

# Banco de Dados I

## 06 - Linguagem de Manipulação de Dados

---

Marcos Roberto Ribeiro



Instituto Federal Minas Gerais - Campus Bambuí

2018

- Até o momento estudamos instruções DDL, ou seja, já temos conhecimento de como manipular as estruturas de um banco de dados;
- Agora vamos estudar as instruções de manipulação de dados (DML);
- Com as instruções DML podemos inserir, modificar e apagar dados nas tabelas criadas com a DDL;
- Além disto, as instruções DML permitem a realização de consulta sobre os dados existentes.

# Banco de Dados Acadêmico I

Em nossos exemplos vamos utilizar o banco de dados acadêmico mostrado a seguir

## aluno

id_aluno: INTEGER NOT NULL [ PK ]
nome_aluno: VARCHAR(50) NOT NULL
cpf: CHAR(11) [ AK ]
data_nascimento: DATE
media: FLOAT

## professor

id_professor: INTEGER NOT NULL [ PK ]
nome_professor: VARCHAR(50) NOT NULL
sobrenome: VARCHAR(50) NOT NULL
area: VARCHAR(20)

## matriculado

id_disciplina: INTEGER NOT NULL [ PFK ]
id_aluno: INTEGER NOT NULL [ PFK ]
nota: FLOAT

## disciplina

id_disciplina: INTEGER NOT NULL [ PK ]
nome_disciplina: VARCHAR(50) NOT NULL
carga_horaria: INTEGER NOT NULL
id_professor: INTEGER [ FK ]



```
CREATE TABLE aluno(  
    id_aluno SERIAL NOT NULL,  
    nome_aluno VARCHAR(50) NOT NULL,  
    cpf CHAR(11),  
    data_nascimento DATE,  
    media FLOAT DEFAULT 0.0,  
    CONSTRAINT aluno_pk PRIMARY KEY (id_aluno),  
    CONSTRAINT aluno_cpf_key UNIQUE (cpf)  
);
```

```
CREATE TABLE professor (  
    id_professor INT NOT NULL,  
    nome_professor VARCHAR(50) NOT NULL,  
    sobrenome VARCHAR(50) NOT NULL,  
    area VARCHAR(20),  
    CONSTRAINT professor_pk PRIMARY KEY (id_professor)  
);
```

```
CREATE TABLE disciplina(  
    id_disciplina INT NOT NULL,  
    nome_disciplina VARCHAR(30) NOT NULL,  
    carga_horaria INT NOT NULL,  
    id_professor INT,  
    CONSTRAINT disciplina_pk PRIMARY KEY (id_disciplina),  
    CONSTRAINT disciplina_fk_professor FOREIGN KEY (id_professor)  
    ↪ REFERENCES professor(id_professor)  
);
```

```
CREATE TABLE matriculado(  
    id_disciplina INT NOT NULL,  
    id_aluno INT NOT NULL,  
    nota FLOAT,  
    CONSTRAINT matriculado_pk PRIMARY KEY (id_disciplina, id_aluno),  
    CONSTRAINT matriculado_fk_aluno FOREIGN KEY (id_aluno)  
    ↪ REFERENCES aluno(id_aluno),  
    CONSTRAINT matriculado_fk_disciplina FOREIGN KEY (id_disciplina)  
    ↪ REFERENCES disciplina(id_disciplina)  
);
```

# Criando Tabelas com Scripts SQL

- As tabelas do banco de dados acadêmico podem ser criadas através de um script SQL disponível na página da disciplina;
- Vamos inicialmente criar o banco de dados no PostgreSQL;

```
CREATE DATABASE academico;
```

- Para criarmos as tabelas com o script vamos executar o seguinte comando:  
`psql -h localhost -U postgres -d academico -f arquivo_script.sql`
- O parâmetro **-d** conecta diretamente ao banco de dados **academico** e o parâmetro **-f** informa os arquivo contendo o script SQL a ser executado.



# Inserindo Dados I

- A inserção de dados é feita através da instrução **INSERT**. Sua sintaxe é a seguinte:

```
INSERT INTO tabela (c1, ..., cn)
VALUES (v1, ..., vn);
```

- Para entendermos melhor vamos utilizar inserir alguns dados na tabela *aluno*:

```
INSERT INTO aluno(nome_aluno, cpf, data_nascimento, media)
VALUES ('José', NULL, '1990-01-20', 0.0);
INSERT INTO aluno(nome_aluno, data_nascimento)
VALUES
('João', '1993-09-10'),
('Maria', '1989-05-15'),
('Ana', '1992-04-21');
```

## Inserindo Dados II

- Podemos notar que para atribuírmos um valor nulo a um campo podemos omiti-lo ou inserir **NULL**;
- Porém se um campo que possui valor padrão for omitido, o valor inserido será o seu valor padrão;
- Para visualizarmos os dados inseridos podemos utilizar a instrução:

```
SELECT * FROM tabela;
```

- Para o nosso exemplo:

```
SELECT * FROM aluno;
```

## Inserindo Dados III

### Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José		1990-01-20	0
2	João		1993-09-10	0
3	Maria		1989-05-15	0
4	Ana		1992-04-21	0

- A instrução **SELECT** é uma instrução de consulta de dados, entraremos em mais detalhes posteriormente, por enquanto vamos usá-la apenas para verificar o resultado das instruções de manipulação de dados.

# Exclusão de dados I

- A exclusão de dados é feita através da instrução **DELETE**;
- Quando vamos excluir um dado devemos ter as mesmas precauções da alteração, ou seja, devemos especificar exatamente qual(ais) registro(s) serão excluídos.
- O formato da instrução **DELETE** é o seguinte:

```
DELETE FROM tabela  
WHERE ch=id;
```

- Como exemplo vamos excluir o aluno João especificando a chave primária deste registro<sup>1</sup>:

```
DELETE FROM aluno  
WHERE id_aluno = 2;
```

# Exclusão de dados II

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José		1990-01-20	0
3	Maria		1989-05-15	0
4	Ana		1992-04-21	0

---

<sup>1</sup>Na verdade, tanto a exclusão quando a alteração podem especificar os registros através de campos que não são chaves primária. Porém, veremos isto com mais detalhes quanto estudarmos a SQL

# Alterando Dados I

- A alteração de dados é feita através da instrução **UPDATE**;
- Quando vamos alterar um dado devemos tomar o cuidado de especificar exatamente qual(ais) registro(s) serão alterados. Normalmente esta especificação é feita através da chave primária.
- O formato da instrução **UPDATE** é o seguinte:

```
UPDATE tabela  
SET c1=d1, ..., cn=dn  
WHERE ch=id;
```

- A instrução **SET** determina quais alterações devem ser feitas. A instrução **WHERE** especifica onde as alterações acontecerão;

## Alterando Dados II

- Como exemplo vamos alterar o CPF e a data de nascimento do aluno João:

```
UPDATE aluno
SET cpf='01234567890', data_nascimento='1991-12-23'
WHERE id_aluno = 3;
```

### Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José		1990-01-20	0
4	Ana		1992-04-21	0
3	Maria	01234567890	1991-12-23	0

# Banco de Dados para Consultas I

Enquanto estudamos a SQL vamos considerar novamente o banco de dados acadêmico, mas agora com os seguintes dados:

**Tabela *aluno***

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José	111111111111	1990-01-20	85
2	João		1993-09-10	84.8
3	Maria	333333333333	1989-05-15	66.9
4	Ana	444444444444	1992-04-21	70.5

Se os dados da tabela **aluno** forem apagados para uma nova inserção pode ser conveniente reiniciar a sequência sobre o campo **id\_aluno** com a instrução:



# Banco de Dados para Consultas II

```
ALTER SEQUENCE aluno_id_aluno_seq RESTART;
```

## Tabela *professor*

id_professor	nome_professor	sobrenome	area
10	Roberto	Silva	Computação
20	Carlos	Alves	Matemática
30	José	Teodoro	Administração
40	Rodrigo	Martins	Engenharia

# Banco de Dados para Consultas III

Tabela *disciplina*

id_disciplina	nome_disciplina	carga_horaria	id_professor
100	Algoritmos	80	10
200	Banco de Dados	80	10
300	Cálculo	60	20
400	Álgebra	60	20
500	Empreendedorismo	40	30
600	Redes	80	

# Banco de Dados para Consultas IV

Tabela *matriculado*

id_disciplina	id_aluno	nota
100	1	89.5
200	1	78
300	1	90
400	1	82.5
100	2	88.7
200	2	81
400	2	77.5
500	2	92
100	3	72.5
200	3	52.8
400	3	83.3
100	4	71
200	4	70
300	3	59

Como exercício altere os dados do banco de dados anterior para possuir estas informações.

# Consultas em Bancos de Dados

- As consultas são realizadas com a instrução **SELECT**;
- Até o momento utilizamos apenas a instrução **SELECT \* FROM *tabela***. Porém a instrução pode conter também as cláusulas **WHERE** e **ORDER BY**:

```
SELECT c1,...,cn  
FROM tabelas  
WHERE condições  
ORDER BY ordenação;
```

- A cláusula **SELECT** delimita quais campos devem ser retornados pela consulta<sup>2</sup>. A cláusula **FROM** informa quais tabelas estão envolvidas;
- Já a cláusula **WHERE** é opcional e delimita os resultados da consulta através de certas condições;
- A cláusula **ORDER BY** determina como o resultado da consulta deve ser ordenado.

---

<sup>2</sup>Já observamos que podemos usar o **\*** para retornar todos os campos

# A Cláusula SELECT

- Uma consulta simples utilizando as cláusulas SELECT e FROM poderia ser *selecionar o nome e a data de nascimento dos alunos*:

```
SELECT nome_aluno, data_nascimento FROM aluno;
```

## Resultado

nome_aluno		data_nascimento
-----+-----		
José		1990-01-20
João		1993-09-10
Maria		1989-05-15
Ana		1992-04-21

# Renomeando Campos e Tabelas

- Também é possível renomear campos e tabelas, isto será útil quando for necessário realizar junções de tabelas. Como exemplo vamos modificar a consulta anterior<sup>3</sup>:

```
SELECT a.nome_aluno AS aluno,  
       a.data_nascimento AS nascimento  
FROM aluno AS a;
```

## Resultado

aluno		nascimento
-----	+	-----
José		1990-01-20
João		1993-09-10
Maria		1989-05-15
Ana		1992-04-21

<sup>3</sup>A renomeação não muda os nomes no banco de dados, ela ocorre apenas dentro da consulta

# Realizando Cálculos I

- Além dos próprios campos da tabela podemos realizar cálculos através dos operadores aritméticos como + (adição), - (subtração), \* (multiplicação), / (divisão). Por exemplo:

```
SELECT 102 * 30 as conta;
```

## Resultado

conta

-----

3060

## Realizando Cálculos II

- Também podemos realizar operações aritméticas sobre campos numéricos. Suponha que a média dos alunos deva ser reduzida em 20% e acrescida de 25 pontos:

```
SELECT nome_aluno AS aluno,  
       media * 0.8 + 25 AS media  
FROM aluno;
```

### Resultado

aluno		media
-----	+	-----
José		93
João		92.84
Maria		78.52
Ana		81.4



# Concatenando Dados Textuais

- Para concatenarmos dados textuais em uma consulta podemos usar o || (dois pipes);
- Como exemplo vamos selecionar o nome completo dos professores:

```
SELECT nome_professor || ' ' || sobrenome AS professor
FROM professor;
```

## Resultado

professor

-----  
Roberto Silva  
Carlos Alves  
José Teodoro  
Rodrigo Martins

# A Cláusula WHERE

- A cláusula **WHERE** possibilita expressar mais precisamente quais registros devem ser retornados pela consulta;
- As condições desta cláusula são condições booleanas formadas pela combinação de *comparações* e dos conectivos lógicos **AND**, **OR** e **NOT**;
- As comparações podem depender do tipo do campo e dos valores armazenados no mesmo;
- Primeiramente vamos estudar comparações envolvendo tipos de campos numéricos e temporais;
- As comparações mais comuns sobre campos numéricos e temporais envolvem os operadores de comparação: **<**, **<=**, **=**, **<>**, **>**, **>=**, **!=** e **IN**.

# Exemplos com WHERE I

*“Obter os alunos que nasceram depois de 1990”:*

```
SELECT * FROM aluno
WHERE data_nascimento >= '1991-01-01';
```

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
2	João		1993-09-10	84.8
4	Ana	444444444444	1992-04-21	70.5

## Exemplos com WHERE II

*“obter as disciplinas com carga horária de 40 ou 60”:*

```
SELECT * FROM disciplina  
WHERE carga_horaria IN (40,60);
```

### Resultado

id_disciplina	nome_disciplina	carga_horaria	id_professor
300	Cálculo	60	20
400	Álgebra	60	20
500	Empreendedorismo	40	30

## Considerando Intervalos

- Outro tipo de comparação sobre campos numéricos envolve a verificação de intervalos através das palavras chaves **BETWEEN** e **AND**. Como exemplo vamos considerar a consulta *“obter os alunos que nasceram na década de 1980”*:

```
SELECT * FROM aluno
WHERE data_nascimento BETWEEN '1980-01-01' AND '1989-12-31';
```

### Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
3	Maria	33333333333	1989-05-15	66.9

# Combinando Condições

- Agora vamos fazer uma combinação com duas comparações numéricas através da consulta *“obter os alunos com data de nascimento a partir de 1990 e média maior que 80 pontos”*:

```
SELECT * FROM aluno
WHERE data_nascimento >= '1990-01-01'
      AND media > 80;
```

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José	11111111111	1990-01-20	85
2	João		1993-09-10	84.8

# Comparações sobre Campos Textuais

- Os operadores de comparação <, <=, =, <>, > e >= também podem ser usados para comparar campos textuais (**CHAR** ou **VARCHAR**). Entretanto, quando utilizados, o valor comparado deve ser exatamente igual ao dado procurado.
- Quando desejamos filtrar consultas considerando campos textuais podemos utilizar os operadores **LIKE**, **ILIKE** e **SIMILAR TO**;
- Estes operadores permitem a comparação entre campos textuais e certos padrões de caracteres;
- A construção destes padrões é realizada através da combinação de caracteres caracteres simples (letras, números, espaço, ...), de caracteres especiais % e \_ e também de expressões regulares;
- O caractere \_ representa um caractere qualquer;
- O caractere % representa uma combinação de qualquer quantidade de caracteres;

# Exemplo de Comparação Textual I

Para entendermos melhor vamos considerar a consulta “*obter os alunos cujos nomes comecem com a letra J*”:

```
SELECT * FROM aluno
WHERE nome_aluno LIKE 'J%';
```

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
1	José	111111111111	1990-01-20	85
2	João		1993-09-10	84.8



## Exemplo de Comparação Textual II

Outro exemplo poderia ser a consulta “obter os alunos cujos nomes possuam três letras”:

```
SELECT * FROM aluno
WHERE nome_aluno LIKE '___';
```

### Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
4	Ana	444444444444	1992-04-21	70.5

- Podemos usar **NOT LIKE** para especificar que um padrão não deve ser retornado;
- Também podemos combinar os tipos de comparações visto até o momento através dos conectivos **AND**, **OR** e **NOT**;

# Operador ILIKE

- O operador **ILIKE** é similar ao operador **LIKE**, porém não diferencia letras maiúsculas de minúsculas;
- Como exemplo na consulta *“obter os professores que possuam a letra a (maiúscula ou minúscula) no sobrenome”*:

```
SELECT * FROM professor  
WHERE sobrenome ILIKE '%a%';
```

## Resultado

id_professor	nome_professor	sobrenome	area
10	Roberto	Silva	Computação
20	Carlos	Alves	Matemática
40	Rodrigo	Martins	Engenharia

# Operador SIMILAR TO

- O operador **SIMILAR TO** além dos caracteres especiais `_` e `%`, também permite o uso de expressões regulares;
- Exemplo: *“Os alunos cujo nome possua a letra a independentemente de estar acentuada”*<sup>4</sup>:

```
SELECT * FROM aluno
WHERE nome_aluno SIMILAR TO '%(a|á|à|ã|â|ä)%';
```

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
2	João		1993-09-10	84.8
3	Maria	33333333333	1989-05-15	66.9
4	Ana	44444444444	1992-04-21	70.5

<sup>4</sup>O PostgreSQL possui extensões mais adequadas para lidar com caracteres especiais.



# Extensão unaccent

- O PostgreSQL possui uma extensão chamada *unaccent* que facilita muito quando temos que trabalhar com caracteres especiais (acentos e cedilha)
- É preciso instalar o pacote **postgresql-contrib** para usar tal extensão
- Para usar a extensão, é preciso criá-la dentro de cada banco com o seguinte comando:

```
CREATE EXTENSION unaccent;
```

- Depois podemos, usar a função *unaccent* da seguinte maneira:

```
SELECT * FROM aluno  
WHERE UNACCENT(LOWER(nome_aluno)) ILIKE '%a%';
```

## Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
2	João		1993-09-10	84.8
3	Maria	33333333333	1989-05-15	66.9
4	Ana	44444444444	1992-04-21	70.5

## Comparação com Valores Nulos

- A consideração de valor nulo em um campo é feita através das seguintes comparações:
  - campo IS NULL** Retorna os registros nos quais o campo possui valor nulo;
  - campo IS NOT NULL** Retorna os registros nos quais o campos não possui valor nulo;
- Como exemplo vamos supor a consulta *“obter os alunos cujo CPF apresente valor nulo”*:

```
SELECT * FROM aluno  
WHERE cpf IS NULL;
```

### Resultado

id_aluno	nome_aluno	cpf	data_nascimento	media
2	João		1993-09-10	84.8

## Ordenação do resultado

- A ordenação dos registros retornados por uma consulta é feita através da cláusula **ORDER BY**;
- Os registros poder ser ordenados de forma crescente ou decrescente considerando um ou mais campos;
- Por padrão a ordenação sobre dos registros é feita de forma crescente, porém podemos impor uma ordenação decrescente informando a palavra chave **DESC** logo após o campo desejado;

## Exemplo com Ordenação

Como exemplo vamos considerar a consulta “*obter as disciplinas ordenadas por carga horária decrescente e nome crescente*”:

```
SELECT * FROM disciplina
ORDER BY carga_horaria DESC, nome_disciplina;
```

### Resultado

id_disciplina	nome_disciplina	carga_horaria	id_professor
100	Algoritmos	80	10
200	Banco de Dados	80	10
600	Redes	80	
400	Álgebra	60	20
300	Cálculo	60	20
500	Empreendedorismo	40	30



# Tratamento de Conjuntos de Dados com SQL

- Um conjunto é uma coleção de elementos únicos, ou seja, os elementos de um conjunto não se repetem;
- Por padrão, o resultado de uma consulta SQL pode conter registros repetidos. Isto acontece porque, na maioria das vezes, os registros repetidos devem ser exibidos e a eliminação destas repetições é uma operação de alto custo para o SGBD.
- No entanto, em alguns casos é importante que o resultado de uma consulta SQL seja um conjunto, ou seja, não contenha registros repetidos. Para atender a estes casos deve ser usada a palavra chave **DISTINCT** juntamente com a cláusula **SELECT**;

## Exemplo com DISTINCT

Para exemplificarmos vamos resolver a consulta “*obter as possíveis cargas horárias das disciplinas*”. Uma primeira consulta poderia ser:

```
SELECT carga_horaria  
FROM disciplina;
```

### Resultado

carga_horaria
80
80
60
60
40
80

Podemos notar que foram retornados registros repetidos. Para que isto não aconteça devemos usar a consulta:

```
SELECT DISTINCT carga_horaria  
FROM disciplina;
```

### Resultado

carga_horaria
40
60
80

# Tratamento de Conjuntos de Dados com SQL

- Além de podermos retornar o resultado de uma consulta como um conjunto, a SQL permite também realizar as operações de *união*, *interseção* e *diferença* de conjuntos;
- Para realizarmos as referidas operações devemos juntar duas consultas SQL com as palavras chaves **UNION** (para união), **INTERSECT** (para interseção) ou **EXCEPT** (para diferença).
- É importante observar que os resultados das duas consultas devem ser compatíveis, isto é, devemos realizar estas operações sobre registros que possuam o mesmo número de campos e os campos correspondentes devem ser do mesmo tipo;

# Tratamento de Conjuntos de Dados com SQL

- Como exemplo de consulta utilizando a operação de união temos *obter os nomes de todos os professores e alunos*

```
SELECT nome_professor AS nome
FROM professor
UNION
SELECT nome_aluno AS nome
FROM aluno;
```

## Resultado

nome

-----  
Carlos  
João  
Rodrigo  
Maria  
Ana  
José  
Roberto

- Podemos notar que o nome **José** foi retornado apenas uma vez. Para mostrar repetições pode ser usada a palavra chave **ALL** após **UNION**;
- Como exercício reescreva esta consulta usando as operações **EXCEPT** e **INTERSECT**. E depois usando a palavra chave **ALL**.

# Testando Valores

- Quando é preciso testar algum valor podemos utilizar a palavra chave **CASE**;

Por exemplo, se desejarmos transformar as notas dos alunos em conceito:

```
SELECT *,  
  CASE  
    WHEN nota >= 90 THEN 'A'  
    WHEN nota >= 75 THEN 'B'  
    WHEN nota >= 60 THEN 'C'  
    WHEN nota >= 40 THEN 'D'  
    ELSE 'E'  
  END AS conceito  
FROM matriculado;
```

## Resultado

id_disciplina	id_aluno	nota	conceito
100	1	89.5	B
200	1	78	B
300	1	90	A
400	1	82.5	B
100	2	88.7	B
200	2	81	B
400	2	77.5	B
500	2	92	A
100	3	72.5	C
200	3	52.8	D
400	3	83.3	B
100	4	71	C
200	4	70	C
300	3	59	D

# Funções de Agregação

- Em determinadas consultas é necessário agrupar valores e obter informações sobre tais agrupamentos como soma, média e outras;
- Nestas consultas podemos utilizar as chamadas funções de agregação.

As funções de agregação mais comuns são:

**AVG()** Média;

**COUNT()** Contagem;

**MAX()** Máximo;

**MIN()** Mínimo;

**SUM()** Soma;

- A maneira mais simples de utilizar funções de agregação é informar apenas a função de agregação na cláusula **SELECT**. Por exemplo, para obtermos a quantidade de alunos podemos usar a consulta:

```
SELECT COUNT(*)  
FROM aluno;
```

count

-----

4

## Funções de Agregação I

- A função **COUNT** aceita qualquer tipo de campo (inclusive \*), as demais funções aceitam apenas campos numéricos. Como exemplo, poderíamos estar interessados saber qual a maior média dentre os alunos através da consulta:

```
SELECT MAX(media)  
FROM aluno;
```

## Funções de Agregação II

- Na maioria das vezes precisamos de agregar valores agrupando por um determinado campo. Nestas situações devemos utilizar uma consulta no seguinte formato:

```
SELECT c1,...,cn, FUNÇÃO(campo)
FROM tabela
WHERE condições
GROUP BY c1,...,cn
HAVING condições agregação;
```

- Consultas neste formato retornam o resultado da agregação agrupado pelos campos *c1*, ..., *cn* informados na cláusula **GROUP BY**;
- A cláusula **HAVING** é semelhante à cláusula **WHERE**, porém as condições são impostas sobre as funções de agregação.



## Exemplos com Funções de Agregação

- Suponhamos a consulta “*obter a carga horária de cada professor*”. Neste caso podemos utilizar a consulta:

```
SELECT id_professor,  
       ↪ SUM(carga_horaria)  
FROM disciplina  
GROUP BY id_professor;
```

### Resultado

id_professor	sum
-----+-----	
	80
20	120
10	160
30	40

- Considerando novamente a consulta anterior, mas desta vez apenas os professores com carga horária superior a 100:

```
SELECT id_professor, SUM(carga_horaria)  
FROM disciplina  
GROUP BY id_professor  
HAVING SUM(carga_horaria) > 100;
```

## Consultas com Funções de Agregação Mais Complexas

- Agora vamos considerar uma consulta um pouco mais complexa. Obter a média para cada *id\_aluno* considerando as disciplinas 100 e 200 e médias superiores a 80 na tabela *matriculado*.

```
SELECT id_aluno, AVG(nota) AS media
FROM matriculado
WHERE id_disciplina = 100 OR id_disciplina = 200
GROUP BY id_aluno
HAVING AVG(nota) > 80;
```

### Resultado

id_aluno	media
1	83.75
2	84.85

- Como exercício calcule novamente a média de cada aluno para todas as disciplinas cursadas, porém sem utilizar a função **AVG**;

## Consultas Envolvendo mais de uma Tabela

- Muitas vezes uma consulta pode necessitar de informações disponíveis em mais de uma tabela, neste caso devemos juntar todas as tabelas necessárias em uma única instrução SQL. Consultas deste tipo são chamadas de consultas com junções;
- Existem três tipos de consultas com junção:
  - Junção de Produto Cartesiano** todos os elementos de uma tabela são combinados com todos os elementos de outra tabela;
  - Junção Interna** Os elementos das tabelas são combinados considerando campos que possuem valores em comum nas tabelas;
  - Junção Externa** Os elementos das tabelas são combinadas considerando valores comuns entre campos, e os elementos que não possuem valores em comum são combinados com valores nulos.

# Junção de Produto Cartesiano

Na junção de produto cartesiano o SGBD combina todos os registros de uma tabela com todos os registros de outra tabela. Este tipo de junção é pouco usado, mas pode ser útil em determinadas situações;

Por exemplo, “*Listas todas as duplas possíveis de nomes entre professores e alunos*”, há duas maneiras possíveis:

```
SELECT nome_aluno,  
       ↪ nome_professor  
FROM aluno, professor;
```

```
SELECT nome_aluno,  
       ↪ nome_professor  
FROM aluno  
CROSS JOIN professor;
```

nome_aluno	nome_professor
-----	-----
José	Roberto
José	Carlos
José	José
José	Rodrigo
João	Roberto
João	Carlos
João	José
João	Rodrigo
Maria	Roberto
Maria	Carlos
Maria	José
Maria	Rodrigo
Ana	Roberto
Ana	Carlos
Ana	José
Ana	Rodrigo

## Junção Interna

- No caso da junção interna, o SGBD verifica se um determinado campos entre duas tabelas possui o mesmo valor nas duas tabelas, somente neste caso o registro será retornado;
- Quando há mais de duas tabelas envolvidas, o SGBD combina as tabelas duas a duas;
- Normalmente são utilizados os campos de chave estrangeira nas comparações;<sup>5</sup>
- Como exemplo vamos tentar obter os nomes dos professores e as disciplinas ministradas pelos mesmos, neste caso precisaremos de duas tabelas;
- Podemos observar que estas tabelas estão relacionadas através da chave estrangeira presente no campo **id\_professor** da tabela **disciplina**, então vamos usar esta chave estrangeira para efetuar a consulta;

---

<sup>5</sup>Na verdade podemos usar campos que não são chaves estrangeiras, entretanto a utilização de chaves estrangeiras é mais eficiente

## Exemplo Junção Interna

Exemplo “Listar os nomes dos professores e as disciplinas ministradas pelos mesmos”, há duas maneiras possíveis:

```
SELECT p.nome_professor,  
       d.nome_disciplina  
FROM professor AS p,  
     disciplina AS d  
WHERE p.id_professor =  
       ↪ d.id_professor;
```

```
SELECT p.nome_professor,  
       d.nome_disciplina  
FROM professor AS p  
INNER JOIN disciplina AS d  
ON p.id_professor =  
    ↪ d.id_professor;
```

### Resultado

nome_professor	nome_disciplina
-----+-----	
Roberto	Algoritmos
Roberto	Banco de Dados
Carlos	Cálculo
Carlos	Álgebra
José	Empreendedorismo

As palavras chaves **INNER JOIN** podem ser substituídas simplesmente por **JOIN**

## Exemplo Junção Interna I

Agora vamos considerar outra consulta, “*Listar os nomes dos alunos, as disciplinas cursadas pelos mesmos e suas respectivas notas*”:

```
SELECT a.nome_aluno,  
       d.nome_disciplina,  
       m.nota  
FROM aluno AS a,  
     disciplina AS d,  
     matriculado AS m  
WHERE a.id_aluno = m.id_aluno  
     AND d.id_disciplina =  
     ↪ m.id_disciplina;
```

```
SELECT a.nome_aluno,  
       d.nome_disciplina,  
       m.nota  
FROM aluno AS a  
JOIN matriculado AS m  
ON a.id_aluno = m.id_aluno  
JOIN disciplina AS d  
ON d.id_disciplina =  
   ↪ m.id_disciplina;
```

## Exemplo Junção Interna II

### Resultado:

nome_aluno	nome_disciplina	nota
José	Algoritmos	89.5
José	Banco de Dados	78
José	Cálculo	90
José	Álgebra	82.5
João	Algoritmos	88.7
João	Banco de Dados	81
João	Álgebra	77.5
João	Empreendedorismo	92
Maria	Algoritmos	72.5
Maria	Banco de Dados	52.8
Maria	Álgebra	83.3
Ana	Algoritmos	71
Ana	Banco de Dados	70
Maria	Cálculo	59



# Junção Natural

A junção interna pode ser feita também com as palavras chaves **NATURAL JOIN**, desde que os campos a serem relacionados nas tabelas possuam o mesmo nome. Por exemplo:

```
SELECT a.nome_aluno,  
       d.nome_disciplina,  
       m.nota  
FROM aluno AS a  
NATURAL JOIN matriculado AS  
    ↪ m  
NATURAL JOIN disciplina AS  
    ↪ d;
```

nome_aluno	nome_disciplina	nota
-----+-----+-----		
José	Algoritmos	89.5
José	Banco de Dados	78
José	Cálculo	90
José	Álgebra	82.5
João	Algoritmos	88.7
João	Banco de Dados	81
João	Álgebra	77.5
João	Empreendedorismo	92
Maria	Algoritmos	72.5
Maria	Banco de Dados	52.8
Maria	Álgebra	83.3
Ana	Algoritmos	71
Ana	Banco de Dados	70
Maria	Cálculo	59

## Junção Externa

- Na junção externa, o SGBD retorna os registros relacionados entre as tabelas (de forma semelhante a junção interna), mas também retorna os registros não relacionados combinando-os com valores nulos;

- Há três tipos de junção externa:

**LEFT OUTER JOIN** Todos os registros da tabela esquerda mesmo quando não exista registros correspondentes na tabela direita;

**RIGHT OUTER JOIN** Todos os registros da tabela direita mesmo quando não exista registros correspondentes na tabela esquerda;

**FULL OUTER JOIN** Esta operação apresenta todos os dados das tabelas à esquerda e à direita, mesmo que não possuam correspondência em outra tabela.

# LEFT OUTER JOIN

- Como exemplo vamos “obter os todos professores (mesmo aqueles que não tem disciplina) e suas respectivas disciplinas”:<sup>6</sup>

```
SELECT p.nome_professor,  
       d.nome_disciplina  
FROM professor AS p  
LEFT OUTER JOIN disciplina  
  ↪ AS d  
ON p.id_professor =  
  ↪ d.id_professor;
```

## Resultado

nome_professor	nome_disciplina
-----+-----	
Roberto	Algoritmos
Roberto	Banco de Dados
Carlos	Cálculo
Carlos	Álgebra
José	Empreendedorismo
Rodrigo	

<sup>6</sup>A palavra chave **OUTER** pode ser omitida.

# RIGHT OUTER JOIN

- Como exemplo vamos “obter os todas as disciplinas (mesmo aquelas sem professor) e seus respectivos professores”:

```
SELECT p.nome_professor,  
       d.nome_disciplina  
FROM professor AS p  
RIGHT JOIN disciplina AS d  
ON p.id_professor =  
   ↪ d.id_professor;
```

## Resultado

nome_professor		nome_disciplina
-----+-----		
Roberto		Algoritmos
Roberto		Banco de Dados
Carlos		Cálculo
Carlos		Álgebra
José		Empreendedorismo
		Redes

# FULL OUTER JOIN

- Como exemplo vamos “obter os todas as disciplinas (mesmo aquelas sem professor) e seus respectivos professores (mesmo aqueles sem disciplina)”:

```
SELECT p.nome_professor,  
       d.nome_disciplina  
FROM professor AS p  
FULL JOIN disciplina AS d  
ON p.id_professor =  
   ↪ d.id_professor;
```

## Resultado

nome_professor	nome_disciplina
-----+-----	
Roberto	Algoritmos
Roberto	Banco de Dados
Carlos	Cálculo
Carlos	Álgebra
José	Empreendedorismo
	Redes
Rodrigo	

# Consultas Usando a Mesma Tabela Mais de uma Vez

As vezes é preciso usar uma mesma tabela mais de uma vez. Por exemplo: “Obter as disciplinas ministradas pelo mesmo professor”;

```
SELECT d1.nome_disciplina,  
    ↪ p.nome_professor  
FROM disciplina AS d1,  
    professor AS p,  
    disciplina AS d2  
WHERE d1.id_professor =  
    ↪ p.id_professor  
    AND d1.id_professor =  
    ↪ d2.id_professor  
    AND d2.id_disciplina <>  
    ↪ d1.id_disciplina;
```

## Resultado

nome_disciplina	nome_professor
Algoritmos	Roberto
Banco de Dados	Roberto
Cálculo	Carlos
Álgebra	Carlos

Foi necessário usar a tabela disciplina duas vezes (**d1** e **d2**), pois precisávamos comparar duas disciplinas com o mesmo professor (e com **id\_disciplina** diferentes).

# LIMIT e OFFSET

- No PostgreSQL podemos utilizar as palavras chaves **LIMIT** e **OFFSET** para controlar a quantidade de quais linhas do resultado da consulta devem ser retornadas;
- Como exemplo retornar da segunda a quinta disciplina:

```
SELECT id_disciplina,  
       nome_disciplina  
FROM disciplina  
LIMIT 4 OFFSET 1;
```

## Resultado

id_disciplina	nome_disciplina
200	Banco de Dados
300	Cálculo
400	Álgebra
500	Empreendedorismo

# Criando Tabelas a partir de Consultas

- Quando estudamos as instruções DDL, vimos como criar tabelas especificando seus campos e restrições. Outra maneira de criar tabelas é a partir do resultado de uma consulta. Como por exemplo:

```
CREATE TABLE adn AS
SELECT a.nome_aluno, d.nome_disciplina, m.nota
FROM aluno AS a, disciplina AS d, matriculado AS m
WHERE a.id_aluno = m.id_aluno
      AND d.id_disciplina = m.id_disciplina;
```

- Agora tente selecionar os dados da tabela ADN;
- A tabela criada com a instrução **CREATE TABLE ... AS** cria uma tabela com os campos e dados retornados pela consulta, mas sem nenhum tipo de restrição (como chaves primárias e estrangeiras). Como criar estas restrições?
- Outra instrução interessante é **CREATE TEMPORARY TABLE ... AS** que cria uma tabela temporária para o usuário corrente que existira apenas enquanto o mesmo estiver conectado no banco de dados.



# Manipulando Dados a partir de Resultados de Consultas

- As instruções DML **INSERT**, **UPDATE**, **DELETE** podem ser combinadas com a instrução **SELECT** para automatizar a manipulação de dados;
- Como exemplo, vamos matricular todos os alunos na disciplina de “Redes” (*id\_disciplina = 600*):

```
INSERT INTO matriculado  
(id_disciplina, id_aluno, nota)  
SELECT 600, id_aluno, 0 FROM aluno;
```

- Vamos apagar estes dados para não afetar os próximos exemplos:

```
DELETE FROM matriculado WHERE id_disciplina = 600;
```

# UPDATE com SELECT

- Para exemplificar o um **UPDATE** usando um **SELECT**, vamos atualizar a média dos alunos com base nas notas da tabela *matriculado*:

```
CREATE TEMPORARY TABLE media AS  
SELECT id_aluno, AVG(nota) AS media  
FROM matriculado  
GROUP BY id_aluno;
```

```
UPDATE aluno AS a  
SET media = m.media  
FROM media AS m  
WHERE a.id_aluno = m.id_aluno;
```

# Funções para Dados Temporais

- As principais funções para dados temporais são:
  - `current_date` Retorna a data atual do sistema;
  - `current_time` Retorna a hora atual do sistema;
  - `date_part()` Retorna uma parte de um dado temporal como ano, mês, hora, etc.;
  - `extract()` Retorna uma parte de um dado temporal como ano, mês, hora, etc.;
  - `now()` Retorna a data e a hora atuais do sistema;
- Exemplos:

```
SELECT current_date, current_time, now();
```

```
SELECT nome_aluno, date_part('month', data_nascimento) FROM aluno;
```

```
SELECT nome_aluno, extract(year from data_nascimento) FROM aluno;
```

- Sub-consultas são consultas SQL inseridas dentro de outra instrução SQL;
- A maneira mais comum de utilizar sub-consulta é nas cláusulas **WHERE** ou **HAVING** de um **SELECT**;
- As sub-consultas podem utilizar as seguintes cláusulas:
  - **EXISTS**;
  - **IN**;
  - **ANY** ou **SOME**;
  - **ALL**.

## Sub-Consultas com EXISTS

- A cláusula **EXISTS** permite verificar a existência de dados em uma sub-consulta para retornar ou não os dados da consulta principal;
- Como exemplo vamos considerar a consulta “Obter os nomes dos alunos que já obtiveram uma nota maior que 80”

```
SELECT a.nome_aluno
FROM aluno AS a
WHERE EXISTS(
    SELECT 1
    FROM matriculado AS m
    WHERE nota > 80
    AND a.id_aluno = m.id_aluno);
```

### Resultado

nome\_aluno

-----  
José

João

Maria

- Observe que a sub-consulta não precisa retornar um campo específico para o **EXISTS**, apenas retornar ou não uma linha.

## Exemplo com NOT EXISTS

Outro exemplo de consulta seria: *“Obter o nome dos alunos que não cursaram disciplinas ministradas pelo professor José”*

```
SELECT a.nome_aluno
FROM aluno AS a
WHERE NOT EXISTS(
    SELECT 1
    FROM matriculado AS m,
         disciplina as D
    WHERE m.id_disciplina =
↪ d.id_disciplina
    AND d.id_professor = 30
    AND m.id_aluno = a.id_aluno);
```

### Resultado

nome_aluno
-----
José
Maria
Ana

## Exemplo com CASE WHEN EXISTS

Agora um exemplo combinando a cláusula **EXISTS** com o **CASE**: “Listar todas as disciplina informando quais os alunos João e José cursaram juntos”

```
SELECT d.nome_disciplina,  
       CASE WHEN EXISTS(  
           SELECT id_disciplina, COUNT(*)  
           FROM matriculado AS m  
           WHERE (id_aluno=1 OR  
↪ id_aluno=2)  
               AND m.id_disciplina =  
↪ d.id_disciplina  
           GROUP BY id_disciplina  
           HAVING COUNT(*) = 2)  
       THEN 'S' ELSE 'N' END AS jj  
FROM disciplina AS d;
```

### Resultado

nome_disciplina	jj
-----+-----	
Algoritmos	S
Banco de Dados	S
Cálculo	N
Álgebra	S
Empreendedorismo	N
Redes	N

## Sub-Consultas com IN

- Sub-consultas com a cláusula **IN** ou **NOT IN** permitem verificar se o valor de um campo é retornado ou não pela sub-consulta;
- Como exemplo vamos considerar a consulta a mesma consulta feita com o **EXISTS**: “Obter os nomes dos alunos que já obtiveram uma nota maior que 80”.<sup>7</sup>

```
SELECT a.nome_aluno
FROM aluno AS a
WHERE id_aluno IN (
    SELECT id_aluno
    FROM matriculado AS m
    WHERE nota > 80);
```

### Resultado

nome_aluno
-----
José
João
Maria

<sup>7</sup>As sub-consultas com **EXISTS** são mais eficientes do que as consultas com **IN**, **ANY**, **SOME** ou **ALL** pois a sub-consulta com **EXISTS** é processada em paralelo com a consulta principal.



## Sub-Consultas com ANY ou SOME

- As cláusulas **ANY** e **SOME** são equivalentes;<sup>8</sup>
- Sub-consultas estas cláusulas permitem comparar o valor de um campo com os valores de outro campo retornado pelo sub-consulta;
- Como exemplo vamos considerar: *“Obter os alunos cuja média seja superior à média de alguma disciplina que o mesmo cursou”*

```
SELECT nome_aluno, media
FROM aluno
WHERE media > ANY (
    SELECT AVG(nota)
    FROM matriculado
    GROUP BY id_disciplina);
```

### Resultado

nome_aluno		media
-----+-----		
José		85
João		84.8
Ana		70.5

---

<sup>8</sup>IN equivale a = **ANY**.

## Sub-Consultas com ALL


- A cláusula **ALL** compara o valor de um campo com todos os valores retornados pela sub-consulta;
- Como exemplo vamos supor a consulta: *“Obter os alunos cuja média está inferior à média de todos os alunos nas disciplinas cursadas”*


```
SELECT nome_aluno, media
FROM aluno
WHERE media < ALL (
    SELECT AVG(nota)
    FROM matriculado
    GROUP BY id_disciplina);
```

### Resultado

nome_aluno		media
-----+-----		
José		85
João		84.8
Ana		70.5

 (2012).  
**Postgresql documentation.**

 Elmasri, R. and Navathe, S. B. (2011).  
***Sistemas de banco de dados.***  
Pearson Addison Wesley, São Paulo, 6 edition.

 Ramakrishnan, R. and Gehrke, J. (2008).  
***Sistemas de gerenciamento de banco de dados.***  
McGrawHill, São Paulo, 3 edition.