

# Banco de Dados II

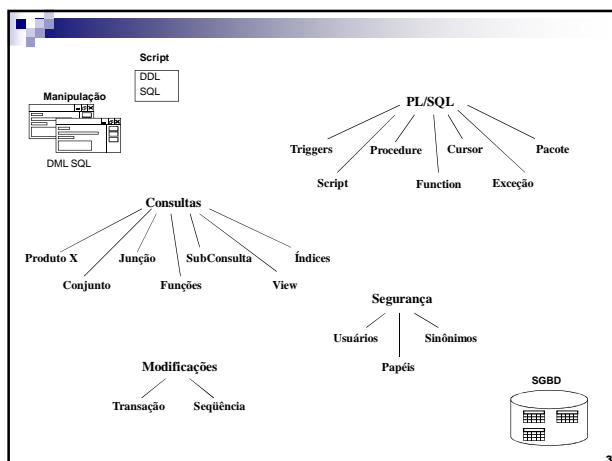
## DML – Parte I

Osmar de Oliveira Braz Junior

1

## Objetivos

- Construir expressões utilizando a estrutura básica select, from e where;
- Utilizar Expressões Regulares em condições de consultas;
- Realizar a composição de relações;
- Construir expressões utilizando operadores de conjunto;
- Utilizar funções agregadas em expressões de consulta agrupadas ou não;
- Criar expressões utilizando subconsultas;
- Realizar a modificação da base de dados controlada por transações
- Definir índices para melhorar a performance de consultas;
- Criar expressões armazenadas em visões;
- Utilizar seqüências para gerar identificadores em inserções.



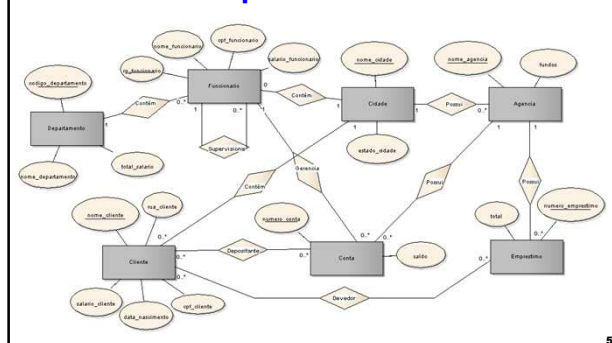
3

## 1. Linguagem de Manipulação de Dados

- Uma vez que o esquema esteja compilado e o banco de dados esteja populado, usa-se uma linguagem para fazer a manipulação dos dados, a **DML** (Data Manipulation Language - Linguagem de Manipulação de Dados).
- Por manipulação entendemos:
  - A recuperação das informações armazenadas no banco de dados;
  - Inserção de novas informações no banco de dados;
  - A remoção das informações no banco de dados;
  - A modificação das informações no banco de dados;

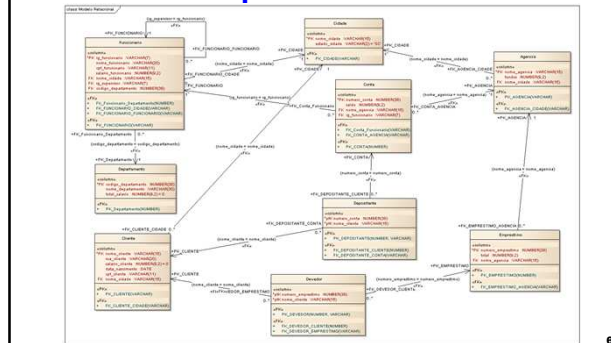
4

## 2. Estruturas Básicas(Esquema) financeira2.sql



5

## 2. Estruturas Básicas(Esquema) financeira2.sql



6

## 2. Estruturas Básicas(Esquema) financeira.sql

- **Cidade** = (nome\_cidade, estado\_cidade)
- **Agencia** = (nome\_agencia, nome\_cidade, fundos)
- **Departamento** = (codigo\_departamento, nome\_departamento, total\_salario)
- **Funcionario** = (rg\_funcionario, nome\_funcionario, cpf\_funcionario, salario\_funcionario, rg\_supervisor, nome\_cidade, codigo\_departamento)
- **Cliente** = (nome\_cliente, rua\_cliente, nome\_cidade, salario\_cliente, data\_nascimento, cpf\_cliente)
- **Emprestimo** = (numero\_emprestimo, nome\_agencia, total)
- **Devedor** = (nome\_cliente, numero\_emprestimo)
- **Conta** = (numero\_conta, nome\_agencia, saldo, rg\_funcionario)
- **Depositante** = (nome\_cliente, numero\_conta)

7

## 3. Estruturas Básicas

- A estrutura básica consiste de três cláusulas: select, from e where.
- **SELECT** – Ela é usada para relacionar os atributos desejados no resultado de uma consulta.
- **FROM** – Ela associa as relações que serão pesquisadas durante a evolução de uma expressão.
- **WHERE** – Ela consiste em um predicado envolvendo atributos da relação que aparece na cláusula from.

8

## 3. Estruturas Básicas

- Uma consulta típica em SQL tem a seguinte forma:

```
SELECT A1, A2, ..., An
FROM R1, R2, ..., Rn
WHERE P;
```

- Cada Ai representa um atributo em cara Ri uma relação. P é um predicado.

9

## 3. Estruturas Básicas 3.1. Cláusula Select

- O comando **select** permite a seleção de tuplas e atributos em uma ou mais tabelas.
- Exemplo: “Encontre os nomes de todos os clientes”.

```
SELECT nome_cliente
FROM cliente;
```

10

## 3. Estruturas Básicas 3.1. Cláusula Select

- *Especificador Distinct*

Eliminação da duplicidade

```
SELECT DISTINCT nome_cidade
FROM cliente;
```

- *Operador \**

Seleciona todos os atributos de uma tabela

```
SELECT *
FROM cliente;
```

11

## 3. Estruturas Básicas 3.1. Cláusula Select

- *Expressões Aritméticas*

A cláusula select pode conter expressões aritméticas envolvendo os operadores +, -, / e \* e operandos constantes ou atributos das tuplas:

- Exemplo: “Mostre o nome dos clientes e o seu salário reajustado em 10%”.

```
SELECT nome_cliente, salario_cliente * 1.1
FROM cliente;
```

12

### 3. Estruturas Básicas

#### 3.2. Cláusula Where

##### ■ Critérios

Considere a consulta “encontre todos os nomes dos clientes que ganham mais de 2400”.

- Esta consulta pode ser escrita em SQL como:

```
SELECT nome_cliente
FROM CLIENTE
WHERE salario_cliente > 2400;
```

13

### 3. Estruturas Básicas

#### 3.2. Cláusula Where

##### ■ Conectores Lógicos

A SQL usa conectores lógicos **and**, **or** e **not** na cláusula where. Os operandos dos conectivos lógicos podem ser expressões envolvendo operadores de comparação **<**, **<=**, **>**, **>=**, **=** e **!=**.

- Pode ser utilizando ainda o **<>** no lugar do **!=**.

14

### 3. Estruturas Básicas

#### 3.2. Cláusula Where

##### ■ Cláusula BETWEEN

- Operador de comparação **between** para simplificar a cláusula where;
- Especifica que um valor pode ser menor ou igual a algum valor e maior ou igual a algum outro valor.
- Se desejarmos encontrar os números de empréstimos cujos montantes estejam entre 90 mil dólares e 100 mil dólares, podemos usar a comparação **between** escrevendo:  

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total between 90000 and 100000;
```
- em vez de  

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total >=90000 and total <= 100000;
```

- negação **not between**.

15

### 3. Estruturas Básicas

#### 3.3. Cláusula From

- A cláusula **FROM** por si só define um produto cartesiano das relações da cláusula.
- Uma vez que a junção natural é definida em termos de **produto cartesiano**, uma seleção é uma projeção são um meio relativamente simples de escrever a expressão SQL para uma junção natural.

16

### 3. Estruturas Básicas

#### 3.3. Cláusula From

- Escrevemos a expressão em álgebra relacional:

$$\pi_{\text{nome\_cliente, numero\_emprestimo}} \left( \sigma_{\text{devedor.numero\_emprestimo = emprestimo.numero\_emprestimo}} (\text{devedor} \times \text{emprestimo}) \right)$$

para a consulta “para todos os clientes que tenham um empréstimo em um banco, encontre seus nomes e números de empréstimos”.

- Em SQL, essa consulta pode ser escrita como:

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE
    devedor.numero_emprestimo =
    emprestimo.numero_emprestimo;
```

17

### 3. Estruturas Básicas

#### 3.3. Cláusula From

- Note que a SQL usa a notação **nome\_relação.nome\_atributo**, como na álgebra relacional para evitar ambiguidades nos casos em que um atributo aparecer no esquema mais de uma relação.

18

### 3. Estruturas Básicas

#### 3.4. Operação Rename(alias)

- A SQL proporciona um mecanismo para rebatizar tanto relações quanto atributos, usando a cláusula **as**, da seguinte forma:  
`nome_antigo as nome_novo`
- Considere novamente a consulta usada anteriormente:  

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo =
      emprestimo.numero_emprestimo;
```
- Por exemplo se desejarmos que o nome do atributo `nome_cliente` seja substituído pelo nome `nome_devedor`, podemos reescrever a consulta como:  

```
SELECT distinct nome_cliente as devedor,
      devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo =
      emprestimo.numero_emprestimo;
```

19

### 3. Estruturas Básicas

#### 3.5. Variáveis Tuplas

- Variáveis tuplas são definidas na cláusula **from** por meio do uso da cláusula **as**.
- Com isto é possível renomear uma relação
- Variáveis tuplas são úteis para comparação de duas tuplas de mesma relação.

20

### 3. Estruturas Básicas

#### 3.5. Variáveis Tuplas

- Exemplo  
Funcionário  
  - (`rg_funcionario`, `nome_funcionario`, `cpf_funcionario`, `salario_funcionario`, `rg_supervisor`, `nome_cidade`, `codigo_departamento`)
- Encontre o nome de todos funcionários e o nome do seu supervisor (Supervisor também é um funcionário):  

```
SELECT distinct
      F.nome_funcionario, S.nome_funcionario
FROM Funcionario AS F, Funcionario AS S
WHERE F.RG_Funcionario = S.RG_Supervisor;
```

21

### 3. Estruturas Básicas

#### 3.6. Operações em String

- As operações em strings mais usadas são as checagens para verificação de coincidências de pares, usando o operador **like**. Indicaremos esses pares por meio do uso de dois caracteres especiais:
  - Porcentagem (%) : o caracter % compara qualquer substring.
  - Sublinhado ( \_ ) : o caracter \_ compara qualquer caracter.
- Para pesquisar diferenças em vez de coincidências, use o operador **not like**.

22

### 3. Estruturas Básicas

#### 3.6. Operações em String

- Exemplos  
Comparações desse tipo são sensíveis ao tamanho das letras; isto é, minúsculas não são iguais a maiúsculas, e vice-versa.
- Para ilustrar considere os seguintes exemplos:
  - "**Pedro**%" corresponde a qualquer string que comece com "Pedro"
  - "%**inh**%" corresponde a qualquer string que possua uma substring "inh", por exemplo "huguinho", "zezinho" e "luizinho"
  - "**\_\_\_**" corresponde a qualquer string com exatamente três caracteres
  - "**\_\_\_**%" corresponde a qualquer string com pelo menos três caracteres

23

### 3. Estruturas Básicas

#### 3.6. Operações em String

- Consulta  
Considere a consulta "encontre os nomes de todos os clientes cujos os nomes possuam a substring 'Silva'".
- Esta consulta pode ser escrita assim:  

```
SELECT nome_cliente
FROM cliente
WHERE nome_cliente LIKE '%Silva%';
```

24

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- As expressões regulares são uma maneira de descrever um conjunto de string com base em características comuns compartilhados por cada string no conjunto.
- Elas podem ser usadas para pesquisar, editar ou manipular texto e dados.

25

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Função **REGEXP\_LIKE** semelhante ao like

```
SELECT * FROM CLIENTE
WHERE nome_cliente LIKE 'Joao%';
```

```
SELECT * FROM CLIENTE
WHERE REGEXP_LIKE(nome_cliente, '^Joao');
```

- Para negar use **NOT REGEXP\_LIKE**

26

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Para diferenciar maiúsculas e minúsculas adicione um terceiro parâmetro ao operador
  - c – modo normal
  - i – ignorando a diferença de maiúsculas e minúsculas(O i deve ser minúsculo).

```
SELECT * FROM CLIENTE
WHERE REGEXP_LIKE(nome_cliente, '^Joao', 'i');
```

27

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

Operadores de Expressões Regulares(Meta Caracteres)	
Operador	Descrição
( )	Trata a expressão ou o conjunto de literais como uma subexpressão.
[...]	O par de colchetes delimita uma lista de uma ou mais expressões: combinações de elementos, símbolos, classes equivalentes, classes de caractere ou expressões de dimensão.
[^...]	Uma expressão de não igualdade. Indica que a lista de expressões dentro dos colchetes não deve ser encontrada.
[. . .]	O uso do ponto especifica uma combinação de elementos de acordo com o local. Muito útil em situações onde dois ou mais caracteres são necessários para especificar um elemento, como por exemplo, na especificação de um limite entre "a" e "ch" utilizaremos [a..[ch]].
[...]	Especifica uma classe de caracteres.
[...=...]	Especifica uma classe de equivalência, por exemplo, [e=] representa "e", "é", "ê", "ë".
.	O ponto combina qualquer caractere.
?	Combina zero ou uma ocorrência da subexpressão que o precede.

28

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

Operadores de Expressões Regulares(Meta Caracteres)	
Operador	Descrição
*	Combina zero, um ou mais ocorrências da subexpressão que o precede.
+	Combina uma ou mais ocorrências da subexpressão que o precede.
{n1}	Combina precisamente n1 ocorrências da subexpressão que o precede.
{n1, }	Combina n1 ou mais ocorrências da subexpressão que o precede.
{n1, n2}	Combina as ocorrências entre n1 e n2, inclusive os limites, da subexpressão que o precede.
\	Dependendo do contexto a contra barra é apenas uma contra barra, se estiver precedendo outro operador este é transformado em literal, por exemplo, \+ é o valor literal do mais.
\n1	Referencia anterior, repetição de "n1 vezes" da subexpressão dentro da expressão anterior.
	Operador lógico "OU". Utilizado para separar duas expressões, onde uma delas é combinada. Ex.: (joão maria)
^	Início de linha. Ex.: ^A strings que se iniciem com A.
\$	Fim de linha. Ex.: B\$ strings que se terminem com B.

29

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

Classes de Caracteres POSIX	
Operador	Descrição
[ :alnum: ]	Caracteres alfanuméricos. Inclui letras e números, omitindo pontuação. [A-Za-z0-9]
[ :alpha: ]	Caracteres do alfabeto. Apenas letras. [A-Za-z]
[ :blank: ]	Caracteres que formam espaços.
[ :cntrl: ]	Caracteres de controle (que não são impressos).
[ :graph: ]	Todas as classes de caractere combinadas, [:punct:], [:upper:], [:lower:], [:digit:].
[ :lower: ]	Caracteres Minúsculos [a-z]
[ :print: ]	Caracteres que podem ser impressos.
[ :punct: ]	Caracteres de pontuação. [!?:~]
[ :space: ]	Caracteres de espaço que não podem ser impressos. [\t\n\r\f\v]
[ :upper: ]	Caracteres maiúsculos. [A-Z]
[ :xdigit: ]	Caracteres hexadecimais.

30

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

Intervalo de Caracteres	
Operador	Descrição
[A-Z]	Todos os caracteres do alfabeto maiúsculos.
[a-z]	Todos os caracteres do alfabeto minúsculos.
[0-9]	Todos os dígitos numéricos.
[1-9]	Todos os dígitos numéricos, menos zero.
[A-Za-z]	Todos caracteres do alfabeto maiúsculos e minúsculos.

31

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar o formato da Hora (hh:mm)

- ...
- `[0-9]{2}:[0-9]{2}`
- `[012][0-9]:[0-9]{2}`
- `[012][0-9]:[0-5][0-9]`
- `([01][0-9]|2[0-3]):[0-5][0-9]`

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar numero decimal formado 999.99

- `[0-9]+`
- `[0-9]+(\.[0-9]+)?`
- `^[0-9]+(\.[0-9]+)?$`
- Permite um conjunto de [0-9] uma ou mais vezes seguido de um conjunto opcional(zero ou uma vez) de [0-9] uma ou mais vezes.
- `^` e `$` delimitam a expressão regular.

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar o formato da Data (dd/mm/aaaa)

- .../.../....
- `[0-9]{2}/[0-9]{2}/[0-9]{4}`
- `[0123][0-9]/[0-9]{2}/[0-9]{4}`
- `[0123][0-9]/[01][0-9]/[0-9]{4}`
- `[0123][0-9]/[01][0-9]/[12][0-9]{3}`
- `([012][0-9]|3[01])/[01][0-9]/[12][0-9]{3}`
- `([012][0-9]|3[01])/[01-9]|1[012]/[12][0-9]{3}`
- `([01-9]|12[0-9]|3[01])/[01-9]|1[012]/[12][0-9]{3}`

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar o formato do Telefone (9999-9999)

- ....-....
- `[0-9]{4}-[0-9]{4}`
- `\(..\) [0-9]{4}-[0-9]{4}`
- `\(..\) ?[0-9]{4}-[0-9]{4}`
- `\(0xx..\) ?[0-9]{4}-[0-9]{4}`
- `\(0xx[0-9]{2}\) ?[0-9]{4}-[0-9]{4}`
- `\(0xx[0-9]{2}\) ?[0-9]{4}-[0-9]{4}`

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Encontrar o nome e cpf dos funcionários que onde foram digitados corretamente ou seja somente 11 números.

```
SELECT nome_funcionario, cpf_funcionario
FROM funcionario
WHERE REGEXP_LIKE(CPF_FUNCIONARIO, '^[:digit:]{11}$');
```

- Para encontrar os que foram digitados errado utilize **NOT REGEXP\_LIKE**

36

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Encontrar o nome e data de nascimento dos clientes que nasceram entre 2000 e 2005.

```
SELECT nome_cliente, data_nascimento
FROM cliente
WHERE REGEXP_LIKE(TO_CHAR(data_nascimento, 'YYYY'), '^200[0-5]$');
```

37

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar o cpf do funcionário.

```
ALTER TABLE funcionario ADD ck_funcionario_cpf
CHECK (REGEXP_LIKE(cpf_funcionario, '^d{3}.^d{3}.^d{3}-d{2}$'));
```

- Validar o email do cliente.

```
ALTER TABLE cliente ADD (CONSTRAINT ck_cliente_email
CHECK (REGEXP_LIKE(email,
'^([[:alnum:]]+)[[:@]]([[:alnum:]]+).(com|net|org|edu|gov|mil)$')));
```

38

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- Validar o numero do telefone de cliente no formato (XX) XXXX-XXXX.

```
ALTER TABLE cliente ADD (CONSTRAINT ck_cliente_telefone
CHECK (REGEXP_LIKE(numero_telefone,
'^\([[:digit:]]{2}\) [[:digit:]]{4}-[[:digit:]]{4}$')));
```

39

### 3. Estruturas Básicas

#### 3.7. Expressão Regular

- REGEXP\_SUBSTR
  - Para obter trechos de texto, para informar a posição de início da busca passe um terceiro parâmetro.
- REGEXP\_INSTR
  - Para saber a posição de início
- REGEXP\_COUNT
  - Para saber o número de vezes de ocorrência do texto
- REGEXP\_REPLACE
  - Para substituir texto

40

### 3. Estruturas Básicas

#### 3.8. Precedência dos Operadores

Ordem de Avaliação	Operadores
1	Operadores Aritméticos
2	Operador de Concatenação
3	Condições de Comparação
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Condição lógica NOT
7	Condição lógica AND
8	Condição lógica OR

Para modificar a ordem de avaliação utilize parênteses.

41

### 3. Estruturas Básicas

#### 3.9. Ordenação

- A SQL oferece ao usuário algum controle sobre a ordenação por meio da qual as tuplas de uma relação serão apresentadas. A cláusula **order by** faz com que as tuplas do resultado de uma consulta apareçam em uma determinada ordem.
- Para listar em ordem alfabética todos os clientes, escrevemos:

```
SELECT nome_cliente, salario_cliente
FROM cliente
ORDER BY nome_cliente;
```
- Por default, a cláusula **order by** é em ordem **ascendente**.
- Especificação
  - **desc** para ordem descendente
  - **asc** para ordem ascendente.
- A ordenação pode ser realizada por diversos atributos.

42

### 3. Estruturas Básicas

#### 3.9. Ordenação

- Para listar em ordem alfabética as cidades dos clientes:

```
SELECT distinct nome_cidade
FROM cliente
ORDER BY cidade_cliente asc;
```

CLIENTE
NOME_CIDADE
Itajaí
Itajaí
Itajaí
Joinville
Joinville

43

### 3. Estruturas Básicas

#### 3.9. Ordenação

- Para listar em ordem alfabética as cidades e o nome dos clientes em ordem alfabética descendentes:

```
SELECT nome_cidade, nome_cliente
FROM cliente
ORDER BY nome_cidade asc, nome_cliente desc;
```

CLIENTE	
NOME_CIDADE	NOME_CLIENTE
Itajaí	Zeca
Itajaí	João
Itajaí	Antônio
Joinville	Zeca
Joinville	João

44

### 3. Estruturas Básicas

- Resolver os exercícios de DML Select (ExercicioDML\_Select.pdf)

45

### 4. Composição de Relações

- Utilizamos junção (join) de tabelas para extrair dados de mais de uma tabela.

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE R1.A1 = R2.A2;
```

- Uma condição deve estar presente na cláusula WHERE, justificando a relação entre as tabelas envolvidas.
- Deve ser usado um prefixo para as colunas que tiverem o mesmo nome em mais de uma tabela.

46

### 4. Composição de Relações

- Um **produto cartesiano** é formado nas seguintes condições:
  - Uma condição de junção(join) é omitida
  - Uma condição de junção(join) é inválida
  - Todas as linhas da primeira tabela são multiplicadas com todas as linhas da segunda tabela
- Para impedir a formação de um produto cartesiano, sempre inclua uma condição de restrição válida na cláusula **WHERE**.

47

### 4. Composição de Relações

EMPREGADO (14 linhas)

EMPID	ENOME	DEPTID
7839	KING	10
7698	BLAKE	30
...	...	...
7934	MILLER	10

DEPARTAMENTO (4 linhas)

DEPTID	DNOME	DLOCAL
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

"Produto Cartesiano:  
14\*4=56 linhas"

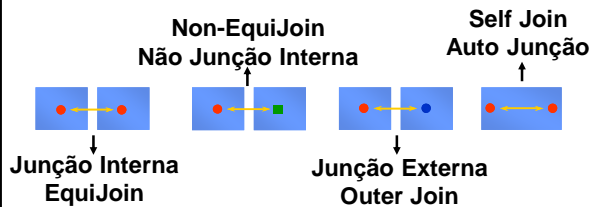
ENOME	DNOME
KING	CONTABILIDADE
BLAKE	CONTABILIDADE
...	...
KING	PESQUISA
BLAKE	PESQUISA
...	...
56 rows selected.	

48



## 4. Composição de Relações

### 4.1 Tipos de Junção



49

## 4. Composição de Relações

### 4.1 Tipos de Junção

- Cláusulas de Junção
  - Inner join => join
  - Left outer join => left join
  - Right outer join => right join
  - Full outer join => full join
- Condições de junção
  - Natural
  - On <predicado>
  - Using (A1, A2, ..., An)

50

## 4. Composição de Relações

### 4.2 Junção Interna

- Junção Interna (Inner Join) é quando tuplas são incluídas no resultado de uma consulta somente se existir uma correspondente na outra relação.
  - `SELECT tabela1.atributo1, tabela2.atributo2`
  - `FROM tabela1 INNER JOIN tabela2`
  - `ON tabela1.atributo1 = tabela2.atributo2;`
- Se o nome dos atributos for igual a cláusula ON e desnecessária, para isto usa-se o a Junção Natural com cláusula NATURAL.
  - `SELECT tabela1.atributo1, tabela2.atributo2`
  - `FROM tabela1 NATURAL INNER JOIN tabela2 ;`
  - `OU`
  - `SELECT tabela1.atributo1, tabela2.atributo2`
  - `FROM tabela1 NATURAL JOIN tabela2 ;`

51

## 4. Composição de Relações

### 4.2 Junção Interna

- A condição de junção USING(A1, A2, ..., An) é similar à condição de junção natural exceto pelo fato de que seus atributos de junção são os A1, A2, .. An em vez de todos os atributos comuns a ambas as relações.
- Os atributos A1, A2, ..., An devem ser somente os atributos comuns a ambas as relações e eles aparecem apenas uma vez no resultado da junção
  - `SELECT tabela1.atributo1, tabela2.atributo2`
  - `FROM tabela1 NATURAL INNER JOIN tabela2 USING (atributo1);`
  - `OU`
  - `SELECT tabela1.atributo1, tabela2.atributo2`
  - `FROM tabela1 NATURAL JOIN tabela2 USING (atributo1);`

52

## 4. Composição de Relações

### 4.2 Junção Interna

EMPREGADO			DEPARTAMENTO		
EMPID	ENOME	DEPTID	DEPTID	DNOME	DLOCAL
7839	KING	10	10	CONTABILIDADE	NEW YORK
7698	BLAKE	30	30	VENDAS	CHICAGO
7782	CLARK	10	10	CONTABILIDADE	NEW YORK
7566	JONES	20	20	PESQUISA	DALLAS
7654	MARTIN	30	30	VENDAS	CHICAGO
7499	ALLEN	30	30	VENDAS	CHICAGO
7844	TURNER	30	30	VENDAS	CHICAGO
7900	JAMES	30	30	VENDAS	CHICAGO
7521	WARD	30	30	VENDAS	CHICAGO
7902	FORD	20	20	PESQUISA	DALLAS
7369	SMITH	20	20	PESQUISA	DALLAS
...			...		

14 linhas selecionadas.

14 linhas selecionadas.

Foreign key Primary key

53

## 4. Composição de Relações

### 4.3 Resultado da Junção Interna

```
SQL> SELECT empregado.empid, empregado.enome,
2      empregado.deptid, dept.deptno,
3      empregado.dlocal
4      FROM empregado, departamento
5      WHERE empregado.deptid=departamento.deptid;
```

EMPID	ENOME	DEPTID	DEPTID	DLOCAL
7839	KING	10	10	NEW YORK
7698	BLAKE	30	30	CHICAGO
7782	CLARK	10	10	NEW YORK
7566	JONES	20	20	DALLAS
...				

14 linhas selecionadas.

- Usando as cláusulas FROM e WHERE para realizar o junção

54

## 4. Composição de Relações

### 4.3 Resultado da Junção Interna

```
SQL>SELECT empregado.empid, empregado.enome,
2      empregado.deptid, departamento.deptid,
3      departamento.dlocal
FROM empregado INNER JOIN departamento ON
      empregado.deptid=departamento.deptid;
```

EMPID	ENOME	DEPTID	DEPTID	DLOCAL
7839	KING	10	10	NEW YORK
7698	BLAKE	30	30	CHICAGO
7782	CLARK	10	10	NEW YORK
7566	JONES	20	20	DALLAS
...				

14 linhas selecionadas.

- Use o INNER JOIN ou simplesmente JOIN para realizar junção sem where.

55

## 4. Composição de Relações

### 4.3 Resultado da Junção Natural

```
SQL>SELECT empregado.empid, empregado.enome,
2      deptid, departamento.dlocal
3 FROM empregado NATURAL INNER JOIN departamento;
```

EMPID	ENOME	DEPTID	DLOCAL
7839	KING	10	NEW YORK
7698	BLAKE	30	CHICAGO
7782	CLARK	10	NEW YORK
7566	JONES	20	DALLAS
...			

14 linhas selecionadas.

- Use o NATURAL INNER JOIN (Junção Natural) ou NATURAL JOIN quando os atributos da junção tiverem nomes iguais

56

## 4. Composição de Relações

### 4.3 Resultado da Junção Natural com Using

```
SQL>SELECT empregado.empid, empregado.enome,
2      deptid, departamento.dlocal
3 FROM empregado INNER JOIN departamento USING(deptid);
```

EMPID	ENOME	DEPTID	DLOCAL
7839	KING	10	NEW YORK
7698	BLAKE	30	CHICAGO
7782	CLARK	10	NEW YORK
7566	JONES	20	DALLAS
...			

14 linhas selecionadas.

- Use a cláusula USING para especificar que atributos devem ser usados na junção

57

## 4. Composição de Relações

### 4.4 Ambigüidade

- Use prefixos para distinguir os nomes das colunas quando utilizar múltiplas tabelas.
- Utilize-se de alias quando existirem colunas com mesmo nome em mais de uma tabela.
- Utilize o operador **rename** (as)

58

## 4. Composição de Relações

### 4.5 Alias de Tabela

- Objetivo: Simplificar as consultas com produto cartesiano

```
SQL> SELECT empregado.empid, empregado.enome,
2      empregado.deptid, departamento.deptid,
3      departamento.dlocal
4 FROM empregado, departamento
5 WHERE empregado.deptid=departamento.deptid;
```

```
SQL> SELECT e.empid, e.enome, e.deptid,
2      d.deptid, d.dlocal
3 FROM empregado e, departamento d
4 WHERE e.deptid=d.deptid;
```

59

## 4. Composição de Relações

### 4.5 Alias de Tabela

- Objetivo: Simplificar as consultas com join

```
SQL> SELECT empregado.empid, empregado.enome,
2      empregado.deptid, departamento.deptid,
3      departamento.dlocal
4 FROM empregado INNER JOIN departamento ON
      empregado.deptid=departamento.deptid;
```

```
SQL> SELECT e.empid, e.enome, e.deptid,
2      d.deptid, d.dlocal
3 FROM empregado e INNER JOIN departamento d ON
      e.deptid=d.deptid;
```

60

#### 4. Composição de Relações

##### 4.6 Junção de mais de duas Tabelas

CLIENTE		PEDIDO		ITEM
NOME	CLIENTEID	CLIENTEID	PEDIDOID	
JOCKSPORTS	100	101	610	
TKB SPORT SHOP	101	102	611	
VOLLYRITE	102	104	612	
JUST TENNIS	103	106	601	
K+T SPORTS	105	102	602	
SHAPE UP	106	106		
WOMENS SPORTS	107	106		
...	...	...		
9 linhas selecionadas.		21 linhas		

PEDIDOID	ITEMID
610	3
611	1
612	1
601	1
602	1
...	...
64 linhas selecionadas	

61

#### 4. Composição de Relações

##### 4.6 Junção de mais de duas Tabelas

```
SQL> SELECT cliente.nome, cliente.clienteid,
        pedido.clienteid, pedido.pedidoid,
        item.pedidoid, item.itemid
2 FROM cliente, pedido, item
3 WHERE cliente.clienteid = pedido.clienteid
4        and pedido.pedidoid = item.pedidoid;
```

OU

```
SQL> SELECT cliente.nome, cliente.clienteid,
        pedido.clienteid, pedido.pedidoid,
        item.pedidoid, item.itemid
2 FROM (cliente INNER JOIN pedido
3       ON cliente.clienteid = pedido.clienteid)
4      INNER JOIN item
5      ON pedido.pedidoid = item.pedidoid;
```

62

#### 4. Composição de Relações

##### 4.6 Junção de mais de duas Tabelas

```
SQL> SELECT cliente.nome, clienteid, pedidoid, item.itemid
2 FROM (cliente NATURAL INNER JOIN pedido)
3      NATURAL INNER JOIN item;
```

- Os atributos utilizados na junção natural podem se especificados somente uma vez na cláusula SELECT(clienteid e pedidoid).

63

#### 4. Composição de Relações

##### 4.7 Não Junção Interna

###### EMPREGADO

EMPID	ENOME	SAL
7839	KING	5000
7698	BLAKE	2850
7782	CLARK	2450
7566	JONES	2975
7654	MARTIN	1250
7499	ALLEN	1600
7844	TURNER	1500
7900	JAMES	950
...		
14 linhas selecionadas.		

###### SALGRADE

GRADE	MESAL	MASAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

“O salário na tabela EMPREGADO está entre o menor e o maior salário da tabela SALGRADE”

64

#### 4. Composição de Relações

##### 4.7 Resultado Não Junção Interna

```
SQL> SELECT e.ename, e.sal, s.grade
2 FROM empregado e, salgrade s
3 WHERE e.sal
4      BETWEEN s.mesal AND s.masal;
```

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1
...		
14 linhas selecionadas.		

65

#### 4. Composição de Relações

##### 4.7 Resultado Não Junção Interna

```
SQL> SELECT e.ename, e.sal, s.grade
2 FROM emp e INNER JOIN salgrade s ON e.sal
        BETWEEN s.mesal AND s.masal;
```

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1
...		
14 linhas selecionadas.		

- A diferença da junção interna tradicional é a não utilização do operador de igualdade.

66

## 4. Composição de Relações

### 4.8 Junção Externa

- Junção externa é quando tuplas são incluídas no resultado sem que exista uma tupla correspondente na outra relação.

EMPREGADO		DEPARTAMENTO	
ENAME	DEPTID	DEPTID	DNAME
KING	10	10	CONTABILIDADE
BLAKE	30	30	VENDAS
CLARK	10	10	CONTABILIDADE
JONES	20	20	PESQUISA
...	...	40	OPERACOES

Nenhum empregado no departamento OPERACOES

67

## 4. Composição de Relações

### 4.8 Junção Externa

- Utiliza-se junção externa para listar tuplas que não usualmente se reúnem numa condição join.
- O operador de junção externa é o sinal de adição (+) ou **OUTER JOIN** tanto para a esquerda(LEFT) como a direita(RIGHT)

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE R1.A1(+) = R2.A1;
```

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE R1.A1 = R2.A2(+);
```

68

## 4. Composição de Relações

### 4.8 Junção Externa

```
SELECT R1.A1, R2.A2
FROM R1 RIGHT OUTER JOIN R2 ON
      R1.A1 = R2.A2;
```

```
SELECT R1.A1, R2.A2
FROM R1 LEFT OUTER JOIN R2 ON
      R1.A1 = R2.A2;
```

- RIGHT OUTER JOIN = Junção Externa a Direita
- LEFT OUTER JOIN = Junção Externa a Esquerda
- FULL OUTER JOIN = Junção Externa Total
  - Junção Externa a Esquerda + Junção Externa a Direita

Recomendável pela Oracle usar OUTER do que (+)

69

## 4. Composição de Relações

### 4.9 Utilização da Junção Externa

```
SQL> SELECT e.ename, d.deptid, d.dname
2 FROM empregado e, departamento d
3 WHERE e.deptid(+) = d.deptid
4 ORDER BY e.deptid;
```

ENAME	DEPTID	DNAME
KING	10	CONTABILIDADE
CLARK	10	CONTABILIDADE
...	40	OPERACOES

15 linhas selecionadas.

70

## 4. Composição de Relações

### 4.9 Utilização da Junção Externa

```
SQL> SELECT e.ename, d.deptid, d.dname
2 FROM empregado e RIGHT OUTER JOIN
      departamento d ON e.deptid = d.deptid
3 ORDER BY e.deptid;
```

ENAME	DEPTID	DNAME
KING	10	CONTABILIDADE
CLARK	10	CONTABILIDADE
...	40	OPERACOES

15 linhas selecionadas.

71

## 4. Composição de Relações

### 4.10 Auto Junção

EMPREGADO			EMPREGADO (GERENTE)	
EMPID	ENAME	GER	EMPID	ENAME
7839	KING		7839	KING
7698	BLAKE	7839	7839	KING
7782	CLARK	7839	7839	KING
7566	JONES	7839	7698	BLAKE
7654	MARTIN	7698	7698	BLAKE
7499	ALLEN	7698		

"GER na tabela Empregado é idêntica a EMPID na tabela Gerente"

72

## 4. Composição de Relações

### 4.10 Auto Junção

```
SQL> SELECT trab.enome||' trabalha para '||ger.enome
2 FROM empregado trab, empregado ger
3 WHERE trab.ger = ger.empid;
```

```
TRAB.ENOME|TRABALHA PARA|GER.ENOME
-----
BLAKE trabalha para KING
CLARK trabalha para KING
JONES trabalha para KING
MARTIN trabalha para BLAKE
...
13 linhas selecionadas.
```

|| realiza a concatenação de Literais.

73

## 4. Composição de Relações

### 4.10 Auto Junção

```
SQL> SELECT trab.enome||' trabalha para '||ger.enome
2 FROM empregado trab INNER JOIN empregado ger
ON trab.ger = ger.empid;
```

```
TRAB.ENOME|TRABALHA PARA|GER.ENOME
-----
BLAKE trabalha para KING
CLARK trabalha para KING
JONES trabalha para KING
MARTIN trabalha para BLAKE
...
13 linhas selecionadas.
```

74

## 4. Composição de Relações

- Resolver os exercícios de DML  
Join(ExercicioDML\_Join.pdf)

75

## 4. Composição de Relações

- A companhia dos pilotos que voam para o Brasil.

```
R.
SELECT distinct companhia
FROM ((piloto p INNER JOIN escala E
ON P.codigo_piloto=E.codigo_piloto)
INNER JOIN voo V
ON E.codigo_voo=V.codigo_voo)
INNER JOIN aeroporto A
ON V.aeroporto_destino = A.codigo_aeroporto
WHERE A.pais = 'Brasil';

OU

SELECT distinct companhia
FROM ((piloto INNER JOIN escala
ON piloto.codigo_piloto=escala.codigo_piloto)
INNER JOIN voo
ON escala.codigo_voo=voo.codigo_voo)
INNER JOIN aeroporto
ON voo.aeroporto_destino=aeroporto.codigo_aeroporto
WHERE aeroporto.pais = 'Brasil';
```

76

## 4. Composição de Relações

- O código e horário dos vôos internos de todos os países.

```
R.
SELECT distinct origem.pais, voo.codigo_voo, voo.hora
FROM voo INNER JOIN aeroporto destino
ON voo.aeroporto_destino = destino.codigo_aeroporto
INNER JOIN aeroporto origem
ON voo.aeroporto_origem = origem.codigo_aeroporto
WHERE origem.pais = destino.pais;
```

77

## 4. Composição de Relações

- O código de todos os vôos internacionais da Tam.

```
R.
SELECT distinct V.codigo_voo
FROM piloto P INNER JOIN escala E
ON P.codigo_piloto= E.codigo_piloto
INNER JOIN voo V
ON E.codigo_voo=V.codigo_voo
INNER JOIN aeroporto destino
ON V.aeroporto_destino = destino.codigo_aeroporto
INNER JOIN aeroporto origem
ON V.aeroporto_origem = origem.codigo_aeroporto
WHERE (origem.pais <> 'Brasil' or destino.pais <> 'Brasil')
and companhia = 'Tam';
```

78

## 4. Composição de Relações

- Selecione o nome dos pilotos e seu avião estando ele escalado ou não.

R.

```
SELECT distinct p.nome_piloto, e.aviao  
FROM escala e RIGHT OUTER JOIN piloto p  
ON P.codigo_piloto= E.codigo_piloto
```

ou

```
SELECT distinct piloto.nome_piloto, escala.aviao  
FROM escala RIGHT OUTER JOIN piloto  
ON piloto.codigo_piloto= escala.codigo_piloto
```

79

## 4. Operações de Conjuntos

- Os operadores **union**, **intersect** e **except** operam relações e correspondem às operações  $\cup$ ,  $\cap$  e  $-$  da álgebra relacional.
- Como a união, interseção e diferença de conjuntos da álgebra relacional, as relações participantes das operações precisam ser compatíveis (**MESMO DOMÍNIO**), isto é, elas precisam ter o mesmo conjunto de atributos.


80

## 4. Operações de Conjunto

### 4.1. União

- Para encontrar todos os clientes do banco que possuem empréstimo, uma conta ou ambos, escrevemos:

```
(SELECT nome_cliente  
FROM depositante)  
UNION  
(SELECT nome_cliente  
FROM devedor);
```



- A operação de união, ao contrário da cláusula select, automaticamente elimina as repetições.
- Se desejarmos obter todas as repetições, teremos de escrever **union all** no lugar de **union**:


81

## 4. Operações de Conjunto

### 4.2. Interseção

- Para encontrar todos os clientes que tenham tanto empréstimo quanto contas, escrevemos:

```
(SELECT nome_cliente  
FROM depositante)  
INTERSECT  
(SELECT nome_cliente  
FROM devedor);
```



- A operação intersect automaticamente elimina todas as repetições.
- Se desejarmos obter todas as repetições, teremos de escrever **intersect all** no lugar de **intersect**.


82

## 5. Operações de Conjunto

### 5.3. Exceto(Subtração)

- Para encontrar todos os clientes que tenham uma conta e nenhum empréstimo no banco, escrevemos:

```
(SELECT nome_cliente  
FROM depositante)  
EXCEPT  
(SELECT nome_cliente  
FROM devedor);
```



- A operação **except** (**minus** no oracle) automaticamente elimina todas as repetições.
- Se desejarmos obter todas as repetições, teremos de escrever **except all** no lugar de **except**.

83

## 5. Operações de Conjunto

- Resolver os exercícios de DML Conjunto(ExercicioDML\_Conjunto.pdf)

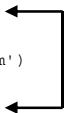
84

## 5. Operações de Conjunto

- O nome dos pilotos que voam tanto para a Varig como para Tam.

R.

```
(SELECT nome_piloto
FROM piloto
WHERE companhia='Tam')
UNION
(SELECT nome_piloto
FROM piloto
WHERE companhia='Varig')
```



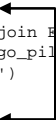
85

## 5. Operações de Conjunto

- As companhias que **só** voam de MD11.

R.

```
(SELECT Companhia
FROM Piloto inner join ESCALA
on piloto.codigo_piloto = escala.codigo_piloto
where aviao='MD11')
MINUS
(SELECT Companhia
FROM Piloto inner join ESCALA
on piloto.codigo_piloto = escala.codigo_piloto
where aviao<>'MD11')
```



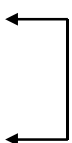
86

## 5. Operações de Conjunto

- O maior salário pago a um do piloto.

R.

```
(SELECT salario
FROM piloto)
MINUS
(SELECT p2.salario
FROM piloto p1, piloto p2
WHERE p1.salario > p2.salario)
```




87

## 5. Operações de Conjunto

- O nome do piloto que ganha o maior salário.

R.

```
SELECT nome_piloto
FROM piloto,
((SELECT salario
FROM piloto)
MINUS
(SELECT p2.salario salario
FROM piloto p1, piloto p2
WHERE p1.salario > p2.salario)) maior
WHERE piloto.salario = maior.salario
```




88

## 5. Operações de Conjunto

- Selecione o nome dos pilotos e seu avião estando ele escalado ou não. (Não usar junção externa)

R.

```
((SELECT nome_piloto, 'null' aviao
FROM piloto)
MINUS
(SELECT nome_piloto, 'null' aviao
FROM piloto, escala
WHERE piloto.codigo_piloto = escala.codigo_piloto))
UNION
(SELECT nome_piloto, aviao
FROM piloto, escala
WHERE piloto.codigo_piloto = escala.codigo_piloto)
```



Todos os pilotos com a coluna que não existe em branco ou nula.

Subtrair dos pilotos com escala.

Unir com os pilotos que possuem escala

89

## 6. Funções Agregadas

- Funções agregadas ou de grupo são funções que tomam uma coleção (um conjunto ou subconjunto) e valores como entrada, retornando um valor simples. A SQL oferece cinco funções agregadas pré-programadas:

- Média (average): **avg**.
- Mínimo (minimum): **min**.
- Máximo (maximum): **max**.
- Total (total): **sum**.
- Contagem (count): **count**.

90

## 6. Funções Agregadas

- Para “encontrar a média dos salários dos clientes”, escrevemos:

```
SELECT AVG(salario_cliente)
FROM cliente;
```

91

## 6. Funções Agregadas

### CLIENTE

NOME_CLIENTE	SALARIO_CLIENTE
FELIX	10000
JOAO	2000
PEDRO	300
MARIA	500
MAURICIO	5000
GEAN	770
OSMAR	15000
JOAQUIM	900
FERNANDO	800
JOANA	520
JOAQUIM	5030
MARIANA	120
JOANA DA SILVA	860
MARIDEL	9200
JOAO PEDRO	130
GUILHERME	1130

16 linhas selecionadas.

Média dos salários

AVG(SALARIO_CLIENTE)
3266,25

92

## 6. Funções Agregadas

- Para “encontrar a média dos salários dos clientes que mais de 150.00”, escrevemos:

```
SELECT AVG(salario_cliente)
FROM cliente
WHERE salario_cliente > 150;
```

93

## 6. Funções Agregadas

### CLIENTE

NOME_CLIENTE	SALARIO_CLIENTE
FELIX	10000
JOAO	2000
PEDRO	300
MARIA	500
MAURICIO	5000
GEAN	770
OSMAR	15000
JOAQUIM	900
FERNANDO	800
JOANA	520
JOAQUIM	5030
MARIANA	120
JOANA DA SILVA	860
MARIDEL	9200
JOAO PEDRO	130
GUILHERME	1130

16 linhas selecionadas.

Média Salário > 150

AVG(SALARIO_CLIENTE)
3250,625

94

## 6. Funções Agregadas

- Para “encontrar a média, o maior, o menor e a soma para os salários dos clientes”, escrevemos:

```
SQL> SELECT AVG(salario_cliente), MAX(salario_cliente),
2 MIN(salario_cliente), SUM(salario_cliente)
3 FROM cliente;
```

AVG(SALARIO_CLIENTE)	MAX(SALARIO_CLIENTE)	MIN(SALARIO_CLIENTE)	SUM(SALARIO_CLIENTE)
3266,25	15000	120	52260

95

## 6. Funções Agregadas

### 6.1. Cláusula Group By

- Existem circunstâncias em que gostaríamos de aplicar uma função agregada não somente a um conjunto de tuplas, mas também a um grupo de conjunto de tuplas o que é possível usando a cláusula SQL **group by**.
- O atributo ou atributos fornecidos em uma cláusula **group by** são usados para formar grupos. Tuplas com o mesmos valores em todos os atributos da cláusula **group by** são colocados em um grupo.

96



## 6. Funções Agregadas

### 6.1. Cláusula Group By

- Encontrar “a média dos salários dos clientes em cada uma das cidades”, escrevemos:

```
SELECT nome_cidade, avg(salario_cliente)
FROM cliente
GROUP BY nome_cidade;
```

97

## 6. Funções Agregadas

### CLIENTE

NOME_CIDADE	SALARIO_CLIENTE
BLUMENAU	9200
BLUMENAU	860
CURITIBA	5030
CURITIBA	2000
CURITIBA	800
FLORIANOPOLIS	15000
FLORIANOPOLIS	770
FLORIANOPOLIS	10000
FLORIANOPOLIS	5000
ITAJAI	130
PALHOCA	900
RIO DE JANEIRO	120
RIO DE JANEIRO	300
RIO DE JANEIRO	520
SAO PAULO	500
TUBARAO	1130

16 linhas selecionadas.

NOME_CIDADE	AVG(SALARIO_CLIENTE)
BLUMENAU	5030
CURITIBA	2610
FLORIANOPOLIS	7692,5
ITAJAI	130
PALHOCA	900
RIO DE JANEIRO	313,33
SAO PAULO	500
TUBARAO	1130

8 linhas selecionadas

98

## 6. Funções Agregadas

### 6.1. Cláusula Group By

- Encontrar “a média dos salários dos clientes quem ganham mais de 150,00 em cada uma das cidades”, escrevemos:

```
SELECT nome_cidade, avg(salario_cliente)
FROM cliente
WHERE salario_cliente > 150
GROUP BY nome_cidade;
```

99

## 6. Funções Agregadas

### 6.2. Cláusula Having

- Às vezes, é mais interessante definir condições e aplicá-las a grupos do que aplicá-las a tuplas.
- Por exemplo, encontrar “as cidades que possuem média dos salários dos clientes maior que 1100.00”.

```
SELECT nome_cidade, avg(salario_cliente)
FROM cliente
GROUP BY nome_cidade
HAVING avg(salario_cliente) > 1100;
```

100

## 6. Funções Agregadas

### 6.2. Cláusula Having

- Essa condição não se aplica a uma única tupla, mas em cada grupo determinado pela cláusula **group by**.
- Para exprimir tal consulta, usamos a cláusula **having** da SQL.
- Os predicados da cláusula **having** são aplicados depois da formação dos grupos, assim poderão ser usadas funções agregadas.

101

## 6. Funções Agregadas

### 6.2. Cláusula Having

- Por exemplo, encontrar “as cidades que possuem média dos salários dos clientes maior que 1100.00 e o salario seja superior a 120.00”.

```
SELECT nome_cidade, avg(salario_cliente)
FROM cliente
WHERE salario_cliente > 120
GROUP BY nome_cidade
HAVING avg(salario_cliente) > 1100;
```

102

## 6. Funções Agregadas

### 6.3. RollUp

- **RollUp** permite obter os totais da função utilizada no agrupamento depois de exibir os dados.
- Encontre “a quantidade de clientes por cidade e o total geral”, escrevemos:  

```
SELECT nome_cidade, COUNT (*)
FROM cliente
GROUP BY ROLLUP(nome_cidade);
```
- Você pode escolher mais de uma coluna para ver os totais.

103

## 6. Funções Agregadas

### 6.3. RollUp

```
SELECT nome_cidade, COUNT (*)
FROM cliente
GROUP BY ROLLUP(nome_cidade);
```

NOME_CIDADE	COUNT(*)
1 BLUMENAU	2
2 CURITIBA	3
3 FLORIANOPOLIS	4
4 ITAJAI	1
5 PALHOCA	1
6 RIO DE JANEIRO	3
7 SAO PAULO	1
8 TUBARAO	1
9 (null)	16

104

## 6. Funções Agregadas

### 6.3. Cube

- **Cube** é similar ao **RollUp**, só que calcula todos os subtotais relativos à consulta antes do agrupamento.
- Encontre “a quantidade de clientes por cidade e rua os subtotais e o total geral”, escrevemos:

```
SELECT nome_cidade, rua_cliente, COUNT(*)
FROM cliente
GROUP BY CUBE(nome_cidade, rua_cliente);
```

105

## 6. Funções Agregadas

### 6.3. Cube

NOME_CLIENTE	RUA_CLIENTE	COUNT(*)
1 (null)	(null)	16
2 (null)	rua_cliente 1	1
3 (null)	rua_cliente 2	1
4 (null)	rua_cliente 3	1
5 (null)	rua_cliente 4	1
6 (null)	rua_cliente 5	1
7 (null)	rua_cliente 7	1
8 (null)	rua_cliente 8	1
9 (null)	rua_cliente 9	1
10 (null)	rua_cliente 10	1
11 (null)	rua_cliente 11	1
12 (null)	rua_cliente 12	1
13 (null)	rua_cliente 13	1
14 (null)	rua_cliente 14	1
15 (null)	rua_cliente 15	1
16 (null)	rua_cliente 16	1
17 (null)	rua_cliente 6	1
18 ITAJAI	(null)	1
19 ITAJAI	rua_cliente 15	1
20 PALHOCA	(null)	1

21 PALHOCA	rua_cliente 8	1
22 TUBARAO	(null)	1
23 TUBARAO	rua_cliente 16	1
24 BLUMENAU	(null)	2
25 BLUMENAU	rua_cliente 13	1
26 BLUMENAU	rua_cliente 14	1
27 CURITIBA	(null)	3
28 CURITIBA	rua_cliente 2	1
29 CURITIBA	rua_cliente 9	1
30 CURITIBA	rua_cliente 11	1
31 SAO PAULO	(null)	1
32 SAO PAULO	rua_cliente 4	1
33 FLORIANOPOLIS	(null)	4
34 FLORIANOPOLIS	rua_cliente 1	1
35 FLORIANOPOLIS	rua_cliente 5	1
36 FLORIANOPOLIS	rua_cliente 7	1
37 FLORIANOPOLIS	rua_cliente 6	1
38 RIO DE JANEIRO	(null)	3
39 RIO DE JANEIRO	rua_cliente 3	1
40 RIO DE JANEIRO	rua_cliente 10	1
41 RIO DE JANEIRO	rua_cliente 12	1

106

## 6. Funções Agregadas

### 6.4. Listas Agregadas

- A função **LISTAGG** (list aggregate | lista agregada), é uma função SQL de grupo específica do Oracle(11g.r2). É uma função analítica, que agrupa os dados na mesma linha dentro de uma instrução SELECT.
- Pode ser utilizada sozinha para produzir conjuntos de dados agrupados por linhas simples, ou em conjunto com a função GROUP BY.

107

## 6. Funções Agregadas

### 6.4. Listas Agregadas

- Por exemplo “Agrupe o nome dos funcionários, criando uma lista separada por ;”.
- Podemos escrever a seguinte expressão:

```
SELECT LISTAGG(nome_funcionario, ';' )
WITHIN GROUP (ORDER BY nome_funcionario) NOME_FUNCIONARIOS
FROM funcionario;
```

Separador dos dados agrupados.

Especifica a ordem dos dados agrupados.

NOME_FUNCIONARIOS
1 ANTONIO; CARLOS; GOKU; HEITOR; HUGUINHO; JOAO; JOSE; LUIZ; MARIA; PEDRO; TERESA; ZEZINHO

108

## 6. Funções Agregadas

### 6.4. Listas Agregadas

- Por exemplo “Liste para cada departamento o nome de seus funcionários”:
- Podemos escrever a seguinte expressão:

```
SELECT nome_departamento,
       nome_funcionario
FROM   departamento, funcionario
WHERE  departamento.codigo_departamento =
funcionario.codigo_departamento;
```

1	NOME_DEPARTAMENTO	NOME_FUNCIONARIO
1	CENTRAL	JOAO
2	CENTRAL	MARIA
3	CENTRAL	PEDRO
4	CENTRAL	LUIZ
5	RH	ZEZINHO
6	RH	HUGUINHO
7	VENDAS	HEITOR
8	VENDAS	JOSE
9	VENDAS	ANTONIO
10	VENDAS	CARLOS
11	COMPRAS	TERESA
12	COMPRAS	GOKU

109

## 6. Funções Agregadas

### 6.4. Listas Agregadas

- Mas para criar uma lista separada por “,” para os nomes de funcionários para cada departamento, usamos a função ListAgg desta forma:

```
SELECT nome_departamento,
       LISTAGG(nome_funcionario, ',' ) WITHIN GROUP (
           ORDER BY nome_funcionario) NOME_FUNCIONARIOS
FROM   departamento, funcionario
WHERE  departamento.codigo_departamento=funcionario.codigo_departamento
GROUP BY
       nome_departamento;
```

1	NOME_DEPARTAMENTO	NOME_FUNCIONARIOS
1	CENTRAL	JOAO; LUIZ; MARIA; PEDRO
2	COMPRAS	GOKU; TERESA
3	RH	HUGUINHO; ZEZINHO
4	VENDAS	ANTONIO; CARLOS; HEITOR; JOSE

110

## 6. Funções Agregadas

- Outras funções podem ser vistas na Apostila no Anexo 2 item 7.5.
- Algumas funções são comuns a outros S.G.B.D.

111

## 6. Funções Agregadas

### Funções Matemáticas

FUNÇÕES	EXEMPLO	RESULTADO
ABS	ABS(-10.0)	Retorna o valor absoluto de um número ou expressão matemática.
ACOS	ACOS(15)	Retorna o arco-coseno(em radianos) de um número ou expressão
ATAN	ATAN(15)	Retorna o arco-tangente(em radianos) de um número ou expressão
COS	COS(45)	Retorna o co-seno de um ângulo expresso em radianos.
EXP	EXP(18)	Retorna e (a base dos logaritmos naturais) elevado à uma potência especificada (e <sup>n</sup> ).
LN	LN(10)	Retorna o logaritmo natural de um número positivo.
LOG	LOG(10,2)	Retorna o logaritmo, base m, de n. log(m,n).
MOD	MOD(3,2)	Retorna o resto da divisão do primeiro argumento pelo segundo argumento.
POWER	POWER(2,5)	Retorna m elevado a n-ésima potência, power(m,n).
ROUND	ROUND(100.1234,2)	Retorna um número arredondado em um número de casas especificadas.
SIN	SIN(90)	Retorna o seno de um ângulo medido em radianos.
SQRT	SQRT(9)	Retorna a raiz quadrada de um número não negativo ou expressão matemática.
TAN	TAN(180)	Retorna a tangente de um ângulo que é especificado em radianos.
TRUNC	TRUNC(100.1234,2)	Retorna um número truncado em um número de casas especificadas.

112

## 6. Funções Agregadas

### Funções de Data

FUNÇÕES	EXEMPLO	RESULTADO
ADD_MONTHS	ADD_MONTHS(HIREDATE,5)	Adiciona 5 meses na data HIREDATE
LAST_DAY	LAST_DAY(SYSDATE)	Retorna a data tomando como parâmetro o 'FMT'
MONTHS_BETWEEN	MONTHS_BETWEEN(HIREDATE,SYSDATE)	Calcula o número de meses BETWEEN entre as datas
NEXT_DAY	NEXT_DAY(HIREDATE,FRIDAY)	Procura uma sexta-feira após HIREDATE
ROUND	ROUND(SYSDATE,MONTH)	Retorna uma data arredonda para o segundo argumento.
SYSDATE	SYSDATE	Retorna a data e o horário atuais do sistema
SYSTIMESTAMP	SYSTIMESTAMP	Retorna a data e o horário atuais do sistema com TimeStamp
TIME	TIME	Retorna o horário atual do sistema operacional do sistema.
TRUNC	TRUNC(SYSDATE,FMT)	Trunca a data para a primeira data do 'FMT'

113

## 6. Funções Agregadas

### Funções de Data

- Recuperando o nome dos clientes que nascem em um determinado dia.  

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'DD') = 01;
```
- Recuperando o nome dos clientes que nascem em um determinado mês.  

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'MM') = 8;
```
- Recuperando o nome dos clientes que nascem em um determinado ano.  

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'YYYY') = 1998;
```

114

## 6. Funções Agregadas Formato de Data

Formato	Descrição	Formato	Descrição
AM	AM ou PM	MON	Mês abreviado ('NOV')
CC	Século	MONTH	Mês por extenso ('NOVEMBER')
D	Dia da semana (1-7)	PM	AM ou PM
DAY	Dia da semana ('SUNDAY')	RR	Ano com 2 dígitos - especial
DD	Dia do mês (1-31)	RRRR	Ano com 4 dígitos
DDD	Dia do ano	SS	Segundos do minuto (0 - 59)
DY	Dia da semana abreviado ('SUN')	SSSS	Segundos do dia
FM	Tira os Brancos ou Zeros da esquerda	W	Semana do Mês
HH	Hora do dia (0-12)	WW	Semana do Ano
HH24	Hora do dia (0-24)	YEAR	Ano por extenso
MI	Minutos da Hora	YY	Ano com 2 dígitos
MM	Mês com 2 dígitos	YYYY	Ano com 4 dígitos

115

## 6. Funções Agregadas Funções de Data

- Mostrar os dias do ano a partir da data do sistema utilizando to\_char.

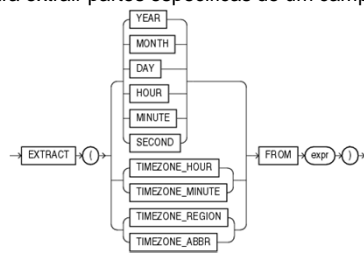
```
SELECT sysdate,
       to_char(sysdate, 'DD')
FROM dual;
```

SYSDATE	TO_CHAR(SYSDATE,'DD')
1 21/05/2014 19:30:40 21	

116

## 6. Funções Agregadas Funções de Data

- Função extract
  - Utilizado para extrair partes específicas de um campo datetime.



117

## 6. Funções Agregadas Funções de Data

- Mostrar os dias do ano a partir da data do sistema utilizando extract.

```
SELECT sysdate,
       extract(day from sysdate)
FROM dual;
```

SYSDATE	EXTRACT(DAYFROMSYSDATE)
1 21/05/2014 19:32:42	21

118

## 6. Funções Agregadas Funções de Caracteres

FUNÇÕES	EXEMPLO	RESULTADO
ASCII	ASCII('CASA')	Retorna a representação decimal do caractere dado.
CHR	CHR(45)	Retorna o caractere correspondente ao código decimal ANSI.
CONCAT	CONCAT('boni','dia')	Retorna uma string concatenando as duas strings passadas como argumento. Como alternativa pode-se usar o    double pipe.
INITCAP	INITCAP('casa')	Retorna uma string cuja primeira letra foi convertida minúscula.
INSTR	INSTR('string','i')	Retorna um número inteiro com a primeira ocorrência da letra do segundo argumento na string do primeiro argumento.
LOWER	LOWER('CAsa')	Retorna uma string cujas letras maiúsculas foram convertidas para letras minúsculas.
LEFT	LEFT('Casa',2)	Retorna o número de caracteres mais à esquerda especificados a partir de uma string.
LENGTH	LENGTH('Casa')	Retorna o número de caracteres contidos em uma string.
LPAD	LPAD('senha',10,'*')	Retorna a string completando com * até o 10 caractere a esquerda.
LTRIM	LTRIM(' Casa')	Retorna uma string sem espaços em brancos na esquerda.

119

## 6. Funções Agregadas Funções de Caracteres

FUNÇÕES	EXEMPLO	RESULTADO
REPLACE	REPLACE('XUXXU','X','X')	Retorna a string com a substituição de todas as ocorrências do segundo argumento pelo valor do terceiro argumento.
RIGHT	RIGHT('Casa',2)	Retorna o número de caracteres mais à direita especificados a partir de uma string.
RPAD	RPAD('senha',10,'*')	Retorna a string completando com * até o 10 caractere a direita.
RTRIM	RTRIM('Casa ')	Retorna uma string sem espaços em brancos na direita.
SOUNDEX	SOUNDEX('Joao')	Retorna a expressão fonética de uma string. Para a String 'Joao' retorna J000.
STR	STR(10)	Retorna uma representação em forma de string de um número ou expressão.
STRING	STRING(10,45)	Retorna uma string de comprimento especificado que consiste de um único caractere.
SUBSTR	SUBSTR('string',1,3)	Retorna uma substring da posição inicial(1) até a posição final(3).
TRIM	TRIM(' Casa ')	Retorna uma string sem espaços em brancos na esquerda e direita.
UPPER	UPPER(' casa')	Retorna uma string cujas letras minúsculas foram convertidas para letras maiúsculas.

120

## 6. Funções Agregadas

### Funções de Caracteres

- Consultar um literal que contenha espaços e somente em maiúsculo.

```
SELECT nome_cliente
FROM cliente
WHERE upper(trim(nome_cliente)) = 'PEDRO';
```

121

## 6. Funções Agregadas

- Resolver os exercícios de DML Funções Agregadas (ExercicioDML\_Funcao.pdf)

122

## 7. Valores Nulos

- Podemos usar a palavra-chave **null** como predicado para testar a existência de valores nulos.
- Assim, para encontrar os números de empréstimos que aparecem na relação empréstimo com valores nulos para total, escrevemos:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total is null;
```

- O predicado **is not null** testa a ausência de valores nulos.

123

## 8. Subconsultas Aninhadas

- A SQL proporciona um mecanismo para o **aninhamento** de subconsultas.
- Um subconsulta é uma expressão **select-from-where** aninhada dentro de uma outra consulta.
- As aplicações mais comuns para as **subconsultas** são testes para membros de conjuntos, comparações de conjuntos e cardinalidade de conjuntos

124

## 8. Subconsultas Aninhadas

### 8.1. Membros de Conjuntos(IN)

- A SQL permite verificar se uma tupla é membro ou não de uma relação. O conectivo **in** testa os membros de um conjunto, no qual o conjunto é a coleção de valores produzidos pela cláusula select. O conectivo **not in** verifica a ausência de membros de um conjunto.
- Precisamos encontrar todos os clientes que contrairam empréstimos no banco e que aparecem na lista dos correntistas. Faremos isso por meio do aninhamento da subconsulta em uma consulta externa. O resultado dessa forma é:  

```
SELECT distinct nome_cliente
FROM devedor
WHERE nome_cliente IN (SELECT nome_cliente
                        FROM depositante);
```
- Para ilustrar o uso do **not in** "encontre todos os clientes que tenham um empréstimo no banco, mas que não tenham uma conta no banco":

125

## 8. Subconsultas Aninhadas

### 8.1. Membros de Conjuntos(IN)

- ```
SELECT distinct nome_cliente
FROM devedor
WHERE nome_cliente IN (SELECT nome_cliente
                       FROM depositante);
```
- ```
SELECT distinct nome_cliente
FROM devedor, depositante
WHERE devedor.nome_cliente = depositante.nome_cliente;
```
- ```
SELECT distinct nome_cliente
FROM devedor inner join depositante using nome_cliente;
```
- ```
(SELECT distinct nome_cliente
FROM devedor)
INTERSECT
(SELECT distinct nome_cliente
FROM depositante);
```

126

## 8. Subconsultas Aninhadas

### 8.1. Membros de Conjuntos(IN)

- Pode-se utilizar constantes dentro da cláusula **IN**.
- ```
SELECT distinct nome_agencia
FROM conta
WHERE numero_conta IN (1,2,3,4,5);
```
- ```
SELECT distinct numero_conta
FROM conta
WHERE nome_agencia IN
('AGENCIAL', 'AGENCIA2');
```

127

## 8. Subconsultas Aninhadas

### 8.2. Comparação de Conjuntos(some)

- Considere a consulta “encontre os nomes de todas as agências que tenham fundos maiores que ao menos uma agência localizada no Brooklyn”. Anteriormente escreveríamos esta consulta da seguinte forma:  

```
SELECT distinct T.nome_agencia
FROM agencia as T, agencia as S
WHERE T.fundos > S.fundos and
      S.nome_cidade = 'Brooklyn';
```

128

## 8. Subconsultas Aninhadas

### 8.2. Comparação de Conjuntos(some)

- A SQL, de fato, oferece alternativas de estilos para escrever a consulta precedente. A frase “maior que ao menos uma” é representada em SQL por **> some**. Esse construtor permite-nos reescrever a consulta em uma forma que remonta intimamente sua formulação em português.

```
SELECT nome_agencia
FROM agencia
WHERE fundos > SOME(SELECT fundos
                     FROM agencia
                     WHERE
                       nome_cidade = 'Brooklyn');
```

- A SQL permite além disso comparações **< some**, **<= some**, **>= some**, **= some** e **<> some**. Verifique que **= some** é idêntico a **in**, enquanto, **<> some** não é a mesma coisa que **not in**.

129

## 8. Subconsultas Aninhadas

### 8.2. Comparação de Conjuntos(all)

- Agora modificaremos ligeiramente nossa consulta.
- Encontraremos os nomes de todas as agências que tenham fundos maiores que cada uma das agências do Brooklyn.
- O construtor **> all** corresponde à frase “maior que todos”. Usando esse construtor, escrevemos a consulta como segue:

```
SELECT nome_agencia
FROM agencia
WHERE fundos > ALL(SELECT fundos
                   FROM agencia
                   WHERE
                     nome_cidade = 'Brooklyn');
```

- Como ocorre para **some**, a SQL permite além disso comparações **< all**, **<= all**, **>= all**, **= all** e **<> all**. Como exercício, verifique que **= some** é idêntico a **in**, enquanto, **<> some** não é a mesma coisa que **not in**. Verifique que **<> all** é idêntica a **not in**.

130

## 8. Subconsultas Aninhadas

### 8.3. Relações Vazias

- A SQL possui meios de testar se o resultado de uma subconsulta possui alguma tupla.
- O construtor **exists** retorna o valor **true**(verdadeiro) se o argumento de uma subconsulta é não-vazio.
- Por meio do construtor **exists** podemos escrever a consulta “encontre todos os clientes que tenham tanto conta quanto empréstimo no banco” de outro modo:

```
SELECT nome_cliente
FROM devedor
WHERE EXISTS (SELECT *
              FROM depositante
              WHERE depositante.nome_cliente =
                devedor.nome_cliente);
```

- Podemos testar a não existência de tuplas na subconsulta por meio do construtor **not exists**.

131

## 8. Subconsultas Aninhadas

### 8.4. Tuplas Repetidas

- A SQL contém recursos para testar se uma subconsulta possui alguma tupla repetida em seu resultado. O construtor **unique** retorna o valor **true**(verdadeiro) caso o argumento da subconsulta não possua nenhuma tupla repetida.
- Usando o construtor **unique** podemos escrever a consulta “encontre todos os clientes que tenham somente uma conta na agência Perryridge”, como segue:

```
SELECT T.nome_cliente
FROM depositante AS T
WHERE UNIQUE (SELECT R.nome_cliente
              FROM conta, depositante AS R
              WHERE T.nome_cliente = R.nome_cliente AND
                R.numero_conta = conta.numero_conta AND
                Conta.nome_agencia = 'Perryridge');
```

- Podemos testar a existência de tuplas repetidas em uma subconsulta por meio do construtor **not unique**.

132

## 8. Subconsultas Aninhadas

### 8.5. Subconsulta Escalar

- Uma expressão de **subconsulta** escalar é uma subconsulta contida na lista select para retornar um único valor.

```
SELECT saldo,  
       (select max(saldo)  
        from conta) maior  
FROM conta;
```

133

## 8. Subconsultas Aninhadas

### 8.5. Subconsulta Escalar

- Pode-se utilizar **subconsulta** também na cláusula from.

```
SELECT distinct dep.nome_cliente  
FROM (select nome_cliente  
      from depositante) dep,  
     (select nome_cliente  
      from devedor) dev  
WHERE dep.nome_cliente = dev.nome_cliente;
```

134

## 8. Subconsultas Aninhadas

- Resolver os exercícios de DML  
Subconsulta  
(ExercicioDML\_Subconsulta.pdf)

135

## 8. Subconsultas Aninhadas

- O código dos pilotos que não estão escalados.

```
R.  
SELECT Companhia  
FROM Piloto  
WHERE codigo_piloto IN (  
    SELECT Codigo_piloto  
    FROM Escala  
    WHERE aviao='MD11');
```

136

## 8. Subconsultas Aninhadas

- As companhias que não voam de MD11.\*

```
R.  
SELECT piloto.companhia  
FROM Piloto, Escala  
WHERE piloto.codigo_piloto = escala.codigo_piloto  
    and aviao <> 'MD11'  
    and piloto.companhia not in (  
    SELECT piloto.companhia  
    FROM Piloto, Escala  
    WHERE piloto.codigo_piloto =  
          escala.codigo_piloto  
    and aviao='MD11');
```

137

## 8. Subconsultas Aninhadas

- As companhias que só voam de MD11.\*

```
R.  
SELECT piloto.companhia  
FROM Piloto, Escala  
WHERE piloto.codigo_piloto = escala.codigo_piloto  
    and aviao='MD11'  
    and piloto.companhia not in (  
    SELECT piloto.companhia  
    FROM Piloto, Escala  
    WHERE piloto.codigo_piloto =  
          escala.codigo_piloto  
    and aviao <> 'MD11');
```

138

## 8. Subconsultas Aninhadas

- O nome dos pilotos que voam de MD11 ou de 737.

```
R.
SELECT nome_piloto
FROM piloto
WHERE codigo_piloto IN (SELECT codigo_piloto
                        FROM escala
                        WHERE aviao = 'MD11' or aviao = '737');

Ou
SELECT nome_piloto
FROM piloto
WHERE codigo_piloto IN (SELECT codigo_piloto
                        FROM escala
                        WHERE aviao IN ('MD11','737'));
```

139

## 8. Subconsultas Aninhadas

- O nome das companhias que paguem salários maiores que ao menos um salário pago por companhias brasileiras.

```
R.
SELECT companhia
FROM piloto
WHERE salario > SOME(SELECT salario
                     FROM piloto
                     WHERE pais = 'Brasil');
```

140

## 8. Subconsultas Aninhadas

- O nome das companhias que paguem salários maiores que todos os salários pago por companhias brasileiras.

```
R.
SELECT companhia
FROM piloto
WHERE salario > ALL(SELECT salario
                   FROM piloto
                   WHERE
                   pais = 'Brasil');
```

141

## 8. Subconsultas Aninhadas

- O nome dos pilotos que estão escalados. (Usar exists)

```
R.
SELECT nome_piloto
FROM piloto
WHERE EXISTS(SELECT *
             FROM escala
             WHERE
             escala.codigo_piloto =
             piloto.codigo_piloto);
```

142

## 9. Modificações no BD

### 9.1. Inserção

- Para inserir dados em relação podemos especificar uma tupla a ser inserida ou escrever uma consulta cujo resultado é um conjunto de tuplas a inserir. A inserção é expressa por:

```
INSERT INTO r(A1, A2,..., AN)
VALUES (V1, V2,...,VN);
```

- Em que *r* é a relação *Ai* representa os atributos a serem inseridos e *Vi* os valores contidos nos atributos *Ai*.

143

## 9. Modificações no BD

### 9.1.1. Inserção mais complexa

- A instrução **insert** pode conter subconsultas que definem os dados a serem inseridos:

```
INSERT INTO r_temp(A1, A2,..., AN)
SELECT (A1, A2,..., AN)
FROM r;
```

- Com isto a instrução **insert** insere mais de uma tupla.

144



## 9. Modificações no BD

### 9.1.2. Inserção com Null

- Na inserção também podemos utilizar a palavra-chave **null** para que se construa uma única expressão de insert.

```
INSERT INTO
cliente(nome_cliente, rua_cliente, nome_cidade,
salario_cliente, data_nascimento, cpf_cliente)
VALUES ('João da Silva', null, null, null, null, null);
```

- Para preencher os outros atributos de cliente basta trocar **null** pelo valor desejado.

145

## 9. Modificações no BD

### 9.1.3. Inserção de Data

- Para inserir datas no banco de dados é necessário especificar o formato utilizado pelo sistema gerenciador de banco de dados.
- As datas geralmente são consideradas como texto na inserção e convertidas para um tipo específico depois de armazenadas(Date).

- '01/02/1999'
- '01/02/1999 11:20:21'

146

## 9. Modificações no BD

### 9.1.3. Inserção de Data

```
INSERT INTO cliente(nome_cliente,
rua_cliente,
nome_cidade,
salario_cliente,
data_nascimento)
VALUES('FELIX', 'rua_cliente
1', 'FLORIANOPOLIS', 10000,
to_date('01/02/1999', 'DD/MM/YYYY'));
```

147

## 9. Modificações no BD

### 9.1.3. Inserção de Data

- to\_date** é uma função do Oracle
  - dois parâmetros.
    - DD** dia com dois dígitos(1-31),
    - MM** mês com dois dígitos(1-12),
    - YYYY** ano com quatro dígitos,
    - HH** hora com dois dígitos(0-12),
      - HH24** hora com dois dígitos(0-23),
    - MI** minuto com dois dígitos(0-59)
    - SS** segundo com dois dígitos(0-59).

- No apêndice 2 item 9.5.3.2 existe outros formatos para as datas.

148

## 9. Modificações no BD

### 9.1.3. Inserção de Data

```
SELECT nome_cliente, rua_cliente,
nome_cidade, salario_cliente,
to_char(data_nascimento, 'DD/MM/YYYY')
FROM cliente;
```

149

## 9. Modificações no BD

### 9.2. Alteração

- Em determinadas situações, podemos querer modificar valores das tuplas sem, no entanto alterar todos os valores. Para esse fim, o comando update pode ser usado.

```
UPDATE r
SET A1 = V1, A2 = V2, ... NA = VN
WHERE p;
```

- em **r** é a relação, **A<sub>i</sub>** representa o atributo a ser atualizado e **V<sub>i</sub>** o valor a ser atualizado no atributo **A<sub>i</sub>**, e **P** um predicado. A cláusula where pode ser omitida nos casos de atualização de todas as de **r**.

150

## 9. Modificações no BD

### 9.3. Remoção

- A remoção de dados é expresso muitas vezes do mesmo modo que uma consulta.
- Podemos remover somente tuplas inteiras; não podemos excluir valores de um atributo em particular.
- Em SQL, a remoção é expressa por:

```
DELETE FROM r  
WHERE P;
```

- ☐ P representa um predicado
- ☐ r uma relação.
- O comando delete encontra primeiro todas as tuplas t em r para as quais  $P(t)$  é verdadeira e então removê-las de r.
- A cláusula where pode ser omitida nos casos de remoção de todas as de r.

151

## 9. Modificações no BD

- Resolver os exercícios de DML  
Modificação  
(ExercicioDML\_Modificacao.pdf)

152

## 10. Transações

- Transação é uma unidade **atômica** de trabalho que atua sobre um banco de dados.
- Uma transação pode ser constituída por **uma** ou **mais** operações de acesso à base de dados.
- Todas as operações devem ser bem-sucedidas, caso contrário os efeitos da transação devem ser **revertidos**.

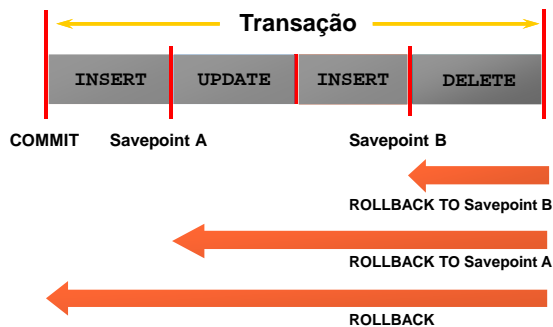
153

## 10. Transações

- Inicia com o **primeiro** comando SQL emitido para a base de dados
- Finaliza com os seguintes eventos:
  - ☐ COMMIT ou ROLLBACK
  - ☐ Comandos DDL executam commit automático
  - ☐ Saída do Usuário
  - ☐ Queda do Sistema

154

## 10. Transações



155

## 10. Transações

### 10.1. COMMIT

- Uma transação bem-sucedida termina quando um comando **COMMIT** é executado.
- O comando **COMMIT** finaliza e efetiva todas as alterações feitas na base de dados durante a transação.
- **Grava** uma transação no banco de dados.
- Sintaxe:

```
COMMIT ;
```

156

## 10. Transações

### 10.1. COMMIT

#### Alteração de Dados

```
SQL> UPDATE funcionario
2 SET      codigo_departamento = 3
3 WHERE    rg_funcionario = '1112';
1 linha atualizada.
```

#### Commit nos Dados

```
SQL> COMMIT;
Commit concluído.
```

157

## 10. Transações

### 10.2. ROLLBACK

- Restaura uma transação.
- Recupera o banco de dados para a posição que esta após o último comando **COMMIT** ser executado.
- Sintaxe:  
**ROLLBACK;**

158

## 10. Transações

### 10.2. ROLLBACK

- Descarta todas as mudanças da transação.
  - Os valores anteriores são recuperados.
  - Os bloqueios são desfeitos.

```
SQL> DELETE FROM funcionario;
14 linhas deletadas.
SQL> ROLLBACK;
Rollback concluído.
```

159

## 10. Transações

### 10.3. Pontos de Transação

- Salvando
  - Marca um ponto de início de transação.
  - Sintaxe:  
**SAVEPOINT <nome\_do\_ponto>;**
- Recuperando
  - Executa um ROLLBACK até um ponto de transação salva.
  - Sintaxe:  
**ROLLBACK [TO SAVEPOINT <nome\_do\_ponto>];**

160

## 10. Transações

### ■ Operação 1

```
SELECT *
FROM profbd2.cidadetemp;
```

### ■ Operação 2

```
UPDATE profbd2.cidadetemp
set populacao=populacao*1.1;
COMMIT;
```

161

## 10. Transações

### 10.4. Locks

- Locks são recursos de compartilhamento de dados, que permitem que o dado seja atualizado e pesquisados dentro de um ambiente multi-usuário de maneira segura e que lhes garante confiabilidade e integridade.
  - garantir que somente um usuário esteja atualizando
  - vários usuários possam pesquisar
- Há, normalmente, independente da nomenclatura dada por fornecedores de SGBDs, dois níveis de locks importantes:

162

## 10. Transações

### 10.4. Locks

- **Lock Exclusivos - XLOCKS (eXclusive Locks)**
  - Usados para garantir o uso de um determinado dado por um único usuário. É utilizado em casos de atualizações.
- **Locks Compartilhados - SLOCKS (Shared Locks),**
  - Usados para permitir que mais de um usuário acesse o mesmo dado ao mesmo tempo.

163

## 10. Transações

### 10.5. Estado dos Dados Antes de COMMIT/ROLLBACK

- O estado anterior do dado pode ser recuperado.
- Outros usuários não podem ver as alterações efetuadas.
- As linhas afetadas pela transação são bloqueadas (locked) até se completar a transação.

164

## 10. Transações

### 10.5. Estado dos dados Após COMMIT/ROLLBACK

- Os dados são alterados definitivamente na base de dados.
- O valor anterior dos dados não são recuperados.
- Todos os usuários vêem o mesmo resultado.
- Os bloqueios (locks) são desfeitos, liberando os dados para os outros usuários.
- Todos os Savepoints são eliminados.

165

## 10. Transações

### 10.6. Rollforward

- Quando da reinicialização de um banco de dados após uma falha, algumas transações podem ter sido perdidas na memória, embora um comando Commit já tenha sido emitido. Isto quer dizer que no log de transações a transação é considerada completa. No entanto, os efeitos não foram registrados em definitivo na base de dados.
- Isto quer dizer que esta transação deve ser completada novamente. Usando o log de transações o SGDB "sabe" que partes da transação ainda não foram gravados em definitivo. As etapas que ainda faltam ser gravadas são então executadas até ser encontrado o comando COMMIT gravado no log de transações. A este processo chamamos **refazer a transação** ou **rollforward**.
- O rollforward pode ser apenas executado quando do processo de reinicialização do banco de dados.

166

## 10. Transações

### 10.7. Syncpoint

- Alguns SGDBs permitem que cada transação completada seja imediatamente gravada na base, isto gera uma sobrecarga de gravação contra o banco, o que ocasiona uma **queda** de performance.
- Cada operação que constitui uma transação pode ser mantida em memória (gerenciada por **páginas**, **gerenciadores de cache**, etc) e registrada contra o log de transações, sendo gravada contra a base em intervalos de tempo pré-determinados.
- Quando terminado o intervalo, todos os efeitos das alterações mantidos em memória são gravados efetivamente contra a base, sem prejuízo do gerenciamento de lock em curso. A este "alarme" que permite disparar a efetivações contra a base chamamos **syncpoint**.
  - Os syncpoints sincronizam log de transações, base de dados e memória.

167

## 11. Índice

- O método básico que os **SGDBs** utilizam para **localizar tuplas** consiste em varrer sistematicamente as tabelas em busca dos registros que satisfazem uma determinada chave (condição de seleção).
- Se uma consulta envolve um número muito grande de registros, o processo de busca pode apresentar **desempenho indesejável**.
- Os índices têm como principal característica **acelerar o processamento de consultas** submetidas ao banco de dados.

168

## 11. Índice

### ■ Classificação de índices

#### □ Índice Ordenado

##### ■ Primário

##### □ Denso

##### □ Esparso

##### ■ Secundário

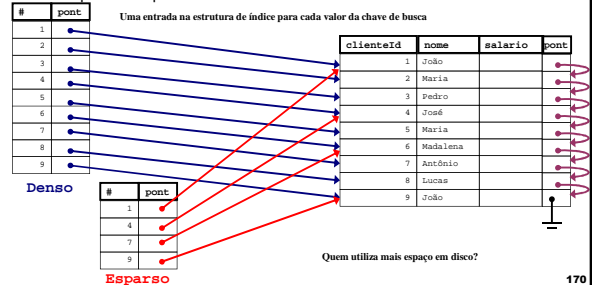
#### □ Índice Hash

169

## 11. Índice

### ■ Índice Primário (Clusterizado/Agrupamento)

- Definido sobre uma chave de busca que determina a **ordem física** das tuplas no arquivo de índice. Ex Chave Primária

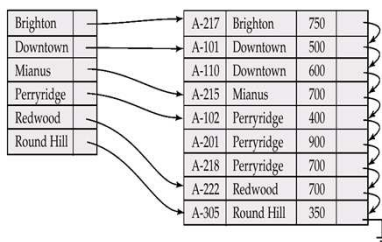


170

## 11. Índice

### ■ Índice Primário (Clusterizado/Agrupado)

- A chave de busca de um índice primário normalmente é a **chave primária**, mas não necessariamente;



171

## 11. Índice

### ■ Índice Denso

- Sequência de blocos contendo apenas as chaves das tuplas e os ponteiros para os próprios tuplas

- Índice denso = (chave, ponteiro) -> tuplas

### ■ Índice Esparso

- Usa menos espaço de armazenamento que o índice denso ao custo de um tempo um pouco maior para localizar uma tupla dada a sua chave

- Índice esparso = (chave, ponteiro) -> blocos de dados
- Aponta para o 1o. registro do bloco

172

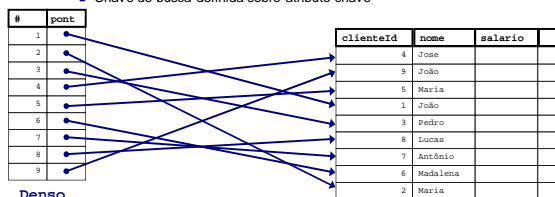
## 11. Índice

### ■ Índice Secundário (Não Clusterizado, Não Agrupado)

- A chave de busca não determina a ordem física das tuplas
- Índices secundários determinam uma ordem (lógica) das tuplas

#### □ Exemplo

- Chave de busca definida sobre atributo chave



**Índice secundário é sempre denso**

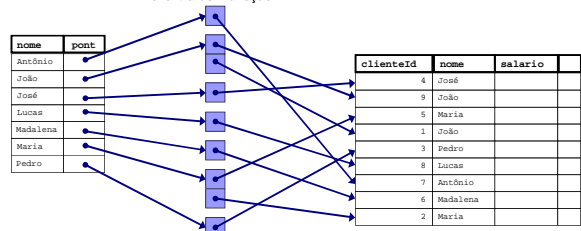
173

## 11. Índice

### ■ Índice Secundário

#### □ Exemplo

- Chave de busca definida por atributo não chave
- Nível extra de indireção

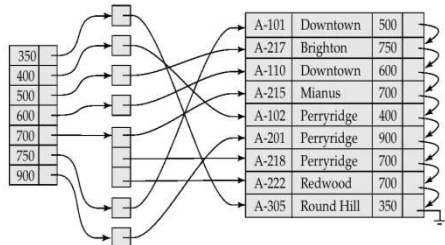


174

## 11. Índice

### Índice Secundário

- Exemplo: índice secundário sobre o campo saldo de conta



## 11. Índice

### Índice Hash

- Estrutura de índice organizada como um arquivo hash

#### Hashing Estático

- Bucket** representa uma unidade armazenamento de um conjunto de tuplas
- Função de Hash(h)

- $h: K \rightarrow B$ , onde
  - $K$  representa o conjunto de chaves de busca
  - $B$  é o conjunto do endereço de buckets

#### Eficiente para consultas do tipo *exact match*

- Pode ser utilizado para implementar o operador físico de *seleção* com igualdade sobre atributos chaves
  - `Select  $A_1, A_2, A_3, \dots, A_n$`
  - `From s`
  - `where  $A_k = c$`

176

## 11. Índice

### Hashing Estático

- A função de hashing trabalha com um número fixo de registros possíveis para a tabela, evitando com isto que duas chaves tenham o mesmo valor de hashing. Uma alteração na quantidade de registros pode causar uma necessidade de chamar da função hash novamente.

### Hashing Dinâmico

- A função de hashing não depende da quantidade de registros da tabela, necessita de muito mais elaboração por parte do fabricante pois os limites dos registros possíveis em uma tabela deve ser levado em consideração.

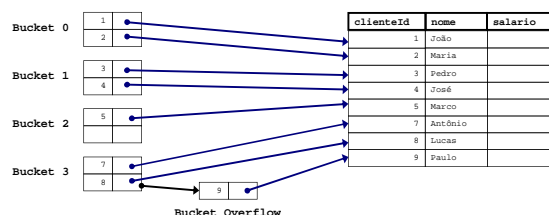
177

## 11. Índice

### Índice Hash

#### Hashing Estático

- $h(clientId) = clientId \bmod 2$



178

## 11. Índice

- Técnicas de organização ou estruturas de dados para arquivos de índice são chamadas de **Métodos de Acesso**.
- Em sistemas relacionais, a estrutura mais comumente utilizada para representar índice são as **Árvore B+**.
- Uma **Árvore B+** é uma árvore balanceada, cujas folhas contêm uma seqüência de pares chave-ponteiro, onde as chaves são ordenadas pelo seu valor.
- A **Árvore-B+** é ideal para aplicações que requerem tanto acesso seqüencial quanto aleatório.

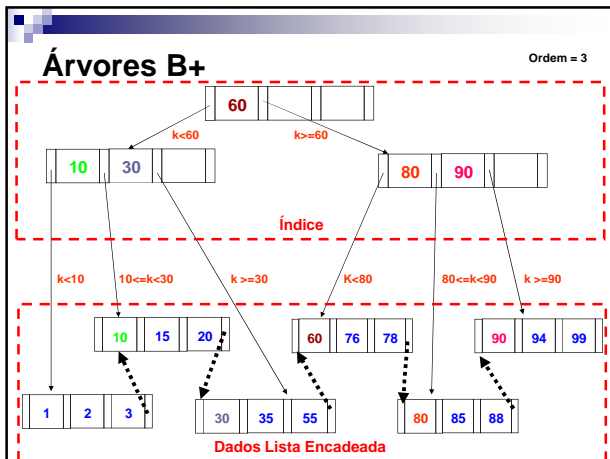
179

## 11. Índice

### Estrutura da **Árvore B+**:

- Todas as chaves são mantidas em folhas;
- As chaves são repetidas em nós não-folha formando um índice para localizar registros individuais;
- As folhas são ligadas através de uma **lista duplamente encadeada** oferecendo um caminho seqüencial para percorrer as chaves.
- Isso permite o armazenamento dos dados em um arquivo, e do índice em outro arquivo separado

180



## 11. Índice

- A **SQL-padrão** não provê nenhuma maneira de o usuário ou administrador de banco de dados controlar quais índices são criados e mantidos no sistema de banco de dados.
- São importantes para o processamento eficiente de transações, incluindo transações de atualizações e consultas.
- Os índices também são importantes para a implementação eficiente das restrições de integridade.
- Um índice é criado pelo comando `create index`, que tem a seguinte forma:

```
CREATE INDEX <nome_do_indice>
ON <nome_da_tabela> (<lista_atributos>);
```

182

## 11. Índice

- Se desejamos declarar que a chave de procura é uma chave candidata, adicionamos o atributo **unique** à definição do índice:

```
CREATE [UNIQUE] INDEX <nome_do_indice>
ON <nome_tabela> (lista_de_atributos);
```

- Removendo índice  
`DROP INDEX <nome_do_indice>;`
- Listando os índices criados  
`SELECT Index_name FROM user_indexes;`
- Listando as tabelas e seus índices  
`SELECT Table_name, Index_name FROM user_indexes;`

183

## 11. Índice

- **DDL Tabela CidadeTemp**

```
CREATE TABLE cidadetemp(
  cidadeId INT,
  nome VARCHAR(15),
  uf VARCHAR(2),
  populacao NUMERIC(9,0),
  CONSTRAINT pk_cidadeId_temp
    PRIMARY KEY(cidadeId));
```

- **Povoamento da Tabela CidadeTemp**

```
DECLARE
  A NUMBER(3) := 0;
BEGIN
  FOR A IN 1..2000000 LOOP
    INSERT INTO cidadetemp VALUES(A, 'CIDADE' || A, 'UF', A);
  END LOOP;
  COMMIT;
END;
```

184

## 11. Índice

### ■ Operação 1

```
SELECT * FROM profbd2.cidadetemp
WHERE cidadeId = 599999;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	2
TABLE ACCESS (BY INDEX ROWID)	CIDADETEMP	1	2
INDEX (UNIQUE SCAN)	PK_CIDADEID_TEMP	1	1
Access Predicates			
CIDADEID=599999			

185

## 11. Índice

### ■ Operação 2

```
SELECT * FROM profbd2.cidadetemp
WHERE nome = 'CIDADE599999';
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		100	2478
TABLE ACCESS (FULL)	CIDADETEMP	100	2478
Filter Predicates			
NOME='CIDADE599999'			

186

## 11. Índice

### ■ Criando o índice!

```
CREATE INDEX  
ix_cidadetemp_nome on  
cidadetemp(nome);
```

187

## 11. Índice

### ■ Operação 3-Executando novamente!

```
SELECT * FROM profbd2.cidadetemp  
WHERE nome = 'CIDADE599999';
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		100	1552
TABLE ACCESS (BY INDEX ROWID)	CIDADETEMP	100	1552
INDEX (RANGE SCAN)	IX_CIDADETEMP_NOME	6686	3
Access Predicates			
NOME='CIDADE599999'			

188

## 11. Índice

### ■ Apago o índice anterior!

```
DROP INDEX ix_cidadetemp_nome;
```

### ■ Criando o índice único!

```
CREATE UNIQUE INDEX  
ix_cidadetemp_nome on  
cidadetemp(nome);
```

189

## 11. Índice

### ■ Operação 4-Executando novamente!

```
SELECT * FROM profbd2.cidadetemp  
WHERE nome = 'CIDADE599999';
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	3
TABLE ACCESS (BY INDEX ROWID)	CIDADETEMP	1	3
INDEX (UNIQUE SCAN)	IX_CIDADETEMP_NOME	1	2
Access Predicates			
NOME='CIDADE599999'			

190

## 11. Índice

### ■ Operação 5

```
SELECT * FROM profbd2.cidadetemp  
WHERE nome like '%59999';
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		100	2476
TABLE ACCESS (FULL)	CIDADETEMP	100	2476
Filter Predicates			
AND			
NOME IS NOT NULL			
NOME LIKE '%59999'			

191

## 11. Índice

### ■ Operação 6

```
SELECT * FROM profbd2.cidadetemp  
WHERE nome like '59999%';
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		100	2460
TABLE ACCESS (BY INDEX ROWID)	CIDADETEMP	100	2460
INDEX (RANGE SCAN)	IX_CIDADETEMP_NOME	15043	61
Access Predicates			
NOME LIKE '59999%'			
Filter Predicates			
NOME LIKE '59999%'			

192



## 12. Visões

- **Visões** ou **View** é uma tabela virtual ou lógica que permite a visualização e manipulação do conteúdo de uma ou mais tabelas.
- Uma visão tem a aparência e o funcionamento **parecido** com a de uma tabela normal, só que as suas colunas podem ser oriundas de diversas tabelas diferentes. As tabelas que dão origem às colunas são chamadas de **tabelas-base**.
- Um dos objetivos do uso de **views** é restringir o acesso a certas porções dos dados por questões de segurança, além de pré-definir (armazenar) certas consultas através de tabelas virtuais que poderão ser utilizadas por outras consultas.

193

## 12. Visões

### 12.1 Visões Normais

- Alguns bancos de dados trabalham com dois tipos de **views**. O Oracle trabalha com **views** “**normais**” e **views** “**materializadas**”.
- A **View Normal** é “**materializada**” no momento da consulta. Com isto não é necessário fazer cópias dos dados.
- A consulta à **view** é resolvida sobre as próprias tabelas originais, o que minimiza qualquer perda de desempenho ou espaço.

194

## 12. Visões

### 12.1 Visões Normais

- Definimos uma visão em SQL usando o comando **create view**. Para definir a visão, precisamos dar-lhe um nome e definir a consulta que processará essa visão. A forma do comando **create view** é:

```
CREATE [or replace] VIEW
<nome_da_visão>
AS <expressão_da_consulta>;
```

- em que <expressão da consulta> é qualquer expressão de consulta válida e o nome da visão é representado por <nome\_da\_visão>. A opção **or replace** recria uma visualização já existente.

195

## 12. Visões

### 12.1 Visões Normais

- Como exemplo, considere uma visão composta dos nomes de agências e nomes de clientes que tenham uma conta ou um empréstimo na agência. Suponha que desejamos que essa visão seja denominada **todos\_clientes**. Definimos essa visão como segue:

```
CREATE VIEW VW_todos_clientes AS
( SELECT nome_agencia, nome_cliente
  FROM depositante, conta
  WHERE depositante.numero_conta = conta.numero_conta)

UNION

( SELECT nome_agencia, nome_cliente
  FROM devedor, emprestimo
  WHERE emprestimo.numero_emprestimo =
    devedor.numero_emprestimo);
```

196

## 12. Visões

### 12.1 Visões Normais

- Os nomes dos atributos de visão devem ser especificados explicitamente, conforme segue:

```
CREATE VIEW VW_emprestimo_total_agencia (nome_agencia,
emprestimo_total) AS
SELECT nome_agencia, sum(total)
FROM emprestimo
GROUP BY nome_agencia;
```

- A visão anterior cede para cada agência a soma dos totais de todos os empréstimos da agência. Uma vez que a expressão **soma (total)** não possui nome, o nome do atributo é especificado explicitamente na definição da visão (**emprestimo\_total**).

197

## 12. Visões

### 12.1 Visões Normais

- Os nomes de visão podem aparecer em qualquer lugar onde o nome de uma relação aparece. Usando a visão **todos\_clientes**, podemos encontrar todos os clientes da agência Agência1, escrevendo:

```
SELECT nome_cliente
FROM VW_todos_clientes
WHERE nome_agencia = 'AGENCIA1';
```

198

## 12. Visões

### 12.1 Visões Normais

- Apagando visões

```
DROP VIEW <nome_da_visão>;
```

- Listando as visões criadas

```
SELECT View_Name  
FROM User_Views;
```

ou

```
SELECT * FROM Cat  
WHERE TABLE_TYPE = 'VIEW';
```

199

## 12. Visões

### 12.2 Visões Materializadas

- As **Views Materializadas**, são *views* que a cada requisição ou chamada acessa dados em tabelas virtuais gerenciadas pelo banco de dados, aos quais são previamente otimizadas para que o retorno dos dados seja feita de forma mais otimizada, para dados que precisam sofrer algum processamento (cálculo) no SGBD.
- Esses dados são atualizados sob demanda, ou seja, quando solicitado pelo usuário ou quando o mesmo é programado para que seja de forma automática.
  - Indicado para ambientes que trabalham com **Data Warehouse** pois são utilizados principalmente para consultas de grandes volumes de dados.

200

## 12. Visões

### 12.2 Visões Materializadas

- A sintaxe para criar Views Materializadas é:

```
CREATE MATERIALIZED VIEW <nome_da_visão>  
BUILD [IMMEDIATE | DEFERRED]  
REFRESH [FAST | COMPLETE | FORCE ]  
ON [COMMIT | DEMAND ]  
[[ENABLE | DISABLE] QUERY REWRITE]  
AS <expressão_da_consulta>;
```

- Visões materializadas precisam ser apagadas para serem modificadas, OR REPLACE não funciona.

201

## 12. Visões

### 12.2 Visões Materializadas

- A cláusula **BUILD** possui as seguintes opções:
  - **IMMEDIATE**: Preenchida imediatamente após sua criação.
  - **DEFERRED**: Preenchida na primeira atualização solicitada.
- A cláusula **REFRESH** possui as seguintes opções:
  - **FAST**: Somente alterações entre o intervalo de tempo.
  - **COMPLETE**: Recria toda a estrutura da *view* materializada mesmo que não seja necessário.
  - **FORCE**: Faz o FAST ser for possível, caso contrário faz o processo COMPLETE.
- A atualização pode ser acionada de duas maneiras.
  - **ON COMMIT**: A atualização é desencadeada por uma alteração de dados em uma das tabelas base. Você não precisa lembrar de atualizar as *views*.
  - **ON DEMAND**: A atualização é iniciada por um pedido manual ou uma tarefa agendada (DBMS\_MVIEW).
  - **START WITH**: A atualização ocorre e uma data e hora.
- A cláusula de consulta **QUERY REWRITE** diz ao banco se a visão materializada deve ser reescrita. Por default é **DISABLE**.

202

## 12. Visões

### 12.2 Visões Materializadas

- Reescrevendo uma visão normal para uma visão materializada temos:

```
CREATE MATERIALIZED VIEW VWM_emprestimo_total_agencia  
AS  
SELECT nome_agencia, sum(total) emprestimo_total  
FROM emprestimo  
GROUP BY nome_agencia;
```

- Equivale:

```
CREATE MATERIALIZED VIEW VWM_EMPRESTIMO_TOTAL_AGENCIA  
BUILD IMMEDIATE  
REFRESH FORCE  
ON DEMAND  
DISABLE QUERY REWRITE  
AS SELECT nome_agencia, sum(total) emprestimo_total  
FROM emprestimo  
GROUP BY nome_agencia;
```

203

## 12. Visões

### 12.2 Visões Materializadas

- Listando as visões materializadas criadas

```
SELECT mview_name, staleness  
FROM User_Mviews;
```

- **Staleness** indica se a visão foi atualizada ou não (NEEDS\_COMPILE OU FRESH).

- Atualizando uma visão materializada

```
BEGIN  
DBMS_MVIEW.REFRESH('VWM_emprestimo_total_agencia');  
END;
```

- Apagando visões materializadas

```
DROP MATERIALIZED VIEW <nome_da_visão>;
```

204

## 12. Visões

- Resolver os exercícios de DML Visão (ExercicioDML\_View.pdf)

205

## 12. Visões

- 1) Os dados de todos os pilotos de companhias brasileiras.

R.

```
CREATE OR REPLACE VIEW
vw_piloto_brasil As
SELECT *
FROM piloto
WHERE upper(pais) = 'BRASIL';
```

206

## 12. Visões

- 3) O nome de todos os pilotos escalados.

R.

```
CREATE OR REPLACE VIEW vw_piloto_escalado As
SELECT distinct nome_piloto
FROM piloto p, voo v, escala e
WHERE p.codigo_piloto = e.codigo_piloto
AND e.codigo_voo = v.codigo_voo;
```

207

## 12. Visões

- 6) O nome de todos os pilotos, junto com seu salário e gratificação.

R.

```
CREATE OR REPLACE VIEW vw_piloto_gratificacao As
SELECT nome_piloto, salario,
       salario*1.2 gratificacao
FROM piloto;
```

Ou

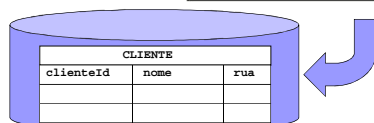
```
CREATE OR REPLACE VIEW
vw_piloto_gratificacao(nome_piloto, gratificacao)
As
SELECT nome_piloto, salario,
       salario*1.2
FROM piloto;
```

208

## 13. Seqüência

```
SELECT
NVL(MAX(clienteId),0)+1
FROM cliente;
```

NVL retorna o primeiro argumento se não for nulo, caso contrário retorna o segundo argumento(0).



209

## 13. Seqüência

### ■ Opção 1

```
SELECT NVL(MAX(clienteId),0)+1
FROM cliente;
```

- Funciona muito bem para sistemas monousuários, por estar a transação ocorrendo em uma única sessão

210

## 13. Seqüência

### ■ Opção 2

```
SELECT id
FROM geradorid
WHERE tabela='CLIENTE';
```

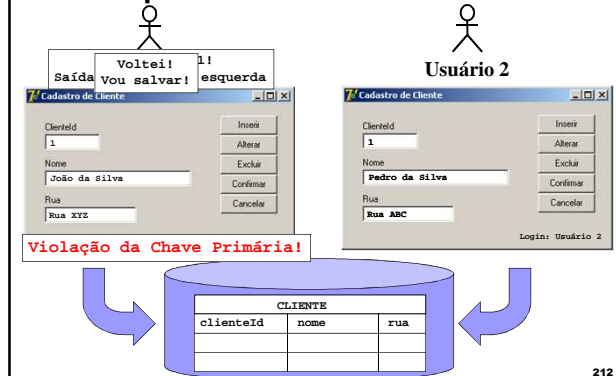
```
UPDATE geradorid
SET id = id + 1
WHERE tabela='CLIENTE';
```

- Criar uma tabela para armazenar os identificadores das tabelas.
- Fazer uma consulta e depois incrementar o valor do identificador(ID).

GERADORID	
tabela	id
CLIENTE	1

211

## 13. Seqüência



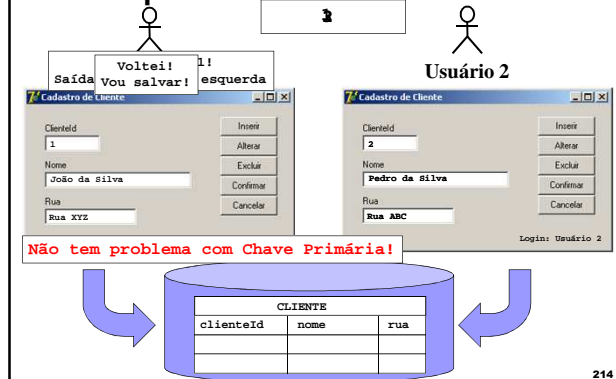
212

## 13. Seqüência

- Se dois usuários em sessões distintas executarem o sistema um não tem conhecimento da transação concorrente.
- A propriedade (**ISOLAMENTO**) das propriedades **ACID** esta sendo violada.
  - ATOMICIDADE - Tudo (commit) ou nada (rollback)
  - CONSISTÊNCIA - Sem violação de restrições de integridade
  - ISOLAMENTO - Alterações concorrentes invisíveis
  - DURABILIDADE - Persistência das atualizações confirmadas

213

## 13. Seqüência



214

## 13. Seqüência

### ■ Opção 3

```
SELECT id
FROM geradorid
WHERE tabela='CLIENTE';
```

```
UPDATE geradorid
SET id = id + 1
WHERE tabela='CLIENTE';
```

215

## 13. Seqüência

- As seqüências são usadas para facilitar o processo de criação de identificadores únicos de um registro em um banco de dados. Uma seqüência nada mais é do que um contador automático incrementado toda vez que é acessado. O número gerado pelo contador pode ser usado para atualizar um campo do tipo código do produto ou código do cliente em uma tabela, garantido que não haja duas linhas com o mesmo código.
- Quando uma seqüência é criada, ela adota alguns valores-padrão que são adequados para a maioria dos casos. Uma seqüência padrão tem as seguintes características:
  - Começa sempre a partir do número 1
  - Tem ordem ascendente
  - É aumentada em 1

216

## 13. Seqüência

### 13.1. Estrutura

- O comando SQL usado para a criação de uma seqüência é o comando **create sequence**.  
Sintaxe básica:  

```
CREATE SEQUENCE <nome_da_seqüencia>
[ start with <numero_inicio> ]
[ increment by <numero_incremento> ]
[ minvalue <numero_minimo> ]
[ nominvalue ]
[ maxvalue <numero_maximo> ]
[ nomaxvalue ]
[ cycle/nocycle ]
[ cache/nocache ]
```

  - **start with** – valor inicial.
  - **Increment by** – Indica o valor de aumento.
  - **minvalue** – valor mínimo que a seqüência poderá ter.
  - **nominvalue** – Indica que a seqüência não tem valor mínimo predefinido.
  - **maxvalue** – Indica o valor máximo que a seqüência poderá ter.
  - **nomaxvalue** – Indica que a seqüência não tem valor máximo predefinido.
  - **cycle/nocycle** – indica que, ao atingir o valor máximo, a seqüência deve retornar ao valor inicial. Por sua vez **nocycle** impede que a seqüência volte ao seu ciclo.
  - **cache/nocache** – armazena as seqüências em cache na memória para acesso mais rápido

217

## 13. Seqüência

### 13.2. Comandos

- Para obter o valor atual de uma seqüência, sem incrementar o seu valor uma pseudo-coluna chamada **CURRVAL** deve ser usada.
- Para obter e incrementar o valor da seqüência, uma pseudo-coluna chamada **NEXTVAL** deve ser usada.
- Essas duas pseudo-colunas podem ser usadas nas seguintes ocasiões:
  - Em um comando INSERT, como valor da cláusula VALUES.
  - Na cláusula SET do comando update.
  - Na lista de colunas do comando SELECT. No caso do comando SELECT, as pseudocolunas não podem ser usadas se o comando possuir as cláusulas ORDER BY, GROUP BY, DISTINCT ou uma subquery.

218

## 13. Seqüência

### 13.2. Comandos

- **Criando uma Seqüência**  
Uma seqüência somente está disponível para o esquema que a criou. Todas as seqüências criadas nesses exemplos são feitas sob um usuário. O próximo exemplo ilustra a criação da seqüência **sq\_departamento** usando os valores padrão.

```
CREATE SEQUENCE sq_departamento;
ou

CREATE SEQUENCE sq_departamento START WITH 10;
ou

CREATE SEQUENCE sq_departamento START WITH 10
INCREMENT BY 2;
```

219

## 13. Seqüência

### 13.2. Comandos

- **Usando e Acionando uma Seqüência**  
Na primeira vez que a seqüência é acionada ela traz o seu valor inicial.  
  

```
SELECT sq_departamento.nextval FROM dual;
```
- **Visualizando o valor atual da seqüência**  
Utilizar **currval** não incrementa o valor da seqüência.

```
SELECT sq_departamento.currval FROM dual;
```

dual é uma tabela auxiliar de 1 campo e nenhuma linha.

220

## 13. Seqüência

### 13.2. Comandos

- **Usando na inclusão de registros**  
Para incluir registros em **departamento** (**codigo\_departamento**, **nome**) usa-se da seguinte forma:

```
INSERT INTO departamento(codigo_departamento,nome)
VALUES(sq_departamento.nextval, 'ESTOQUE');
/
INSERT INTO departamento(codigo_departamento,nome)
VALUES(sq_departamento.nextval, 'FINANÇAS');
/
```

221

## 13. Seqüência

### 13.2. Comandos

- **Apagando uma seqüência**  
O comando **DROP SEQUENCE** remove a seqüência do esquema no qual foi criado.  
  

```
DROP SEQUENCE <nome_da_seqüencia>;
```
- **Visualizando as seqüências**  
Para o usuário corrente a tabela é **USER\_SEQUENCES**:

```
SELECT sequence_name
FROM USER_SEQUENCES;
```

222

## Bibliografia

### ■ Principal

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

### ■ Complementar

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

Fim