UDESC

Banco de Dados I

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

Profa. Rebeca Schroeder Freitas
Prof. Fabiano Baldo

UDESC

Álgebra Relacional

Seleção

- Retorna **tuplas** que satisfazem um **predicado**
- Notação: $\sigma_{predicado}(relação)$

  (sigma)
- Operadores de comparação: =, <, <=, >, >=, ≠
- Operadores lógicos: ∧ (and) ∨ (or) ¬ (not)

- Exemplo: $\sigma_{z \geq 2}(R)$

R

x	y	z
1	1	1
2	2	2
2	3	3

$resultado$

x	y	z
2	2	2
2	3	3

Projeção

- Retorna um ou mais atributos de interesse
- Notação: $\pi_{lista\_nomes\_atributos}\ (relação)$

  (pi)
- Elimina automaticamente duplicatas

- Exemplo: $\pi_{x,y}\ (R)$

R

x	y	z
1	1	1
2	2	2
2	2	3

$resultado$

x	y
1	1
2	2

Produto Cartesiano

- Retorna todas as combinações de tuplas de duas relações R_1 e R_2
- Grau do resultado: *grau*(R_1) + *grau*(R_2)
- Cardinalidade do resultado: *card*(R_1) * *card*(R_2)
- Notação: *relação1* × *relação2*

  (*cross joint*)
- Exemplo: $(R1 \times R2)$

$R1$

x	y	z
1	1	1
2	2	2
3	3	3

$R2$

w	y
1	1
2	2

$resultado$

x	R1y	z	w	R2y
1	1	1	1	1
1	1	1	2	2
2	2	2	1	1
2	2	2	2	2
3	3	3	1	1
3	3	3	2	2

Atribuição

- Armazena o resultado de uma expressão algébrica em uma **variável de relação**
- Possibilita processamento de consulta por etapas
- Notação: *nomeVariavel = ExpressaoAlgebrica*

  (*assignment*)
- Exemplo:

$R1 = \pi_{codm,\, data}\ (Consultas)$

$R2 = \pi_{codm,\, nome}\ (Medicos)$

$\pi_{codm,\, data}\ (\sigma_{\ Consultas.codm=Medicos.codm}\ (R1 \times R2)\ )$

Renomeação

- Altera o nome de uma relação ou de seus atributos
- Notação: (rho)
  - renomear relação: $\rho_{\ novoNomeRelação}\ (Relação)$
  - renomear colunas: $\rho_{\ novo1 \leftarrow orig1,\ novo2 \leftarrow orig2}\ (Relação)$
- Exemplo: $(R \times \rho_{N1}(R))$ $\rho_{a \leftarrow x,\ b \leftarrow y}(R)$

R

x	y
1	2
3	4

$resultado$

R.x	R.y	N1.x	N1.y
1	2	1	2
1	2	3	4
3	4	1	2
3	4	3	4

R.a	R.b
1	2
3	4

$resultado$

União, Diferença e Intersecção

- Operam somente sobre duas relações compatíveis
  - grau (R_1) = grau (R_2)
  - domínio atributo a_i de R_1 = domínio atributo b_i de R_2

- Resultado:
  - Grau: grau (R_1) [= grau (R_2)]
  - Atributos: nomes dos atributos de R1 (relação esquerda)

União

- Retorna a união das tuplas de duas relações
- Elimina duplicatas automaticamente
- Notação: $relacao_1 \cup relacao_2$
- Exemplo: $(R_1 \cup R_2)$

R_1

x	y
1	2
3	4
2	4

R_2

x	y
1	3
3	4
1	2
3	5

$resultado$

x	y
1	2
3	4
2	4
1	3
3	5

Diferença

- Retorna as tuplas que estão em R_1 e não estão em R_2
- Notação: $relacao_1$ - $relacao_2$
- Exemplo: $(R_1 - R_2)$

R_1

x	y
1	2
3	4
2	4

R_2

x	y
1	3
3	4
1	2
3	5

$resultado$

x	y
2	4

Intersecção

- Retorna as tuplas comuns entre R_1 e R_2
- Notação: $relacao_1 \cap relacao_2$
- Exemplo: $(R_1 \cap R_2)$

R_1

x	y
1	2
3	4
2	4

R_2

x	y
1	3
3	4
1	2
3	5

$resultado$

x	y
1	2
3	4

UDESC

Junção (*Join*)

- Retorna a combinação de tuplas de duas relações R_1 e R_2 que satisfazem um predicado.

- Tipos de junção:
  - *Junção Natural*
  - *Junções Externas*
    - *Junção externa à esquerda*
    - *Junção externa à direita*
    - *Junção externa completa*
  - *Semi-Junção*
  - *Anti-Junção*

Junção Natural (*Natural Join*)

- Junção em que a igualdade é predefinida entre todos os atributos que apresentam o mesmo nome nas relações

  (similar a um produto cartesiano condicional)

- Notação: $relacao_1 \bowtie_{(condição)} relacao_2$
- Exemplo: $(R_1 \bowtie_{(R1.y = R2.y)} R_2)$

R_1

x	y	z
1	**1**	1
1	**1**	2
2	**2**	3

R_2

w	y
7	**1**
4	**2**

$resultado$

x	y	z	w
1	**1**	1	7
1	**1**	2	7
2	**2**	3	4

Junção Natural (*Natural Join*)

- Exemplo: $(R_1 \bowtie R_2)$

R_1

x	y	z
1	**1**	1
1	**1**	2
2	**2**	3

R_2

x	y	w
1	**1**	3
2	**4**	2

$resultado$

x	y	z	w
1	**1**	1	3
1	**1**	2	3

- Exemplo: $(R_1 \bowtie R_2) = (R_1 \times R_2)$

R_1

x	y	z
1	**1**	1
1	**2**	5

R_2

w	t
7	**1**
4	**2**

$resultado$

x	y	z	w	t
1	1	1	7	1
1	1	1	4	2
1	2	5	7	1
1	**2**	5	4	2

Junções Externas (*Outer Joins*)

- Junção em que as tuplas de uma ou ambas as relações que não são combinadas são mesmo assim preservadas
- Tipos:
  - Junção Externa à Esquerda (*left [outer] join*)
    - apenas tuplas da relação à esquerda são preservadas
    - Notação: $relacao_1 \ ⟕_{(condição)} \ relacao_2$
  - *J*unção Externa à Direita (*right [outer] join*)
    - apenas tuplas da relação à direita são preservadas
    - Notação: $relacao_1 \ ⟖_{(condição)} \ relacao_2$
  - *J*unção Externa Completa (*full [outer] join*)
    - tuplas de ambas as relações são preservadas
    - Notação: $relacao_1 \ ⟗_{(condição)} \ relacao_2$

Junções Externas (*Outer Joins*)

- Exemplos:

$(R_1 \bowtie_{(R1.x = R2.x)} R_2)$

x	y	z	a	b
1	1	1	7	3
2	1	2	4	2
3	3	3	-	-
5	5	5	-	-

$(R_1 \bowtie R_2)$

x	y	z	a	b
1	1	1	7	3
2	1	2	4	2
3	3	3	-	-
5	5	5	-	-
4	-	-	4	4

$(R_1 \bowtie R_2)$

x	y	z	a	b
1	1	1	7	3
2	1	2	4	2
4	-	-	4	4

UDESC

Semi-Junção (*Semi Join*)

- Versão de junção semelhante à Junção Natural. Difere apenas no fato de que são preservadas apenas as colunas da relação à esquerda (*left semi join*) ou da relação à direita (*right semi join*)

- Notação: $relacao_1 \ltimes relacao_2$

  $relacao_1 \rtimes relacao_2$

Semi-Junção (*Semi Join*)

- Exemplos:

Nome	Id	Dept
Alfredo	3415	Finanças
Beto	2241	Vendas
Carla	3401	Finanças
Djenifer	2202	Produção

Dept	Gerente
Produção	Zelia
Vendas	Yury
Admin	Willy

$$(R_1 \propto R_2)$$

Nome	Id	Depto
Beto	2241	Vendas
Djenifer	2202	Produção

$$(R_1 \propto R_2)$$

Dept	Gerente
Produção	Zelia
Vendas	Yury

Anti-Junção (*Anti Join*)

- Junção similar a Junção Natural, mas preservam-se apenas as tuplas da relação da esquerda que possuem valor(es) na(s) coluna(s) comum(s) que não aparecem na relação da direita.

- Notação: $relacao_1 \triangleright relacao_2$
- Pode ser escrito como:

$$\mathrm{R} \triangleright S = R - (R \propto S)$$

Anti-Junção (*Anti Join*)

- Exemplos:

Nome	Id	Dept
Alfredo	3415	Finanças
Beto	2241	Vendas
Carla	3401	Finanças
Djenifer	2202	Produção

Dept	Gerente
Produção	Zelia
Vendas	Yury
Admin	Willy

$(R_1 \triangleright R_2)$

Nome	Id	Depto
Alfredo	3415	Finanças
Carla	3401	Finanças

$(R_2 \triangleright R_1)$

Dept	Gerente
Admin	Willy

Ordenar (*Order by*)

- Ordena os resultados baseado em uma ou mais colunas em ordem crescente ou decrescente

tau

- Notação: $\tau_{\text{[coluna ordem]}}(relacao_1)$
- Ordem pode assumir valores: asc, desc
- A lista de *coluna ordem* deve ser separada por vírgula

R_1

x	y	z
1	2	a
4	1	b
2	3	c
2	5	c

$\tau_{x\ desc, y\ asc}(R_1)$

Resultado

x	y	z
4	1	b
2	3	c
2	5	c
1	2	a

Agrupar (*Group by*)

- Agrupa os resultados e permite a realização de alguma função sobre o agrupamento (contagem, soma, ...)

gamma

- Notação: $\gamma_{\text{[função(coluna) -> novo\_nome]}}(relacao_1)$
- Função pode ser: count( ), sum( ), avg( ), min( ), max( )
- A lista de agrupamentos deve ser separada por vírgula

R_1

x	y	z
1	2	a
4	1	b
2	3	c
2	5	c

$\gamma_{count(x)->qtd,\ sum(y)->somaY}(R_1)$

Resultado

qtd	somaY
4	11

UDESC

Divisão

- Operação entre duas relações
  - Dividendo (grau m + n)
  - Divisor (grau n)
- Grau “n”: atributos de mesmo nome nas relações
- Grau “m” ou quociente: atributos da relação dividendo cujos valores associam-se com TODOS os valores da relação divisor

- Notação: $relacao_1 \div relacao_2$

OBS: No RELAX, procure deixar as colunas em comum por último.

Divisão

- Exemplos:

R_1

x	y	z
1	1	1
1	2	1
2	1	1
2	2	2
3	1	3

R_2

z
1

R_3

y	z
1	1

R_4

y
1
2

$R_1 \div R_2$

x	y
1	1
1	2
2	1

$R_1 \div R_3$

x
1
2

$R_1 \div R_4$

x	z
1	1

OBS: No RELAX, ocorre problema ao usar uma coluna intermediária (ex: R_4:y). O sistema utiliza associando a última coluna (R_1:z)

Atualização de Relações

- Exclusão
  - Notação: *relação <=== relação – expressãoConsulta*

    relação <=== expressãoConsulta
- Inclusão
  - Notação: relação <=== relação ∪ Expr

    onde Expr é um conjunto de tuplas
- Alteração
  - Notação: $\delta_{\{nome\_atributo <=== Expr\}}(relação)$

    (delta) onde Expr é uma expressão aritmética ou um valor constante

Atualização de Relações

Exemplos:

R_1

x	y	z
1	1	1
2	1	3

R_2

w	t	v
1	3	1
2	2	2
3	2	3

a) $R_1 = R_1 - (\sigma_{x=1} R_1)$

b) $R_2 = (\sigma_{t=2} R_2)$

c) $R_1 = R_1 \cup \{(1,2,2),(1,2,3)\}$

d) $\text{temp} = \pi_w (\sigma_{t=2} R2)$

$R_1 = R_1 \cup (\text{temp} \times \{(3,3)\})$

e) $\delta_{x <-- x+1} R_1$

f) $temp = \sigma_{t=2} R_2$

$R_2 = R_2 - temp$

$\delta_{w <-- w-1} temp$

$R_2 = R_2 \cup temp$

UDESC

Cálculo Relacional

UDESC

Cálculo Relacional

Cálculo Relacional de Tupla

- Forma Geral

$$\{t, v, ..., x \mid P(t, v, ..., x)\}$$

variáveis livres

predicado aplicado à *t, v, ..., x*

- Variável livre
  - assume valores de tuplas de uma ou mais relações
  - participa da resposta da consulta
- Predicado
  - expressão lógica que, se verdadeira para uma associação de valores das tuplas atribuídas a *t, v, ...x*, retorna os valores destas variáveis na resposta da consulta

UDESC

Cálculo Relacional

Exemplos (Seleção e Projeção)

- buscar os dados dos pacientes que estão com sarampo

  $\{p \mid p \in Pacientes \wedge p.doença = \text{'sarampo'}\}$

- buscar os dados das consultas, exceto aquelas marcadas para os médicos com código 46 e 79

  $\{c \mid c \in Consultas \wedge \neg (c.codm = 46 \vee c.codm = 79)\}$

- buscar o número e a capacidade dos ambulatórios do terceiro andar

  $\{a.nroa, a.capacidade \mid a \in Ambulatórios \wedge a.andar = 3\}$

UDESC

Cálculo Relacional

Exemplos (Produto ou Junção)

- buscar o nome dos médicos que têm consulta marcada e as datas das suas consultas

  {m.nome, c.data | m ∈ Médicos ∧ c ∈ Consultas ∧ m.codm = c.codm}

- buscar os nomes dos médicos ortopedistas e o número e andar dos ambulatórios onde eles atendem

  {m.nome, a,nroa, a.andar | m ∈ Médicos ∧ m.especialidade = 'ortopedia' ∧ a ∈ Ambulatórios ∧ m.nroa = a.nroa}

UDESC

Cálculo Relacional
Quantificador Existencial

Exemplos

- buscar o nome dos médicos que atendem em ambulatórios do segundo andar

$\{m.nome \mid m \in Médicos \wedge \exists\ a \in \textbf{Ambulatórios}\ (a.andar = 2 \wedge m.nroa = a.nroa)\}$

- buscar o nome e a doença dos pacientes que têm consulta marcada com o médico João da Silva

$\{p.nome, p.doença \mid p \in Pacientes \wedge \exists\ c \in \textbf{Consultas}\ (p.codp = c.codp \wedge \exists\ m \in \textbf{Médicos}\ (c.codm = m.codm \wedge c.nome = \text{'João da Silva'}))\}$

UDESC

Cálculo Relacional
Quantificador Universal

Exemplos

- buscar o nome dos médicos que têm consulta marcada com todos os pacientes

$\{m.nome \mid m \in Médicos \wedge \forall\ p \in \mathbf{Pacientes}\ (\exists\ c \in Consultas\ (p.codp = c.codp \wedge c.codm = m.codm))\}$

- buscar o nome dos pacientes que têm consulta marcada com todos os médicos ortopedistas

$\{p.nome \mid p \in Pacientes \wedge \forall\ m \in \mathbf{Médicos}\ (\mathbf{m.especialidade = 'ortopedia'} \Rightarrow \exists\ c \in Consultas\ (c.codm = m.codm \wedge c.codp = p.codp))\}$

PostgreSQL

SQL – Structured Query Language

Microsoft® SQL Server®

Criando Tabelas

- Para definir uma nova tabela e sua estrutura:

```
CREATE TABLE nome_tabela(
    nome_atributo_1 tipo_1 [*RESTRIÇÕES*],
    [{nome_atributo_n tipo_n [*RESTRIÇÕES*]},]
    [PRIMARY KEY (nome(s)_atributo(s)),]
    [FOREIGN KEY (nome_atributo)
       REFERENCES nome_tabela (nome_atributo)]
  );
```

Criando Tabelas

Exemplo

```
CREATE TABLE Cursos(
    id INT,
    nome VARCHAR(30) NOT NULL,
    PRIMARY KEY (id)
);
```

```
CREATE TABLE Alunos(
    matricula INT,
    nome VARCHAR(50) NOT NULL,
    curso_id INT,
    PRIMARY KEY (matricula),
    FOREIGN KEY (curso_id)
        REFERENCES Cursos (id)
  );
```

UDESC

Criando Tabelas

- Principais tipos de dados:
  - Numéricos: englobam números inteiros de vários tamanhos e pontos flutuantes (ver tabela a seguir)
  - Cadeia de Caracteres: podem ser de tamanho fixo [`char(n)`] ou variável [`varchar(n)`, `text`]
  - Booleano: assume valores TRUE e FALSO, mas pode ser também um valor NULL
  - Date: formato `aaaa-mm-dd`
  - Time: formato `hh:mm:ss`
  - Timestamp: formado por data e hora

UDESC

Criando Tabelas

- Detalhes sobre dados numéricos:

nome	tamanho armazenamento	faixa de valores
tinyint	1 bytes	-128 a 127
smallint	2 bytes	-32.768 a +32.767
int	4 bytes	-2.147.483.648 a +2.147.483.647
bigint	8 bytes	-9.223.372.036.854.775.808 a ...
decimal	Variável	Sem limite
numeric	Variável	Sem limite
real	4 bytes	Precisão de 6 dígitos decimais
double precision	8 bytes	Precisão de 15 dígitos decimais

Criando Tabelas

- É possível utilizar atributos opcionais para dados numéricos:
  - `UNSIGNED`: indica que um tipo numérico não terá sinal e por consequência a faixa de valores é alterada
  - `ZEROFILL`: indica que os espaços vazios do campo serão preenchidos com zeros [PostgreSQL não implementa]
  - `(n)`: É possível especificar o "tamanho de apresentação", ex: `int`(5) => 00032 (OBS: faixa de valores não muda)
  - `(i,d)`: para os tipos `decimal` e `numeric` é possível especificar o número de dígitos totais e decimais, ex: `decimal`(6,2) => -9.999,99 até +9.999,99

Criando Tabelas

- Restrições / Atributos Opcionais
  - `NOT NULL`: indica que o campo não pode ser nulo, ou seja, precisa necessariamente ter um valor
  - `UNIQUE`: indica que um campo deve ser único para cada registro / linha da tabela (não pode haver duplicatas
  - `AUTO_INCREMENT`: indica que será utilizado um valor incrementável automaticamente para o campo

    OBS: no PostgreSQL não há está restrição, mas sim um tipo de dado serial que possui este comportamento

https://www.postgresql.org/docs/9.4/static/ddl-constraints.html

Criando Tabelas

- Restrições / Atributos Opcionais
  - `DEFAULT valor`: indica um valor padrão fixo para o campo caso um valor não seja especificado
  - `CHECK (predicado)`: indica que deverá ser realizar verificação de integridade de acordo com o predicado;
  - `PRIMARY KEY`: indica que um campo é chave primária;
  - `REFERENCES tabela (campo)`: indica que um campo é chave estrangeira

OBS: estas restrições podem ser definidas também no final

https://www.postgresql.org/docs/9.4/static/ddl-constraints.html

Criando Tabelas

- Exemplos com Restrições

```
CREATE TABLE Materiais(
     id SERIAL PRIMARY KEY,
     nome VARCHAR(50) UNIQUE NOT NULL,
     preco NUMERIC(6,2)
);
```

```
CREATE TABLE Disciplinas(
       sigla CHAR(3) PRIMARY KEY,
       curso_id INT REFERENCES cursos (id),
       ch SMALLINT
);
```

UDESC

Criando Tabelas

- Exemplos com Restrições (forma alternativa)

```
CREATE TABLE Materiais(
     id SERIAL,
     nome VARCHAR(50) NOT NULL,
     preco NUMERIC(6,2) CHECK (preco > 0),
     preco_vista NUMERIC(6,2),
     PRIMARY KEY (id),
--   CONSTRAINT pk PRIMARY KEY (id),
     UNIQUE (nome),
--   CONSTRAINT nome_diferente UNIQUE (nome),
     CHECK (preco_vista <= preco)
--   CONSTRAINT avista CHECK (preco_vista <= preco)
);
```

Criando Tabelas

- Exemplos com Restrições (forma alternativa)

```
CREATE TABLE Disciplinas(
        sigla CHAR(3),
        curso_id INT,
        ch SMALLINT,
        PRIMARY KEY (sigla),
        CONSTRAINT fk FOREIGN KEY (curso_id)
           REFERENCES cursos (id)
);
```

➢ Múltiplas chaves primárias, ou campos únicos podem ser definidos separando os nomes das colunas com vírgula

Criando Tabelas

- Para as `FOREIGN KEYS`, pode-se especificar o que deve ser feito caso uma operação sobre suas referencias seja realizada:
  - Definir qual a operação considerada:
    - `ON DELETE ; ON UPDATE`
  - Definir o comportamento:
    - NO ACTION: padrão, previne a alteração do registro se este estiver referenciado por outro registro.
    - RESTRICT: similar ao NO ACTION
    - SET NULL / SET DEFAULT: altera os registros que estiverem o referenciando para um valor nulo ou padrão;
    - CASCADE: propaga a alteração ou remoção aos registros que o estiverem referenciando

Criando Tabelas

- ***Storage Engine*** **ou** ***Engine****:* são componentes do SGBD que são responsáveis pela realização das operações SQL

- PostgreSQL suporta e utiliza apenas um único e*ngine* que recebe o seu nome: PostgreSQL
- MySQL suporta e possibilita o uso de diversos *engines*, entre eles: InnoDB, MyISAM, MEMORY, CSV, ARCHIVE, BLACKHOLE, MERGE, FEDERATED, EXAMPLE

```
CREATE TABLE Disciplinas(
      [...]
) engine=MyISAM;
```

```
CREATE TABLE Disciplinas(
      [...]
) engine=InnoDB;
```

UDESC

Criando Tabelas

- No SQL os *engines* mais comuns são MyISAM e InnoDB:
  - *MyISAM*: modelo simples e eficiente
    - ✓ **contras**: não garante restrições de integridade referenciais
    - ✓ **prós**: simples e bom desempenho;
      bom para muitas leituras e poucas escritas;
      bom quando o controle de restrições está no código
  - *InnoDB:* modelo mais robusto
    - ✓ **prós**: garante restrições de integridade referenciais;
      bom para escritas concorrentes;
    - ✓ **contras**: um pouco mais lento do que o MyISAM

Apagando Tabelas

- Para excluir uma tabela existente pode-se utilizar:

```
DROP TABLE nome_tabela;
```

Alterando Tabelas

- Existem diversas alterações possíveis que podem ser feitas em uma tabela já existente:

```
ALTER TABLE nome_tabela
  RENAME TO novo_nome
  SET SCHEMA new_schema
  ADD [COLUMN] nome tipo [{Ris}]
  RENAME [COLUMN] nome_antigo TO nome_novo
  DROP [COLUMN] nome
  ALTER [COLUMN] nome [{SET|DROP} DEFAULT
      | {SET|DROP} NOT NULL | SET DATA TYPE tipo|...]
  ADD CONSTRAINT ri [PK|FK|CHECK|...]
  DROP CONSTRAINT nome_ri
  ...
```

https://www.postgresql.org/docs/9.1/static/sql-altertable.html

Alterando Tabelas

- Alguns exemplos

```
ALTER TABLE alunos
  RENAME TO graduandos

ALTER TABLE graduandos
  SET SCHEMA public

ALTER TABLE graduandos
  ADD COLUMN dt_nasc
  DATE NOT NULL

ALTER TABLE graduandos
  RENAME COLUMN dt_nasc
  TO data_nascimento
```

```
ALTER TABLE graduandos
  DROP COLUMN data_nascimento

ALTER TABLE graduandos
  ADD CONSTRAINT nroMat
  CHECK (matricula > 10000)

ALTER TABLE graduandos
  DROP CONSTRAINT nroMat

ALTER TABLE disciplinas
ALTER COLUMN ch
  SET DEFAULT (32)
```

Criando Índices

- **Índices** são utilizados para acelerar consultas a dados (assim como ocorre em um livro). Entretanto, índices ocupam espaço e devem ser utilizados com parcimônia
  - Índices são definidos automaticamente para PKs

```
CREATE [UNIQUE] INDEX nome_indice
  ON nome_tabela (nome_atrib [{, nome_atrib_n}])

DROP INDEX nome_indice ON nome_tabela
```

UDESC

Criando Sequencias

- O PostgreSQL utiliza **sequências** para valores do tipo serial (auto incrementáveis). As sequências podem ser alteradas com os seguintes comandos

```
ALTER SEQUENCE nome_sequencia
  INCREMENT BY valor -- escolhe o valor de inc.
  RESTART WITH valor -- reinicia com um valor
  MINVALUE valor -- define o valor mínimo
  MAXVALUE valor -- define o valor máximo
```

Inserção de Registros

- Para inserir valores podem-se utilizar dois modelos:
  - ❑ Apresentando todos os valores explicitamente:

```
INSERT INTO nome_tabela
  VALUES ( valores_separados_por_virgula )
```

  - ❑ Identificando quais colunas estão sendo listadas (demais colunas recebem valor padrão ou valor nulo)

```
INSERT INTO nome_tabela ( lista_colunas )
  VALUES ( valores_na_mesma_ordem )
```

OBS: caso algum campo com restrição não nulo não apareça na operação, o SGBD acusará um erro

Inserção de Registros

- Exemplos:

```
INSERT INTO materiais
   VALUES (1, 'livro BAN', 75.00, 70.00);
```

```
INSERT INTO materiais (nome, preco, preco_vista)
  VALUES ('teste', 125.50, 100.00);
```

OBS: caso o campo de sequencia não tenha sido atualizado, a primeira tentativa resultará em erro pois o SGBD tentará utilizar id=1, mas este valor já foi utilizado

```
INSERT INTO materiais (nome, preco, preco_vista)
  VALUES ('mouse', 45.00, 39.99),
         ('teclado', 75.50, 65.00);
```

OBS: nem todo SGBD permite a inserção de múltiplos registros em uma única cláusula

Inserção de Registros

- Exemplos das Restrições de Integridade na inserção:

OBS: o campo nome é obrigatório (NOT NULL). Vamos adicionar um novo registro sem nome e ver o que acontece

```
INSERT INTO materiais (preco, preco_vista)
  VALUES (99.90, 95.00);
```

OBS: a coluna nome foi marcada como UNIQUE. Vamos adicionar um novo registro com o mesmo nome e ver o que acontece

```
INSERT INTO materiais (nome, preco, preco_vista)
  VALUES ('teste', 175.50, 150.00);
```

OBS: preco_vista tem uma restrição assertiva (check) que verifica se este valor é menor do que o preço. E se não for, o que acontece?

```
INSERT INTO materiais (nome, preco, preco_vista)
  VALUES ('caneta atômica', 4.50, 5.00);
```

Alteração/Atualização de Registros

- Para alterar / atualizar registros utiliza-se:

```
UPDATE nome_tabela
   SET nome_atributo = valor
       [{, nome_atributo_n = valor}]
   [WHERE condição]
```

CUIDADO: caso um condicional WHERE não for definido, todos os registros da tabela serão alterados!!!

- Exemplos:

```
UPDATE materiais
 SET valor_vista = 65.00
 WHERE id = 1;
```

```
UPDATE materiais
 SET nome = 'livro EDA',
  preco_vista = preco * 0.9
 WHERE id = 2;
```

Exclusão de Registros

- Para excluir registros utiliza-se:

```
DELETE FROM nome_tabela
   [WHERE condição]
```

CUIDADO: caso um condicional WHERE não for definido, todos os registros da tabela serão excluidos!!!

- Exemplos:

```
DELETE FROM Ambulatorios
```

```
DELETE FROM materiais
WHERE id = 1
```

```
DELETE FROM materiais
WHERE nome = 'livro EDA'
   OR nome = 'livro BAN'
```

Operadores SQL

- Para as expressões condicionais é possível utilizar os seguintes operadores:
  - Operadores de comparação

    `= ; <> ; != ; > ; >= ; < ; <=`
  - Operadores lógicos

    `and; or ; not`

Consultas Básicas

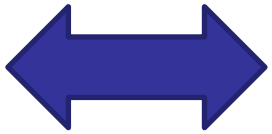
- Para consultar dados de uma tabela utiliza-se:

```
SELECT lista_de_colunas
FROM tabela
[WHERE condição]
```

  - *lista_de_colunas* pode ser substituida por asterisco (*) que representa todos os atributos da tabela

- Mapeamento entre SQL e álgebra relacional:

```
SELECT a1, ..., an
FROM t
WHERE c
```



$$\pi_{a_1,\ldots,an}(\sigma_c(t))$$

Consultas Básicas

- Exemplos:

Álgebra Relacional	SQL
$(Pacientes)$	`SELECT * FROM pacientes`
$\sigma_{idade>18}(pacientes)$	`SELECT * FROM pacientes WHERE idade > 18`
$\pi_{cpf,nome}(t)$	`SELECT cpf, nome FROM pacientes`
$\pi_{cpf,nome}(\sigma_{idade>18}(pacientes))$	`SELECT cpf, nome FROM pacientes WHERE idade > 18`

Consultas Básicas
Particularidade na projeção

- Não há eliminação automática de duplicatas
  - Tabela ≡ coleção
  - Para eliminar duplicatas deve-se usar o termo `distinct`

```
SELECT DISTINCT doenca FROM consultas
```

- É possível renomear os campos / colunas (`AS`)
  - Operador ρ (*rho*) na álgebra relacional

```
SELECT codp AS codigo_paciente,
codm AS codigo_medico, data
FROM consultas
```

Consultas Básicas
Particularidade na projeção

- É possível utilizar operadores aritméticos e funções
  - Quantos grupos de 5 leitos podem ser formados em cada ambulatório?

```
SELECT nroa, capacidade/5 AS cap5
FROM ambulatorios
```

  - Qual o salário líquido dos funcionários sabendo que há um desconto de 12,63% sobre o salário base?

```
SELECT nome, ROUND( salario * 0.8737 , 2) AS
    salario_liquido FROM funcionarios
```

Função **ROUND**: parâmetros (valor : numeric , casas : int | numeric)

Consultas Básicas
Particularidade na projeção

- Pode-se usar funções de agregação / agrupamento
  - ➢ COUNT: contador de ocorrências [registros ou atributos]

OBS: Conta valores não nulos

```
SELECT COUNT(*) FROM medicos
WHERE especialidade = 'ortopedia'
```

  - ➢ MAX / MIN: valor máximo / mínimo de um atributo

```
SELECT MAX(salario) AS maior_salario FROM funcionarios
```

  - ➢ SUM: soma os valores de um dado atributo
    - ❖ Qual é o gasto total com a folha de pagamento dos funcionários?
  - ➢ AVG: contabiliza a média dos valores de um dado atributo
    - ❖ Qual é a média de idade dos funcionários de Florianópolis?

Consultas Básicas
Particularidade na projeção

- Pode-se misturar funções de agregação com `distinct`

```
SELECT COUNT(DISTINCT especialidade)
FROM medicos
```

- Não podem ser combinados outros campos junto com funções de agregação

```
SELECT andar, COUNT(andar)
FROM ambulatorios
```

- Pode-se realizar *casting* de tipos usando: `campo::tipo`

```
SELECT nome, cpf::text, idade::numeric(3,1)
FROM pacientes
```

UDESC

Consultas Básicas
Particularidade na seleção

- Procurar por valores nulos ou não nulos
  - cláusula `IS [NOT] NULL`

```
SELECT cpf, nome
FROM medicos WHERE nroa IS NULL
```

- Procurar por intervalos de valores
  - Cláusula `[NOT] BETWEEN valor1 AND valor2`

```
SELECT * FROM consultas
WHERE hora BETWEEN '13:00' AND '18:00'
```

Consultas Básicas
Particularidade na seleção

- Procurar por pertinência em conjunto ou coleção
  - cláusula `[NOT] IN`

```
SELECT * FROM medicos
WHERE especialidade IN
    ('ortopedia', 'traumatologia')
```

```
SELECT codm, codp, data FROM consultas
WHERE codm IN (
    SELECT codm FROM medicos
    WHERE idade > 40
)
```

Consultas Básicas
Particularidade na seleção

- Procurar por padrões
  - Cláusula `[NOT] LIKE`
  - Pode receber os seguintes padrões

    `%` casa com qualquer cadeia de caracteres

    `_` casa com um único caractere

    [a-d] casa com qualquer caractere entre as letras apresentadas (SQL-Server)

❖ Buscar médicos que nome iniciando por ‘M’

❖ Buscar médicos com um número exato de décadas de vida

```
SELECT * FROM medicos
WHERE nome LIKE 'M%'
```

```
SELECT * FROM medicos
WHERE idade::text LIKE '_0'
```

UDESC

Consultas Básicas
Particularidade na seleção

- Procurar por padrões

  ❖ Buscar por consultas marcadas para o mês de julho

```
SELECT * FROM consultas
WHERE data::text LIKE '%/07/%'
```

  ❖ Buscar por pacientes cujo CPF termina com 20000 ou 30000

```
SELECT * FROM pacientes
WHERE cpf::text LIKE '%20000'
   OR cpf::text LIKE '%30000'
```

União de Tabelas

- SQL implementa a união da álgebra relacional
  - Lembre-se, as tabelas devem ser compatíveis!

Álgebra Relacional

```
relação-1   ∪   relação-2
```

SQL

```
SQL-1   UNION   SQL-2
```

❖ Buscar o nome e o CPF dos médicos e funcionários

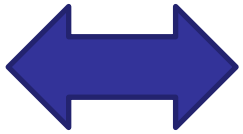
```
SELECT cpf, nome FROM medicos
  UNION
SELECT cpf, nome FROM pacientes
```

Consultas Envolvendo Múltiplas Tabelas

- SQL implementa a operação de **PRODUTO CARTESIANO**
  - Relaciona todos os registros de uma tabela com todos os registros de outra tabela
  - Geralmente é necessário vincular as duas tabelas através de uma condição

- Mapeamento entre SQL e álgebra relacional:

```
SELECT a1, ..., an
FROM t1, ..., tm
WHERE c
```



$$\pi_{a_1,\ldots,an}(\sigma_c (t_1 \times \cdots \times t_m))$$

Consultas Envolvendo Múltiplas Tabelas

- Trazer o CPF e nome dos pacientes e suas respectivas datas de consultas agendadas para o período da tarde

```
SELECT cpf, nome, data
FROM pacientes, consultas
WHERE hora > '12:00' AND
  pacientes.codp = consultas.codp
```

- Trazer os nomes dos médicos que tem a mesma especialidade do médico de nome ‘João’
  [neste caso é necessário renomear uma tabela]

OBS: Quando as colunas são iguais é necessário usar o nome da tabela como prefixo. Assim, é comum renomear as tabelas com nomes mais enxutos

Consultas Envolvendo Múltiplas Tabelas

- SQL implementa as operações de **JUNÇÃO**

```
SELECT a1, ..., an
FROM t1 [INNER] JOIN t2
  ON condição_junção
WHERE c
```

❖ Trazer o CPF e nome dos pacientes e suas respectivas datas de consultas agendadas para o período da tarde

```
SELECT p.cpf, p.nome, c.data
FROM pacientes AS p JOIN consultas AS c
  ON p.codp = c.codp
WHERE hora > '12:00'
```

Consultas Envolvendo Múltiplas Tabelas

- **JUNÇÃO NATURAL**
  - Junção utilizando as colunas que possuem mesmo nome entre as tabelas relacionadas. Assim, a condição não é declarada

```
SELECT a1, ..., an
FROM t1 NATURAL JOIN t2
WHERE c
```

❖ Trazer o CPF e nome dos médicos e suas respectivas datas de consultas agendadas para o período da manhã

```
SELECT m.cpf, m.nome, c.data
FROM medicos AS m NATURAL JOIN consultas AS c
WHERE hora <= '12:00'
```

Consultas Envolvendo Múltiplas Tabelas

- **JUNÇÕES EXTERNAS**
  - Junções que mantêm os elementos (linhas) não relacionados de uma ou mais tabelas no resultado

```
SELECT a1, ..., an
FROM t1 LEFT|RIGHT|FULL [OUTER] JOIN t2
  ON condição_junção
WHERE c
```

❖ Listar todos os pacientes e, caso existam, as datas e horários de suas consultas.

```
SELECT p.cpf, p.nome, c.data
FROM pacientes AS p LEFT JOIN consultas AS c
   ON p.codp = c.cod
```

Consultas Envolvendo Múltiplas Tabelas

- **JUNÇÕES EXTERNAS**

❖ Listar todos os médicos e funcionários (nome e cidade), vinculando aqueles que moram em uma mesma cidade

```
SELECT f.nome, f.cidade, m.nome, m.cidade
FROM funcionarios AS f FULL JOIN medicos AS m
   ON f.cidade = m.cidade
```

Consultas Aninhadas

- Já vimos que é possível realizar consultas alinhadas usando a cláusula [NOT] IN

```
SELECT codm, codp, data FROM consultas
WHERE codm IN (
    SELECT codm FROM medicos
    WHERE idade > 40
)
```

  - É possível realizar operações de diferença e interseção

$\pi_{CPF}(Funcionarios) - \pi_{CPF}(Pacientes)$

```
SELECT cpf FROM funcionarios
WHERE cpf NOT IN (
    SELECT cpf FROM pacientes
)
```

$\pi_{CPF}(Medicos) \cap \pi_{CPF}(Pacientes)$

```
SELECT cpf FROM medicos
WHERE cpf NOT IN (
    SELECT cpf
    FROM pacientes
)
```

Consultas Aninhadas

- Pode-se fazer uso também de **Subconsultas unitárias**
  - Cardinalidade da subconsulta = 1
  - Neste caso não é necessário utilizar cláusula de subconsulta

❖Buscar o nome e CPF dos médicos que possuem a mesma especialidade do que o médico de CPF 10000100000

```
SELECT nome, cpf FROM medicos
WHERE cpf != 10000100000 AND
   especialidade = (
      SELECT especialidade FROM medicos
      WHERE cpf = 10000100000
)
```

Consultas Aninhadas

- Existem ainda as cláusulas ANY , ALL e EXISTS (Cálculo Relacional)

  – **ANY**: testa se uma dada condição é verdadeira para pelo menos um valor da consulta aninhada

  ❖Buscar o nome e a idade dos médicos que são mais velhos do que pelo menos um funcionário

```
SELECT nome, idade FROM medicos
WHERE idade > ANY (
    SELECT idade FROM funcionarios
)
```

Consultas Aninhadas

- Existem ainda as cláusulas ANY , ALL e EXISTS (Cálculo Relacional)

  – **ALL**: testa se uma dada condição é verdadeira para todos os valores de uma consulta aninhada

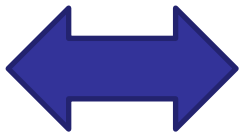
❖Buscar o nome e a idade dos médicos que são mais velhos do que todos os funcionários de Florianópolis

```
SELECT nome, idade FROM medicos
WHERE idade > ALL (
    SELECT idade FROM funcionarios
    WHERE cidade = 'Florianopolis'
)
```

Consultas Aninhadas

- Existem ainda as cláusulas ANY , ALL e EXISTS (Cálculo Relacional)
  - **EXISTS**: testa se um predicado é verdadeiro ou falso; a subconsulta é executada para cada linha da consulta externa
- Mapeamento entre SQL e álgebra relacional:

```
SELECT a1, ..., an
FROM t1
WHERE EXISTS
  (SELECT * FROM t2
   WHERE d > 5 AND
   t2.x = t1.c)
```



$$\{t_1.a_1, \ldots, t_1.an \mid t_1 \in T_1 \wedge \exists\, t_2 \in T_2 \; (t_2.d > 5 \wedge t_2.x = t_1.c)\}$$

Consultas Aninhadas

- Existem ainda as cláusulas ANY , ALL e EXISTS (Cálculo Relacional)

  – **EXISTS**: testa se um predicado é verdadeiro ou falso; a subconsulta é executada para cada linha da consulta externa

❖Buscar o nome dos médicos que possuem uma consulta para o dia 06 de novembro de 2013

$$\{ m.nome \mid m \in medicos \wedge \exists c \in consultas (c.data = \text{'2013/11/06'} \wedge c.codm = m.codm)\}$$

```
SELECT nome FROM medicos AS m
WHERE EXISTS (
    SELECT * FROM consultas
    WHERE data = '2013/11/06'
      AND codm = m.codm )
```

Consultas Aninhadas

- Existem ainda as cláusulas ANY , ALL e EXISTS (Cálculo Relacional)

  – **EXISTS**: testa se um predicado é verdadeiro ou falso; a subconsulta é executada para cada linha da consulta externa

❖Buscar o nome dos funcionários de Florianópolis que nunca se consultaram como pacientes na clínica

$$\{ f.nome \mid f \in funcionarios \wedge f.cidade = \text{'Florianopolis'} \neg\exists\, p \in pacientes (p.cpf = f.cpf)\}$$

?

Consultas Aninhadas

- Subconsulta na cláusula **FROM / JOIN**
  - Consulta externa é feita sobre um subconjunto resposta
  - Útil para otimização filtrando linhas e colunas antecipadamente

❖Buscar os dados dos médicos e a hora das consultas que estão agendadas para o dia 06/11/2013

```
SELECT medicos.*, c.hora
FROM medicos JOIN
   (SELECT codm, hora FROM consultas
    WHERE data = '2013/11/06')
AS c ON medicos.codm = c.codm
```

Consultas Aninhadas

- Subconsulta na cláusula **FROM / JOIN**
  - Consulta externa é feita sobre um subconjunto resposta
  - Útil para otimização filtrando linhas e colunas antecipadamente

❖Buscar os números e andares dos ambulatórios em que médicos de Florianopolis dão atendimento

```
SELECT amb.* FROM
   (SELECT nroa, andar FROM ambulatorios)
AS amb JOIN
   (SELECT nroa FROM medicos
    WHERE cidade = 'Florianopolis')
AS mflo ON amb.nroa = mflo.nroa
```

Ordenar Tuplas Resultantes

- Resultados podem ser ordenadas pela cláusula **ORDER BY**

```
SELECT lista_atributos
FROM lista_tabelas
[WHERE condições]
[ORDER BY nome_atrib_1 [ASC|DESC]
  {[, nome_atrib_n [ASC|DESC]]}
```

- Mapeamento entre SQL e álgebra relacional:

$$\tau_{idade\ asc,\ nome\ desc}(\pi_{nome,idade}(\ Funcionarios\ ))$$

```
SELECT nome, idade FROM pacientes
ORDER BY idade ASC, nome DESC
```

Limitar Tuplas Resultantes

- Resultados podem ser limitados pela cláusula **LIMIT**

```
SELECT lista_atributos
FROM lista_tabelas
[WHERE condições]
[ORDER BY regras]
[LIMIT v1 [,v2]]
```

```
SELECT lista_atributos
FROM lista_tabelas
[WHERE condições]
[ORDER BY regras]
[LIMIT qtd OFFSET ini]
```

- Se apenas v_1 é utilizado ele representa o número de tuplas
- Se v_1 e v_2 forem utilizados, v_1 representa quantos registros iniciais devem ser pulados e v_2 representa o número de tuplas

- No **PostgreSQL** deve-se utilizar **LIMIT** para indicar o número de tuplas e **OFFSET** para indicar quantos registros iniciais devem ser pulados.

Limitar Tuplas Resultantes

- Resultados podem ser limitados pela cláusula **LIMIT**

  ❖ Retornar o nome e a idade dos 3 pacientes mais velhos

```
SELECT nome, idade FROM pacientes
ORDER BY idade DESC LIMIT 3
```

  ❖ Retornar o nome, a idade e o salario dos funcionários que recebem o segundo e terceiro maior salário

```
SELECT nome, salario
FROM funcionarios
ORDER BY salario DESC
LIMIT 1, 2
```

```
SELECT nome, salario
FROM funcionarios
ORDER BY salario DESC
LIMIT 2 OFFSET 1
```

UDESC

Agrupar Tuplas

- Tuplas podem ser agrupados pela cláusula **GROUP BY**
  - Agrupa as tuplas que possuem mesmo valor nas colunas especificadas para o agrupamento
  - Apenas os atributos de agrupamento podem aparecer no resultado final da consulta
  - Geralmente é utilizada alguma **função de agregação** (ex: contagem, somatório) sobre o resultado da consulta

```
SELECT lista_atributos
FROM lista_tabelas
[WHERE condições]
[GROUP BY lista_atributos_agrupamento
  [HAVING condição_para_agrupamento]]
```

Agrupar Tuplas

- Tuplas podem ser agrupados pela cláusula **GROUP BY**
  - Funções de agregação: COUNT, SUM, AVG, MIN, MAX

❖ Listar quantos médicos existem por especialidade

```
SELECT especialidade, COUNT(*)
FROM medicos
GROUP BY especialidade
```

❖ Qual é a média de salários pagos aos funcionários por sua cidade de origem?

```
SELECT cidade, AVG(salario)
FROM funcionarios
GROUP BY cidade
```

Agrupar Tuplas

- Tuplas podem ser agrupados pela cláusula **GROUP BY**
  - Opcionalmente pode-se utilizar a cláusula **HAVING** para aplicar condições sobre os grupos que são formados
  - As condições só podem ser definidas sobre atributos do agrupamento ou sobre funções de agregação

❖ Listar as cidades que são origem de pelo menos mais do que um paciente e informar quantos pacientes são dessas cidades

```
SELECT cidade, COUNT(*)
FROM pacientes
GROUP BY cidade
HAVING COUNT(*) > 1
```

Atualização com Consultas

- Comandos de atualização (INSERT, UPDATE e DELETE) podem incluir comandos de consulta

  - Ex1: a médica Maria pediu para cancelar todas as suas consultas após as 17:00

```
DELETE FROM consultas
WHERE hora > '17:00'
AND codm IN
   (SELECT codm FROM medicos
    WHERE nome = 'Maria')
```

Atualização com Consultas

- Comandos de atualização (INSERT, UPDATE e DELETE) podem incluir comandos de consulta

  - Ex2: a direção da clínica determinou que deve haver sempre dois médicos por ambulatório, caso contrário o médico não deve ter um ambulatório definido/fixo

```
UPDATE medicos
SET nroa = NULL
WHERE NOT EXISTS
   (SELECT * FROM medicos AS m
    WHERE m.codm != medicos.codm
      AND m.nroa = medicos.nroa)
```