7)      FOR UPDATE OF SALARIO ;
8)  EXEC SQL OPEN EMP ;
9)  EXEC SQL FETCH from EMP into :ssn, :pname, :minicial, :unome, :salario ;
10) while (SQLCODE == 0) {
11)     printf("Nome do empregado e:", pname, minit, unome)
12)     prompt("Entre com o aumento de salario: ", aumento) ;
13)     EXEC SQL
14)         update EMPREGADO
15)         set SALARIO = SALARIO + :raise
16)         where CURRENT OF EMP ;
17)     EXEC SQL FETCH from EMP into :ssn, :pname, :minicial, :unome, :salario ;
18) }
19) EXEC SQL CLOSE EMP ;
```

---

# Formato Geral

```
DECLARE <nome do cursor> [INSENSITIVE] [SCROLL] CURSOR  
[WITH HOLD] FOR <especificação da consulta>  
[ORDER BY <especificação da ordenação>]  
[FOR READ ONLY | FOR UPDATE [OF <lista de atributos>]];
```

- UPDATE sem a lista de atributos é para o caso de remoção.
- SCROLL é para barrer o cursor de forma diferente a sequencial. Podemos definir uma orientação de busca diferente de NEXT. Por ex: PRIOR, FIRST, LAST, ABSOLUTE i, RELATIVE i

```
FETCH [[<orientação da busca> ] FROM ] <nome do cursor> INTO <lista  
de busca designada>
```

---

---

## Especificando consultas em tempo de execução usando SQL Dinâmica

- Nos exemplos anteriores as consultas são codificadas como parte do código fonte. Se precisar reformular a consulta, precisa recompilação, etc.
  - Em alguns casos, precisa-se um programa que possa executar consultas diferentes em SQL, ou atualizações, dinamicamente em tempo de execução. Veja exemplo:
-

## Segmento de Programa E3, um segmento de programa C que usa a SQL dinâmica para a atualização de uma tabela.

```
//Segmento de Programa E3:  
0) EXEC SQL BEGIN DECLARE SECTION ;  
1) varchar sqlatualizacadeia [256] ;  
2) EXEC SQL END DECLARE SECTION ;  
  
...  
3) prompt ("Entre com o Comando de Atualizacao: ", sqlatualizacadeia) ;  
4) EXEC SQL PREPARE sqlcomando FROM :sqlatualizacadeia ;  
5) EXEC SQL EXECUTE sqlcomando ;
```

Os comando 4 e 5 podem ser combinados numa única execução:  
EXEC SQL EXECUTE IMMEDIATE :sqlatualizacadeia;

---

# SQL Dinâmico

- Então, permite que os programas construam e executem comandos SQL em tempo de execução.
- Exemplo do uso de SQL dinâmico desde dentro de um programa em C.

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

- O programa com SQL dinâmico contém uma "?", que representa uma variável que deve ser proporcionada na execução do programa.
-

```
EXEC SQL BEGIN DECLARE SECTION;  
const char *stmt = "CREATE TABLE test1 (...);";  
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL EXECUTE IMMEDIATE :stmt;
```

```
EXEC SQL BEGIN DECLARE SECTION;  
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";  
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE mystmt FROM :stmt;
```

```
...
```

```
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

---

```
EXEC SQL BEGIN DECLARE SECTION;

const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";

int v1, v2;

VARCHAR v3;

EXEC SQL END DECLARE SECTION;


EXEC SQL PREPARE mystmt FROM :stmt;

...

EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```



---

# SQLJ: SQL Embutida em Comandos JAVA

- SQLJ: Padrão adotado para embutir SQL em Java. Estudaremos como SQLJ é usada em ORACLE.
  - Geralmente, um tradutor SQLJ converte as declarações SQL em Java, que podem ser executadas por uma interface JDBC.
  - Instalar driver de JDBC.
  - É necessário importar várias bibliotecas de classes.
-

## Importando as classes necessárias para inserir a SQLJ em um programa JAVA no ORACLE, estabelecendo uma conexão e *default context*.

```
1) import java.sql.* ;
2) import java.io.*;
3) import sqlj.runtime.* ;
4) import sqlj.runtime.ref.* ;
5) import oracle.sqlj.runtime.* ;
...
6) DefaultContext cntxt =
7)     oracle.getConnection("<nome url>", "<nome usuario>", "<senha>", true) ;
8) DefaultContext.setDefaultContext(cntxt) ;
...
```

Public static DefaultContext

getConnection(String url, String usuario, String senha, Boolean  
autoCommit) throws SQLException;

---

---

# SQLJ: SQL Embutida em Comandos JAVA

- Nos exemplos seguintes, não serão mostradas os programas completos em JAVA. Só ilustrar o uso de SQLJ

```
1) String dnome, ssn , pnome, fn, unome, ln, datanasc, endereco ;  
2) Char sexo, minicial, mi ;  
3) double salario, sal ;  
4) Integer dno, dnumero ;
```

**Variáveis de programa JAVA usadas nos exemplos SQLJ J1 e J2.**

---

## Segmento de Programa J1, um segmento de programa JAVA com a SQLJ

```
//Segmento de Programa J1:
1)  ssn = readEntry("Entre com o Numero do Seguro Social: ") ;
2)  try {
3)      #sql {select PNAME, MINICIAL, UNOME, ENDERECO, SALARIO
4)          into :pname, :minicial, :unome, :endereco, :salario
5)          from EMPREGADO where SSN = :ssn} ;
6)  } catch (SQLException se) {
7)      System.out.println("Numero do Seguro Social Inexistente: " + ssn) ;
8)      Return ;
9)      }
10) System.out.println(pnome + " " + minicial + " " + unome + " " + endereco + " " +
    salario)
```

Cada operação de Java tem de estabelecer as exceções que podem ser Emitidas.

<tipo de retorno da operação> < nome da operação>(<parâmetros>)  
throws SQLException, IOException ;

---

## Recuperando Diversas Tuplas em SQLJ por meio de Iteradores

- Um “iterator” é um tipo de objeto associado a uma coleção (um ou diversos conjuntos) de tuplas do resultado de uma consulta.
  - Há dois tipos de iteradores:
    - Um iterador designado, que é associado ao resultado de uma consulta por intermédio da lista de nomes e tipos de atributos que nela aparecerem.
    - Um iterador posicional, que lista somente os tipos dos atributos que aparecem no resultado da consulta.
-

## Segmento de programa J2A, um segmento de programa JAVA que usa um iterador designado para a impressão das informações dos empregados de um departamento em particular.

```
//Segmento de Programa J2A:
0)  dnome = readEntry ("Entre com o Nome do Departamento: ") ;
1)  try {
2)      #sql {select DNUMERO into :dnumero
3)          from DEPARTAMENTO where DNOME = :dnome} ;
4)  } catch (SQLException se) {
5)      System.out.println ("Departamento Inexistente: " + dnome) ;
6)      Return ;
7)  }
8)  System.out.println("Informacoes dos Empregados do Departamento: " + dnome) ;
9)  #sql iterator Emp(String ssn, String pnome, String minicial, String unome,
    double salario) ;
10) Emp e = null ;
11) #sql e = {select ssn, pnome, minicial, unome, salario
12)          from EMPREGADO where DNO = :dnumero} ;
13) while (e.next()) {
14)     System.out.println (e.ssn + " " + e.pnome + " " + e.minicial + " " +
        e.unome + " " + e.salario) ;
15) } ;
16) e.close() ;
```

**Segmento de programa J2B, um segmento de programa JAVA que usa um iterator posicional para imprimir as informações dos empregados de um departamento em particular.**

//Segmento de Programa J2B:

```
0)  dnome = readEntry ("Entre com o Nome do Departamento: ") ;
1)  try {
2)      #sql {select DNUMERO into :dnumero
3)          from DEPARTAMENTO where DNAME = :dnome} ;
4)  } catch (SQLException se) {
5)      System.out.println ("Departamento Inexistente: " + dnome) ;
6)      Return ;
7)  }
8)  System.out.println ("Informacoes dos Empregados do Departamento: " + dnome) ;
9)  #sql iterator Emppos (String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e ={select ssn, pnome, minicial, unome, salario
12)     from EMPREGADO where DNO = :dnumero} ;
13) #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)     System.out.println (ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)     #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```



**Segmento de programa J2B, um segmento de programa JAVA que usa um iterator posicional para imprimir as informações dos empregados de um departamento em particular.**

//Segmento de Programa J2B:

```
0)  dnome = readEntry ("Entre com o Nome do Departamento: ") ;
1)  try {
2)      #sql {select DNUMERO into :dnumero
3)          from DEPARTAMENTO where DNOME = :dnome} ;
4)  } catch (SQLException se) {
5)      System.out.println ("Depa
6)      Return ;
7)  }
8)  System.out.println ("Infor
9)  #sql iterator Emppos (String
10) Emppos e = null ;
11) #sql e ={select ssn, pnome, minicia
12)     from EMPREGADO where DNO = :dnumero} ;
13) #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)     System.out.println (ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)     #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

O comportamento do iterator é semelhante ao SQL Embutido.  
Fetch para <variável iterator> nas <variáveis do programa>



---

# JDBC – CONEXÃO JAVA E SGBDS

---

---

## Procedimentos Armazenados em Bds (Stored Procedures) e SQL/PSM

- Procedimentos armazenados: módulos de programas armazenados pelo SGBD no servidor de Bds.
  - Extensões de SQL padrão para considerar construtores para a programação de propósito geral em SQL  $\Rightarrow$  SQL/PSM (SQL/Persistent Stored Modules).
-

---

# Procedimentos armazenados

- Até agora, as técnicas supõe que o programa está num computador cliente. As vezes é útil criar módulos (procedimentos ou funções) armazenados no servidor. Ex:
    - ❑ Se um programa é útil para várias aplicações.
    - ❑ A execução no servidor pode reduzir a transferência de dados.
    - ❑ Esses procedimentos podem ser úteis para aumentar o poder de modelagem proporcionado pelas visões e podem ser úteis para verificar restrições mais complexas.
-

# Procedimentos armazenados

- A forma geral de declarar procedimentos armazenados é:

```
CREATE PROCEDURE <nome do procedimento> ([<parâmetros>])  
[<declarações locais>]  
<corpo do procedimento>;
```

- Para declarar uma função:

```
CREATE FUNCTION <nome da função> ([<parâmetros>])  
RETURNS <tipo de retorno>  
[<declarações locais>]  
<corpo da função>;
```

- Cada parâmetro deve ter um modo de parâmetro:  
IN, OUT ou INOUT.

# Exemplos

```
create procedure proc_num_contas (in nome varchar(20),  
                                out num_cont integer)
```

```
begin
```

```
    select count(*) into num_cont  
    from Dono_Conta  
    where Dono_Conta.Nome_Cliente =  
    proc_num_contas.nome;
```

```
end;
```

```
create function num_contas (nome varchar(20))  
returns integer
```

```
begin
```

```
    declare num_cont integer;  
    select count (*) into num_cont  
    from Dono_Conta  
    where Dono_Conta.Nome_Cliente = nome  
    return num_cont;
```

```
end
```

---

# Procedimentos armazenados

- Se o procedimento é escrito em uma linguagem de programação de propósito geral é usado o seguinte formato:

```
CREATE PROCEDURE <nome do procedimento>  
    ([<parâmetros>])
```

```
LANGUAGE <nome da linguagem de programação>
```

```
EXTERNAL NAME <nome do caminho do arquivo>
```

---

---

# Exemplos

```
create procedure proc_num_contas(in nome varchar(20),  
                                out num integer)  
language C  
external name ' /usr/avi/bin/account_count_proc'
```

```
create function num_contas(nome varchar(20))  
returns integer  
language C  
external name ' /usr/avi/bin/account_count'
```

---

---

# Procedimentos armazenados

- Eles podem ser chamados pelas diversas interfaces. Exemplo:

CALL <nome do procedimento ou função> (<lista de argumentos>);

---



# Procedimentos armazenados em PostgreSQL

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ [ nome_do_argumento ] tipo_do_argumento [, ...] ] )  
    RETURNS tipo_retornado  
{ LANGUAGE nome_da_linguagem  
  | IMMUTABLE | STABLE | VOLATILE  
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT  
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
  | AS 'definição'  
  | AS 'arquivo_objeto', 'símbolo_de_vinculação'  
} ...  
[ WITH ( atributo [, ...] ) ]
```

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$  
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END  
$$ LANGUAGE SQL STRICT;
```

# SQL/PSM

- É possível usar condicionais e laços. Um exemplo de condicional a seguir.
- O formato do laço é:

```
WHILE <condições> DO  
    <lista de declarações>  
END WHILE;
```

```
REPEAT  
    <lista de declarações>  
UNTIL <condições>  
END REPEAT;
```

## Declarando uma função em SQL/PSM

//Função PSM1:

```
0) CREATE FUNCTION DeptTamanho(IN deptnro INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE NroDeEmps INTEGER ;
3) SELECT COUNT(*) INTO NroDeEmps
4) FROM EMPREGADO WHERE DNO = deptnro ;
5) IF NroDeEmps > 100 THEN RETURN "ENORME"
6)     ELSEIF NroDeEmps > 25 THEN RETURN "GRANDE"
7)     ELSEIF NroDeEmps > 10 THEN RETURN "MEDIO"
8)     ELSE RETURN "PEQUENO"
9) END IF ;
```

# PL/pgSQL – Linguagem procedural SQL

```
CREATE OR REPLACE FUNCTION incremento(i integer) RETURNS integer AS $$  
    BEGIN  
        RETURN i + 1;  
    END;  
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION soma_dias(data date,dias integer)  
RETURNS date AS '  
DECLARE  
    nova_data date;  
BEGIN  
    nova_data := data + dias;  
    RETURN nova_data;  
END;  
' LANGUAGE plpgsql;
```