

PostgreSQL - SQL Básico

Juliano Atanzio

2019-12-18

PostgreSQL - SQL Básico

Sobre esta Obra...

Desde que fiz meu primeiro curso de PostgreSQL já havia despertado em mim um desejo muito forte de um dia também ministrar o curso.

Mas junto com meu desejo de ministrar o curso havia um desejo muito forte também de eu fazer meu próprio material de forma que fosse fácil de entender e com muitos exercícios para praticar e entender a teoria.

A primeira tentativa foi baseada no PostgreSQL 8.4, cuja estrutura era a tradução da documentação oficial conforme o assunto abordado, mas com uma diagramação ainda bem insatisfatória.

Nesta obra ainda a estrutura é fortemente baseada na tradução da documentação oficial, mas com diagramação bem mais moderna e fácil de entender. Claro que muitas coisas há muito de minhas próprias palavras, porém o intuito desta obra é trazer uma abordagem prática com exemplos fáceis de serem executados.

Esta obra não tem pretensão alguma de substituir a documentação oficial. A documentação oficial do PostgreSQL está entre as melhores que já tive a oportunidade de conhecer, muito bem organizada, bem como a documentação do FreeBSD.

De forma geral os assuntos tratados são diversas formas de instalação, configuração, backup, migração de versões e outros assuntos relacionados ao sistema gerenciador de banco de dados orientado a objetos PostgreSQL.

Aqueles que tiverem críticas, sugestões ou elogios, estou à disposição para que a cada versão além de conteúdo acrescentado, aperfeiçoamento de exemplos e textos bem como correção de eventuais erros.

Um grande abraço e boa leitura e bom curso, pois você escolheu o melhor SGBD do mundo! :D

Juliano Atanazio

juliano777@gmail.com

Dedicatórias

Àqueles que tanto contribuíram para o avanço da Ciência e Tecnologia, em todas suas vertentes, compartilhando seu conhecimento e assim se tornando imortais.

A todos guerreiros do Software Livre, que mantenham sempre a chama acesa pra seguir em frente e lutar (ad victoriam).

A todas as bandas de Heavy Metal e Rock in Roll que me inspiraram e me entusiasmaram durante todo o trabalho aqui apresentado (a verdadeira música é eterna).

A todos professores que se dedicam ao nobre papel de repassar seu conhecimento a pessoas interessadas em evoluir intelectualmente.

A todas as pessoas de bem que de uma forma ou de outra contribuem para um mundo melhor.

À minha família e amigos.

Ao Criador.

Sumário

1	Apresentação e Laboratório.....	18
1.1	Laboratório.....	29
1.2	Docker.....	30
2	SQL.....	34
2.1	O que é SQL.....	35
2.2	Subdivisões SQL.....	36
2.2.1	DDL.....	36
2.2.2	DML.....	36
2.2.3	DCL.....	36
2.3	Identificadores.....	37
2.4	Operadores.....	39
2.5	Comentários SQL.....	41
3	Tipos de Dados.....	42
3.1	Sobre Tipos de Dados.....	43
3.1.1	Máscaras de Tipos de Dados.....	43
3.1.2	Descobrimo o Tipo de Dado.....	43
3.2	Tipos de Dados Numéricos.....	46
3.2.1	Máscaras Para Formatos Numéricos.....	48
3.3	Tipos de Dados de Data e Hora.....	50
3.3.1	Máscaras de Data e Hora.....	52
3.4	Tipos de Dados para Texto.....	53
3.5	Nulo (Null).....	55
3.6	Booleano.....	56
3.7	Tipos de Dados de Rede.....	57
3.8	Outros Tipos.....	60
3.9	Type Casts: Conversão de Tipos.....	61
4	Interfaces.....	62
4.1	Sobre Interfaces de Acesso a um Banco de Dados.....	63
4.2	psql.....	64
4.2.1	Obtendo Ajuda de Meta Comandos do psql.....	65
4.2.2	Obtendo Ajuda de Comandos SQL Dentro do psql.....	65
4.2.3	Conectando-se a Outro Banco de Dados Dentro do psql.....	65
4.2.4	~/psqlrc.....	66
4.3	Drivers de Linguagens de Programação.....	67
4.4	pgAdmin 3.....	68
4.5	PgAdmin 4.....	69
4.6	phpPgAdmin.....	70
5	Objetos de Bancos de Dados.....	71
5.1	Sobre Objetos de Bancos de Dados.....	72
5.1.1	Criando, Modificando e Apagando Objetos.....	72
5.2	Descrição (Comentário) de Objetos.....	77
6	Tabelas.....	79
6.1	Sobre Tabelas.....	80
6.1.1	Criação de Tabelas.....	80
6.1.2	Alterando Tabelas.....	80
6.1.3	Apagando uma tabela.....	83
6.1.4	TRUNCATE - Redefinindo uma Tabela Como Vazia.....	85
6.1.5	CREATE TABLE IF NOT EXISTS / DROP TABLE IF EXISTS.....	86

6.2	Tabelas Temporárias.....	87
6.3	UNLOGGED TABLE – Tabela Não Logada.....	88
6.3.1	Qual a Utilidade de uma UNLOGGED TABLE?.....	88
6.4	Fillfactor de Tabela.....	92
7	Restrições (Constraints).....	94
7.1	Sobre Restrições (Constraints).....	95
7.2	Valor Padrão (DEFAULT).....	96
7.3	CHECK.....	97
7.4	NOT NULL.....	98
7.5	UNIQUE.....	99
7.6	A Cláusula [NOT] DEFERRABLE.....	100
7.7	Chave Primária / Primary Key.....	102
7.7.1	Chave Primária Composta.....	104
7.8	Chave Estrangeira / Foreign Key.....	105
7.8.1	Cláusulas ON DELETE / ON UPDATE.....	107
8	SELECT – Consultas (Queries).....	109
8.1	Sobre SELECT.....	110
8.2	DISTINCT.....	112
8.3	LIMIT e OFFSET.....	113
8.4	A Cláusula WHERE.....	114
8.5	A Cláusula ORDER BY.....	119
8.6	CASE: Um IF Disfarçado.....	121
8.7	A Função coalesce.....	123
8.8	A Função nullif.....	124
8.9	As Funções greatest e least.....	125
8.10	O Comando TABLE.....	126
9	DML: Inserir, Alterar e Deletar Registros.....	127
9.1	Sobre INSERT.....	128
9.1.1	INSERT Múltiplo.....	129
9.2	Dollar Quoting.....	131
9.3	UPDATE – Alterando Registros.....	133
9.3.1	Atualizando Mais de Um Campo.....	133
9.4	DELETE – Removendo Registros.....	134
9.5	A Cláusula RETURNING.....	135
9.5.1	RETURNING com INSERT.....	135
9.5.2	RETURNING com UPDATE.....	135
9.5.3	RETURNING com DELETE.....	136
9.6	“UPSERT”: INSERT ON CONFLICT.....	137
9.6.1	Sobrescrevendo Registros com UPSERT.....	138
10	Novas Tabelas a Partir de Dados.....	141
10.1	Sobre.....	142
10.2	SELECT INTO.....	143
10.3	CREATE TABLE AS.....	144
10.4	CREATE TABLE ... (like ...)......	145
11	Conjuntos.....	147
11.1	Sobre Conjuntos.....	148
11.2	União (UNION).....	150
11.3	Intersecção (INTERSECT).....	151
11.4	Diferença (EXCEPT).....	152
12	Casamento de Padrões e Expressões Regulares.....	153

12.1	Sobre Casamento de Padrões e Expressões Regulares.....	154
12.2	LIKE (~~) e ILIKE (~~*).....	155
12.3	SIMILAR TO.....	158
12.4	Função substring com Três Parâmetros.....	160
12.5	Expressões Regulares POSIX.....	161
13	Subconsultas.....	163
13.1	Sobre Subconsultas.....	164
13.1.1	Subconsulta no WHERE.....	164
13.1.2	Subconsulta no SELECT.....	165
13.1.3	Subconsulta no FROM.....	165
13.1.4	EXISTS.....	165
13.1.5	IN e NOT IN.....	166
13.1.6	ANY ou SOME.....	167
13.1.7	ALL.....	168
13.1.8	Comparação de Linha Única.....	168
13.2	Subconsultas Laterais.....	169
13.3	Subconsultas em INSERT.....	170
13.4	Subconsultas em UPDATE.....	172
13.4.1	Atualizar Campo por Resultado de Consulta.....	172
13.4.2	Atualizar Campo Copiando o Restante da Linha.....	172
13.4.3	Atualizar uma Tabela Através de Outra Tabela.....	173
13.5	Subconsultas em DELETE.....	174
14	Junções (Joins).....	176
14.1	Sobre Junções.....	177
14.2	NATURAL JOIN (Junção Natural).....	178
14.3	INNER JOIN (Junção Interna).....	179
14.4	OUTER JOIN (Junção Externa) - Definição.....	180
14.5	LEFT OUTER JOIN (Junção Externa à Esquerda).....	181
14.6	RIGHT OUTER JOIN (Junção Externa à Direita).....	183
14.7	FULL OUTER JOIN (Junção Externa Total).....	185
14.8	SELF JOIN (Auto-Junção).....	187
14.9	CROSS JOIN (Junção Cruzada).....	189
14.10	UPDATE com JOIN.....	190
14.11	DELETE com JOIN.....	191
15	Funções Built-ins.....	192
15.1	Sobre Funções.....	193
15.2	Funções Matemáticas.....	194
15.3	Funções de Data e Hora.....	195
15.3.1	extract(...).....	195
15.4	Funções de Strings.....	196
15.4.1	to_char().....	197
15.5	Funções de Sistema.....	199
16	Agrupamentos de Dados.....	200
16.1	Sobre Agrupamentos de Dados.....	201
16.2	Funções de Agregação (ou de Grupos de Dados).....	202
16.3	GROUP BY.....	204
16.4	A Cláusula HAVING.....	205
16.5	A Cláusula FILTER.....	206
17	SEQUENCE – Sequência.....	208
17.1	Sobre Sequência.....	209

17.2	Funções de Manipulação de Sequência.....	211
17.3	Usando Sequências em Tabelas.....	212
17.4	Alterando uma Sequência.....	214
17.5	Apagando uma Sequência.....	215
17.6	TRUNCATE para Reiniciar Sequências de uma Tabela.....	216
17.7	Colunas Identity.....	219
17.7.1	GENERATED BY DEFAULT vs GENERATED ALWAYS.....	221
17.7.2	TRUNCATE em Tabelas com Coluna Identity.....	223
18	UUID.....	225
18.1	Sobre UUID.....	226
18.2	O Módulo uuid-oss.....	227
19	VIEW – Visão.....	230
19.1	Sobre Visão.....	231
19.2	Visões Materializadas.....	234
20	Cursores.....	238
20.1	Sobre Cursores.....	239
20.1.1	Parâmetros.....	239
20.2	FETCH – Recuperando Linhas de um Cursor.....	241
20.3	MOVE – Movendo Cursores.....	243
20.4	CLOSE – Fechando Cursores.....	244
20.5	Apagando e Atualizando a Linha Onde o Cursor Aponta.....	245
21	BLOB.....	247
21.1	Sobre BLOB.....	248
21.2	Removendo BLOBs.....	250
22	Bytea.....	251
22.1	Sobre Bytea.....	252
23	COPY.....	256
23.1	Sobre COPY.....	257
23.2	Formato CSV.....	262
23.3	COPY Remoto.....	263
24	Range Types - Tipos de Intervalos.....	265
24.1	Sobre Tipos de Intervalos.....	266
24.1.1	Simbologia de Limites de Intervalos.....	266
24.1.2	Built-in Range Types.....	266
24.1.3	Extração de Limites Superior e Inferior.....	268
24.1.4	Contenção: O Operador Contém @>.....	268
24.1.5	Contenção: O Operador Contido <@.....	269
24.1.6	Overlaps - Sobreposições.....	269
24.1.7	Intersecção.....	269
24.1.8	Intervalo Vazio.....	270
24.1.9	Sem Limites (Mínimo, Máximo e Ambos).....	271
24.1.10	Aplicação de Conceitos.....	272
25	Tipos de Dados Criados por Usuários.....	274
25.1	Sobre Tipos de Dados Criados por Usuários.....	275
25.1.1	Tipo Concha (Shell Type).....	276
25.2	Tipos Compostos (Composite Types).....	277
25.3	Tipos Enumerados (Enumerated Types).....	279
25.4	Tipos por Faixa (Range Types).....	280
25.5	Tipos Base (Base Types).....	281
26	DOMAIN (Domínio).....	282

26.1	Sobre Domínio.....	283
26.2	DOMAIN vs TYPE.....	284
27	SCHEMA (Esquema).....	286
27.1	Sobre Esquemas.....	287
27.2	Caminho de Busca de Esquema - Schema Search Path.....	290
28	MVCC.....	291
28.1	Sobre MVCC.....	292
28.2	Transação.....	292
28.2.1	Utilidade de uma transação.....	293
28.3	Fenômenos Indesejados em Transações.....	293
28.4	Níveis de Isolamento.....	294
28.5	O Conceito de ACID.....	295
28.6	Travas.....	300
28.7	SAVEPOINT – Ponto de Salvamento.....	302
28.7.1	RELEASE SAVEPOINT.....	304
28.8	SELECT ... FOR UPDATE.....	306
29	PREPARE.....	307
29.1	Sobre PREPARE.....	308
30	PREPARE TRANSACTION.....	311
30.1	Sobre PREPARE TRANSACTION.....	312
31	Arrays.....	315
31.1	Sobre Arrays.....	316
31.2	Arrays PostgreSQL.....	317
31.3	Inserindo Valores de Array.....	318
31.3.1	Implementando Limites em Arrays.....	321
31.4	Consultas em Arrays.....	325
31.4.1	Slicing – Fatiamento de Vetores.....	326
31.5	Modificando Arrays.....	328
31.6	Funções de Arrays.....	329
31.7	Operadores de Arrays.....	331
31.8	Alterando Tipos para Array.....	332
32	INDEX - Índice.....	335
32.1	Sobre Índice.....	336
32.1.1	Tipos de Índices.....	336
32.1.2	Dicas Gerais.....	336
32.2	Índices B-tree.....	337
32.3	Índices Hash.....	339
32.4	Índices GiST.....	341
32.5	Índices SP-GiST.....	343
32.6	Índices GIN.....	345
32.7	Índices BRIN.....	347
32.7.1	Páginas por Faixa.....	350
32.8	Índices Compostos.....	352
32.9	Índices Parciais.....	353
32.10	Fillfactor – Fator de Preenchimento.....	355
32.11	Reconstrução de Índices – REINDEX.....	359
32.12	CLUSTER – Índices Clusterizados.....	361
32.13	Excluindo um Índice.....	364
32.13.1	Antes de Excluir um Índice.....	364
33	Modelos de Banco de Dados – Templates.....	365

33.1	Sobre Templates.....	366
34	Herança.....	369
34.1	Sobre Herança de Tabelas.....	370
34.2	Herança Múltipla.....	377
35	Relacionamentos.....	379
35.1	Cardinalidade.....	380
35.1.1	Simbologia.....	381
35.1.2	Cardinalidade Mínima e Cardinalidade Máxima.....	381
35.2	Relacionamento 1:1 – Um para Um.....	382
35.2.1	Relacionamento (0, 1):(0, 1).....	382
35.2.2	Relacionamento (1, 1):(0, 1).....	385
35.2.3	Relacionamento (0, 1):(1, 1).....	388
35.2.4	Relacionamento (1, 1):(1, 1).....	391
35.2.5	Cardinalidade 1:1 Como Estratégia de Particionamento.....	394
35.3	Relacionamento 1:n – Um para Muitos.....	396
35.3.1	Relacionamento (0, 1):(0, n).....	396
35.3.2	Relacionamento (1, 1):(0, n).....	399
35.4	Relacionamento n:n – Muitos para Muitos.....	401
35.5	Relacionamento Ternário.....	404
36	Particionamento de Tabelas.....	406
36.1	O que é Particionamento de Tabelas.....	407
36.1.1	Benefícios de Particionamento de Tabelas.....	407
36.1.2	Partição Padrão.....	408
36.1.3	Dicas e Boas Práticas de Particionamento.....	408
36.2	Tipos de Particionamento.....	409
36.2.1	Range.....	409
36.2.1.1	List.....	412
36.2.1.2	Hash.....	414

A Apostila

- Sobre a Apostila
- Créditos de Softwares Utilizados

Sobre a Apostila

Para um melhor entendimento, esta apostila possui alguns padrões, tais como:

Comando a Ser Digitado

Cliente de modo texto do PostgreSQL:

```
$ psql
```

Consulta envolvendo condição de filtragem:

```
> SELECT campo1, campo2 FROM tabela WHERE campo3 = 7;
```

São de três tipos, sendo que o sinal que precede o comando indica sua função, que conforme ilustrados acima são respectivamente:

- **\$** Cifrão: Comando executado no shell do sistema operacional que não precisa ser dados como usuário *root* do sistema;
- **#** Sustenido: Comando executado no shell do sistema operacional que é necessário ser *root* do sistema para poder fazê-lo;
- **>** Sinal de Maior: Comando executado em um shell específico de uma aplicação. Comandos pertinentes à própria aplicação e não ao shell do sistema operacional. Às vezes determinada parte do código pode ficar destacada em negrito quando for exposto um novo tema.

Texto Impresso em Tela

	<i>total</i>	<i>used</i>	<i>free</i>	<i>shared</i>	<i>buffers</i>	<i>cached</i>
Mem:	8032	3518	4514	0	1	1866
-/+ buffers/cache:	1650	6382				
Swap:	1906	0	1906			

É um texto que aparecem na tela, são resultantes de um determinado comando. Tais textos podem ser informativos, de aviso ou mesmo algum erro que ocorreu.

Arquivos ou Trechos de Arquivos

```
search mydomain.com
nameserver 10.0.0.200
nameserver 10.0.0.201
```

São demonstrações de como um arquivo deve ser alterado, como deve ser ou partes do próprio.

Se tiver três pontos seguidos separados por espaços significa que há outros

conteúdos que ou não importam no momento ou que há uma continuação (a qual é anunciada).

Aviso

Aviso:

O mal uso deste comando pode trazer consequências indesejáveis.

Precauções a serem tomadas para evitar problemas.

Observação

Obs.:

O comando anterior não necessita especificar a porta se for usada a padrão (5432).

Observações e dicas sobre o assunto tratado.

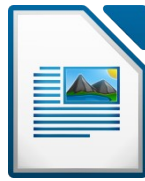
Créditos de Softwares Utilizados

Software utilizados para elaboração desta obra.

Editor de Texto



LibreOffice
The Document Foundation



LibreOffice Writer

www.libreoffice.org

Editor de Gráficos Vetoriais



INKSCAPE

Inkscape

www.inkscape.org

Editor de Imagens



gimp

GIMP

www.gimp.org

Sistema Operacional



Linux

www.linux.com

Distribuições



Ubuntu

www.ubuntu.com



Linux Mint

www.linuxmint.com

1 Apresentação e Laboratório

- Objetivo
- Cenário do Curso
- Sobre o PostgreSQL
- Laboratório
- Docker

Objetivo

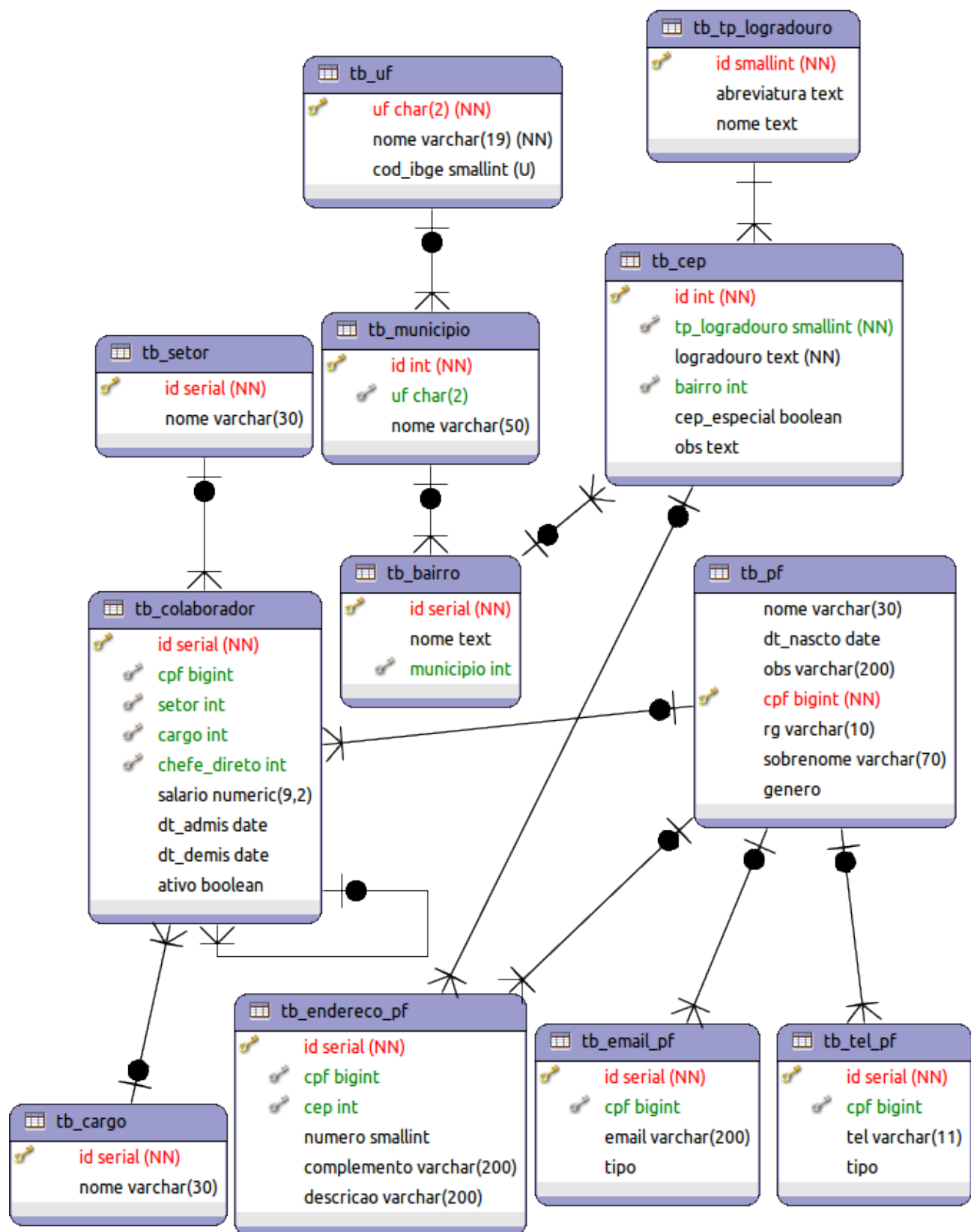
Este curso visa capacitar seus alunos a utilizar a linguagem SQL, que é a linguagem padrão para bancos de dados relacionais.

Serão vistas boas práticas de modelagem de banco de dados, a fim de construir bases de dados bem estruturadas.

Cenário do Curso

Os exercícios desta apostila, em sua maioria, terão como base um banco de dados de uma empresa fictícia, a “Chiquinho da Silva S. A.”.

Tal banco conta com alguns cadastros, cujo DER (Diagrama Entidade-Relacionamento), pode ser conferido na figura da próxima página:



Sobre o PostgreSQL

O que é o PostgreSQL

PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional baseado no POSTGRES, Versão 4.2, desenvolvido na Universidade da Califórnia no Departamento de Ciências da Computação em Berkeley, o qual foi pioneiro em muitos conceitos que vieram a estar disponíveis em alguns bancos de dados comerciais mais tarde.

O PostgreSQL é um descendente open-source do código original de Berkeley code.

Suporta uma grande parte do padrão SQL standard e oferece muitas características modernas:

- Consultas complexas;
- Chaves estrangeiras (foreign keys);
- Gatilhos (triggers);
- Visões (views);
- Integridade transacional.

O PostgreSQL pode também ser estendido pelo usuário para muitos propósitos, por exemplo adicionando novos:

- Tipos de dados;
- Funções;
- Operadores;
- Funções agregadas;
- Métodos de indexação;
- Linguagens procedurais.

Devido à sua licença liberal, o PostgreSQL pode ser usado, modificado, e distribuído por qualquer um gratuitamente para qualquer propósito, seja privado, comercial, ou acadêmico.

O PostgreSQL é um banco de dados objeto-relacional (nada a ver com linguagens de programação orientadas a objetos), em que cada coisa criada é tratada como um objeto, tais como bancos de dados, tabelas, views, triggers, etc.

Os objetos podem ter relacionamento entre si.

História do PostgreSQL

Dadas suas características poderosas e avançadas, como uma peça valiosa de software se tornou livre e com seu código-fonte aberto? Assim como muitos outros importantes projetos de código aberto, a resposta começa na Universidade da Califórnia, em Berkeley (UCB).

O PostgreSQL, originalmente chamado Postgres, foi criado por um professor da UCB do curso de Ciências da Computação chamado Michael Stonebraker, que passou a ser o CTO da Informix Corporation. Stonebraker começou o Postgres em 1986 como um projeto de continuação de seu antecessor, o Ingres, que agora pertence à Computer

Associates.

O nome Postgres é uma referência ao seu antecessor (algo como “após o Ingres”).

O Ingres foi desenvolvido em 1977 até 1985, que tinha sido um exercício de como criar um sistema de banco de dados e de acordo com a clássica teoria de SGBDR (Sistema Gerenciador de Banco de Dados Relacional).

O Postgres, desenvolvido entre 1986 a 1994, foi um projeto destinado a abrir novos caminhos nos conceitos de banco de dados, tais como a exploração de tecnologias “objeto-relacional”.

Stonebraker e seus estudantes de graduação desenvolveram o Postgres por oito anos. Durante esse tempo, o Postgres introduziu *rules*, *procedures*, *time travel*, tipos extensíveis com índices e conceitos objeto-relacionais.

O Postgres foi mais tarde comercializado para se tornar o Illustra que posteriormente foi comprado pela Informix e integrado dentro de seu servidor universal.

A Informix foi adquirida pela IBM em 2001 por um bilhão de dólares.

Em 1995, dois estudantes Ph. D. do laboratório de Stonebraker, Andrew Yu e Jolly Chen substituíram a linguagem de consulta Postgres’ POSTQUEL por um subconjunto estendido de SQL. Eles renomearam o sistema para Postgres95. Em 1996, o Postgres95 se afastou da academia e iniciou uma nova vida, no mundo *open source*, quando um grupo de desenvolvedores dedicados, fora de Berkeley, viu a promessa do sistema e se dedicaram à continuação de seu desenvolvimento. Com grandes contribuições de tempo e habilidade. Com trabalho e conhecimento técnico, esse grupo global de desenvolvimento transformou o Postgres radicalmente. Durante os próximos oito anos, eles trouxeram consistência e uniformidade para a base do código, criaram testes detalhados de regressão para garantia de qualidade, criaram listas de discussão para relatórios de *bugs*, consertaram vários *bugs*, adicionaram incríveis novos recursos, e arredondaram o sistema preenchendo várias lacunas como documentação para desenvolvedores e usuários. Os frutos desse trabalho foi um novo banco de dados que ganhou uma reputação para a estabilidade de uma rocha sólida. Com o início dessa nova vida no mundo *open source*, com muitos novos recursos e melhorias, o sistema de banco de dados teve seu nome atual: PostgreSQL, (“Postgres” ainda é usado como um apelido de fácil pronúncia).

O PostgreSQL começou na versão 6.0, dando crédito a seus muitos anos anteriores de desenvolvimento. Com a ajuda de centenas de desenvolvedores ao redor do mundo, o sistema foi alterado e melhorado em quase todas as áreas. Nos próximos quatro anos (versões de 6.0 a 7.0), grandes melhorias e novos recursos foram feitos como:

- **Controle de Concorrência Multiversão (Multiversion Concurrency Control – MVCC)**

O bloqueio no nível de tabela foi substituído por um sofisticado controle de concorrência multiversão, que permite a quem lê a continuar a ler dados consistentes durante uma atividade de escrita e habilita *backups online* (hot) enquanto o banco de dados está rodando.

- **Importantes Recursos SQL**

Muitas melhorias SQL foram feitas: subconsultas, *defaults*, *constraints* (restrições), *primary keys* (chaves primárias), identificadores entre aspas, coerção literal de *string*, *type casting*, e entrada de inteiros binários e hexadecimais entre outros.

- **Melhoramentos em Tipos Embutidos (built-in)**

Novos tipos nativos foram adicionados incluindo uma ampla gama de tipos de data/tempo e tipos geométricos adicionais.

- **Velocidade**

Maior velocidade e performance foram feitas, aumentando em 20 a 40%, e inicialização do backend foi diminuída em 80%.

Os quatro anos seguintes (versões de 7.0 a 7.4) trouxe o *Write-Ahead Log* (WAL), esquemas (*schemas*) SQL, consultas preparadas, *OUTER JOINS*, consultas complexas, sintaxe SQL92 JOIN, TOAST, suporte a IPv6, padrão SQL de informações de esquema, indexação full-text, auto-vacuum, linguagens procedurais Perl/Python/TCL, suporte melhorado a SSL, uma revisão de otimizador, informações de estatísticas de banco de dados, segurança adicionada, funções de tabela, melhoramentos de log, significantes aumentos de velocidade entre outras coisas. Uma pequena medida de desenvolvimento intensivo do PostgreSQL se reflete nas notas de lançamento. Hoje, a base de usuários do PostgreSQL é maior do que nunca e inclui um considerável grupo de grandes corporações que o usam em ambientes exigentes. Algumas dessas empresas como Afilias e Fujitsu tem feito contribuições significantes ao desenvolvimento do PostgreSQL. E, fiel às suas raízes, ele continua a melhorar em sofisticação e performance, agora mais do que nunca.

A versão 8.0 do PostgreSQL foi a mais aguardada no mercado de bancos de dados corporativos, trazendo novos recursos como tablespaces, stored procedures Java, *Point In Time Recovery* (PITR), e transações aninhadas (*savepoints*). Junto com o lançamento dessa versão veio a característica mais aguardada; uma versão nativa para Windows.

Muitas organizações, agências governamentais e empresas usam o PostgreSQL.

São encontradas instalações em organizações como ADP, Cisco, NTT Data, NOAA, *Research In Motion*, Serviço Florestal dos EUA e a Sociedade Química Americana. hoje, é raro encontrar uma grande corporação ou agência de governo que não usa o PostgreSQL em pelo menos um departamento. Se teve um tempo para considerar seriamente utilizar o PostgreSQL para agilizar sua aplicação ou negócio, que seja agora! ;)

Como se Fala e Como se Escreve

Uma dúvida comum ao PostgreSQL é seu nome. As formas corretas são as seguintes:

- Postgres, pronuncia-se “postígres” (sim, o “s” é pronunciado!);
- PostgreSQL, pronuncia-se “postgres és quiu el”.

Nunca, jamais, em hipótese nenhuma escrever “**postgree**” ou dizer “**postgri**”.

Infelizmente ainda há fontes na Internet com o nome do Postgres escrito erroneamente, o que leva muita gente também a falar errado.

Limites do PostgreSQL

Limite	Valor
Tamanho máximo de um banco de dados	Ilimitado
Tamanho máximo de uma tabela	32 TB
Tamanho máximo de uma linha (registro)	1.6 TB
Tamanho máximo de um campo (coluna)	1 GB
Número máximo de linhas por tabela	Ilimitado
Número máximo de colunas por tabela	250 - 1600 dependendo do tipo de coluna
Número máximo de índices por tabela	Ilimitado

Suporte ao PostgreSQL

Informações gerais podem ser encontradas no link:

<https://www.postgresql.org/support/>

Sobre Software Livre em Geral

Infelizmente, ainda hoje existe um mito a respeito de qualquer software livre. É o mito expresso pelas perguntas: “Se der algum problema? Quem se responsabiliza? Se eu descobrir um bug o mesmo será rapidamente corrigido?”.

Em softwares proprietários, cujo código-fonte é fechado, quem dá suporte é a empresa desenvolvedora. Isso passa uma ideia (falsa) de segurança para quem adquire uma solução proprietária. Há casos (não raros) de softwares proprietários que tem bug descoberto e divulgado por usuários a anos sem ter sido corrigido. Isso mostra a outra face de uma pseudo-segurança dita, pois se o código-fonte está nas mãos de um círculo limitado de pessoas. Isso sem mencionar também que devido à solução proprietária ser fechada não se sabe o que roda além do que é citado para quem o adquire.

Uma solução em Software Livre prima por liberdade, privacidade e segurança.

Há muitas empresas que ajudam (financeiramente ou fornecendo mão de obra) e que veem claramente vantagens para si em colaborar com um software mantido pela Comunidade de Software Livre.

Sobre bugs descobertos em uma aplicação de código-fonte aberto, assim que descobertos, os mesmos são reportados à Comunidade que geralmente soluciona em pouquíssimo tempo.

Quanto à questão de suporte a um software de código-fonte temos as opções de buscar ajuda na Comunidade ou contratando uma empresa de consultoria.

Suporte da Comunidade

- **Sites Oficiais:**

- Global: <https://www.postgresql.org/>
- Wiki: <https://wiki.postgresql.org/>
- Brasileiro: <https://www.postgresql.org.br/>

- **Documentação:**

A documentação do PostgreSQL é muito rica e tem as opções em arquivo PDF conforme a versão (<https://www.postgresql.org/docs/manuals/>) ou on-line (<https://www.postgresql.org/docs/>).

- **Listas de Discussão:**

É um meio rápido de resolução de problemas, contando com uma comunidade forte e vibrante temos as listas:

- Internacional: <https://lists.postgresql.org>
- Brasileira: <https://listas.postgresql.org.br/>

- **Blogs**

São vários os blogs ao redor do planeta sobre PostgreSQL, mas para facilitar temos blogs oficiais da comunidade que aglutinam posts de outros blogs, de forma a termos um conhecimento diversificado através de tutoriais e artigos divulgados nesses.

- Planet PostgreSQL: <http://planet.postgresql.org/>
- Planeta PostgreSQL: <https://planeta.postgresql.org.br/>

- **Outros:**

Fóruns, sites especializados em software livre, grupos de mensageiros de celular, canais IRC e etc são formas alternativas de buscar ajuda e conhecimento sobre PostgreSQL.

Suporte Comercial

Há várias empresas que provêm suporte ao PostgreSQL no mundo inteiro, que podem ser encontradas por região ou especificamente para hosting [2]:

[1] https://www.postgresql.org/support/professional_support/

[2] https://www.postgresql.org/support/professional_hosting/

Derivações do PostgreSQL

Devido à licença permissiva do PostgreSQL pode-se derivá-lo mantendo o código-fonte aberto ou mesmo podendo fechá-lo.

Na Wiki [1] do site global há uma lista de vários desses produtos.

[1] https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases

Segue abaixo alguns exemplos:



Citus / CitusDB

Citus Data

<https://www.citusdata.com/>

Licença: AGPL 3.0

Descrição:

Faz escalonamento horizontal através de commodity servers usando sharding e replicação.

Seu motor de consulta paraleliza consultas SQL vindas desses servidores para possibilitar respostas em tempo real em grandes conjunto de dados.

O Citus estende o PostgreSQL em vez de ser um fork dele, que dá a desenvolvedores e empresas o poder e familiaridade com o PostgreSQL tradicional.



Postgres-XL

PGXLDG

<http://www.postgres-xl.org/>

Licença: BSD

Descrição:

Projetado para ambientes BI e de análise Big Data, tem capacidade MPP (Massively Parallel Processing: Processamento Paralelo em Massa) e com recurso de sharding que distribui os dados através de seus nós.



Greenplum Database

Greenplum

<http://greenplum.org/>

Licença: Apache 2

Descrição:

O Greenplum Database é um SGBD desenhado para data warehousing, processamento analítico em larga escala.

Tem MPP (Massively Parallel Processing: Processamento Paralelo em Massa).

Faz particionamento automático de dados e roda consultas paralelas, o que permite a um cluster de servidores operar como uma única base de dados em um super computador executando dezenas ou centenas de vezes mais rápido do que um banco de dados tradicional.

Suporta SQL, processamento paralelo MapReduce e volumes de dados gigantescos (de centenas de gigabytes a centenas de terabytes).



Netezza

IBM

<https://www.ibm.com/software/data/netezza/>

Licença: Proprietária

Descrição:

Appliance baseada no motor do PostgreSQL, voltada para BI, análise preditiva e *data warehousing*.



Vertica

HP

www.vertica.com

Licença: Proprietária

Descrição:

SGBD SQL analítico para demandas Big Data e BI de alta performance e escalabilidade.



Postgres Advanced Server

Enterprise DB

<http://www.enterprisedb.com>

Licença: Proprietária

Descrição:

Uma solução proprietária baseada no código-fonte original do PostgreSQL que mantém total compatibilidade com o mesmo. Também tem compatibilidade com o Oracle, que devido ao custo de sua licença ser muito menor resulta em redução de custos significativa em casos que haja a necessidade do SGBD ser apenas o Oracle. Outra característica muito interessante é a replicação multi-master nativa. O Postgres Advanced Server é certificado pela Hortonworks para integração com clusters Hadoop.

1.1 Laboratório

Para os exercícios das partes práticas é necessário um ambiente preparado adequadamente.

O ambiente é composto de uma base de dados de exemplo que está em um repositório no [GitHub](https://github.com/juliano777/db_empresa.git)*, que contém uma imagem PNG com o DER (Diagrama Entidade-Relacionamento) e dois arquivos SQL que foram separados em estrutura e os dados da base.

* https://github.com/juliano777/db_empresa.git



1.2 Docker

Docker é uma plataforma de virtualização em contêiner, muito leve, baixo overhead e extremamente versátil.

Docker é a plataforma sugerida para a parte prática e para isso será utilizada uma imagem** específica para o ambiente.

** https://cloud.docker.com/repository/docker/juliano777/postgres_br



Instalação do Docker

```
$ sudo su -c 'wget -O - get.docker.io | bash'
```

Adicionar o usuário (deve ser um sudoer) ao grupo docker e sai:

```
$ sudo usermod -aG docker `whoami` && logout
```

Habilita e inicia o serviço Docker imediatamente:

```
$ sudo systemctl enable --now docker.service
```

Criação do contêiner do curso:

```
$ docker run -itd --name curso_pgsql_sql_basico \
  --hostname postgres juliano777/postgres_br
```

Instalação do git no contêiner:

```
$ docker exec -itu root curso_pgsql_sql_basico \
  sh -c 'apt update && apt install -y git && apt clean'
```

Clonagem do repositório da base de dados:

```
$ docker exec -itu postgres curso_pgsql_sql_basico \  
  sh -c 'git -C ~/ clone https://github.com/juliano777/db_empresa.git'
```

Execução do script SQL com comandos DDL para criação da estrutura da base:

```
$ docker exec -itu postgres curso_pgsql_sql_basico \  
  sh -c 'psql -f ~/db_empresa/db_empresa-schema.sql'
```

Execução do script SQL com comandos DML para inserir os dados da base:

```
$ docker exec -itu postgres curso_pgsql_sql_basico \  
  sh -c 'psql -f ~/db_empresa/db_empresa-data.sql db_empresa'
```

Variável de ambiente de comandos SQL para teste:

```
$ SQL='\  
  SELECT version() AS "Versão do PostgreSQL";\  
  SELECT current_database() AS "Base de Dados";\  
'
```

Execução do teste:

```
$ echo ${SQL} | docker exec -iu postgres curso_pgsql_sql_basico \  
  sh -c "psql db_empresa"
```

```
-----  
Versão do PostgreSQL  
-----  
PostgreSQL 12beta2 on x86_64-pc-linux-gnu, compiled by gcc (Debian 8.3.0-6) 8.3.0, 64-bit  
(1 row)  
  
Base de Dados  
-----  
db_empresa  
(1 row)
```

PostgreSQL

—

SQL Básico

2 SQL

- O que é SQL
- Subdivisões SQL
- Identificadores
- Operadores
- Comentários SQL

2.1 O que é SQL

Structured Query Language – Linguagem Estruturada de Consulta, é a linguagem usada nos SGBDs¹ por padrão, no entanto cada um tem suas particularidades dentro da própria linguagem, tendo implementações diferentes.

O mesmo objetivo pode ser feito de formas SQL diferentes de um SGBD pra outro.

Assim como em linguagens de programação “comuns”, existem palavras reservadas, as quais não podem ser usadas como identificadores.

Cada comando SQL é finalizado com ponto e vírgula (;).

¹ Originalmente da sigla em inglês DBMS (Data Base Management System): Sistema Gerenciador de Banco de Dados; que é um conjunto de aplicativos, cuja função é gerenciar uma base dados. Exemplos: PostgreSQL, Oracle, DB2, MySQL, Firebird SQL, MS SQL Server, Sybase, etc.

2.2 Subdivisões SQL

A linguagem SQL tem algumas divisões, que facilitam o entendimento da mesma, categorizando seus comandos. Sendo que as mais conhecidas, que serão explicadas a seguir, são: DDL, DML e DCL.

2.2.1 DDL

Data Definition Language – Linguagem de Definição de Dados, é a parte da Linguagem SQL que trata, como o próprio nome diz, da definição da estrutura dos dados, cujos efeitos se dão sobre objetos. Criação de bancos de dados, tabelas, views, triggers, etc...

Exemplos: `CREATE` (criação), `ALTER` (alteração), `DROP` (remoção), etc.

2.2.2 DML

Data Manipulation Language – Linguagem de Manipulação de Dados, é a parte da Linguagem SQL que não altera a estrutura e sim os registros de uma base de dados, cujos efeitos se dão sobre registros. São comandos que fazem consultas, inserem, alteram ou apagam registros.

Exemplos: `SELECT` (consulta), `INSERT` (inserção), `UPDATE` (alteração), `DELETE` (remoção), etc.

2.2.3 DCL

Data Control Language – Linguagem de Controle de Dados, é a parte da linguagem SQL referente ao controle de acesso a objetos por usuários e seus respectivos privilégios.

Os principais comandos SQL são:

- `GRANT`: Garante direitos a um usuário;
- `REVOKE`: Revoga (retira) direitos dados a um usuário.

Os direitos dados a um usuário podem ser: `ALL`, `CREATE`, `EXECUTE`, `REFERENCES`, `SELECT`, `TRIGGER`, `USAGE`, `CONNECT`, `DELETE`, `INSERT`, `RULE`, `TEMPORARY`, `UPDATE`, etc.

2.3 Identificadores

Para nomearmos identificadores devemos usar como primeiro caractere letras ou “_” (*underline*).

Um exemplo de uma nomenclatura correta de um identificador:

```
> CREATE DATABASE abc;
```

Um exemplo de uma nomenclatura **incorreta** de um identificador:

```
> CREATE DATABASE 123;
```

```
ERROR:  syntax error at or near "123"  
LINE 1: CREATE DATABASE 123;  
                        ^
```

Tentativa de criação de objeto cujo identificador tem letras maiúsculas:

```
> CREATE DATABASE ABC;
```

O mesmo seria considerado como “abc” e facilmente apagado como se tivesse sido criado com letras minúsculas.

Apagando um objeto:

```
> DROP DATABASE abc;
```

Há uma maneira de fazer com que os identificadores criados sejam “*case sensitive*”, para isso devemos criar os nomes de identificadores entre aspas dupas (“ ”). Inclusive, pode-se até criar identificadores iniciando com caracteres numéricos (o que não é uma boa prática):

```
> CREATE DATABASE "Abc";
```

Uma prática não recomendável:

```
> CREATE DATABASE "123tEste";
```

Listando as bases de dados existentes via meta comando psql:

```
> \l
```

Name	Owner	Encoding	Collate	Ctype	Access privileges
123tEste	postgres	UTF8	pt_BR.UTF-8	pt_BR.UTF-8	
Abc	postgres	UTF8	pt_BR.UTF-8	pt_BR.UTF-8	
postgres	postgres	UTF8	pt_BR.UTF-8	pt_BR.UTF-8	
template0	postgres	UTF8	pt_BR.UTF-8	pt_BR.UTF-8	=c/postgres +
					postgres=CtC/postgres
template1	postgres	UTF8	pt_BR.UTF-8	pt_BR.UTF-8	=c/postgres +
					postgres=CtC/postgres

Listando as bases via catálogo:

```
> SELECT datname FROM pg_database;
```

```
datname
-----
template1
template0
postgres
Abc
123tEste
```

Obs.:

Tabelas de Sistema: Muito úteis para extrairmos informações sobre objetos da instalação do banco de dados.

No último comando foi usada a tabela de sistema pg_database.

Como dito anteriormente, para manipular objetos com nomes fora do padrão, seu identificador deve estar entre aspas:

```
> DROP DATABASE "Abc";
```

```
> DROP DATABASE "123tEste";
```

2.4 Operadores

Operador	Descrição
.	Separador de nome de tabela/coluna
::	Typecast estilo PostgreSQL
[]	Seleção de elemento de array
-	Menos unário
^	Exponenciação
/	Raiz quadrada
/	Raiz cúbica
@	Valor absoluto
*	Multiplicação
/	Divisão
%	Resto de divisão
+	Adição
-	Subtração
IS	IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL ou IS NULL	Teste para nulo
NOTNULL ou NOT NULL	Teste para não nulo
IN	"em"

Operador	Descrição
BETWEEN	"entre"
OVERLAPS	Sobreposição de intervalo de tempo
LIKE ou ~~	Comparação de string case sensitive
ILIKE ou *~~	Comparação de string case insensitive
SIMILAR	Comparação de string por similaridade
<	Menor que
>	Maior que
=	Igualdade, atribuição
<> ou !=	Diferente
>=	Maior ou igual que
<=	Menor ou igual que
NOT	"NÃO" lógico
AND	"E" lógico
OR	"OU" lógico
	Concatena strings
@>	Contém
<@	Contido

Operador de multiplicação:

```
> SELECT 3 * 5;
```

```
?column?
-----
      15
```

Divisão inteira:

```
> SELECT 7 / 2;
```

```
?column?
-----
       3
```

Divisão com ponto flutuante:

```
> SELECT 7 / 2.0;
```

```
?column?
-----
3.5000000000000000
```

Divisão com ponto flutuante com conversão de dado do divisor:

```
> SELECT 7 / 2::float4;
```

```
?column?  
-----  
3.5
```

Uso do operador IN para verificar se o elemento pertence à lista:

```
> SELECT 'verde' IN ('azul', 'amarelo', 'verde');
```

```
?column?  
-----  
t
```

Uso do operador NOT para assegurar se o elemento não pertence à lista:

```
> SELECT 'vermelho' NOT IN ('azul', 'amarelo', 'verde');
```

```
?column?  
-----  
t
```


2.5 Comentários SQL

Há dois tipos de comentários; o de uma única linha e de múltiplas linhas respectivamente:

Comentário de Uma Linha:

```
> SELECT current_database(); -- Exibe a base de dados em que está conectado
```

ou

Comentário de Múltiplas Linhas:

```
> /*  
Comentários SQL  
multilinhas  
*/  
SELECT current_database();
```

Os comentários de uma única linha são feitos com -- e de múltiplas linhas são delimitados entre /* e */.

3 Tipos de Dados

- Sobre Tipos de Dados
- Tipos de Dados Numéricos
- Tipos de Dados de Data e Hora
- Tipos de Dados para Texto
- Nulo (Null)
- Booleano
- Tipos de Dados de Rede
- Outros Tipos
- Type Casts: Conversão de Tipos

3.1 Sobre Tipos de Dados

Na modelagem de uma base de dados é de extrema importância saber que tipo de dado é o ideal para uma determinada entidade.

Para uma mesma finalidade mais de um tipo pode atender, mas pode ocupar mais ou menos espaço, influenciando diretamente na forma de consumo de recursos.

Valores numéricos e booleanos (`true` / `false`) são escritos de forma pura enquanto valores de texto (strings) ou de data são envolvidos por apóstrofes.

3.1.1 Máscaras de Tipos de Dados

É um recurso utilizado para representar um dado de acordo com um padrão especificado para seu tipo.

Muito utilizado pela função `to_char`, por exemplo.

3.1.2 Descobrindo o Tipo de Dado

O PostgreSQL fornece uma função para descobrir que tipo de dado é um determinado valor, a função `pg_typeof()`:

A função `pg_typeof` retorna o OID do tipo de dado do valor que é passado a ela. Isso pode ser útil para resolução de problemas ou construção de consultas SQL dinamicamente.

A função é declarada como retorno `regtype`, que é um apelido (alias) de tipo; o que significa que ele é o mesmo como um OID para propósitos de comparação, mas exibe o nome do tipo.

Descobrir que tipo de dado é 7:

```
> SELECT pg_typeof(7);
```

```
pg_typeof  
-----  
integer
```

Descobrir que tipo de dado é '7':

```
> SELECT pg_typeof('7');
```

```
pg_typeof  
-----  
unknown
```

Retorno dado como desconhecido por não saber exatamente que tipo de dado é a string do parâmetro.

Descobrir que tipo de dado é '7' já convertendo (type cast) para char:

```
> SELECT pg_typeof('7'::char);
```

```
pg_typeof
-----
character
```

Qual o tipo de dado retornado pela função now()?:

```
> SELECT pg_typeof(now());
```

```
pg_typeof
-----
timestamp with time zone
```

Qual o tipo de dado do seguinte valor?:

```
> SELECT pg_typeof('2015-07-01');
```

```
pg_typeof
-----
unknown
```

Outro caso que não há como determinar o tipo de dado. É preciso explicitar o tipo.

Fazendo conversão de tipo de forma a tornar explícito o tipo:

```
> SELECT pg_typeof('2015-07-01'::date);
```

```
pg_typeof
-----
date
```

Que tipo vai ser o número abaixo?:

```
> SELECT pg_typeof(1.5);
```

```
pg_typeof
-----
numeric
```

Nem sempre um número inteiro vai ser um integer:

```
> SELECT pg_typeof(1500000000);
```

```
pg_typeof
-----
bigint
```

Testando um valor booleano:

```
> SELECT pg_typeof(true);
```

```
pg_typeof  
-----  
boolean
```

Qual o tipo retornado pela função type_of?

```
> SELECT pg_typeof(pg_typeof('foo'));
```

```
pg_typeof  
-----  
regtype
```

3.2 Tipos de Dados Numéricos

Nome	Alias	Faixa	Tamanho	Descrição
smallint	int2	-32768 até +32767	2 bytes	Inteiro
integer	int, int4	-2147483648 até +2147483647	4 bytes	
bigint	int8	-9223372036854775808 até +9223372036854775807	8 bytes	
numeric[(p,s)]*	decimal[(p,s)] *	sem limite	variável	Número exato com precisão customizável
real	float4	6 dígitos decimais de precisão	4 bytes	Precisão única de ponto flutuante
double precision	float8	15 dígitos decimais de precisão	8 bytes	
smallserial	serial2	1 até 32767	2 bytes	Auto-incremento inteiro
serial	serial4	1 até 2147483647	4 bytes	
bigserial	serial8	1 até 9223372036854775807	8 bytes	

* p = precisão: quantidade de dígitos (antes do ponto flutuante + depois do ponto flutuante)

s = escala: quantidade de dígitos após o ponto flutuante

Um dado numérico basicamente pode ser um inteiro ou um número decimal.

Seja qual for o valor numérico, o mesmo é expresso sem apóstrofes.

Obs.:

Os tipos seriais são pseudo-tipos. São tipos inteiros na verdade, serial2 → int2, serial4 → int4, serial8 → int8.

Quando um tipo serial é definido para um campo, seu valor é um inteiro cujo valor padrão é o próximo número de um objeto sequência.

Aviso:

Para valores monetários utilize o tipo `numeric`. Não utilize `money`, pois o mesmo é sensível à configuração `lc_monetary` e em uma eventual restauração, se o valor dessa configuração for diferente do original pode haver distorção de valores, enquanto `numeric` é mais “universal”.

Qual o tamanho (em bytes) de um valor para um número do tipo int2?:

```
> SELECT pg_column_size(1::smallint);
```

```
pg_column_size
-----
2
```

Qual o tamanho (em bytes) de um valor para um número do tipo int4?:

```
> SELECT pg_column_size(1::int);
```

```
pg_column_size
-----
4
```

Qual o tamanho (em bytes) de um valor para um número do tipo int8?:

```
> SELECT pg_column_size(1::bigint);
```

```
pg_column_size
-----
8
```

Qual o tamanho (em bytes) de um valor para um número do tipo numeric?:

```
> SELECT pg_column_size(1::numeric);
```

```
pg_column_size
-----
8
```

Qual o tamanho (em bytes) de um valor para um número do tipo real?:

```
> SELECT pg_column_size(1::real);
```

```
pg_column_size
-----
4
```

Uma simples consulta como uma calculadora:

```
> SELECT 5 + 2;
```

```
?column?
-----
7
```

Cálculo com alias:

```
> SELECT 5 + 2 AS resultado;
```

```
resultado
-----
7
```

3.2.1 Máscaras Para Formatos Numéricos

Nome	Descrição
9	Valor com o número especificado de dígitos
0	Valor com zeros à esquerda
. (ponto)	Valor com o número especificado de dígitos
, (vírgula)	Valor com o número especificado de dígitos
PR	Valor negativo entre < e >
S	Sinal preso ao número (utiliza o idioma)
L	Símbolo da moeda (utiliza o idioma)
D	Ponto decimal (utiliza o idioma)
G	Separador de grupo (utiliza o idioma)
MI	Sinal de menos na posição especificada (se número < 0)
PL	Sinal de mais na posição especificada (se número > 0)
SG	Sinal de mais/menos na posição especificada
RN [a]	Algarismos romanos (entrada entre 1 e 3999)
TH ou th	Sufixo de número ordinal
V	Desloca o número especificado de dígitos (veja as notas sobre utilização)
EEEE	Notação científica (ainda não implementada)

Converter 2009 para algarismos romanos:

```
> SELECT to_char(2009, 'RN');
```

```
to_char
-----
MMIX
```

Número 1 ordinal (padrão inglês):

```
> SELECT to_char(1, '99999th');
```

```
to_char
-----
1st
```


Número 3 ordinal (padrão inglês):

```
> SELECT to_char(3, '99999th');
```

```
to_char
-----
      3rd
```

Mudando a configuração de sessão de numeração para o padrão brasileiro:

```
> SET lc_numeric = 'pt_BR.UTF8';
```

Máscara com separadores de milhares (G) e de casas decimais (D):

```
> SELECT to_char(3148.5, '999G999G999D99');
```

```
to_char
-----
 3.148,50
```

Mudando a configuração de sessão de numeração para o padrão americano:

```
> SET lc_numeric = 'en_US.UTF8';
```

Máscara com separadores de milhares (G) e de casas decimais (D):

```
> SELECT to_char(3148.5, '999G999G999D99');
```

```
to_char
-----
 3,148.50
```

Cinco posições com zeros à esquerda:

```
> SELECT to_char(21, '00000');
```

```
to_char
-----
00021
```

Sinal à direita:

```
> SELECT to_char(-3, '999S');
```

```
to_char
-----
      3-
```

Sinal à esquerda:

```
> SELECT to_char(3, 'S999');
```

```
to_char
-----
+3
```

3.3 Tipos de Dados de Data e Hora

Nome	Alias	Descrição
date		Data de calendário (ano, mês, dia)
interval [(p)]		Intervalo de tempo
time [(p)] [without time zone]		Horas
time [(p)] with time zone	timetz	Horas fuso horário
timestamp [(p)] [without time zone]		Data e horas
timestamp [(p)] with time zone	timestamptz	Data e horas com fuso horário

Alguns tipos de data e hora aceitam um valor com uma precisão “p” que especifica a quantidade de dígitos fracionais retidos no campo de segundos. Por exemplo: Se um campo for especificado como do tipo `interval(4)`, se no mesmo for inserido um registro com valor “00:00:33.23439”, o mesmo terá seu valor arredondado para “00:00:33.2344”, devido à customização da precisão para 4 dígitos. A faixa de precisão aceita é de 0 a 6.

Função `now()`; data e hora atual em formato timestamp:

```
> SELECT now();
```

```
-----  
now  
-----  
2016-03-02 12:46:22.697306-03
```

Type cast do retorno da função `now()` extraindo a data:

```
> SELECT now()::date;
```

```
-----  
now  
-----  
2015-06-25
```

Type cast do retorno da função `now()` extraindo a hora:

```
> SELECT now()::time;
```

```
-----  
now  
-----  
12:54:10.882728
```

Type cast do retorno da função `now()` extraindo a hora sem dígitos fracionais:

```
> SELECT now()::time(0) without time zone;
```

```
-----  
now  
-----  
12:59:17
```

Type cast do retorno da função now() extraindo a hora sem dígitos fracionais com timezone (fuso horário):

```
> SELECT now()::time(0) with time zone;
```

```
      now
-----
12:59:25-03
```

Daqui a sete dias:

```
> SELECT now() + '7 days'::interval;
```

```
      ?column?
-----
2015-07-02 13:00:26.391693-03
```

Daqui a sete dias (só a data):

```
> SELECT (now() + '7 days'::interval)::date;
```

```
      date
-----
2015-07-02
```

Daqui a uma semana (só a data):

```
> SELECT (now() + '1 week'::interval)::date;
```

```
      date
-----
2015-07-02
```

Type cast da string para hora:

```
> SELECT '08:00'::time;
```

```
      time
-----
08:00:00
```

Quando será 08:00 mais 7 (sete) horas?:

```
> SELECT '08:00'::time + '7 hours'::interval;
```

```
      ?column?
-----
15:00:00
```

Type cast da string para data:

```
> SELECT '2015-12-31'::date;
```

```
      date  
-----  
2015-12-31
```

3.3.1 Máscaras de Data e Hora

Nome	Descrição
HH	Hora do dia (De 01 a 12)
HH12	Hora do dia (De 01 a 12)
HH24	Hora do dia (De 00 a 23)
MI	Minuto (De 00 a 59)
SS	Segundo (De 00 a 59)
SSSS	Segundo do dia (De 0 a 86399)
AM ou PM	Meridiano
YYYY	Ano (4 dígitos)
YY	Ano (2 dígitos)
Y	Ano (último dígito)

Nome	Descrição
BC ou AD	Era
MONTH	Nome do mês
MM	Mês (De 01 a 12)
DAY	Nome do dia da semana
DD	Dia do mês (De 01 a 31)
D	Dia da semana (De 1 a 7; Domingo = 1)
DDD	Dia do ano (De 1 a 366)
WW	Semana do ano (De 1 a 53)
Y	Ano (último dígito)
BC ou AD	Era

A partir da função now() exibir a data no formato brasileiro e o nome do dia da semana (em inglês):

```
> SELECT to_char(now(), 'DD/MM/YYYY DAY');
```

```
      to_char  
-----  
25/06/2015 THURSDAY
```

3.4 Tipos de Dados para Texto

Nome	Alias	Descrição
character(n)	char(n)	Tamanho fixo e não variável de caracteres
character varying(n)	varchar(n)	Tamanho variável de caracteres com limite
text		Tamanho variável e sem limite

Também conhecidas como “strings” e seus valores são expressos entre apóstrofos.

Uma consulta simples exibindo uma string:

```
> SELECT 'foo';
```

```
?column?  
-----  
foo
```

Exemplo de concatenação de strings:

```
> SELECT 'foo' || 'bar';
```

```
?column?  
-----  
foobar
```

Concatenando duas strings com um espaço entre elas:

```
> SELECT 'Chiquinho' || ' ' || 'da Silva';
```

```
?column?  
-----  
Chiquinho da Silva
```

String com caracteres especiais não interpretados:

```
> SELECT 'Linha1\nLinha2\nLinha3';
```

```
?column?  
-----  
Linha1\nLinha2\nLinha3
```

String com caracteres especiais interpretados:

```
> SELECT E'Linha1\nLinha2\nLinha3';
```

```
?column?  
-----  
Linha1  +  
Linha2  +  
Linha3
```

Como inserir apóstrofo em uma string:

```
> SELECT 'Copo d''água';
```

```
?column?  
-----  
Copo d'água
```

Utilizando dollar quoting:

```
> SELECT $$Copo d'água$$;
```

```
?column?  
-----  
Copo d'água
```

String com caracteres especiais de tabulação e nova linha:

```
> SELECT E'\tLinha1\nLinha2\nLinha3';
```

```
?column?  
-----  
      Linha1+  
Linha2      +  
Linha3
```

3.5 Nulo (Null)

Um valor nulo é simplesmente um valor não preenchido, o que não deve ser confundido com espaço.

Em PostgreSQL string vazia não é nula:

```
> SELECT '' IS NULL;
```

```
?column?  
-----  
f
```

Testando se nulo é nulo:

```
> SELECT NULL IS NULL;
```

```
?column?  
-----  
t
```

Testando se uma string é nula:

```
> SELECT 'x' IS NOT NULL;
```

```
?column?  
-----  
t
```

A função substring neste caso retorna um valor nulo:

```
> SELECT substring('foo', 'bar') IS NULL;
```

```
?column?  
-----  
t
```

3.6 Booleano

Nome	Alias	Descrição
boolean	bool	Valor lógico booleano (true/false)

Admite apenas dois estados “true” ou “false”. Um terceiro estado, “unknown”, é representado pelo valor nulo (`null`).

Valores literais válidos para o estado “true”: `TRUE`, `'t'`, `'true'`, `'y'`, `'yes'`, `'1'`.

Para o estado “false”, são aceitos: `FALSE`, `'f'`, `'false'`, `'n'`, `'no'`, `'0'`.

Teste lógico verdadeiro e falso: > <code>SELECT true AND false;</code> <i>?column?</i> ----- <i>f</i>	Teste lógico verdadeiro ou falso: > <code>SELECT true OR false;</code> <i>?column?</i> ----- <i>t</i>
----------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------

Teste lógico não falso: > <code>SELECT NOT false;</code> <i>?column?</i> ----- <i>t</i>	Teste lógico não verdadeiro: > <code>SELECT NOT true;</code> <i>?column?</i> ----- <i>f</i>
--------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------

Teste lógico negando o resultado de dentro do parênteses:

```
> SELECT NOT (true AND false);  
  
?column?  
-----  
t
```

Teste lógico OR:

```
> SELECT '1' OR '0';  
  
?column?  
-----  
t
```


3.7 Tipos de Dados de Rede

Nome	Descrição
cidr	Endereços de redes IPv4 ou IPv6
inet	Endereços de hosts ou redes IPv4 ou IPv6
macaddr	MAC address (endereço físico)

Um recurso muito interessante e desconhecido de muitas pessoas.

O PostgreSQL fornece nativamente tipos de dados para lidar com endereços de rede que são muito mais eficientes do que se esses dados fossem guardados como strings, além de fazer validação.

Endereço de rede IPv6 como string:

```
> SELECT pg_column_size('2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128'::text);

 pg_column_size 
-----
                40
```

Endereço de rede IPv6 como cidr:

```
> SELECT pg_column_size('2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128'::cidr);

 pg_column_size 
-----
                22
```

Endereço de rede IPv4 como string:

```
> SELECT pg_column_size('192.168.200.100/32'::text);

 pg_column_size 
-----
                22
```

Endereço de rede IPv4 como cidr:

```
> SELECT pg_column_size('192.168.200.100/32'::cidr);

 pg_column_size 
-----
                10
```

Endereço de rede IPv4 inválido:

```
> SELECT '192.168.0.0/33'::cidr;

ERROR:  invalid input syntax for type cidr: "192.168.0.0/33"
LINE 1: SELECT '192.168.0.0/33'::cidr;
          ^
```

Endereço de rede IPv6 como string:

```
> SELECT pg_column_size('2001:4f8:3:ba:2e0:81ff:fe22:d1f1'::text);

 pg_column_size 
-----
                36
```

Endereço de host IPv6:

```
> SELECT pg_column_size('2001:4f8:3:ba:2e0:81ff:fe22:d1f1'::inet);

 pg_column_size 
-----
                22
```

Endereço de host IPv4 como string:

```
> SELECT pg_column_size('192.168.200.100'::text);

 pg_column_size 
-----
                19
```

Endereço de host IPv4 como inet:

```
> SELECT pg_column_size('192.168.200.100'::inet);

 pg_column_size 
-----
                10
```

Entrada inválida de endereço IPv4:

```
> SELECT '192.168.0.256'::inet;

ERROR:  invalid input syntax for type inet: "192.168.0.256"
LINE 1: SELECT '192.168.0.256'::inet;
          ^
```

Endereço MAC como string:

```
> SELECT pg_column_size('00:00:00:00:00:00'::text);  
  
pg_column_size  
-----  
21
```

Endereço MAC como macaddr:

```
> SELECT pg_column_size('00:00:00:00:00:00'::macaddr);  
  
pg_column_size  
-----  
6
```

Entrada inválida de endereço MAC:

```
> SELECT '1a:2b:3c:4d:5e:7g'::macaddr;  
  
ERROR:  invalid input syntax for type macaddr: "1a:2b:3c:4d:5e:7g"  
LINE 1: SELECT '1a:2b:3c:4d:5e:7g'::macaddr;
```

3.8 Outros Tipos

Além de todos os tipos de dados apresentados aqui existem outros. A variedade é muito grande, cujas categorias têm: binários, enumerados, geométricos, de rede, busca textual, UUID, XML, JSON, arrays, compostos, de faixa e etc. Para mais informações, no psql digite o comando:

```
> \dT+ pg_catalog.*
```

Obs.:

O “+” (mais) nos comandos do psql, dá mais informações quando inserido ao final de um comando.

Na documentação oficial, maiores informações sobre os tipos de dados estão disponíveis em:

<http://www.postgresql.org/docs/current/static/datatype.html>

3.9 Type Casts: Conversão de Tipos

Faz uma conversão de um tipo de dado para outro:

Sintaxe SQL ISO

`CAST (expressão AS tipo)`

ou

Sintaxe PostgreSQL

`expressão::tipo`

Conversão de string para inteiro padrão SQL ANSI:

```
> SELECT (CAST (('7' || '0') AS int2)) + (CAST ('7' AS int2)) AS "Resultado";
```

```
Resultado
-----
       77
```

Tudo o que é envolvido por apóstrofes (') é considerado como uma string.

O operador duplo pipe (||) concatena de strings. No exemplo resulta na string '70'.

Se caracteres numéricos não estiverem envolvidos por apóstrofes são considerados como números.

As strings foram convertidas e então com o operador (+) foram somadas.

E se tentarmos somar sem fazer conversão?:

```
> SELECT ('7' || '0') + '7';
```

```
ERROR: operator does not exist: text + unknown
LINE 1: SELECT ('7' || '0') + '7';
                        ^
```

```
HINT: No operator matches the given name and argument type(s). You might need to add explicit type casts.
```

Erro ocasionado por Falta de Conversão.

Conversão de string para inteiro padrão PostgreSQL:

```
> SELECT ('7' || '0')::int2 + '7'::int2 AS "Resultado";
```

```
Resultado
-----
       77
```

4 Interfaces

- Sobre Interfaces de Acesso a um Banco de Dados
- psql
- Drivers de Linguagens de Programação
- pgAdmin 3
- pgAdmin 4
- phpPgAdmin

4.1 Sobre Interfaces de Acesso a um Banco de Dados

Um SGBD não seria tão útil se não tivéssemos como acessar os dados que ele guarda. Para isso precisamos de uma interface para interagir.

Uma interface pode ser um cliente modo texto, um aplicativo desktop ou uma interface web.

Um sistema que uma pessoa utiliza para se fazer um cadastro ou consultas também é uma interface.

4.2 psql

O psql é o cliente padrão do PostgreSQL.

Sua interface é totalmente em linha de comando e considerado o melhor em sua categoria dentre todos os outros SGBDs.

Quando é compilado com a biblioteca *libreadline*, facilita muito a digitação de comandos com o recurso de complemento automático de comandos, apertando a tecla <Tab>.

Sintaxe de Uso:

```
psql [opções]... [nome_da_base [usuário]]
```

Conexão a um Servidor PostgreSQL via psql:

```
$ psql -U postgres -h 192.168.56.2 postgres
```

O comando acima fez com que tentasse uma conexão com o usuário postgres, no host 172.158.50.1 na base de dados postgres.

Utilizando Here Document para comandos:

```
$ psql << EOF
SELECT 2 + 5;
SHOW data_directory;
EOF

?column?
-----
          7
(1 row)

      data_directory
-----
/home/28532757871/pgsql/data
```

Parâmetro -c para comando:

```
$ psql -c "SELECT 'foo' AS test;"

 test
-----
  foo
```


Passando comandos via pipe:

```
$ echo "SELECT 5 + 2; SELECT 'foo';" | psql
```

```
?column?
-----
      7

?column?
-----
foo
```

4.2.1 Obtendo Ajuda de Meta Comandos do psql

O psql possui comandos próprios, inerentes ao aplicativo em si, que não são SQL. Para consultar essa ajuda do psql, dê o seguinte comando dentro de sua interface:

Help de Comandos Internos do psql:

```
> \?
```

4.2.2 Obtendo Ajuda de Comandos SQL Dentro do psql

Podemos também pedir uma ajuda ao psql sobre comandos SQL. Essa ajuda é feita com o comando `\help [comando_SQL]` ou `\h [comando_SQL]`.

Por exemplo, se quisermos saber como se cria uma tabela:

Help de Comandos Internos do psql:

```
> \h CREATE TABLE
```

4.2.3 Conectando-se a Outro Banco de Dados Dentro do psql

No PostgreSQL é preciso uma base para se conectar, mesmo que implicitamente.

Por exemplo, para nos conectarmos à base `template1`, dentro do psql fazemos:

```
> \c template1
```

O comando utilizado para se conectar a outra base é um meta comando do psql.

Por exemplo, para nos conectarmos à base `template1`, dentro do `psql` fazemos:

```
> SELECT current_database();

current_database
-----
template1
```

4.2.4 ~/.psqlrc

O arquivo `.psqlrc`, que fica no diretório do usuário é um nome de arquivo padrão que, se existir, o `psql` o lê e executa seu conteúdo.

O conteúdo do `psqlrc` pode ser tanto meta comandos do `psql` ou comandos SQL.

É muito útil para fazer alguns ajustes.

Criando `.psqlrc` e seu conteúdo:

```
$ cat << EOF > ~/.psqlrc
\set HISTCONTROL ignoredups
\set COMP_KEYWORD_CASE upper
\x auto
EOF
```

Comandos passados:

- `\set HISTCONTROL ignoredups`: Ignorar comandos duplicados, ou seja, só uma ocorrência de um comando dado é registrada no histórico;
- `\set COMP_KEYWORD_CASE upper`: Autocomplemento de comandos SQL convertendo para letras maiúsculas;
- `\x auto`: Modo de expansão de tela automático. Caso as colunas não caibam na horizontal, a exibição será na vertical.

4.3 Drivers de Linguagens de Programação

Para interagir com um SGBD em uma linguagem de programação é preciso ter o driver apropriado.

Segue abaixo linguagens de programação, seu *driver* para PostgreSQL e link correspondente:

Python: [psycopg2](#) [1]
Java: [JDBC](#) [2]
Perl: [DBI:Pg](#) [3]
C: [libpq](#) [4]
PHP: [PDO](#) [5]
C#: [Npgsql](#) [6]
C++: [libpqxx](#) [7]
Lua: [LuaPgSQL](#) [8]
Ruby: [Ruby pg](#) [9]
Go: [pq](#) [10]
Node.js: [node-postgres](#) [11]
Rust: [Rust-Postgres](#) [12]

- [1] <http://initd.org/psycopg>
- [2] <https://jdbc.postgresql.org>
- [3] <https://metacpan.org/pod/DBD::Pg>
- [4] <http://www.postgresql.org/docs/current/static/libpq.html>
- [5] http://php.net/manual/pt_BR/ref.pdo-pgsql.connection.php
- [6] <http://www.npgsql.org>
- [7] <http://pqxx.org/development/libpqxx>
- [8] <https://github.com/arcaeos/luapgsql>
- [9] <https://github.com/ged/ruby-pg>
- [10] <https://github.com/lib/pq>
- [11] <https://github.com/brianc/node-postgres>
- [12] <https://github.com/sfackler/rust-postgres>

4.4 pgAdmin 3

Para quem prefere uma interface gráfica para interagir com o SGBD, se tratando de PostgreSQL, uma das alternativas mais conhecidas é o pgAdmin 3, cujo site oficial é <http://www.pgadmin.org/>.

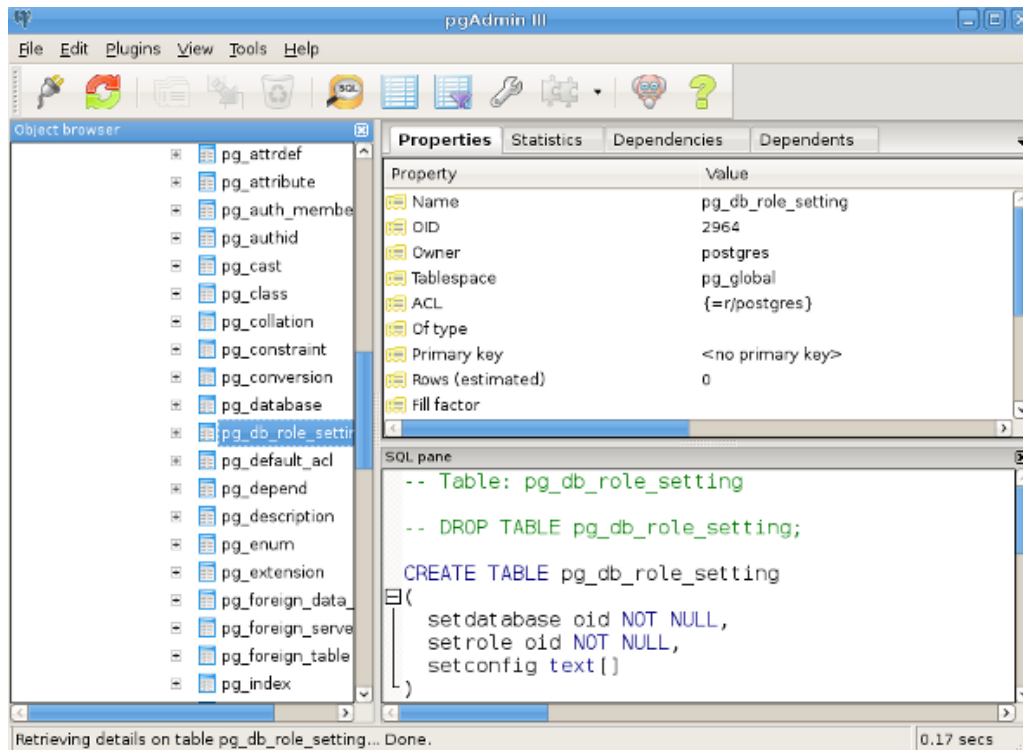


Figura 1: pgAdmin 3 exibindo parte da estrutura de uma base

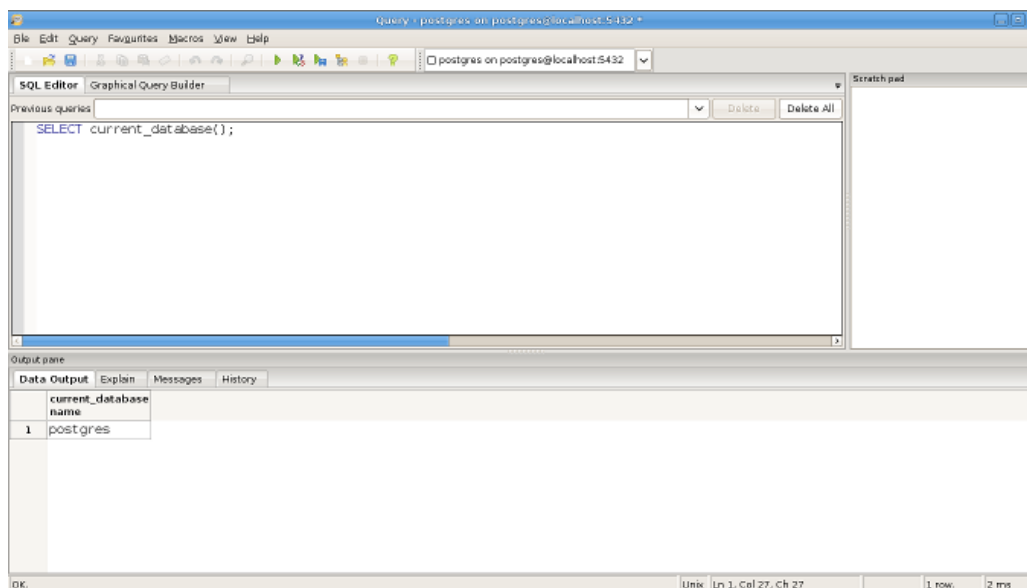
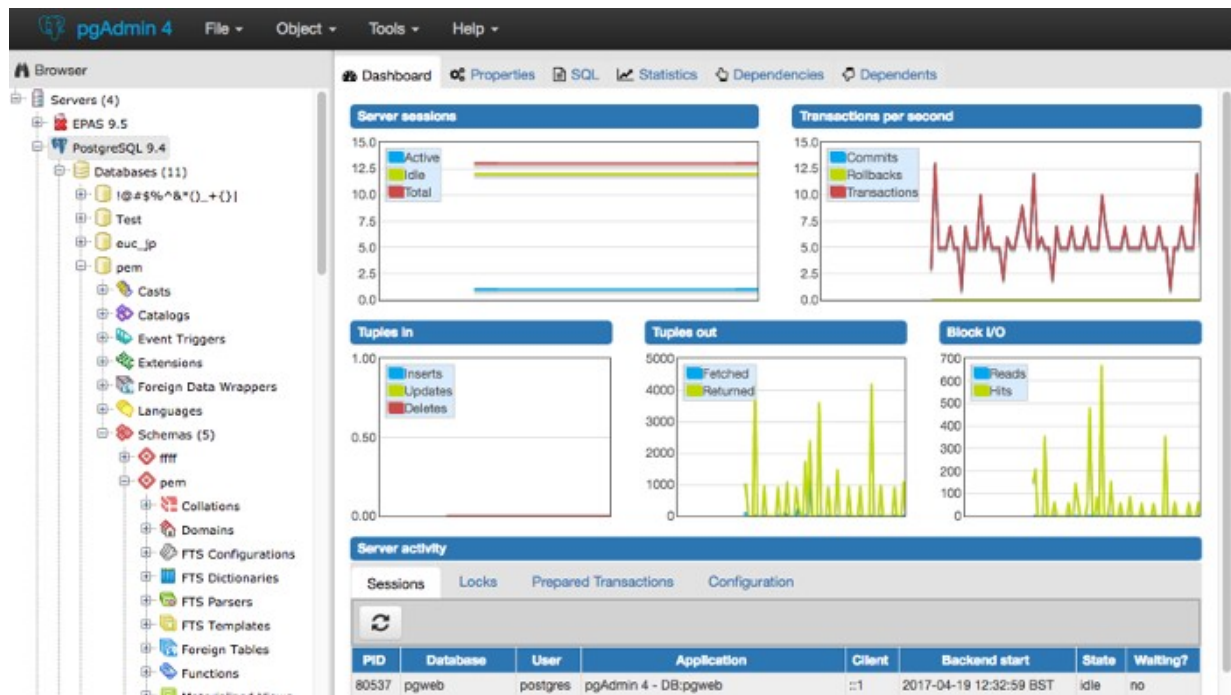


Figura 2: Uma consulta executada no pgAdmin3

4.5 PgAdmin 4

O PgAdmin 4, diferente do PgAdmin 3 tem interface web escrita em Python com o *framework* Flask [1].

<https://www.pgadmin.org/>



[1] <http://flask.pocoo.org/>

4.6 phpPgAdmin

O phpPgAdmin é uma interface web similar ao phpMyAdmin (<http://www.phpmyadmin.net/>), que é para o MySQL.

Assim como o pgAdmin 3 é muito fácil de usar, seu site oficial é <http://phppgadmin.sourceforge.net>.

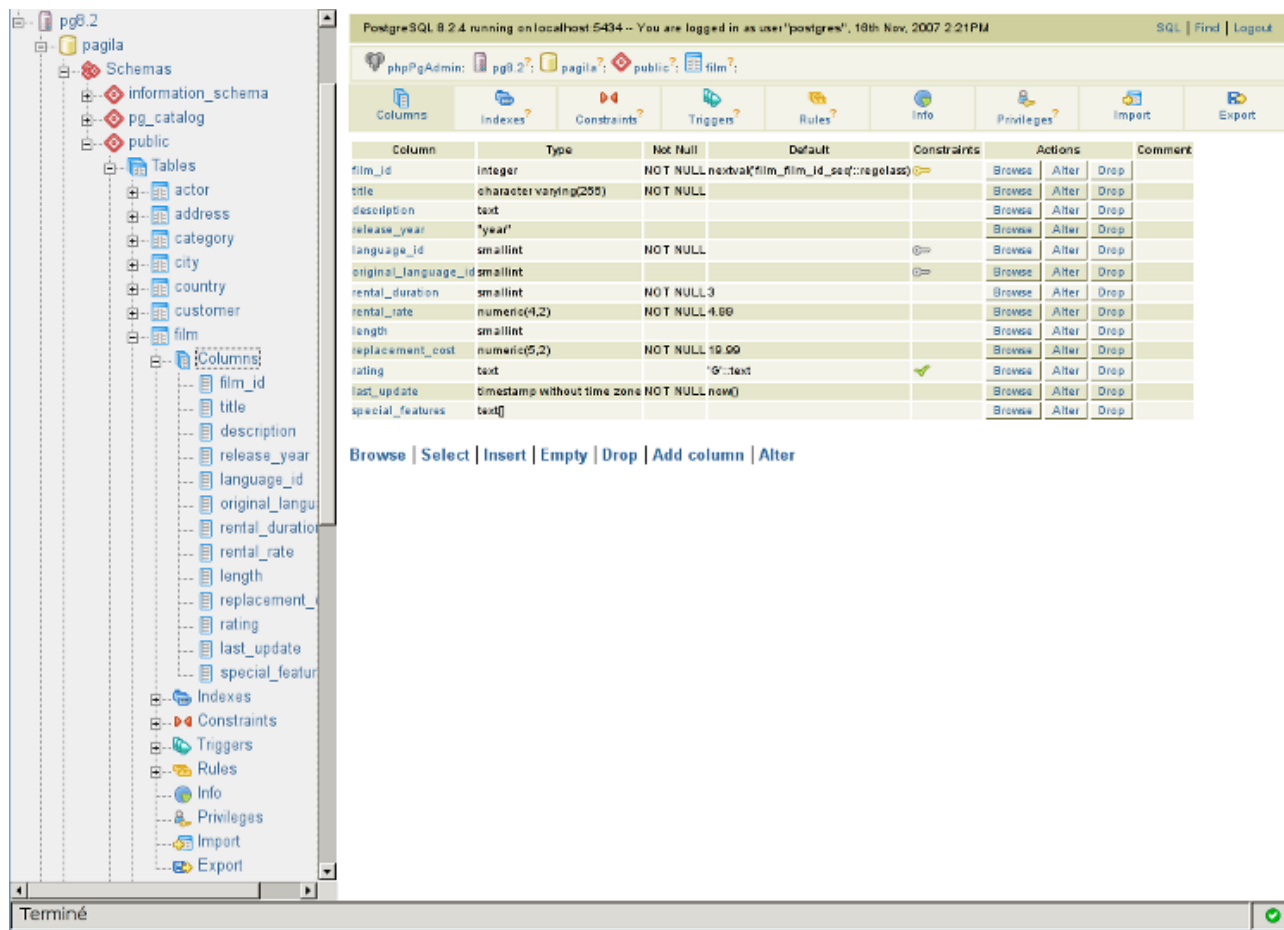


Figura 3: Interface Web phpPgAdmin

5 Objetos de Bancos de Dados

- Sobre Objetos de Bancos de Dados
- Descrição (Comentário) de Objetos

5.1 Sobre Objetos de Bancos de Dados

São componentes da estrutura de uma base de dados, de diversas naturezas (funções que desempenham) e que podem ter relacionamento entre si e / ou interdependência.

Exemplos: banco de dados (base de dados), tabela, campo de tabela (coluna), sequência, visão (*view*), gatilho (*trigger*), usuário (*role*), etc.

5.1.1 Criando, Modificando e Apagando Objetos

Para criar, alterar ou remover objetos em uma base de dados utilizamos respectivamente os comandos DDL: CREATE, ALTER e DROP.

Criação de uma base de dados:

```
> CREATE DATABASE db_teste;
```

Meta comando do psql para listar bases de dados na instância:

```
> \l
```

List of databases					
Name	Owner	Encoding	Collate	Ctype	Access privileges
db_empresa	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
db_teste	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres + postgres=CTc/postgres

Listando bases de dados via catálogo do sistema:

```
> SELECT datname FROM pg_database;
```

```
datname
-----
template1
template0
postgres
db_empresa
db_teste
```

Acessando a base:

```
> \c db_teste
```


Criação de uma tabela sem campos dentro da base db_teste:

```
> CREATE TABLE tb_teste();
```

Meta comando para listar tabelas na base de dados:

```
> \dt
```

```
           List of relations
Schema |   Name   | Type  | Owner
-----+-----+-----+-----
public | tb_teste | table | postgres
```

Listando tabelas da base via catálogo de sistema:

```
> SELECT tablename FROM pg_tables
   WHERE schemaname NOT IN ('pg_catalog', 'information_schema');
```

```
tablename
-----
tb_teste
```

Exibindo a estrutura da tabela:

```
> \d tb_teste
```

```
Table "public.tb_teste"
Column | Type | Modifiers
-----+-----+-----
```

A tabela tem sua estrutura vazia, pois não tem campos.

Adicionando um campo na tabela tb_teste:

```
> ALTER TABLE tb_teste ADD COLUMN campo INT;
```

Verificando a estrutura da tabela após a adição de um campo:

```
> \d tb_teste
```

```
Table "public.tb_teste"
Column | Type  | Modifiers
-----+-----+-----
campo  | integer |
```

Renomeando a tabela:

```
> ALTER TABLE tb_teste RENAME TO tb_1;
```

Criação de um objeto do tipo sequência:

```
> CREATE SEQUENCE sq_1;
```

Listando sequências:

```
> \ds
```

```
           List of relations
 Schema | Name | Type   | Owner
-----+-----+-----+-----
 public | sq_1 | sequence | postgres
```

Listando sequências via catálogo de sistema:

```
> SELECT relname FROM pg_class WHERE relkind = 'S';
```

```
 relname
-----
 sq_1
```

Alterando o valor padrão da coluna para o próximo valor da sequência

```
> ALTER TABLE tb_1 ALTER COLUMN campo SET DEFAULT nextval('sq_1');
```

Exibindo a estrutura da tabela tb_1 após a alteração:

```
> \d tb_1
```

```
           Table "public.tb_1"
 Column | Type          | Modifiers
-----+-----+-----
 campo  | integer       | default nextval('sq_1':regclass)
```

Tentativa de remoção da sequência sq_1, que está atrelada ao valor padrão de campo da tabela tb_1:

```
> DROP SEQUENCE sq_1;
```

```
ERROR: cannot drop sequence sq_1 because other objects depend on it
DETAIL: default for table tb_1 column campo depends on sequence sq_1
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Não foi possível apagar a sequência devido ao fato de haver dependência de objeto.

Remoção de sequência em cascata:

```
> DROP SEQUENCE sq_1 CASCADE;
```

```
NOTICE: drop cascades to default for table tb_1 column campo
```

A mensagem diz que a remoção da sequência foi feita, mas como consequência, devido à dependência do campo da tabela tb_1, a restrição DEFAULT foi apagada.

Exibindo a estrutura da tabela tb_1 após apagar a sequência que ela dependia:

```
> \d tb_1
```

```
Table "public.tb_1"
Column | Type      | Modifiers
-----+-----+-----
campo  | integer   |
```

Criação de uma tabela de teste:

```
> CREATE TEMP TABLE tb_cpf(
    id serial primary key,
    cpf varchar(11));
```

Inserir um dado:

```
> INSERT INTO tb_cpf (cpf) VALUES ('12345678901');
```

Tentativa de alteração de tipo:

```
> ALTER TABLE tb_cpf ALTER cpf TYPE bigint;
```

```
ERROR: column "cpf" cannot be cast automatically to type bigint  
HINT: You might need to specify "USING cpf::bigint".
```

Foi exibida uma mensagem de erro, pois nesse caso é necessário fazer um cast com a cláusula USING.

Alterando o tipo de dado com USING:

```
> ALTER TABLE tb_cpf ALTER cpf TYPE bigint USING cpf::bigint;
```

5.2 Descrição (Comentário) de Objetos

Descrição de objetos, ou também conhecida como comentário de objeto é uma ótima prática para se inserir informações sobre um determinado objeto, que serve também como documentação.

Tais informações são visíveis quando adicionamos o caractere de soma “+” quando pedimos a descrição de um objeto. Esse sinal de soma significa algo como “mais detalhes”.

Criação de uma tabela pra testes:

```
> CREATE TABLE tb_bairro(  
    id serial PRIMARY KEY,  
    nome text,  
    obs text);
```

Listando objetos (tabelas e sequências):

```
> \d
```

List of relations			
Schema	Name	Type	Owner
public	tb_1	table	postgres
public	tb_bairro	table	postgres
public	tb_bairro_id_seq	sequence	postgres

Exibindo somente tabelas:

```
> \dt
```

List of relations			
Schema	Name	Type	Owner
public	tb_1	table	postgres
public	tb_bairro	table	postgres

Exibindo somente tabelas com maiores informações:

```
> \dt+
```

List of relations					
Schema	Name	Type	Owner	Size	Description
public	tb_1	table	postgres	0 bytes	
public	tb_bairro	table	postgres	8192 bytes	

Inserindo um comentário (descrição) na tabela tb_bairro:

```
> COMMENT ON TABLE tb_bairro IS 'Tabela de bairros';
```

Exibindo descrição de tabelas:

```
> \dt+
```

```

              List of relations
 Schema | Name      | Type  | Owner  | Size  | Description
-----+-----+-----+-----+-----+-----
 public | tb_1      | table | postgres | 0 bytes |
 public | tb_bairro | table | postgres | 8192 bytes | Tabela de bairros
```

Inserindo descrição no campo nome da tabela tb_bairro:

```
> COMMENT ON COLUMN tb_bairro.nome IS 'Nome do bairro';
```

Informações detalhadas da tabela tb_bairro:

```
> \d+ tb_bairro
```

```

              Table "public.tb_bairro"
 Column | Type          | Modifiers                               | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id     | integer       | not null default nextval('tb_bairro_id_seq'::regclass) | plain   |               |
 nome   | text          |                                         | extended |               | Nome do bairro
 obs    | text          |                                         | extended |               |
Indexes:
    "tb_bairro_pkey" PRIMARY KEY, btree (id)
```

Removendo a descrição da tabela:

```
> COMMENT ON TABLE tb_bairro IS NULL;
```

Verificando informações detalhadas da tabela:

```
> \dt+ tb_bairro
```

```

              List of relations
 Schema | Name      | Type  | Owner  | Size  | Description
-----+-----+-----+-----+-----+-----
 public | tb_bairro | table | postgres | 8192 bytes |
```

6 Tabelas

- Sobre Tabelas
- Tabelas Temporárias
- UNLOGGED TABLE – Tabela Não Logada
- Fillfactor de Tabela

6.1 Sobre Tabelas

Tabela é o objeto de uma base de dados, cuja função é guardar dados.

Possui colunas (campos) e linhas (registros).

Cada coluna tem sua estrutura da seguinte forma, na sequência: nome, tipo e modificadores.

6.1.1 Criação de Tabelas

Sintaxe geral de criação de uma Tabela:

```
CREATE TABLE nome_tabela(  
    nome_campo tipo_dado [DEFAULT expressao_padrao]  
    [CONSTRAINT nome_restricao tipo_restricao],  
    ...  
);
```

Exemplo de Criação de uma Tabela:

```
> CREATE TABLE tb_2(  
    campo1 int2 NOT NULL,  
    campo2 date);
```

6.1.2 Alterando Tabelas

Às vezes pode acontecer de após termos criado algumas tabelas, haver a necessidade de modificá-las.

Ao pedirmos ajuda ao psql, conforme mostrado a seguir, contém de maneira resumida e objetiva os meios para se fazer as alterações de acordo com o que for preciso.

Ajuda do Comando SQL “ALTER TABLE”:

```
> \help ALTER TABLE
```

```
...
```

Acessando a base db_empresa:

```
> \c db_empresa
```


Exibir estrutura da tabela após alteração feita:

```
> \d tb_colaborador
```

```
Table "public.tb_colaborador"
  Column      |      Type      | Modifiers
-----+-----+-----
 id           | integer        | not null default nextval('sq_colaborador'::regclass)
 cpf          | character varying(11) |
 setor       | integer        |
 cargo       | integer        |
 chefe_direto | integer        |
 salario     | numeric(9,2)   |
 dt_admis    | date           | default now()
 dt_demis    | date           |
 ativo       | boolean        | default true
Indexes:
    "pk_colaborador" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_cargo_colaborador" FOREIGN KEY (cargo) REFERENCES tb_cargo(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
    "fk_pf_colaborador" FOREIGN KEY (cpf) REFERENCES tb_pf(cpf) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
```

Adicionando uma nova coluna (do tipo smallint) à tabela tb_colaborador:

```
> ALTER TABLE tb_colaborador ADD COLUMN nova_coluna int2;
```

Exibir estrutura da tabela após alteração feita:

```
> \d tb_colaborador
```

```
Table "public.tb_colaborador"
  Column      |      Type      | Modifiers
-----+-----+-----
 id           | integer        | not null default nextval('sq_colaborador'::regclass)
 cpf          | character varying(11) |
 setor       | integer        |
 cargo       | integer        |
 chefe_direto | integer        |
 salario     | numeric(9,2)   |
 dt_admis    | date           | default now()
 dt_demis    | date           |
 ativo       | boolean        | default true
 nova_coluna  | smallint       |
Indexes:
    "pk_colaborador" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_cargo_colaborador" FOREIGN KEY (cargo) REFERENCES tb_cargo(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
    "fk_pf_colaborador" FOREIGN KEY (cpf) REFERENCES tb_pf(cpf) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
```

Mudando o tipo de dados de um campo:

```
> ALTER TABLE tb_colaborador ALTER nova_coluna TYPE numeric(7, 2);
```

Exibir estrutura da tabela após alteração feita:

```
> \d tb_colaborador
```

```
Table "public.tb_colaborador"
  Column      |      Type      |      Modifiers
-----+-----+-----
 id           | integer        | not null default nextval('sq_colaborador'::regclass)
 cpf          | character varying(11) |
 setor       | integer        |
 cargo       | integer        |
 chefe_direto | integer        |
 salario     | numeric(9,2)   |
 dt_admis    | date           | default now()
 dt_demis    | date           |
 ativo       | boolean        | default true
 nova_coluna  | numeric(7,2)   |
Indexes:
    "pk_colaborador" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_cargo_colaborador" FOREIGN KEY (cargo) REFERENCES tb_cargo(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
    "fk_pf_colaborador" FOREIGN KEY (cpf) REFERENCES tb_pf(cpf) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES
    tb_colaborador(id)
```

Alterando o nome de um campo:

```
> ALTER TABLE tb_colaborador RENAME COLUMN nova_coluna TO ultima_coluna;
```

Exibir estrutura da tabela após alteração feita:

```
> \d tb_colaborador
```

```
Table "public.tb_colaborador"
  Column      |      Type      |      Modifiers
-----+-----+-----
 id           | integer        | not null default nextval('sq_colaborador'::regclass)
 cpf          | character varying(11) |
 setor       | integer        |
 cargo       | integer        |
 chefe_direto | integer        |
 salario     | numeric(9,2)   |
 dt_admis    | date           | default now()
 dt_demis    | date           |
 ativo       | boolean        | default true
 ultima_coluna | numeric(7,2)   |
Indexes:
    "pk_colaborador" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_cargo_colaborador" FOREIGN KEY (cargo) REFERENCES tb_cargo(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
    "fk_pf_colaborador" FOREIGN KEY (cpf) REFERENCES tb_pf(cpf) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES
    tb_colaborador(id)
```

Apagando um campo de uma tabela:

```
> ALTER TABLE tb_colaborador DROP COLUMN ultima_coluna;
```

Exibir estrutura da tabela após alteração feita:

```
> \d tb_colaborador
```

```
Table "public.tb_colaborador"
  Column      |      Type      |      Modifiers
-----+-----+-----
 id           | integer        | not null default nextval('sq_colaborador'::regclass)
 cpf          | character varying(11) |
 setor       | integer        |
 cargo       | integer        |
 chefe_direto | integer        |
 salario     | numeric(9,2)   |
 dt_admis    | date           | default now()
 dt_demis    | date           |
 ativo       | boolean        | default true
Indexes:
    "pk_colaborador" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_cargo_colaborador" FOREIGN KEY (cargo) REFERENCES tb_cargo(id) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES tb_colaborador(id)
    "fk_pf_colaborador" FOREIGN KEY (cpf) REFERENCES tb_pf(cpf) ON UPDATE CASCADE ON DELETE CASCADE
    "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_colaborador_colaborador" FOREIGN KEY (chefe_direto) REFERENCES
tb_colaborador(id)
```

Criação de uma tabela para teste:

```
> CREATE TABLE tb_foo(column1 int2);
```

Renomeando a tabela:

```
> ALTER TABLE tb_foo RENAME TO tb_foo_bar;
```

6.1.3 Apagando uma tabela

O comando “DROP”, como já foi mencionado, serve pra eliminar objetos, tais como uma tabela.

Conexão ao banco de dados postgres:

```
> \c postgres
```

Criação da tabela de estados:

```
> CREATE TEMP TABLE tb_uf(
    uf char(2) PRIMARY KEY,
    nome text);
```

Criação da tabela de cidades, que depende da tabela de estados:

```
> CREATE TEMP TABLE tb_cidade(  
    id SERIAL PRIMARY KEY,  
    nome TEXT,  
    uf CHAR(2),  
    CONSTRAINT fk_uf_cidade_uf FOREIGN KEY (uf) REFERENCES tb_uf (uf));
```

tb_cidade possui um relacionamento com tb_uf, pois há uma chave estrangeira em tb_cidade que referencia o campo de chave primária de tb_uf.

Criação de uma tabela qualquer sem nenhum relacionamento:

```
> CREATE TEMP TABLE tb_zero();
```

Apagando a tabela sem relacionamento:

```
> DROP TABLE tb_zero;
```

Tentativa de apagar tb_uf que tem tb_cidade como sua dependente por relacionamento:

```
> DROP TABLE tb_uf;
```

```
ERROR: cannot drop table tb_uf because other objects depend on it  
DETAIL: constraint fk_uf_cidade_uf on table tb_cidade depends on table tb_uf  
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Descrição de tb_uf que entre outras informações diz que é referenciada por tb_cidade:

```
> \d tb_uf  
  
      Table "pg_temp_2.tb_uf"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 uf      | character(2)   | not null  
 nome    | text           |  
Indexes:  
    "tb_uf_pkey" PRIMARY KEY, btree (uf)  
Referenced by:  
    TABLE "tb_cidade" CONSTRAINT "fk_uf_cidade_uf" FOREIGN KEY (uf) REFERENCES tb_uf(uf)
```

Descrição de tb_cidade que exibe também que ela referencia tb_uf através de uma chave estrangeira:

```
> \d tb_cidade
```

```

      Table "pg_temp_2.tb_cidade"
Column |      Type      | Modifiers
-----+-----+-----
id      | integer        | not null default nextval('tb_cidade_id_seq'::regclass)
nome    | text           |
uf      | character(2)   |
Indexes:
    "tb_cidade_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_uf_cidade_uf" FOREIGN KEY (uf) REFERENCES tb_uf(uf)
```

Remoção forçada (em cascata) de tb_uf:

```
> DROP TABLE tb_uf CASCADE;
```

```
NOTICE: drop cascades to constraint fk_uf_cidade_uf on table tb_cidade
```

Ao remover em cascata tb_uf, foi dada a informação que a chave estrangeira em tb_cidade foi removida e agora no campo uf não há qualquer modificador fazendo menção ao antigo relacionamento:

```
> \d tb_cidade
```

```

      Table "pg_temp_2.tb_cidade"
Column |      Type      | Modifiers
-----+-----+-----
id      | integer        | not null default nextval('tb_cidade_id_seq'::regclass)
nome    | text           |
uf      | character(2)   |
Indexes:
    "tb_cidade_pkey" PRIMARY KEY, btree (id)
```

6.1.4 TRUNCATE - Redefinindo uma Tabela Como Vazia

O comando TRUNCATE apaga todos os registros de uma tabela, portanto, deve ser usado com cautela.

Apesar de apagar todos os dados de uma tabela é um comando de definição e não de manipulação, ou seja, é um comando DDL.

Sintaxe:

```
TRUNCATE tabela [CASCADE];
```

No banco db_empresa, criar a tabela:

```
> CREATE TABLE tb_pf80
  AS SELECT * FROM tb_pf
  WHERE dt_nascto BETWEEN '1980-01-01' AND '1989-12-31';
```

Redefinindo a Tabela "tb_pf80" Como Vazia:

```
> TRUNCATE tb_pf80;
```

Apagando a tabela de teste:

```
> DROP TABLE tb_pf80;
```

Aviso:

Para usarmos o `TRUNCATE` em tabelas que outros objetos são dependentes, temos que fazer isso em cascata (`CASCADE`).

6.1.5 CREATE TABLE IF NOT EXISTS / DROP TABLE IF EXISTS

`IF NOT EXISTS` é uma cláusula interessante para criação de objetos, sendo eles; `TABLE` (tabela), `FOREIGN TABLE` (tabela estrangeira) e `EXTENSION` (extensão).

Sua função é fazer com que se o objeto de mesmo nome já existir ele apenas retorna um informe em vez de um erro.

Sintaxe:

```
CREATE TIPO_DE_OBJETO IF NOT EXISTS nome_do_objeto ...;
```

```
DROP TIPO_DE_OBJETO IF EXISTS nome_do_objeto ...;
```

Execute o comando abaixo duas vezes:

```
> CREATE TEMP TABLE IF NOT EXISTS tb_tmp();
```

```
NOTICE: relation "tb_tmp" already exists, skipping
```

Em vez de uma mensagem de erro temos um aviso simples.

De forma análoga, para apagar um objeto temos `DROP ... IF EXISTS`:

```
> DROP TABLE IF EXISTS tb_tmp;
```

6.2 Tabelas Temporárias

São tabelas que só podem ser acessadas diretamente pelo usuário que a criou e na sessão corrente. Assim que o usuário criador da mesma se desconectar da sessão em que ela foi criada, a tabela temporária deixará de existir.

A sintaxe de criação é igual à de uma tabela normal, apenas acrescenta-se a palavra “TEMPORARY” ou “TEMP” após `CREATE`.

São 3 (três) as características de uma tabela temporária:

1. Ficam em um esquema especial (e temporário) e seus dados só podem ser vistos pelo backend que a criou;
2. São gerenciadas pelo buffer local em vez do gerenciador de buffer compartilhado (shared buffer);
3. Não geram logs de transação.

Exemplo de criação de uma tabela temporária:

```
> CREATE TEMP TABLE tb_tmp (campo int4);
```

6.3 UNLOGGED TABLE – Tabela Não Logada

Uma tabela “*unlogged*” é uma tabela em que os dados não são escritos para o WAL (*Write-Ahead Log*: logs de transação), que faz com que fique consideravelmente mais rápida do que uma tabela comum. Porém, não são seguras contra falhas (*crash*): uma tabela *unlogged* é automaticamente truncada após uma falha ou um desligamento inapropriado.

O conteúdo de uma tabela UNLOGGED não é replicado para servidores *standby*.

Qualquer índice criado em uma tabela desse tipo é automaticamente também “*unlogged*”, porém índices GiST atualmente não são suportados e não podem ser criados em uma tabela *unlogged*.

É um tipo de tabela rápida, mas insegura. Não deve ser utilizada para operações que precisem de persistência de dados.

6.3.1 Qual a Utilidade de uma UNLOGGED TABLE?

Tabelas não logadas são muito úteis onde não é importante a durabilidade dos dados. Por exemplo: cache de dados, informações temporárias acessíveis a todas sessões abertas, cargas ETL e etc.

Criação da tabela não logada:

```
> CREATE UNLOGGED TABLE tb_teste_unlogged (campo int);
```

Criação de uma tabela comum:

```
> CREATE TABLE tb_teste_logged (campo int);
```

Ligar o cronômetro de comandos do psql:

```
> \timing
```

Inserir três milhões de registros na tabela unlogged:

```
> INSERT INTO tb_teste_unlogged (campo) SELECT generate_series(1, 3000000);
```

```
Time: 1922,384 ms
```

Inserir três milhões de registros na tabela comum:

```
> INSERT INTO tb_teste_logged (campo) SELECT generate_series(1, 3000000);
```

```
Time: 3806,184 ms
```


Sair do psql:

```
> \q
```

ou

```
<Ctrl> + D
```

Matar todos processos do PostgreSQL:

```
$ killall postgres
```

Iniciar o serviço do PostgreSQL:

```
$ pg_ctl start
```

Entrar no psql:

```
$ psql
```

Verificar a tabela não logada:

```
> TABLE tb_teste_unlogged LIMIT 5;
```

```
 campo
-----
 1
 2
 3
 4
 5
```

Inserir mais valores na tabela unlogged:

```
> INSERT INTO tb_teste_unlogged VALUES (6), (7), (8), (9), (10);
```

Sair do psql:

```
> \q
```

ou

```
<Ctrl> + D
```

Matar todos os processos do PostgreSQL de forma abrupta:

```
$ killall -9 postgres
```

Iniciar o serviço do PostgreSQL:

```
$ pg_ctl start
```

Entrar no psql:

```
$ psql
```

Verificar a tabela não logada:

```
> TABLE tb_teste_unlogged;
```

```
campo  
-----
```

Devido à falha provocada, a tabela foi truncada.

No catálogo pg_class, utilizando de expressão regular que case com os nomes das tabelas criadas, verificar o tipo de persistência:

```
> SELECT  
    oid, relfilenode, relname, relpersistence, relkind  
FROM pg_class  
WHERE relname ~ '^tb_.*logged';
```

oid	relfilenode	relname	relpersistence	relkind
17102	17102	tb_teste_logged	p	r
17099	17099	tb_teste_unlogged	u	r

Descrição das colunas:

oid: Coluna de sistema que armazena o ID do objeto;

relfilenode: Nome do arquivo em disco referente ao objeto, inicialmente é igual ao oid;

relname: Nome do objeto;

relpersistence: Tipo de persistência, sendo que p = permanente e u = unlogged;

relkind: Tipo de relação, sendo que a mais comum é "r" para tabelas.

Uma tabela comum a persistência é um "p", enquanto que uma tabela não logada é representada por um "u". Ou seja, se relpersistence for "u" é *unlogged*.

Mudar tabela não logada para logada:

```
> ALTER TABLE tb_teste_unlogged SET LOGGED;
```

Mudar tabela logada para não logada:

```
> ALTER TABLE tb_teste_logged SET UNLOGGED;
```

Verificando o tipo de persistência e outros detalhes:

```
> SELECT
    oid, relfilenode, relname, relpersistence, relkind
FROM pg_class
WHERE relname ~ '^tb_.*logged';
```

oid	relfilenode	relname	relpersistence	relkind
17102	17108	tb_teste_logged	u	r
17099	17105	tb_teste_unlogged	p	r

Quando uma tabela muda seu tipo de persistência de dados, ela tem que ser totalmente reescrita, sendo necessário criar um novo arquivo. Isso explica a mudança de relfilenode após alterar as tabelas.

6.4 Fillfactor de Tabela

`Fillfactor` para uma tabela é uma porcentagem entre 10 e 100 (sendo que 100 é o padrão). Quando um valor menor é definido, operações de `INSERT` faz o preenchimento das páginas até o valor indicado, o espaço restante é reservado para operações de `UPDATE`. Isso dá uma chance para uma cópia de uma linha modificada ser alocada na mesma página que a linha original, que é mais eficiente do que alocar em uma página diferente.

Para uma tabela cujas entradas nunca são modificadas (tabela estática), empacotamento completo (`fillfactor = 100`) é a melhor escolha, mas para tabelas cujos dados são modificados intensamente, valores menores para `fillfactor` é mais apropriado. Esse parâmetro não pode ser ajustado para tabelas `TOAST`.

Habilitamos o cronômetro de comandos no `psql`:

```
> \timing
```

Criação da primeira tabela com `fillfactor` igual a 100 (cem):

```
> CREATE TABLE tb_ff100(  
    id serial PRIMARY KEY,  
    campo int);
```

Popular a tabela com trezentos mil registros:

```
> INSERT INTO tb_ff100 (campo) SELECT generate_series(1, 300000);
```

Média: 1793,763 ms

Verificar o tamanho da tabela após os `INSERTs`:

```
> SELECT pg_size_pretty(pg_relation_size('tb_ff100'));
```

```
pg_size_pretty  
-----  
10 MB
```

Atualizar todos os registros:

```
> UPDATE tb_ff100 SET campo = campo + 1;
```

```
Time: 2182,260 ms
```

Tamanho da tabela após o UPDATE:

```
> SELECT pg_size_pretty(pg_relation_size('tb_ff100'));

pg_size_pretty
-----
21 MB
```

Criação da segunda tabela com fillfactor igual a 50 (cinquenta):

```
> CREATE TABLE tb_ff50 (
  id serial PRIMARY KEY,
  campo int)
  WITH (fillfactor = 50);
```

Popular a tabela com trezentos mil registros:

```
> INSERT INTO tb_ff50 (campo) SELECT generate_series(1, 300000);
```

Média: 2152,716 ms

Verificar o tamanho da tabela após os INSERTs:

```
> SELECT pg_size_pretty(pg_relation_size('tb_ff50'));

pg_size_pretty
-----
21 MB
```

Atualizar todos os registros:

```
> UPDATE tb_ff50 SET campo = campo + 1;
```

Time: 815,024 ms

Tamanho da tabela após o UPDATE:

```
> SELECT pg_size_pretty(pg_relation_size('tb_ff50'));

pg_size_pretty
-----
21 MB
```

Resumo dos Testes		
Fillfactor	100	50
Média INSERT Inicial (ms)	1793,763	2152,716
Tamanho da Tabela após INSERT (MB)	10	21
UPDATE (ms)	2182,260	815,024
Tamanho da Tabela após UPDATE (MB)	21	21

7 Restrições (Constraints)

- Sobre Restrições
- Valor padrão (DEFAULT)
- CHECK
- NOT NULL
- UNIQUE
- A Cláusula [NOT] DEFERRABLE
- Chave Primária / Primary Key
- Chave Estrangeira / Foreign Key

7.1 Sobre Restrições (Constraints)

São regras que inserimos para que uma coluna não receba dados indesejados.

São regras para manter a base de dados consistente de acordo com regras de negócio.

7.2 Valor Padrão (DEFAULT)

Quando uma estrutura de uma tabela é construída, pode se definir a determinados campos, caso os mesmos não forem preenchidos (valores nulos), ter um valor atribuído automaticamente. Ou seja, se nada for atribuído a ele pelo usuário, o sistema o fará.

Criação da tabela de teste para a restrição DEFAULT:

```
> CREATE TEMP TABLE tb_teste_default(  
    nome varchar(15),  
    uf char(2) DEFAULT 'SP');
```

O exemplo de criação de tabela acima, mostra o campo “uf”, o qual se não for inserido algum valor, o sistema, por padrão o preencherá com o valor “SP”.

Para testarmos, vamos fazer algumas inserções.

Sem declarar uf terá seu valor padrão implícito:

```
> INSERT INTO tb_teste_default (nome) VALUES ('Guarulhos');
```

Inserindo explicitamente o valor de uf:

```
> INSERT INTO tb_teste_default (nome, uf) VALUES ('Santo André', 'SP');
```

Inserção com o valor padrão (DEFAULT) para o Campo uf:

```
> INSERT INTO tb_teste_default (nome, uf) VALUES ('Mauá', DEFAULT);
```

Inserção com um valor diferente do padrão para uf:

```
> INSERT INTO tb_teste_default (nome, uf) VALUES ('João Pessoa', 'PB');
```

Todos os dados da tabela:

```
> SELECT * FROM tb_teste_default;
```

nome	uf
Guarulhos	SP
Santo André	SP
Mauá	SP
João Pessoa	PB

7.3 CHECK

Especifica que valores são permitidos em uma determinada coluna.

Restrição CHECK declarada na criação da tabela:

```
> CREATE TEMP TABLE tb_check(  
    cod_prod int2,  
    preco numeric(7, 2) CHECK(preco > 0));
```

No exemplo evita que o campo `preco` tenha um valor inserido que não seja menor do que zero.

Tentativa de inserção de valor negativo para o campo `preco`:

```
> INSERT INTO tb_check (cod_prod, preco) VALUES (147, -10);
```

```
ERROR: new row for relation "tb_check" violates check constraint "tb_check_preco_check"  
DETAIL: Failing row contains (147, -10.00).
```

A mensagem de erro avisa que houve uma violação de restrição do tipo check.

7.4 NOT NULL

Por padrão o preenchimento do valor de uma coluna não é obrigatório, ou seja, é como se declarássemos a coluna com o modificador `NULL`.

Para forçarmos o preenchimento de um valor de uma coluna como obrigatório utilizamos a cláusula `NOT NULL`.

Declaração da restrição `NOT NULL` na criação da tabela:

```
> CREATE TEMP TABLE tb_not_null(  
    campo1 int2,  
    campo2 int2 NOT NULL);
```

Propositalmente omitida a coluna `campo2` que tem restrição `NOT NULL`:

```
> INSERT INTO tb_not_null (campo1) VALUES (3);
```

```
ERROR: null value in column "campo2" violates not-null constraint  
DETAIL: Failing row contains (3, null).
```

Ao inserirmos valor em uma tabela, se um determinado campo não for declarado ou ele será nulo (`null`), ou se foi definida uma restrição `DEFAULT`, o valor será o padrão dessa restrição `DEFAULT`.

7.5 UNIQUE

Força a unicidade de valores na coluna.

UNIQUE ao criar a tabela:

```
> CREATE TEMP TABLE tb_unique(  
    campo1 int2,  
    campo2 int2 UNIQUE);
```

Obs.:

Toda vez que uma restrição UNIQUE é criada, um índice é criado para ela implicitamente.

Verificando a estrutura da tabela:

```
> \d tb_unique  
  
      Table "public.tb_unique"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 campo1 | smallint |  
 campo2 | smallint |  
Indexes:  
    "tb_unique_campo2_key" UNIQUE CONSTRAINT, btree (campo2)
```

Inserção de Valores para Testar a Restrição UNIQUE:

```
> INSERT INTO tb_unique (campo1, campo2) VALUES  
    (1991, 2007),  
    (1991, 2008),  
    (1992, 2007);  
  
ERROR:  duplicate key value violates unique constraint "tb_unique_campo2_key"  
DETAIL:  Key (campo2)=(2007) already exists.
```

A mensagem de erro informa que a restrição UNIQUE foi violada devido à tentativa de inserção com duplicidade de valores.

7.6 A Cláusula [NOT] DEFERRABLE

Uma *constraint* que não é deferida (`NOT DEFERRABLE`) será verificada imediatamente após cada comando.

A verificação das restrições que são deferidas / adiadas (`DEFERRABLE`) pode ser adiada até o final da transação.

`NOT DEFERRABLE` é o padrão. Atualmente, apenas as *constraints* `UNIQUE`, `PRIMARY KEY`, `EXCLUDE` e `REFERENCES` (chave estrangeira) aceitam essa cláusula.

`NOT NULL` e `CHECK` são `NOT DEFERRABLE`.

Constraints deferíveis não podem ser usadas como regentes de conflitos em um comando `INSERT` que inclui uma cláusula `ON CONFLICT DO UPDATE`.

Criação de tabela que será a fonte dos dados de teste:

```
> CREATE TEMP TABLE tb_source(n int);
```

Inserindo os dados na tabela fonte, fazer 2 (duas) vezes:

```
> INSERT INTO tb_source (n) SELECT generate_series(1, 1000000);
```

É necessário fazer o comando duas vezes para gerar dados repetidos.

Criação de tabela cuja cláusula `UNIQUE` é deferível:

```
> CREATE TABLE tb_deferrable(x int UNIQUE DEFERRABLE);
```

Criação de tabela cuja cláusula `UNIQUE` é não é deferível:

```
> CREATE TABLE tb_not_deferrable(x int UNIQUE NOT DEFERRABLE);
```

Habilitando cronômetro do psql:

```
> \timing on
```

Inserindo dados na tabela deferível:

```
> INSERT INTO tb_deferrable (x) SELECT n FROM tb_source;

ERROR: duplicate key value violates unique constraint "tb_deferrable_x_key"
DETAIL: Key (x)=(1) already exists.
Time: 10327,644 ms
```

Inserindo dados na tabela não deferível:

```
> INSERT INTO tb_not_deferrable (x) SELECT n FROM tb_source;

ERROR: duplicate key value violates unique constraint "tb_not_deferrable_x_key"
DETAIL: Key (x)=(1) already exists.
Time: 5642,530 ms
```

Redefinindo a tabela fonte como vazia:

```
> TRUNCATE tb_source;
```

Inserindo os dados na tabela fonte, desta vez sem repetir:

```
> INSERT INTO tb_source (n) SELECT generate_series(1, 2000000);
```

Inserindo dados na tabela deferível:

```
> INSERT INTO tb_deferrable (x) SELECT n FROM tb_source;

Time: 10751,544 ms
```

Inserindo dados na tabela não deferível:

```
> INSERT INTO tb_not_deferrable (x) SELECT n FROM tb_source;

Time: 11266,687 ms
```

Conclusão

No primeiro teste com dados repetidos o comando foi mais rápido e abortado imediatamente, enquanto na tabela com DEFERRABLE demorou mais pois a verificação foi feita posteriormente.

No teste sem repetição a tabela deferível se mostrou mais rápida para inserir, pois

não haviam dados repetidos que violassem a *constraint* UNIQUE.

7.7 Chave Primária / Primary Key

Seus valores são únicos (UNIQUE) e não nulos (NOT NULL), tecnicamente podemos dizer que: PRIMARY KEY = NOT NULL + UNIQUE.

Toda vez que criamos uma chave primária um índice (index) é atribuído para a mesma.

Declaração da chave primária na criação da tabela na mesma linha do campo:

```
> CREATE TEMP TABLE tb_pk1(  
    cod_prod int2 PRIMARY KEY,  
    preco float4 CHECK(preco > 0));
```

Verificando a estrutura:

```
> \d tb_pk1  
  
      Table "public.tb_pk1"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 cod_prod | smallint       | not null  
  preco   | real           |  
Indexes:  
    "tb_pk1_pkey" PRIMARY KEY, btree (cod_prod)  
Check constraints:  
    "tb_pk1_preco_check" CHECK (preco > 0::double precision)
```

Declaração da chave primária no final:

```
> CREATE TEMP TABLE tb_pk2(  
    cod_prod int2,  
    preco float4 CHECK(preco > 0),  
    PRIMARY KEY(cod_prod));
```

Verificando a estrutura:

```
> \d tb_pk2  
  
      Table "public.tb_pk2"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 cod_prod | smallint       | not null  
  preco   | real           |  
Indexes:  
    "tb_pk2_pkey" PRIMARY KEY, btree (cod_prod)  
Check constraints:  
    "tb_pk2_preco_check" CHECK (preco > 0::numeric)
```

Chave primária nomeando-a na criação da tabela na mesma linha do campo:

```
> CREATE TEMP TABLE tb_pk3(  
    cod_prod int2 CONSTRAINT pk_3 PRIMARY KEY,  
    preco float4 CHECK(preco > 0));
```

Verificando a estrutura:

```
> \d tb_pk3
```

```
      Table "public.tb_pk3"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 cod_prod | smallint       | not null  
  preco  | real           |  
Indexes:  
    "pk_3" PRIMARY KEY, btree (cod_prod)  
Check constraints:  
    "tb_pk3_preco_check" CHECK (preco > 0::numeric)
```

Declaração e nomeação da chave primária no final:

```
> CREATE TEMP TABLE tb_pk4(  
    cod_prod int2,  
    preco float4 CHECK(preco > 0),  
    CONSTRAINT pk_4 PRIMARY KEY(cod_prod));
```

Verificando a estrutura:

```
> \d tb_pk4
```

```
      Table "public.tb_pk4"  
  Column |      Type      | Modifiers  
-----+-----+-----  
 cod_prod | smallint       | not null  
  preco  | real           |  
Indexes:  
    "pk_4" PRIMARY KEY, btree (cod_prod)  
Check constraints:  
    "tb_pk4_preco_check" CHECK (preco > 0::numeric)
```

7.7.1 Chave Primária Composta

Até o momento lidamos apenas com chaves primárias simples. Ou seja, há apenas um campo compondo a mesma.

Chaves primárias compostas têm dois ou mais campos.

Esse tipo de chave é mais utilizada em tabelas associativas.

Criação de tabela com chave primária composta:

```
> CREATE TEMP TABLE tb_pk_dois_campos(  
  campo1 int2,  
  campo2 int2,  
  CONSTRAINT pk_dois_campos PRIMARY KEY (campo1, campo2));
```

A chave primária foi criada para os campos campo1 e campo2.

Em casos como esse, é o conjunto de valores dos campos que é avaliado para condição UNIQUE.

Inserção de dados:

```
> INSERT INTO tb_pk_dois_campos (campo1, campo2) VALUES (0, 0), (0, 1);
```

Nas duas inserções o valor 0 é repetido no campo1 e variando no campo2. Isso é permitido, pois o que a chave primária composta avalia é a unicidade do conjunto e não de apenas um campo.

Provocando Erro com Inserção de Valor Repetido:

```
> INSERT INTO tb_pk_dois_campos (campo1, campo2) VALUES (0, 0);
```

```
ERROR: duplicate key value violates unique constraint "pk_dois_campos"  
DETAIL: Key (campo1, campo2)=(0, 0) already exists.
```


7.8 Chave Estrangeira / Foreign Key

Especifica que o valor da coluna deve corresponder a um valor que esteja na coluna da tabela referenciada.

Os **valores dos campos referenciados devem ser únicos**. Chamamos isso de **integridade referencial**.

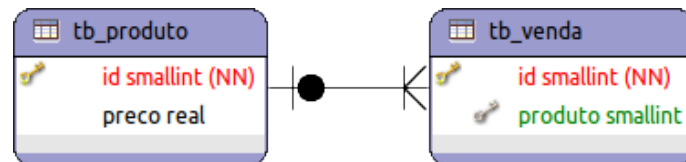


Figura 4: Relacionamento e chave estrangeira.

Acima está representado o relacionamento de venda com produto, vejamos como as tabelas foram criadas.

Criação da Tabela tb_produto:

```
> CREATE TEMP TABLE tb_produto(  
  id int2 PRIMARY KEY,  
  preco float4 CHECK(preco > 0));
```

Criação da Tabela tb_venda:

```
> CREATE TEMP TABLE tb_venda(  
  id int2 PRIMARY KEY,  
  produto int2 REFERENCES tb_produto (id));
```

O campo produto da tabela tb_venda, faz referência à chave primária da tabela tb_produto.

Assim como acontece com chaves primárias, também podemos nomear chaves estrangeiras:

Apagando a tabela:

```
> DROP TABLE tb_venda;
```

Criação da Tabela tb_venda com a chave estrangeira previamente nomeada:

```
> CREATE TEMP TABLE tb_venda(  
  id int2 PRIMARY KEY,  
  produto int2,  
  CONSTRAINT fk_prod FOREIGN KEY (produto) REFERENCES tb_produto (id));
```

Tentativa de exclusão da tabela referenciada:

```
> DROP TABLE tb_produto;
```

```
ERROR: cannot drop table tb_produto because other objects depend on it  
DETAIL: constraint fk_prod on table tb_venda depends on table tb_produto  
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Não foi possível apagar a tabela referenciada por causa da dependência.

Descrição da estrutura de tb_venda:

```
> \d tb_venda
```

```
Table "public.tb_venda"  
Column | Type | Modifiers  
-----+-----+-----  
id      | smallint | not null  
produto | smallint |  
Indexes:  
    "tb_venda_pkey" PRIMARY KEY, btree (id)  
Foreign-key constraints:  
    "fk_prod" FOREIGN KEY (produto) REFERENCES tb_produto(id)
```

Agora vamos usar a cláusula CASCADE para apagá-la efetivamente:

```
> DROP TABLE tb_produto CASCADE;
```

```
NOTICE: drop cascades to constraint fk_prod on table tb_venda
```

A chave estrangeira em tb_venda foi apagada também.

Verificando a tabela:

```
> \d tb_venda
```

```
Table "public.tb_venda"  
Column | Type | Modifiers  
-----+-----+-----  
id      | smallint | not null  
produto | smallint |  
Indexes:  
    "tb_venda_pkey" PRIMARY KEY, btree (id)
```

7.8.1 Cláusulas ON DELETE / ON UPDATE

Quando o dado na coluna referenciada é mudado, certas ações são executadas no dado dessa coluna.

As cláusulas `ON DELETE / ON UPDATE` especificam a ação a ser executada quando uma linha referenciada na tabela referenciada está sendo apagada / atualizada.

Ações:

- **NO ACTION** (padrão)

Produz um erro indicando que seria criada uma violação da chave estrangeira.

Se a constraint é deferida (*deferred*), esse erro será produzido na hora da checagem da constraint se ainda houver quaisquer linhas de referência.

- **RESTRICT**

Gera um erro indicando que um `DELETE` ou um `UPDATE` criará uma violação de chave estrangeira.

Similar a `NO ACTION`, exceto que checa se é não deferida (*not deferrable*).

- **CASCADE**

Propaga a ação para os registros na tabela referenciadora.

- **SET NULL**

Valores de tabelas referenciadoras são mudados para nulos.

- **SET DEFAULT**

Valores de tabelas referenciadoras são mudados para valores padrão da tabela referenciadora. Tais valores padrão têm que constar na tabela referenciada.

Criação de tabela referenciada:

```
> CREATE TEMP TABLE tb_marca(  
    id serial2 PRIMARY KEY,  
    nome text);
```

Criação de tabela referenciadora:

```
> CREATE TEMP TABLE tb_modelo(  
    id serial PRIMARY KEY,  
    marca int2 REFERENCES tb_marca (id) ON DELETE SET NULL ON UPDATE CASCADE,  
    modelo text);
```

Inserindo dados na tabela referenciada:

```
> INSERT INTO tb_marca (nome) VALUES
    ('Ford'), ('GM'), ('Volkswagen'), ('Fiat')
    RETURNING id, nome;
```

id	nome
1	Ford
2	GM
3	Volkswagen
4	Fiat

Inserindo dados na tabela referenciadora:

```
> INSERT INTO tb_modelo (marca, modelo) VALUES
    (1, 'Corcel'),
    (2, 'Chevette'),
    (3, 'Fusca'),
    (4, '147');
```

Apagando uma linha na tabela referenciada:

```
> DELETE FROM tb_marca WHERE id = 1;
```

Verificando os registros na tabela referenciadora:

```
> TABLE tb_modelo;
```

id	marca	modelo
7	2	Chevette
8	3	Fusca
9	4	147
6		Corcel

Atualizando uma linha na tabela referenciada:

```
> UPDATE tb_marca SET id = 33 WHERE id = 3;
```

Verificando os registros na tabela referenciadora:

```
> TABLE tb_modelo;
```

id	marca	modelo
7	2	Chevette
9	4	147
6		Corcel
8	33	Fusca

8 SELECT – Consultas (Queries)

- Sobre SELECT
- DISTINCT
- LIMIT e OFFSET
- A Cláusula WHERE
- A Cláusula ORDER BY
- CASE: Um IF Disfarçado
- A Função coalesce
- A Função nullif
- As Funções greatest e least
- O Comando TABLE

8.1 Sobre SELECT

Em um banco de dados não só armazenamos, como também buscamos informações.

O ato de buscar informações em uma base é fazer uma consulta; selecionar os dados. Ou se preferir, a palavra *query*, que em inglês significa consulta.

O comando `SELECT` é o mais utilizado para fazer consultas.

Sua sintaxe geral é:

```
SELECT campo1, campo2, campo3, ..., campoN FROM tabela;
```

É possível selecionar um ou mais campos conforme visto acima.

Caso desejar selecionar todos os campos sem precisar declará-los, o caractere “*” (asterisco) significa todos os campos.

Sintaxe:

```
SELECT * FROM tabela;
```

A seguir exemplos práticos com a base de dados de exemplo que é nosso laboratório de teste.

Verificando a estrutura da tabela de setores da empresa:

```
> \d tb_setor
```

```
Table "public.tb_setor"
Column |          Type          | Modifiers
-----+-----+-----
id      | integer                | not null default nextval('sq_setor'::regclass)
nome    | character varying(30)  |
Indexes:
    "pk_setor" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "tb_colaborador" CONSTRAINT "fk_setor_colaborador" FOREIGN KEY (setor) REFERENCES
tb_setor(id) ON UPDATE CASCADE ON DELETE CASCADE
```

A tabela de setores tem 2 (dois) campos: `id` e `nome`.

Selecionando todos os campos declarando explicitamente:

```
> SELECT id, nome FROM tb_setor;
```

```
id | nome
---+-----
1  | Presidência
2  | RH
3  | Logística
4  | Vendas
5  | Operações
```

Selecionando todos os campos declarando implicitamente com o caractere *:

```
> SELECT * FROM tb_setor;
```

```
id | nome
---+-----
1 | Presidência
2 | RH
3 | Logística
4 | Vendas
5 | Operações
```

Aviso.:

Evite o uso do *, pois há um *overhead* para o servidor determinar quais campos selecionar.

8.2 DISTINCT

A palavra-chave `DISTINCT` faz com que os resultados de uma consulta sejam únicos, elimina linhas repetidas. É um recurso interessante para selecionarmos os valores não repetidos de um resultado.

Selecionando todos os valores possíveis do campo `context` do catálogo `pg_settings`:

```
> SELECT DISTINCT context FROM pg_settings;
```

```
 context
-----
 backend
  user
 internal
 postmaster
 superuser
 sighup
 superuser-backend
```

Cada valor de `context` exibido tem mais de uma ocorrência, no entanto, cada um só teve uma representação.

Também podemos trazer um resultado distinto da combinação de dois ou mais campos:

```
> SELECT DISTINCT context, vartype FROM pg_settings LIMIT 5;
```

```
 context | vartype
-----+-----
 sighup  | integer
 superuser | integer
 user    | string
 sighup  | string
 user    | integer
```


8.3 LIMIT e OFFSET

LIMIT e OFFSET combinados faz com que os resultados possam ser paginados, ou seja, exibir um conjunto de linhas por vez.

LIMIT faz a limitação de quantos registros exibir.

OFFSET determina quantos ficarão de fora.

Limitando o resultado a 3 linhas:

```
> SELECT cpf, id, setor, salario FROM tb_colaborador LIMIT 3;
```

cpf	id	setor	salario
11111111111	1	1	20000.00
23625814788	2	1	10000.00
33344455511	3	2	4500.00
12345678901	4	1	5000.00
10236547895	5	2	3500.00

Tirando do resultado as 35 primeiras linhas:

```
> SELECT cpf, id, setor, salario FROM tb_colaborador OFFSET 35;
```

cpf	id	setor	salario
35999999143	36	5	1000.00
33777777201	37	5	1000.00
71833333250	38	5	1000.00
35900220143	39	5	750.00
33011111201	40	5	750.00

A palavra-chave OFFSET significa algo como um conjunto (set) fora (off). Ou seja, retirar do resultado os N primeiros registros.

Fazendo paginação combinando LIMIT e OFFSET:

```
> SELECT cpf, id, setor, salario FROM tb_colaborador LIMIT 5 OFFSET 5;
```

cpf	id	setor	salario
14725836944	6	2	3500.00
36925814788	7	2	1000.00
95184736277	8	2	1000.00
77789951452	9	3	4500.00
65456457214	10	3	3500.00

Unindo LIMIT e OFFSET podemos fazer paginação dos resultados com a limitação de quantas linhas exibir com LIMIT e determinando quantos resultados ficarão de fora com OFFSET. Assim progressivamente caminhando entre os registros.

8.4 A Cláusula WHERE

É o recurso utilizado para fazer filtragem de resultados.

Algo como “SELECIONAR os campos x, y, z da tabela ONDE um determinado campo tenha um valor igual a ...”.

Só são exibidos os resultados que atenderem a condição estipulada na consulta.

A ordem dos campos não faz diferença.

Exibir cpf e salario onde o setor seja igual a 2:

```
> SELECT cpf, salario FROM tb_colaborador WHERE setor = 2;
```

cpf	salario
33344455511	4500.00
10236547895	3500.00
14725836944	3500.00
36925814788	1000.00
95184736277	1000.00

Selecionar os campos agrupados dentro de parênteses:

```
> SELECT (cpf, salario) FROM tb_colaborador WHERE setor = 2;
```

row
(33344455511,4500.00)
(10236547895,3500.00)
(14725836944,3500.00)
(36925814788,1000.00)
(95184736277,1000.00)

A coluna resultante foi nomeada como "row".

Fazendo uso do recurso de apelido de coluna:

```
> SELECT (cpf, salario) AS "CPF e Salário" FROM tb_colaborador WHERE setor = 2;
```

CPF e Salário
(33344455511,4500.00)
(10236547895,3500.00)
(14725836944,3500.00)
(36925814788,1000.00)
(95184736277,1000.00)

Por ter letras maiúsculas e espaços, foi necessário colocar o apelido da coluna entre aspas.

Selecionando campos e dando apelidos mais amigáveis a eles para exibição:

```
> SELECT cpf AS "CPF", salario AS "Salário" FROM tb_colaborador WHERE setor = 2;
```

CPF	Salário
33344455511	4500.00
10236547895	3500.00
14725836944	3500.00
36925814788	1000.00
95184736277	1000.00

Uma nova consulta utilizando o recurso de apelido de coluna:

```
> SELECT cpf AS "CPF", salario AS "Salário", id AS matricula FROM tb_colaborador WHERE setor = 2;
```

CPF	Salário	matricula
33344455511	4500.00	3
10236547895	3500.00	5
14725836944	3500.00	6
36925814788	1000.00	7
95184736277	1000.00	8

Para apelidos sem espaços ou letras maiúsculas não é preciso de aspas.

Exibir cpf e id onde o cargo seja 7 e o chefe direto seja diferente de 21:

```
> SELECT cpf, id FROM tb_colaborador WHERE cargo = 7 AND chefe_direto != 21;
```

cpf	id
36925814788	7
95184736277	8

Conforme os campos selecionados buscar o que tiver o campo obs como não nulo:

```
> SELECT nome, sobrenome, cpf FROM tb_pf WHERE obs IS NOT NULL;
```

nome	sobrenome	cpf
Chiquinho	da Silva	11111111111

Foi exibido o único registro em que o campo obs não é nulo.

Um campo utilizado como critério de consulta pode ou não ser exibido.

Quando um campo é utilizado como critério de consulta e não é selecionado, chamamos de coluna invisível.

Selecionar nome, sobrenome e cpf da tabela tb_pf em que obs não seja preenchida e genero seja igual a "f":

```
> SELECT nome, sobrenome, cpf FROM tb_pf WHERE obs IS NULL AND genero = 'f';
```

nome	sobrenome	cpf
Aldebarina	Ferreira	23625814788
Wolfrâmia	Santos	33344455511
Tungstênia	Santana	12345678901
Urânia	Gomes	10236547895
Carmezilda	Gonçalves	95184736277
Valverinda	Ramalho	77789951452
Acabézia	Monteiro	96385274133
Aldebranda	Luz	85274136944
Estrogofina	Carvalho	96325874100
Edervina	Silva	71856987250
Dalclézia	Gomes	26854774475
Claudemira	Santos	25455545544
Edonina	Oliveira	35655754578
Carvézia	Guerra	56987569872
Alfrinalda	Monteiro	65842484247
Torvelina	Arruda	54858547451
Esmeringalda	Barreiras	87557845484
Gioconda	Pereira	71822227250
Murila	Vieira	70000007250
Maria	dos Santos	71833333250

Aviso:

Para verificar se um valor é nulo não se usa o sinal de igual "=" mas sim o operador IS (e. g. IS NULL).

O mesmo se aplica para buscar também valores **não nulos** (e. g. IS NOT NULL).

Um critério de busca pode ter várias condições:

```
> SELECT id FROM tb_colaborador WHERE setor = 2 OR setor = 3;
```

id
3
5
6
7
8
9
10
11
12
13
14
15

Quando uma consulta tem mais de uma condição com a lógica OR, pode ser facilmente substituída por IN:

```
> SELECT id FROM tb_colaborador WHERE setor IN (2, 3);
```

```
id
----
3
5
6
7
8
9
10
11
12
13
14
15
```

O comando anterior é outro exemplo que podemos melhorar a sintaxe, através do agrupamento de comparações.

Selecionar o que não está entre os valores entre parênteses:

```
> SELECT id FROM tb_colaborador WHERE setor NOT IN (3, 2, 5, 4);
```

```
id
----
1
2
4
```

Consulta com várias condições AND:

- | | |
|----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <p>1)</p> <pre>> SELECT id FROM tb_colaborador WHERE cargo = 7 AND chefe_direto = 2 AND ativo = true;</pre> | <p>2)</p> <pre>> SELECT id FROM tb_colaborador WHERE cargo = 7 AND chefe_direto = 2 AND ativo;</pre> |
| <p>3)</p> <pre>> SELECT id FROM tb_colaborador WHERE (cargo, chefe_direto, ativo) = (7, 2, true);</pre> | |

```
id
----
7
8
```

As três consultas são equivalentes. Na primeira o campo booleano (ativo) seu valor foi explícito, na segunda houve uma abreviação e na terceira foi feito uso de interpolação.

Selecionar o id de quem tem seu salário entre 1700 a 2000:

```
> SELECT id FROM tb_colaborador
    WHERE salario >= 1700
    AND salario <= 2000;
```

```
> SELECT id FROM tb_colaborador
    WHERE salario BETWEEN 1700 AND 2000;
```

```
id
---
17
18
19
20
21
```

Ambas consultas produzem o mesmo resultado, mas com o uso de BETWEEN é mais fácil de entender.

Selecionar o que não está entre 750 e 5000 de salário:

```
> SELECT id FROM tb_colaborador WHERE salario NOT BETWEEN 750 AND 5000;
```

```
id
---
1
2
```

8.5 A Cláusula ORDER BY

Quando é feita uma consulta sem se especificar um critério de ordenação, os registros são exibidos por ordem de inserção ou atualização.

Através da cláusula `ORDER BY` ordena-se consultas de acordo com os valores de uma determinada coluna.

Consultar funcionários do setor 2 ordenando pelo salário:

```
> SELECT cpf, salario FROM tb_colaborador WHERE SETOR = 2 ORDER BY salario;
```

cpf	salario
36925814788	1000.00
95184736277	1000.00
10236547895	3500.00
14725836944	3500.00
33344455511	4500.00

Por padrão consultas com a cláusula `ORDER BY` são exibidas em ordem ascendente (ASC).

Ordem crescente explícita:

```
> SELECT cpf, salario FROM tb_colaborador WHERE SETOR = 2 ORDER BY salario ASC;
```

cpf	salario
36925814788	1000.00
95184736277	1000.00
10236547895	3500.00
14725836944	3500.00
33344455511	4500.00

O resultado foi o mesmo da consulta anterior, pois o efeito foi o mesmo, apenas foi explicitada a ordem.

Ordem decrescente (DESC):

```
> SELECT cpf, salario FROM tb_colaborador WHERE setor = 2 ORDER BY salario DESC;
```

cpf	salario
33344455511	4500.00
10236547895	3500.00
14725836944	3500.00
36925814788	1000.00
95184736277	1000.00

É possível também especificarmos mais de uma coluna e tipos de ordem diferentes pra cada, por exemplo consultar primeiramente pelo salário decrescente e cpf crescente.

Ordens diferentes por campo:

```
> SELECT cpf, salario FROM tb_colaborador WHERE setor = 2
   ORDER BY salario DESC, cpf ASC;
```

cpf	salario
33344455511	4500.00
10236547895	3500.00
14725836944	3500.00
36925814788	1000.00
95184736277	1000.00

Uma consulta levando-se em conta primeiro o sobrenome (sname) e depois o nome:

```
> SELECT * FROM tb_pf ORDER BY 4, 6, 1 LIMIT 5;
```

nome	dt_nascto	obs	cpf	rg	sobrenome	genero
Urânia	1983-01-03		10236547895	8741235678	Gomes	f
Chiquinho	1956-02-08	The Big Boss!!!	11111111111	1234567890	da Silva	m
Tungstênia	1970-10-03		12345678901	2345678910	Santana	f
Alderovaldo	1991-07-18		14700258369	1111119875	Souza	m
Estrôncio	1980-09-13		14725836944	1287425671	dos Santos	m

Ao invés dos nomes das colunas podemos usar a classificação posicional.

Levando-se em conta que os campos são numerados da esquerda pra direita e iniciando por 1, temos cpf=4, sobrenome=6 e nome=1. Essa numeração é conforme as colunas exibidas, sendo assim, a primeira coluna a ser exibida vai ser a coluna 1.

Apelidos de colunas também são considerados e podem ser utilizados como critério de ordenamento:

```
> SELECT dt_nascto, nome||' '||sobrenome AS "Nome Completo"
   FROM tb_pf ORDER BY "Nome Completo" LIMIT 5;
```

dt_nascto	Nome Completo
1990-02-02	Acabêzia Monteiro
1988-01-01	Adofrino Dias
1978-03-01	Aldebarina Ferreira
1983-01-03	Aldebranda Luz
1991-07-18	Alderovaldo Souza

dt_nascto como coluna invisível:

```
> SELECT nome, sobrenome FROM tb_pf ORDER BY dt_nascto LIMIT 5;
```

nome	sobrenome
Chiquinho	da Silva
Rondonino	Sales
Tungstênia	Santana
Edervalson	Torres
Estriga	Souto

Colunas invisíveis de ordenação, são assim chamadas aquelas que são usadas como critério de classificação, mas não aparecem no resultado. A coluna dt_nascto não aparece no resultado.

8.6 CASE: Um IF Disfarçado

O comando CASE, para quem conhece linguagens de programação, pode ser comparado ao IF.

Em vez de usarmos a sequência “IF, THEN, ELSE”, usamos “CASE (caso), WHEN (quando), THEN (então), ELSE (senão)” e finalizamos com um “END (fim)”.

O bloco delimitado por CASE e END é representado por uma coluna na consulta.

CASE para o campo cargo:

```
> SELECT id, CASE cargo
      WHEN 1 THEN 'Presidência'
      WHEN 3 THEN 'Gerência'
      ELSE 'Operacional'
      END, salario FROM tb_colaborador LIMIT 5;
```

id	case	salario
1	Presidência	20000.00
2	Presidência	10000.00
3	Gerência	4500.00
4	Operacional	5000.00
5	Operacional	3500.00

Repare que a coluna case é a que representa o comando CASE.

Podemos dar um apelido para a coluna deixando-a de uma forma mais legível:

```
> SELECT
      id,
      CASE cargo
        WHEN 1 THEN 'Presidência'
        WHEN 3 THEN 'Gerência'
        ELSE 'Operacional'
      END AS "Nível", -- Apelido para a coluna
      salario FROM tb_colaborador LIMIT 5;
```

id	Nível	salario
1	Presidência	20000.00
2	Presidência	10000.00
3	Gerência	4500.00
4	Operacional	5000.00
5	Operacional	3500.00

Caso o cargo seja igual a 1; retornará Presidência, igual a 3; retornará Gerência, senão; retornará Operacional.

Sintaxe com operador logo após o nome do campo:

```
> SELECT DISTINCT
  CASE
    WHEN setor = 1 THEN 'Presidência'
    WHEN setor = 2 THEN 'RH'
    WHEN setor = 3 THEN 'Logística'
    WHEN setor = 4 THEN 'Vendas'
    WHEN setor = 5 THEN 'Operações'
    ELSE 'Outros'
  END AS "Setor"
FROM tb_colaborador;
```

```
      Setor
-----
      RH
Logística
Operações
Presidência
Vendas
```

8.7 A Função coalesce

Em certas situações, quando há valores nulos em uma tabela é necessário preenchê-los provisoriamente com algum valor. O comando `coalesce` faz isso.

A tabela `tb_pf` (pessoa física) será usada, pois tem valores nulos no campo `obs`:

```
> SELECT nome, sobrenome, coalesce(obs, '...') FROM tb_pf LIMIT 5;
```

nome	sobrenome	coalesce
Genovézio	Gomes	...
Aldebarina	Ferreira	...
Wolfrâmia	Santos	...
Tungstênia	Santana	...
Urânia	Gomes	...

O comando `coalesce` fez com que os valores do campo `obs` que são nulos fossem mascarados. Ou seja, em vez de um vazio, no exemplo é exibida reticências, conforme determinado como string do segundo parâmetro.

8.8 A Função nullif

É uma função que retorna um valor nulo se o primeiro valor for igual ao segundo valor, caso contrário retornará o primeiro valor.

Iguais retorna nulo:

```
> SELECT nullif(1, 1);
```

```
 nullif  
-----
```

Retorna 5:

```
> SELECT nullif(5, 2);
```

```
 nullif  
-----  
      5
```

Retorna 2:

```
> SELECT nullif(2, 5);
```

```
 nullif  
-----  
      2
```

8.9 As Funções greatest e least

As funções greatest e least selecionam respectivamente o maior ou menor valor de uma lista de qualquer número de expressões.

As expressões devem ser todas compatíveis com o tipo de dados comum, que será o tipo de resultado.

Valores nulos na lista serão ignorados. O resultado será nulo apenas se todas as expressões avaliadas forem também nulas.

Essas funções não fazem parte do padrão SQL, mas são uma extensão comum.

Alguns outros bancos de dados farão o retorno nulo se qualquer argumento for nulo, em vez de somente quando todos forem nulos.

<p>Qual é o maior número?:</p> <pre>> SELECT greatest(2, 1, -9, 55, 20); greatest ----- 55</pre>	<p>Qual é o menor número?:</p> <pre>> SELECT least(2, 1, -9, 55, 20); least ----- -9</pre>
<p>Qual das letras é a última em ordem alfabética?:</p> <pre>> SELECT greatest('n', 'w', 'b'); greatest ----- w</pre>	<p>Qual das letras é a última em ordem alfabética?:</p> <pre>> SELECT least('n', 'w', 'b'); least ----- b</pre>
<p>Um valor nulo contra um inteiro:</p> <pre>> SELECT greatest(NULL, 5); greatest ----- 5</pre>	<p>Um valor nulo contra um inteiro:</p> <pre>> SELECT least(NULL, 5); least ----- 5</pre>

8.10 O Comando TABLE

O comando TABLE tem o mesmo efeito que SELECT * FROM, ou seja:

TABLE = SELECT * FROM

Selecionar todos os dados de tb_cargo limitando a 5 linhas:

```
> TABLE tb_cargo LIMIT 5;
```

id	nome
1	Presidente
2	Secretária
3	Gerente
4	Assistente
5	Office boy

É possível ordenar:

```
> TABLE tb_cargo ORDER BY id DESC LIMIT 5;
```

id	nome
9	Faxineiro
8	Vendedor
7	Ajudante Geral
6	Moto boy
5	Office boy

Não é possível usar a cláusula WHERE:

```
> TABLE tb_cargo WHERE id = 1;
```

```
ERROR: syntax error at or near "WHERE"
LINE 1: TABLE tb_cargo WHERE id = 1;
                        ^
```

9 DML: Inserir, Alterar e Deletar Registros

- Sobre INSERT
- Dollar Quoting
- UPDATE – Alterando Registros
- DELETE – Removendo Registros
- A Cláusula RETURNING
- UPSERT – ON CONFLICT

9.1 Sobre INSERT

O comando INSERT cria novos registros em uma tabela.

Sintaxe Geral:

```
INSERT INTO tabela (coluna1, coluna2, ..., colunaN)
VALUES (valor1, valor2, ..., valorN);
```

ou

```
INSERT INTO tabela
VALUES (valor1, valor2, ..., valorN);
```

Exemplos:

Inserção em tb_pf I:

```
> INSERT INTO tb_pf (nome, dt_nascto, obs, cpf, rg, sobrenome, genero)
VALUES
('Alzerbina', '1960-02-22', 'Nada...', '23549878125', '7485966352', 'Marques', 'f');
```

Inserção em tb_pf II:

```
> INSERT INTO tb_pf VALUES
('Branestézio', '1975-11-05', 'Uma pessoa...', '43549878135', '7415916352',
'Almeida', 'm');
```

O efeito de inserção nos exemplos I e II é o mesmo. Em ambos os casos há valores declarados para todos os campos.

Nota-se que no exemplo II os campos não foram declarados. Quando todos os campos terão valores a serem inseridos não há necessidade de declará-los.

Inserção de valores para campos não numéricos, tais como strings, datas, tempo, etc, o valor deve ficar entre apóstrofes.

Tipos numéricos, palavras-chave, chamada de funções ou identificadores não são envolvidos por apóstrofes.

Criação da tabela de teste:

```
> CREATE TEMP TABLE tb_teste_ins(
    campo1 INT2 DEFAULT 0,
    campo2 DATE,
    campo3 VARCHAR(10));
```


Insert I:

```
> INSERT INTO tb_teste_ins (campo1, campo2, campo3) VALUES (5, now(), 'abcde');
```

Em `now()`, a qual retorna data e hora corrente.

Insert II:

```
> INSERT INTO tb_teste_ins (campo1, campo2, campo3)
VALUES (DEFAULT, '2012-10-17', 'xyz');
```

Para preenchimento do valor de `DEFAULT` foi utilizada.

Selecionando todos os dados da tabela:

```
> SELECT * FROM tb_teste_ins;
```

<i>campo1</i>	<i>campo2</i>	<i>campo3</i>
5	2012-10-16	abcde
0	2012-10-17	xyz

9.1.1 INSERT Múltiplo

Desde a versão 8.0, o PostgreSQL suporta este recurso que facilita a inserção de dados em uma tabela.

Seu uso consiste em, com apenas 1 (um) INSERT fazer várias inserções.

Sintaxe Geral de INSERT Múltiplo

```
INSERT INTO tabela [(campo1, campo2, ..., campoN)] VALUES
(valor1, valor2, ..., valorN),
(valor1, valor2, ..., valorN),
(valor1, valor2, ..., valorN),
(valor1, valor2, ..., valorN);
```

Cada inserção é delimitada por parênteses e separada por vírgulas.

5 (Cinco) novos registros com 1 (Um) INSERT:

```
> INSERT INTO tb_teste_ins VALUES
  (3, current_timestamp, current_user),
  (DEFAULT, now(), '123'),
  (0, '2012-07-13', '5340z'),
  (478, '2002-03-01', (5 + 2)::VARCHAR),
  (DEFAULT, now(), '');
```

Todos os dados da tabela:

```
> TABLE tb_teste_ins;
```

campo1	campo2	campo3
5	2012-10-16	abcde
0	2012-10-17	xyz
3	2012-10-16	postgres
0	2012-10-16	123
0	2012-07-13	5340z
478	2002-03-01	7
0	2012-10-16	

9.2 Dollar Quoting

Dollar Quoting é uma forma alternativa de inserir strings sem usar apóstrofos e de escapar caracteres dando uma maior segurança para o banco de dados, um grande aliado contra um tipo de ataque conhecido como “*SQL Injection*”.

O pode ser feito com o marcador de string vazia:

```
$$sua string$$
```

ou com marcador nomeado:

```
$marcador$sua string$marcador$
```

Quando o marcador é nomeado, o primeiro caractere só pode ser letras ou underline (_).

Criação da tabela de teste:

```
> CREATE TEMP TABLE tb_dollar_quoting(campo text);
```

Inserindo valores:

```
> INSERT INTO tb_dollar_quoting VALUES  
  ($$('teste1'$$), ($outro_marcaador$'teste2'$outro_marcaador$));
```

Consulta com marcador simples (string vazia entre os símbolos de dólar):

```
> SELECT campo FROM tb_dollar_quoting WHERE campo = $$'teste1'$$;
```

campo
'teste1'

Consulta utilizando marcador nomeado:

```
> SELECT campo FROM tb_dollar_quoting  
  WHERE campo = $marcador$'teste2'$marcador$;
```

campo
'teste2'

Apagando um registro:

```
> DELETE FROM tb_dollar_quoting
    WHERE campo = $sabracadabra$'teste1'$sabracadabra$ RETURNING campo;

 campo
-----
'teste1'
```

Atualizando um registro:

```
> UPDATE tb_dollar_quoting SET campo = $_123$'''Docstring Python'''$_123$
    WHERE campo = $a_$'teste2'$a_$ RETURNING campo;

 campo
-----
'''Docstring Python'''
```

Inserindo um novo registro:

```
> INSERT INTO tb_dollar_quoting (campo) VALUES ($apóstrofo -> '$$');
```

Verificando a tabela:

```
> TABLE tb_dollar_quoting;

 campo
-----
'''Docstring Python'''
apóstrofo -> '
```

9.3 UPDATE – Alterando Registros

O comando UPDATE é utilizado para mudar registros de uma tabela.

Sintaxe Geral:

```
UPDATE tabela SET coluna = valor WHERE condição;
```

Vamos usar o comando UPDATE para atualizar a tabela `tb_colaborador`, levando-se em conta que os trabalhadores que ganham menos ou igual a R\$ 1.200,00 e estejam ativos receberão 10% de aumento:

Aumento para os que ganham menos ou igual a R\$ 1.200,00:

```
> UPDATE tb_colaborador SET salario = salario * 1.1  
  WHERE salario <= 1200 AND ativo = true;
```

Todos registros da tabela:

```
> TABLE tb_colaborador;
```

Nota-se que ao fazer uma consulta dos registros, após a atualização, os registros afetados saíram de sua ordem original passando agora para o final de todos os registros.

9.3.1 Atualizando Mais de Um Campo

Podemos também fazer atualização de registros referenciando mais de um campo.

Criação de tabela:

```
> CREATE TEMP TABLE tb_upd(cor varchar(10), temperatura int2);
```

Inserção de registros:	Alteração de registros:
<pre>> INSERT INTO tb_upd VALUES ('verde', 77), ('verde', 80), ('amarelo', 11), ('azul', 32), ('verde', 20), ('cinza', 3), ('vermelho', 10), ('verde', 25), ('vermelho', 5), ('vermelho', 20);</pre>	<pre>> UPDATE tb_upd SET (cor, temperatura) = ('verde', 100) WHERE cor = 'vermelho' AND temperatura <= 10;</pre>

9.4 DELETE – Removendo Registros

Para apagar linhas de uma tabela usa-se o comando DELETE, podendo inclusive usar sub-consultas.

Sintaxe Geral:

```
DELETE FROM tabela WHERE coluna condição;
```

Apagar registros em que o valor do campo dt_nascto em tb_pf for nulo:

```
> DELETE FROM tb_pf WHERE dt_nascto IS NULL;
```

9.5 A Cláusula RETURNING

É um recurso muito interessante que faz com que retorne campos inseridos, atualizados ou apagados.

Criação da tabela temporária para os testes:

```
> CREATE TEMP TABLE tb_teste_returning(  
  id serial PRIMARY KEY,  
  campo1 TEXT,  
  campo2 SMALLINT);
```

9.5.1 RETURNING com INSERT

Inserindo e retornando o campo id:

```
> INSERT INTO tb_teste_returning (campo1, campo2) VALUES  
  ('foo', 452),  
  ('bar', 37)  
  RETURNING id;
```

```
 id  
----  
 1  
 2
```

Verificando a tabela:

```
> TABLE tb_teste_returning;
```

```
 id | campo1 | campo2  
----+-----+-----  
  1 | foo    |    452  
  2 | bar    |     37
```

9.5.2 RETURNING com UPDATE

Atualizando e retornando dois campos (separados por vírgula):

```
> UPDATE tb_teste_returning SET campo1 = 'baz'  
  WHERE campo1 = 'bar' RETURNING id, campo2;
```

```
 id | campo2  
----+-----  
  2 |     37
```

Verificando a tabela:

```
> TABLE tb_teste_returning;
```

<i>id</i>	<i>campo1</i>	<i>campo2</i>
1	foo	452
2	baz	37

9.5.3 RETURNING com DELETE

Apagando e retornando todos os campos da linha apagada:

```
> DELETE FROM tb_teste_returning WHERE id = 2 RETURNING *;
```

<i>id</i>	<i>campo1</i>	<i>campo2</i>
2	baz	37

Verificando a tabela:

```
> TABLE tb_teste_returning;
```

<i>id</i>	<i>campo1</i>	<i>campo2</i>
1	foo	452

9.6 “UPSERT”: INSERT ON CONFLICT

Interessante recurso adicionado ao PostgreSQL a partir da versão 9.5.

É feita uma verificação se já existe o registro e se não existir se comporta como um INSERT, se existe pode se comportar como um UPDATE.

Criação de tabela de teste:

```
> CREATE TABLE tb_upsert(  
    id serial primary key,  
    nome text);
```

Inserindo um valor (fazer duas vezes):

```
> INSERT INTO tb_upsert VALUES (1, 'Joana');
```

```
ERROR: duplicate key value violates unique constraint "tb_upsert_pkey"  
DETAIL: Key (id)=(1) already exists.
```

Na primeira vez não teve problema, mas na segunda não foi permitido o INSERT, pois o registro é exatamente igual, inclusive o campo id, que é único apontou um conflito.

Verificando a tabela:

```
> TABLE tb_upsert;
```

```
 id | nome  
----+-----  
  1 | Joana
```

Tentativa de inserir registro, porém, se der conflito não fazer nada:

```
> INSERT INTO tb_upsert VALUES (1, 'Joana') ON CONFLICT DO NOTHING;
```

Tentativa de inserir registro, porém, se der conflito não fazer nada:

```
> INSERT INTO tb_upsert VALUES (1, 'Maria') ON CONFLICT DO NOTHING;
```

Tentativa de inserir registro, porém, se der conflito efetuar uma atualização:

```
> INSERT INTO tb_upsert VALUES (1, 'Maria')  
    ON CONFLICT (id) DO UPDATE SET nome = 'Maria';
```

Verificando a tabela:

```
> TABLE tb_upsert;
```

id	nome
1	Maria

9.6.1 Sobrescrevendo Registros com UPSERT

Criação da tabela de teste:

```
> CREATE TEMP TABLE tb_upsert2(  
    id serial primary key,  
    nome text);
```

Inserir um valor:

```
> INSERT INTO tb_upsert2 VALUES (1, 'Joana');
```

Verificar a tabela:

```
> TABLE tb_upsert2;
```

id	nome
1	Joana

Tentativa de inserir um registro:

```
> INSERT INTO tb_upsert2 VALUES (1, 'Maria') ON CONFLICT DO NOTHING;
```

Verificar a tabela:

```
> TABLE tb_upsert2;
```

id	nome
1	Joana

A tabela permanece a mesma.

Inserindo com a cláusula ON CONFLICT:

```
> INSERT INTO tb_upsert2 VALUES (1, 'Maria')
   ON CONFLICT (id) DO UPDATE SET nome = 'Maria';
```

Verificar a tabela:

```
> TABLE tb_upsert2;
```

id	nome
1	Maria

Verificar a estrutura da tabela:

```
> \d tb_upsert2

Table "pg_temp_2.tb_upsert2"
Column | Type          | Modifiers
-----+-----+-----
id      | integer       | not null default nextval('tb_upsert2_id_seq'::regclass)
nome    | text          |
Indexes:
    "tb_upsert2_pkey" PRIMARY KEY, btree (id)
```

O campo id tem uma sequência (tb_upsert2_id_seq) atrelada a ele. Seu valor padrão é o próximo valor dessa sequência.

Verificando o último valor dessa sequência e se ela já foi chamada:

```
> SELECT last_value, is_called FROM tb_upsert2_id_seq;
```

last_value	is_called
1	f

Chamando o próximo valor da sequência:

```
> SELECT nextval('tb_upsert2_id_seq');
```

nextval
1

Verificando o último valor dessa sequência e se ela já foi chamada:

```
> SELECT last_value, is_called FROM tb_upsert2_id_seq;
```

last_value	is_called
1	t

Após a sequência ter sido usada, o valor de is_called foi alterado para true.

Fazemos essa verificação na sequência para que não dê problemas com a chave primária, que deve ser única.

Inserindo um novo registro na tabela:

```
> INSERT INTO tb_upsert2 (nome) VALUES ('Joana');
```

Verificando a tabela:

```
> TABLE tb_upsert2;
```

id	nome
1	Maria
2	Joana

Inserindo dois registros na tabela sobrescrevendo os pré existentes:

```
> INSERT INTO tb_upsert2 VALUES  
  (1, 'Sara'),  
  (2, 'Laura')  
  ON CONFLICT (id) DO UPDATE SET nome = excluded.nome;
```

Verificando a tabela:

```
> TABLE tb_upsert2;
```

id	nome
1	Sara
2	Laura

Podemos constatar que utilizando excluded.nome_da_coluna podemos sobrescrever completamente os registros que já existiam.

10 Novas Tabelas a Partir de Dados

- Sobre
- SELECT INTO
- CREATE TABLE AS
- CREATE TABLE ... (like ...)

10.1 Sobre

Pode surgir a necessidade de criar uma tabela cuja estrutura seja uma consulta ou mesmo uma tabela pré existente.

`SELECT INTO` e `CREATE TABLE AS` criam uma tabela a partir de uma consulta, mas não copiam sua estrutura.

`CREATE TABLE ... (like ...)` é capaz de copiar a estrutura da tabela de origem, no entanto sem os dados. Isso pode ser facilmente contornado com um posterior `INSERT` com `SELECT` na tabela criada.

10.2 SELECT INTO

Cria uma nova tabela, cujo nome é especificado pela cláusula `INTO`. Essa criação é com base na consulta feita cujos campos terão os mesmos tipos na estrutura gerada. Essa nova tabela não vai ter nem índices e nem *constraints*.

A partir da tabela `tb_pf` criar uma tabela:

```
> SELECT
    nome, dt_nascto, obs, cpf, rg, sobrenome
    INTO tb_pf_mulher
    FROM tb_pf
    WHERE genero = 'f';
```

A consulta não retornou nenhuma linha, mas inseriu tudo na tabela `tb_pf_mulher`.

Os campos criados a partir dessa consulta são dos mesmos tipos que os campos da tabela original.

10.3 CREATE TABLE AS

Conceito igual ao de `SELECT INTO`, mas com uma sintaxe diferente.

Criar uma tabela já com valores inseridos a partir da consulta:

```
> CREATE TABLE tb_pf80
  AS SELECT * FROM tb_pf
  WHERE dt_nascto
  BETWEEN '1980-01-01' AND '1989-12-31';
```


10.4 CREATE TABLE ... (like ...)

Criar uma tabela copiando a estrutura de outra.

Essa cópia pode ser feita entre tabelas normais e / ou tabelas temporárias.

A opção `like` pode incluir (`INCLUDING`) ou excluir (`EXCLUDING`) opções como *constraints*, índices e outros. A palavra-chave `ALL` é usada para todas as opções de inclusão ou exclusão.

<https://www.postgresql.org/docs/current/static/sql-createtable.html>

Criação da tabela temporária a partir da qual outra será criada:

```
> CREATE TEMP TABLE tb_foo(  
  id serial PRIMARY KEY,  
  campo int CHECK (campo > 0));
```

Criação de uma nova tabela a partir da estrutura de outra com todas constraints e índices:

```
> CREATE TABLE tb_bar (like tb_foo INCLUDING ALL);
```

Estrutura de `tb_foo`:

```
> \d tb_foo  
  
          Table "pg_temp_2.tb_foo"  
+-----+-----+  
Column | Type | Modifiers  
+-----+-----+  
id      | integer | not null default nextval('tb_foo_id_seq'::regclass)  
campo   | integer |  
Indexes:  
    "tb_foo_pkey" PRIMARY KEY, btree (id)  
Check constraints:  
    "tb_foo_campo_check" CHECK (campo > 0)
```

Estrutura de `tb_bar`:

```
> \d tb_bar  
  
          Table "public.tb_bar"  
+-----+-----+  
Column | Type | Modifiers  
+-----+-----+  
id      | integer | not null default nextval('tb_foo_id_seq'::regclass)  
campo   | integer |  
Indexes:  
    "tb_bar_pkey" PRIMARY KEY, btree (id)  
Check constraints:  
    "tb_foo_campo_check" CHECK (campo > 0)
```

Como pode se notar a estrutura de ambas é igual, mudando apenas o nome da chave primária e do índice, que é conforme o nome da tabela.

Pode-se inclusive adicionar campos extras à nova tabela criada:

```
> CREATE TABLE tb_baz (  
  like tb_foo INCLUDING ALL,  
  obs text);
```

Estrutura de tb_baz:

```
> \d tb_baz
```

```
Table "public.tb_baz"  
Column | Type      | Modifiers  
-----+-----+-----  
id      | integer   | not null default nextval('tb_foo_id_seq'::regclass)  
campo   | integer   |  
obs     | text      |  
Indexes:  
    "tb_baz_pkey" PRIMARY KEY, btree (id)  
Check constraints:  
    "tb_foo_campo_check" CHECK (campo > 0)
```

11 Conjuntos

- Sobre Conjuntos
- União (UNION)
- Intersecção (INTERSECT)
- Diferença (EXCEPT)

11.1 Sobre Conjuntos

Resultados de duas ou mais consultas podem ser combinados para operações de conjuntos que são elas: união (UNION), intersecção (INTERSECT) e diferença (EXCEPT), cuja sintaxe é:

```
query1 UNION [ALL] query2  
query1 INTERSECT [ALL] query2  
query1 EXCEPT [ALL] query2
```

query1 e query2 são consultas que podem usar qualquer uma das características discutidas aqui.

Operações de conjunto podem também ser aninhadas ou agrupadas, por exemplo:

```
query1 UNION query2 UNION query3
```

que é executada como:

```
(query1 UNION query2) UNION query3
```

Relembrando a teoria de conjuntos de matemática:

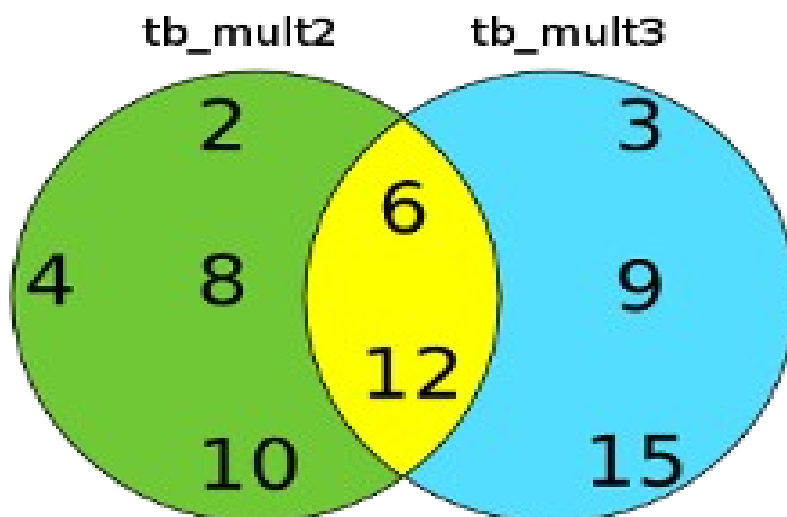


Figura 5: Representação das tabelas tb_mult2 e tb_mult3 em forma de conjuntos

Criação de tabela temporária para múltiplos de 2:

```
> CREATE TEMP TABLE tb_mult2(valor int2);
```

Criação de tabela temporária para múltiplos de 3:

```
> CREATE TEMP TABLE tb_mult3(valor int2);
```

Inserção de valores em tb_mult2:

```
> INSERT INTO tb_mult2 VALUES (2), (4), (6), (8), (10), (12);
```

Inserção de valores em tb_mult3:

```
> INSERT INTO tb_mult3 VALUES (3), (6), (9), (12), (15);
```

11.2 União (UNION)

Efetivamente adiciona o resultado de query2 para o resultado de query1 (embora não há garantia que essa é a ordem que as linhas são atualmente retornadas). Além disso, elimina linhas repetidas de seu resultado, da mesma forma que `DISTINCT`, a não ser que `UNION ALL` seja usado.

Unindo sem repetição `tb_mult2` e `tb_mult3`:

```
> SELECT valor FROM tb_mult2
      UNION
SELECT valor FROM tb_mult3;
```

```
valor
-----
15
12
4
3
6
9
10
8
2
```

Unindo com repetição `tb_mult2` e `tb_mult3`:

```
> SELECT valor FROM tb_mult2
      UNION ALL
SELECT valor FROM tb_mult3;
```

```
valor
-----
2
4
6
8
10
12
3
6
9
12
15
```

Repare que os valores 6 e 12 aparecem duas vezes.

11.3 Intersecção (INTERSECT)

Retorna todas as linhas que estão ambas no resultado de query1 e query2. Linhas repetidas são eliminadas a não ser que `INTERSECT ALL` seja usado.

Elementos comuns entre ambas as tabelas:

```
> SELECT valor FROM tb_mult2  
      INTERSECT  
SELECT valor FROM tb_mult3;
```

```
valor  
-----  
    12  
     6
```

11.4 Diferença (EXCEPT)

Retorna todas as linhas que estão no resultado de query1, mas não no resultado de query2, isso é às vezes chamado de diferença entre duas consultas. Novamente, linhas duplicadas são eliminadas a não ser que `EXCEPT ALL` seja usado.

Para quem vem do SGBD Oracle, essa mesma operação nele é chamada de MINUS.

Elementos que só existem na tabela tb_mult2:

```
> SELECT valor FROM tb_mult2  
      EXCEPT  
SELECT valor FROM tb_mult3;
```

```
valor  
-----  
      8  
      4  
     10  
      2
```


12 Casamento de Padrões e Expressões Regulares

- Sobre Casamento de Padrões e Expressões Regulares
- O Operador LIKE (~~)
- O Operador ILIKE (~~*)
- SIMILAR TO
- Função substring com Três Parâmetros
- Expressões Regulares POSIX

12.1 Sobre Casamento de Padrões e Expressões Regulares

Expressões regulares fornecem uma maneira flexível e concisa de buscar texto (strings) através de expressões apropriadas para isso.

Há três abordagens para casamento de padrões fornecidas pelo PostgreSQL; o operador tradicional `LIKE`, o mais recente operador `SIMILAR TO` (adicionado em SQL:1999), e as expressões regulares estilo POSIX.

Junto com operadores “esta *string* casa com este padrão?”, funções estão disponíveis para extrair ou substituir *substrings* que casem e para dividir uma *string* onde ela combina.

12.2 LIKE (~~) e ILIKE (~~*)

A expressão `LIKE` retorna `true` se a string casa com o padrão fornecido, que pode ter sua lógica invertida (negada) em adição a `NOT`.

Se o padrão não conter sinais de porcentagem (%) ou underscores (_), então o padrão representa apenas a string em si mesma quando `LIKE` atua como um operador de igual.

O `ILIKE` tem como única diferença de `LIKE` o fato de ser *case insensitive*, ou seja, não considera letras maiúsculas ou minúsculas.

Portanto, como exemplos, podem ser tomados os do `LIKE`, só que, logicamente, trocando as palavras `LIKE` por `ILIKE` e invertendo na string da condição maiúsculas por minúsculas e vice-versa.

Sintaxe:

```
string (LIKE|ILIKE) pattern [ESCAPE escape-character]
string NOT (LIKE|ILIKE) pattern [ESCAPE escape-character]
```

Caracteres curinga:

- %: Um, vários ou nenhum caractere na posição;
- _: Um único caractere na posição;
- \: Escape, faz com que se interprete literalmente os caracteres “%” e “_”.

Busca de string com padrão substituindo um caractere:

```
> SELECT 'Japão' LIKE 'Jap_o';
```

```
?column?
-----
t
```

Busca de string com padrão substituindo indefinidos caracteres no início:

```
> SELECT 'Melancia' LIKE '%cia';
```

```
?column?
-----
t
```

Busca de string com escape padrão:

```
> SELECT 'Sinal de porcentagem: %' LIKE '%\%';
```

```
?column?
-----
t
```

Busca de string com escape customizado:

```
> SELECT 'Sinal de porcentagem: %' LIKE '%|%' ESCAPE '|';
```

```
?column?  
-----  
t
```

Busca por string que tenha três caracteres e que "b" esteja no meio:

```
> SELECT 'abc' ~~ '_b_';
```

```
?column?  
-----  
t
```

Pessoas cujo nome inicia com a letra "A":

```
> SELECT nome FROM tb_pf WHERE nome ~~ 'A%' LIMIT 5;
```

```
nome  
-----  
Aldebarina  
Acabézia  
Aldebranda  
Alderovaldo
```

Pessoas cuja segunda letra de seu nome é "a":

```
> SELECT nome FROM tb_pf WHERE nome ~~ '_a%' LIMIT 5;
```

```
nome  
-----  
Carmezilda  
Valverinda  
Marta  
Malzevino
```

Inserção de um registro com caractere especial barra invertida (\):

```
> INSERT INTO tb_pf (cpf, nome, sobrenome)  
VALUES (21548736944, '\alguma coisa', 'Bla bla bla');
```

Consulta para String com Caractere Especial:

```
> SELECT nome FROM tb_pf WHERE nome ~~ '\\%';
```

```
      nome  
-----  
\\alguma coisa
```

12.3 SIMILAR TO

Parecido com LIKE, exceto que ele interpreta o padrão usando a norma de definição SQL de uma expressão regular.

Expressões regulares SQL são um cruzamento curioso entre a notação LIKE e a notação de expressão regular comum.

Como LIKE, SIMILAR TO apenas é bem-sucedido se seu padrão casar com toda a string; o que é diferente do comportamento de uma expressão regular comum que pode combinar qualquer parte da string.

Também como LIKE, SIMILAR TO usa os curingas "_" e "%" que denotam um único caractere e qualquer string, respectivamente (esses são comparáveis a . e .* em expressões regulares POSIX).

Sintaxe:

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

Em adição a essas facilidades herdadas de LIKE, SIMILAR TO suporta estes caracteres de padrões de combinação herdados de expressões regulares POSIX:

Expressão	Descrição
	operador OR.
*	item anterior zero ou mais vezes.
+	item anterior uma ou mais vezes.
?	item anterior zero ou uma vez.
{m}	item anterior exatamente m vezes.

Expressão	Descrição
{m,}	item anterior m ou mais vezes.
{m,n}	item anterior pelo menos m e não mais do que n vezes.
\	escape padrão
()	podem ser usados para agrupar itens em um único item lógico.
[...]	Expressão com colchetes especifica uma classe de caractere, como em expressões regulares POSIX.

Condição em que o segundo caractere do nome seja uma vogal:

```
> SELECT nome FROM tb_pf WHERE nome SIMILAR TO '_[aeiou]%' LIMIT 5;
```

```
nome
-----
Genovézio
Wolfrâmia
Tungstênia
Romirovaldo
Carmezilda
```

Condição em que o segundo caractere do nome não seja uma vogal:

```
> SELECT nome FROM tb_pf WHERE nome SIMILAR TO '_[^aeiou]%' LIMIT 5;
```

```
      nome
-----
Aldebarina
Urânia
Estrôncio
Acabézia
Aldebranda
```

Condição em que o sobrenome não seja nenhum dos declarados entre parênteses, delimitados por pipes:

```
> SELECT nome||' '||sobrenome AS "Nome Completo" FROM tb_pf
       WHERE sobrenome NOT SIMILAR TO '%(Gomes|Santos|Silva|Guerra)' LIMIT 5;
```

```
      Nome Completo
-----
Aldebarina Ferreira
Tungstênia Santana
Romirovaldo Ramires
Carmezilda Gonçalves
Valverinda Ramalho
```

12.4 Função substring com Três Parâmetros

A função `substring` com 3 (três) parâmetros, `substring(string from pattern for escape-character)`, fornece extração de uma *substring* que case com um padrão (*pattern*) de expressão regular SQL.

Como em `SIMILAR TO`, o padrão especificado deve casar com toda a string, ou então a função retornará nulo.

Para indicar a parte do padrão que deve ser retornado em caso de sucesso, o padrão deve conter duas ocorrências do caractere de escape (*character escape*) seguido por aspas ("). O texto que casar com a porção do padrão entre esses marcadores será retornado.

Começa com “P”, tem “SQL” e pode ter mais coisas depois?:

```
> SELECT substring('PostgreSQL: O SGBD!' from 'P%SQL%' for '#');
```

```
-----  
PostgreSQL: O SGBD!
```

Caractere de escape para trazer o texto que casa com o que há dentro do escape:

```
> SELECT substring('PostgreSQL: O SGBD!' from '"P%SQL#"' for '#');
```

```
substring  
-----  
PostgreSQL
```

Caractere de escape para trazer o texto que casa com o que há dentro do escape:

```
> SELECT substring('foobar' from '%"o_b#"' for '#');
```

```
substring  
-----  
oob
```

Caractere de escape para trazer o texto que casa com o que há dentro do escape:

```
> SELECT substring('foobar' from '"o_b#"' for '#');
```

```
substring  
-----
```

O retorno foi nulo, pois não casou com toda a linha de texto.

12.5 Expressões Regulares POSIX

Expressões regulares POSIX fornecem meios mais poderosos para casamento de padrões (*pattern matching*) do que os operadores `LIKE` e `SIMILAR TO`.

Muitas ferramentas Unix como `egrep`, `sed` ou `awk` usa uma linguagem de casamento de padrões similar ao que é descrito aqui.

Operador	Descrição	Operador	Descrição
<code>~</code>	Combina com a expressão regular, case sensitive	<code>!~</code>	Não combina com a expressão regular, case sensitive
<code>~*</code>	Combina com a expressão regular, case insensitive	<code>!~*</code>	Não combina com a expressão regular, case insensitive

Teste com expressão regular que pega um pedaço da string:

```
> SELECT 'Uma frase qualquer' ~ 'Uma';
```

```
?column?  
-----  
t
```

Diferente de `LIKE` e `SIMILAR TO`, houve uma combinação porque com expressões regulares POSIX basta casar um pedaço da string, assim não exigindo que seja a linha inteira. Um comportamento similar ao utilitário Unix `grep`.

Começa com "U"?:

```
> SELECT 'Uma frase qualquer' ~ '^U';
```

```
?column?  
-----  
t
```

Qualquer sequência antes, depois um “f” e depois qualquer sequência de caracteres e um “r”:

```
> SELECT 'Uma frase qualquer' ~ '.*f.*r';
```

```
?column?  
-----  
t
```

“Uma” casa com “uma”?

```
> SELECT 'Uma frase qualquer' ~ 'uma';
```

```
?column?  
-----  
f
```

Utilizando o operador insensitive:

```
> SELECT 'Uma frase qualquer' ~* 'uma';
```

```
?column?  
-----  
t
```

Não combina?:

```
> SELECT 'Uma frase qualquer' !~ 'uma';
```

```
?column?  
-----  
t
```

Não combina também insensitive?:

```
> SELECT 'Uma frase qualquer' !~* 'uma';
```

```
?column?  
-----  
f
```

13 Subconsultas

- Sobre Subconsultas
- Subconsultas Laterais
- Subconsultas em INSERT
- Subconsultas em UPDATE
- Subconsultas em DELETE

13.1 Sobre Subconsultas

Também conhecidas como *subqueries*, são `SELECT`s embutidos dentro de outro `SELECT` que têm por finalidade flexibilizar consultas. Esse recurso está disponível no PostgreSQL desde a versão 6.3.

Aviso:

Para consultas que envolvam mais de uma tabela é uma boa prática dar apelido a cada uma, mesmo que a mesma tabela seja usada como subconsulta de si mesma.

13.1.1 Subconsulta no WHERE

Exibir os CPFs dos colaboradores cujo salário seja maior do que média com 90% de acréscimo:

```
> SELECT c1.cpf FROM tb_colaborador c1
   WHERE c1.salario > (SELECT (avg(c2.salario) * 1.9) FROM tb_colaborador c2);
```

```
      cpf
-----
11111111111
23625814788
12345678901
```

Mesmo sendo a mesma tabela (`tb_colaborador`), a mesma foi apelidada como “c1” para a tabela da consulta principal e “c2” para a tabela da subconsulta e seus campos foram chamados conforme o contexto.

13.1.2 Subconsulta no SELECT

Exibir o CPF e a diferença do salário relativa à média:

```
> SELECT
  c1.cpf "CPF",
  c1.salario -
  (SELECT avg(c2.salario) FROM tb_colaborador c2)::numeric(7, 2)
  "Diferença da Média"
FROM tb_colaborador c1
LIMIT 5;
```

CPF	Diferença da Média
11111111111	17445.00
23625814788	7445.00
33344455511	1945.00
12345678901	2445.00
10236547895	945.00

13.1.3 Subconsulta no FROM

O uso de uma subconsulta no FROM faz com que ela funcione como se fosse uma tabela:

```
> SELECT cpf FROM (SELECT * FROM tb_colaborador WHERE cargo = 1) AS tb_chefe;
```

cpf
11111111111
23625814788

13.1.4 EXISTS

Se a subconsulta retornar pelo menos uma linha seu resultado é “true” (verdadeiro).

Se existir algum registro em tb_pf cujo nome seja “Chiquinho”, exibir id, cpf e salario de tb_colaborador:

```
> SELECT c.id, c.cpf, c.salario FROM tb_colaborador c
  WHERE EXISTS
  (SELECT true FROM tb_pf p WHERE nome = 'Chiquinho' LIMIT 1)
LIMIT 5;
```

id	cpf	salario
1	11111111111	20000.00
2	23625814788	10000.00
3	33344455511	4500.00
4	12345678901	5000.00
5	10236547895	3500.00

Exibir nome de tb_pessoa se existir algum registro em tb_colaborador que salário seja maior que 5000 e setor seja 5:

```
> SELECT p.nome FROM tb_pessoa p
   WHERE EXISTS
   (SELECT c.cpf FROM tb_colaborador c WHERE c.salario > 5000 AND c.setor = 5 LIMIT 1);
```

```
nome
-----
```

A condição da subconsulta não foi satisfeita, por isso também não retornou nada para a consulta principal.

Obs.:

É aconselhável que a subconsulta utilizada em `WHERE EXISTS` utilize `LIMIT 1` para evitar *overhead* e agilizar o resultado, pois só é preciso uma linha para satisfazer a condição.

13.1.5 IN e NOT IN

A subconsulta deve retornar apenas uma coluna, cujos valores serão avaliados para estarem (`IN`) ou não (`NOT IN`) dentro dos valores esperados.

Selecionar id de tb_colaborador onde o cpf está entre os valores retornados da subconsulta:

```
> SELECT c.id FROM tb_colaborador c
   WHERE c.cpf IN (SELECT p.cpf FROM tb_pf p
   WHERE p.dt_Nascto BETWEEN '1982-01-01' AND '1983-12-31')
LIMIT 5;
```

```
id
----
5
10
14
22
24
```

Selecionar id de tb_colaborador onde o cpf não está entre os valores retornados da subconsulta e o setor deve ser igual a 3:

```
> SELECT c.id FROM tb_colaborador c WHERE c.cpf
    NOT IN (SELECT p.cpf FROM tb_pf p
    WHERE p.dt_Nascto BETWEEN '1982-01-01' AND '1983-12-31') AND setor = 3;
```

```
id
---
9
11
12
13
15
```

13.1.6 ANY ou SOME

Tanto faz usar qualquer uma das palavras-chave, pois o efeito é o mesmo.

A subconsulta deve retornar apenas uma coluna, cujos valores serão avaliados para a condição da consulta principal.

Se a subconsulta não trouxer nenhum resultado a consulta inteira retornará nulo.

Quantos colaboradores nasceram na década de 1980?:

```
> SELECT count(*) FROM tb_colaborador c
    WHERE c.cpf = ANY
    (SELECT p.cpf FROM tb_pf p
    WHERE p.dt_nascto BETWEEN '1980-01-01' AND '1989-12-31');
```

```
count
-----
18
```

13.1.7 ALL

A subconsulta deve retornar exatamente uma coluna. A expressão à esquerda é avaliada e comparada a cada linha da subconsulta. O resultado de ALL é verdadeiro se todas as linhas forem verdadeiras (incluindo o caso onde a subconsulta não retorna nenhuma linha). O resultado é falso se qualquer resultado falso for encontrado. O resultado é nulo se a comparação não retornar falso para qualquer linha e retornar nulo para no mínimo uma.

Exibir cpf e salário para o id 1 se o salário for maior ou igual do que qualquer linha retornada pela subconsulta:

```
> SELECT c1.cpf, c1.salario FROM tb_colaborador c1
   WHERE c1.id = 1 AND c1.salario >= ALL
   (SELECT c2.salario FROM tb_colaborador c2 WHERE c2.setor = 1);
```

```
      cpf      | salario
-----+-----
11111111111 | 20000.00
```

13.1.8 Comparação de Linha Única

A subconsulta deve retornar exatamente quantas colunas são necessárias para a expressão à esquerda e somente uma linha. Essa linha retornada é o valor utilizado para avaliar a expressão.

Exibir cpf e salário onde o id é 1 e se o salário for maior do que a média de toda empresa:

```
> SELECT c1.cpf, c1.salario FROM tb_colaborador c1
   WHERE c1.id = 1
 AND c1.salario > (SELECT avg(c2.salario) FROM tb_colaborador c2);
```

```
      cpf      | salario
-----+-----
11111111111 | 20000.00
```


13.2 Subconsultas Laterais

Um recurso incluído a partir da versão 9.3, subconsultas de `FROM` podem ser precedidas pela palavra-chave `LATERAL`. Isso permite-lhes fazer referência a colunas fornecidas antes de `FROM`.

Sem a palavra-chave, cada subconsulta é avaliada de forma independente e por isso não pode se fazer referência cruzada a qualquer outro item de `FROM`.

Aviso:

Não é compatível com junções (*joins*) `FULL` e `RIGHT`.

Criação da tabela de teste:

```
> SELECT generate_series(1, 10) AS numero INTO tb_foo;
```

Tentativa de consulta com subconsulta lateral:

```
> SELECT tb_foo.numero, bar.cubo FROM tb_foo,
       (SELECT (tb_foo.numero ^ 3) AS cubo) AS bar;
```

```
ERROR:  invalid reference to FROM-clause entry for table "tb_foo"
LINE 1: ...ECT tb_foo.numero, bar.cubo FROM tb_foo, (SELECT (tb_foo.num...
```

```
HINT:  There is an entry for table "tb_foo", but it cannot be referenced from this part of the query.
```

Tentativa com consulta lateral com a cláusula `LATERAL` declarada:

```
> SELECT tb_foo.numero, bar.cubo FROM tb_foo,
       LATERAL (SELECT (tb_foo.numero ^ 3) AS cubo) AS bar;
```

```
 numero | cubo
-----+-----
 1 |      1
 2 |      8
 3 |     27
 4 |     64
 5 |    125
 6 |    216
 7 |    343
 8 |    512
 9 |    729
10 |   1000
```

Com a declaração da cláusula `LATERAL` foi possível referenciar `bar.cubo`.

13.3 Subconsultas em INSERT

Método similar ao `CREATE TABLE AS`, é uma forma de passarmos dados de uma tabela para outra.

Para exemplo, vamos primeiro criar uma tabela temporária, cuja finalidade será conter apenas pessoas que o nome comece com a letra “A”.

Criação de Tabela de Teste:

```
> CREATE TEMP TABLE tb_pf_a(  
    cpf INT8,  
    rg VARCHAR(10),  
    sobrenome VARCHAR(70),  
    genero tp_genero,  
    CONSTRAINT pk_pf PRIMARY KEY (cpf)  
) INHERITS (tb_pessoa);
```

Inserção com SELECT:

```
> INSERT INTO tb_pf_a SELECT * FROM tb_pf WHERE nome ~ '^A';
```

Criação de uma nova tabela de teste:

```
> CREATE TEMP TABLE tb_foo(  
    id serial2 PRIMARY KEY,  
    campo1 text,  
    campo2 text,  
    campo3 smallint);
```

Populando a tabela com um simples INSERT múltiplo:

```
> INSERT INTO tb_foo (campo1, campo2, campo3) VALUES  
    ('A', 'A', 1),  
    ('B', 'B', 1),  
    ('C', 'C', 1);
```

Verificando os dados:

```
> TABLE tb_foo;
```

id	campo1	campo2	campo3
1	A	A	1
2	B	B	1
3	C	C	1

Inserindo uma nova linha copiando a linha onde id é igual a 1 e atribuindo o valor 2 à coluna campo3:

```
> INSERT INTO tb_foo (campo1, campo2, campo3)
  SELECT f.campo1, f.campo2, 2 FROM tb_foo f
  WHERE f.id = 1;
```

O INSERT anterior é muito útil quando é preciso uma nova registro em que muitos campos já tem seus valores que precisamos em outro registro.

Verificando a tabela:

```
> TABLE tb_foo;
```

<i>id</i>	<i>campo1</i>	<i>campo2</i>	<i>campo3</i>
1	A	A	1
2	B	B	1
3	C	C	1
4	A	A	2

13.4 Subconsultas em UPDATE

13.4.1 Atualizar Campo por Resultado de Consulta

O intuito é fazer o `UPDATE` em um campo, cujo valor será o retorno de uma consulta que retorna uma linha.

Todos os trabalhadores que ganham menos ou igual a R\$ 1.050,00 terão seus salários reajustados para o valor igual à média daqueles que ganham menos que R\$ 2.000,00:

```
> UPDATE tb_colaborador
  SET salario = (SELECT avg(c.salario)
                FROM tb_colaborador c
                WHERE c.salario < 2000)
  WHERE salario <= 1050;
```

13.4.2 Atualizar Campo Copiando o Restante da Linha

Quando já existe uma linha, cujas colunas têm quase todos os valores desejáveis e queremos atualizar uma ou mais linha através dessa, podemos selecionar seu conteúdo alterando apenas a parte que é diferente.

Só para lembrar a tabela `tb_foo`:

```
> TABLE tb_foo;
```

<i>id</i>	<i>campo1</i>	<i>campo2</i>	<i>campo3</i>
1	A	A	1
2	B	B	1
3	C	C	1
4	A	A	2

A linha de id 3 será atualizada conforme a de id 1, exceto o campo 3 e o campo id, é claro:

```
> UPDATE tb_foo
SET
    campo1 = f.campo1,
    campo2 = f.campo2,
    campo3 = f.campo3
FROM
(
    SELECT
        campo1,
        campo2,
        50 AS campo3
    FROM tb_foo
    WHERE id = 1
) AS f
WHERE id = 3;
```

Verificando a tabela:

```
> TABLE tb_foo;

id | campo1 | campo2 | campo3
---+-----+-----+-----
1  | A      | A      | 1
2  | B      | B      | 1
4  | A      | A      | 2
3  | A      | A      | 50
```

13.4.3 Atualizar uma Tabela Através de Outra Tabela

O propósito é atualizar uma tabela através de outra.

Verificando se as tabelas de teste existem:

```
> DROP TABLE IF EXISTS tb_cor, tb_cor_2;
```

Criação das tabelas de teste:

```
> CREATE TEMP TABLE tb_cor(
    id serial primary key,
    nome text);
```

```
> CREATE TEMP TABLE tb_cor_2(
    id serial primary key,
    nome text);
```

Inserindo valores diferentes entre as tabelas:

```
> INSERT INTO tb_cor (nome) VALUES
    ('Verde'),
    ('Amarelo'),
    ('Azul');
```

```
> INSERT INTO tb_cor_2 (nome) VALUES
    ('Verde'),
    ('Preto'),
    ('Branco');
```

Atualizando a primeira conforme os valores da segunda:

```
> UPDATE tb_cor
SET nome = c2.nome
FROM tb_cor_2 c2
WHERE tb_cor.id = c2.id;
```

Verificando:

```
> TABLE tb_cor;

id | nome
---+-----
1  | Verde
2  | Preto
3  | Branco
```

13.5 Subconsultas em DELETE

Imagine que um grupo de amigos que gostam de futebol de mesa (também conhecido como jogo de botão) resolvem fazer um campeonato e montam até um sistema.

O campeonato conta apenas com times europeus.

Criação da tabela de país de origem dos times:

```
> CREATE TEMP TABLE tb_pais_origem(  
  id serial PRIMARY KEY,  
  nome TEXT);
```

Criação da tabela de times:

```
> CREATE TEMP TABLE tb_time(id serial PRIMARY KEY,  
  nome text,  
  pais_origem int,  
  FOREIGN KEY (pais_origem) REFERENCES tb_pais_origem(id));
```

Tabela de países sendo populada:

```
> INSERT into tb_pais_origem (nome) VALUES  
  ('Alemanha'),  
  ('Inglaterra'),  
  ('Itália'),  
  ('Espanha');
```

Tabela de times sendo populada:

```
> INSERT into tb_time (nome, pais_origem) VALUES  
  ('Bayern de Munique', 1),  
  ('Hamburgo', 1),  
  ('Borussia Dortmund', 1),  
  ('Juventus', 3),  
  ('Milan', 3),  
  ('Real Madri', 4),  
  ('Manchester United', 2),  
  ('New Castle', 2),  
  ('Barcelona', 4);
```

Tabelas preenchidas, agora é hora de verificar o resultado:

```
> SELECT t.nome AS "Time", p.nome AS "País" FROM tb_time AS t
      INNER JOIN tb_pais_origem AS p
      ON t.pais_origem = p.id;
```

Time	País
Bayern de Munique	Alemanha
Hamburgo	Alemanha
Borussia Dortmund	Alemanha
Juventus	Itália
Milan	Itália
Real Madri	Espanha
Manchester United	Inglaterra
New Castle	Inglaterra
Barcelona	Espanha

Na consulta foi usado um recurso que será visto posteriormente, que possibilita pegar dados de diferentes tabelas em uma única consulta. Nesse caso foi usada uma junção interna, também conhecida como **INNER JOIN**.

Houve um imprevisto!

Justamente os representantes dos times espanhóis, em cima da hora, avisaram que não vão mais poder participar do torneio.

Então agora teremos que excluir esses times :(

Apagando os times espanhóis da tabela e retornando o resultado:

```
> DELETE FROM tb_time WHERE pais_origem = (
      SELECT p.id pais
      FROM tb_pais_origem AS p
      WHERE p.nome = 'Espanha')
RETURNING nome;
```

nome
Real Madri
Barcelona

14 Junções (Joins)

- Sobre Junções
- NATURAL JOIN (Junção Natural)
- INNER JOIN (Junção Interna)
- OUTER JOIN (Junção Externa) - Definição
- LEFT OUTER JOIN (Junção Externa à Esquerda)
- RIGHT OUTER JOIN (Junção Externa à Direita)
- FULL OUTER JOIN (Junção Externa Total)
- SELF JOIN (Auto-Junção)
- CROSS JOIN (Junção Cruzada)
- UPDATE com JOIN
- DELETE com JOIN

14.1 Sobre Junções

Junções ou *Joins*, em inglês, como o próprio nome diz é uma junção. Na verdade, não somente uma, mas podem ser várias.

Junção é envolver duas ou mais tabelas em uma mesma consulta.

Ao usarmos mais de uma tabela em uma consulta pode haver nomes de colunas idênticos nas tabelas envolvidas.

Tal inconveniente pode ser resolvido colocando o nome da tabela e um ponto antes do nome da coluna. No entanto, muitas vezes isso pode se tornar um tanto cansativo devido ao fato de se digitar coisas a mais, então usamos os chamados *alias* de tabelas (apelidos de tabelas).

Os apelidos de tabelas são feitos como os apelidos de colunas podendo omitir ou não a palavra-chave `AS`.

```
nome_da_tabela [AS] apelido
```

Utilizando a base de dados `db_empresa`, adicionalmente criaremos uma tabela para veículos com seus respectivos dados.

Criação da tabela de veículos da empresa:

```
> CREATE TABLE tb_veiculo(  
    id serial primary key,  
    marca text,  
    modelo text,  
    colaborador_fk int8 -- motorista  
    REFERENCES tb_colaborador (id) -- Chave estrangeira  
);
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_veiculo (marca, modelo, colaborador_fk) VALUES  
    ('Fiat', '147', 1),  
    ('Volkswagen', 'Variant', NULL),  
    ('Ford', 'Corcel I', 2),  
    ('Chevrolet', 'Chevette', 1),  
    ('Simca', 'Chambord', NULL);
```

Verificando a tabela:

```
> TABLE tb_veiculo;
```

id	marca	modelo	colaborador_fk
1	Fiat	147	1
2	Volkswagen	Variant	
3	Ford	Corcel I	2
4	Chevrolet	Chevette	1
5	Simca	Chambord	

Há dois veículos que não têm colaborador associado.

14.2 NATURAL JOIN (Junção Natural)

Faz uma junção implícita tomando como base as colunas de mesmo nome nas tabelas envolvidas.

É recomendável que ao invés de usar `NATURAL JOIN` se use `INNER JOIN`, pois essa última explicita qual é o critério de vínculo entre tabelas deixando a leitura mais amigável.

Junção natural entre colaboradores e pessoa física cujo salário seja maior ou igual que R\$ 5.000,00:

```
> SELECT p.nome, p.sobrenome, c.salario
   FROM tb_colaborador c
   NATURAL JOIN tb_pf p
  WHERE c.salario >= 5000;
```

nome	sobrenome	salario
Chiquinho	da Silva	20000.00
Aldebarina	Ferreira	10000.00
Tungstênia	Santana	5000.00

Com `EXPLAIN ANALYZE` verificar o plano de execução da consulta:

```
> EXPLAIN ANALYZE
SELECT p.nome, p.sobrenome, c.salario
   FROM tb_colaborador c
   NATURAL JOIN tb_pf p
  WHERE c.salario >= 5000;
```

```

                                QUERY PLAN
-----
Hash Join  (cost=12.47..40.50 rows=360 width=250) (actual time=0.054..0.067 rows=3 loops=1)
  Hash Cond: (c.cpf = p.cpf)
    -> Seq Scan on tb_colaborador c  (cost=0.00..23.50 rows=360 width=22) (actual time=0.014..0.025 rows=3 loops=1)
        Filter: (salario >= '5000'::numeric)
        Rows Removed by Filter: 37
    -> Hash  (cost=11.10..11.10 rows=110 width=244) (actual time=0.031..0.031 rows=41 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 11kB
        -> Seq Scan on tb_pf p  (cost=0.00..11.10 rows=110 width=244) (actual time=0.005..0.016 rows=41 loops=1)
Planning time: 0.202 ms
Execution time: 0.100 ms
```

Podemos verificar que internamente o PostgreSQL faz a junção utilizando como critério o campo `cpf` de ambas as tabelas (*Hash Cond: (c.cpf = p.cpf)*).

14.3 INNER JOIN (Junção Interna)

INNER JOIN ou simplesmente INNER, é o tipo de junção padrão, o qual retorna as informações apenas de acordo com as linhas que obedecem as definições de relacionamento. Existe uma ligação lógica para se fazer a junção, a qual é declarada explicitamente.

Para o critério de junção pode-se usar a cláusula ON que especifica qual a condição usada ou USING que apenas diz qual campo com o mesmo nome em ambas as tabelas deve ser utilizado.

INNER JOIN utilizando ON:	INNER JOIN com USING:
<pre>> SELECT p.nome, p.sobrenome, c.salario FROM tb_colaborador c INNER JOIN tb_pf p ON c.cpf = p.cpf WHERE c.salario >= 5000;</pre>	<pre>> SELECT p.nome, p.sobrenome, c.salario FROM tb_colaborador c INNER JOIN tb_pf p USING (cpf) WHERE c.salario >= 5000;</pre>
<pre> nome sobrenome salario -----+-----+----- Aldebarina Ferreira 10000.00 Tungstênia Santana 5000.00 Chiquinho da Silva 20000.00 </pre>	

Junções equivalentes sem e com a cláusula INNER JOIN:	
<pre>> SELECT c.id AS "Matrícula", p.nome ' ' p.sobrenome "Nome Completo", c.salario "Salário" FROM tb_colaborador c, tb_pf p WHERE c.salario > 4500 AND c.cpf = p.cpf;</pre>	<pre>> SELECT c.id AS "Matrícula", p.nome ' ' p.sobrenome "Nome Completo", c.salario "Salário" FROM tb_colaborador c INNER JOIN tb_pf p USING (cpf) WHERE c.salario > 4500;</pre>
<pre> Matrícula Nome Completo Salário -----+-----+----- 2 Aldebarina Ferreira 10000.00 4 Tungstênia Santana 5000.00 1 Chiquinho da Silva 20000.00 </pre>	

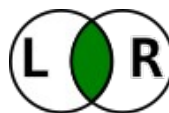
Que colaborador dirige qual carro?:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
  INNER JOIN tb_veiculo v
    ON (c.id = v.colaborador_fk);
```

```

Matrícula | Marca - Modelo
-----+-----
1 | Fiat - 147
2 | Ford - Corcel I
1 | Chevrolet - Chevette

```



Uma intersecção, a qual mostra somente casos em que há correspondência em ambos os lados.

14.4 OUTER JOIN (Junção Externa) - Definição

Assim como na INNER JOIN, existe uma ligação lógica, mas não retorna apenas as informações que satisfaçam a regra da junção. OUTER JOINS podem ser dos tipos:

- LEFT OUTER JOIN: retorna todos os registros da tabela à esquerda;
- RIGHT OUTER JOIN: retorna todos os registros da tabela à direita;
- FULL OUTER JOIN: retorna todos os registros de ambos os lados.

É de uso opcional a palavra OUTER.

Para os exercícios serão inseridos dados na tabela tb_pf, que não tenham correspondência na tabela tb_colaborador.

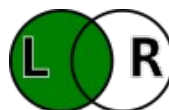
14.5 LEFT OUTER JOIN (Junção Externa à Esquerda)

LEFT OUTER JOIN ou simplesmente LEFT JOIN traz também dados não relacionados, os quais estão na tabela à esquerda da cláusula JOIN.

Se não existir dados relacionados entre ambas as tabelas (à esquerda e à direita da cláusula JOIN), a coluna relacionada entre ambas conterá como valor apenas nulo.

Tabela tb_colaborador à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
LEFT JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
LIMIT 10;
```

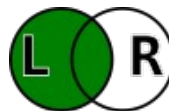


Matrícula	Marca - Modelo
1	Chevrolet - Chevette
1	Fiat - 147
2	Ford - Corcel I
3	
4	
5	
6	
7	
8	
9	

Observa-se que vários colaboradores não têm veículos associados para si.

Tabela tb_veiculo à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_veiculo v
LEFT JOIN tb_colaborador c
  ON (c.id = v.colaborador_fk);
```



Matrícula	Marca - Modelo
1	Fiat - 147
	Volkswagen - Variant
2	Ford - Corcel I
1	Chevrolet - Chevette
	Simca - Chambord

Nota-se que há dois veículos que ainda não têm colaboradores associados.

Tabela tb_colaborador à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
LEFT JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
WHERE v.colaborador_fk IS NULL
LIMIT 10;
```



Matrícula	Marca - Modelo
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Somente colaboradorres que não têm veículos associados.

Tabela tb_colaborador à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_veiculo v
LEFT JOIN tb_colaborador c
  ON (c.id = v.colaborador_fk)
WHERE c.id IS NULL;
```



Matrícula	Marca - Modelo
	Volkswagen - Variant
	Simca - Chambord

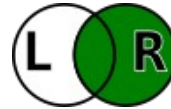
Somente veículos que não têm colaboradores associados.

14.6 RIGHT OUTER JOIN (Junção Externa à Direita)

RIGHT OUTER JOIN ou também simplesmente RIGHT JOIN, tem seu funcionamento igual ao de LEFT JOIN, mas com sua lógica orientada para a direita em vez da esquerda.

Tabela tb_colaborador à direita:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_veiculo v
RIGHT JOIN tb_colaborador c
  ON (c.id = v.colaborador_fk)
LIMIT 10;
```



Matrícula	Marca - Modelo
1	Chevrolet - Chevette
1	Fiat - 147
2	Ford - Corcel I
3	
4	
5	
6	
7	
8	
9	

Nem todos colaboradores têm veículos associados.

Tabela tb_veiculo à direita:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
RIGHT JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk);
```



Matrícula	Marca - Modelo
1	Fiat - 147
	Volkswagen - Variant
2	Ford - Corcel I
1	Chevrolet - Chevette
	Simca - Chambord

Tanto colaboradores como veículos podem ou não serem associados entre si.

Tabela tb_colaborador à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
RIGHT JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
WHERE v.colaborador_fk IS NULL;
```



Matrícula	Marca - Modelo
	Volkswagen - Variant
	Simca - Chambord

Somente veículos sem associação a um colaborador.

Tabela tb_colaborador à esquerda:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_veiculo v
RIGHT JOIN tb_colaborador c
  ON (c.id = v.colaborador_fk)
WHERE v.colaborador_fk IS NULL
LIMIT 10;
```



Matrícula	Marca - Modelo
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Somente colaboradores sem associação a um veículo.

14.7 FULL OUTER JOIN (Junção Externa Total)

FULL OUTER JOIN ou simplesmente FULL JOIN, tendo correspondência ou não entre esquerda e direita retorna resultados de ambos os lados.

Usando FULL JOIN para relacionar colaboradores com os veículos que dirigem na empresa:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
FULL JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
LIMIT 10;
```

Matrícula	Marca - Modelo
1	Fiat - 147
	Volkswagen - Variant
2	Ford - Corcel I
1	Chevrolet - Chevette
	Simca - Chambord
20	
25	
26	
27	
11	



Notamos aqui que nem todos os colaboradores dirigem um veículo e nem todo veículo possui um motorista cadastrado.

Repare que em ambos os lados há valores nulos.

Somente veículos que ainda não têm um colaborador associado:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
FULL JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
WHERE c.id IS NULL;
```

Matrícula	Marca - Modelo
	Volkswagen - Variant
	Simca - Chambord



Colaboradores e veículos sem associação entre si:

```
> SELECT
  c.id "Matrícula",
  v.marca||' - '||v.modelo "Marca - Modelo"
FROM tb_colaborador c
FULL JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
WHERE v.colaborador_fk IS NULL
LIMIT 10;
```

Matrícula	Marca - Modelo
	Volkswagen - Variant
	Simca - Chambord
25	
26	
17	
33	
10	
15	
21	
32	



Com ajuda de um INNER JOIN com a tabela tb_pf podemos agora ver os nomes dos funcionários:

```
> SELECT
  p.nome||' '||p.sobrenome "Colaborador",
  v.marca||' - '||v.modelo "Marca - Modelo"

FROM tb_colaborador c
INNER JOIN tb_pf p
  ON (p.cpf = c.cpf)
FULL JOIN tb_veiculo v
  ON (c.id = v.colaborador_fk)
LIMIT 10;
```

Colaborador	Marca - Modelo
Chiquinho da Silva	Chevrolet - Chevette
Chiquinho da Silva	Fiat - 147
Aldebarina Ferreira	Ford - Corcel I
Wolfrâmia Santos	
Tungstênia Santana	
Urânia Gomes	
Estrôncio dos Santos	
Romirovaldo Ramires	
Carmezilda Gonçalves	
Valverinda Ramalho	

14.8 SELF JOIN (Auto-Junção)

Nem sempre JOINS são usadas para unir dados de duas ou mais tabelas. Há um caso especial que se faz uma auto-junção. Ou seja, é feita uma junção de uma tabela consigo mesma. Para evitar conflitos, usa-se *alias*es.

Descobrir o cpf do chefe direto de cada funcionário:

```
> SELECT
    colab_1.id,
    colab_1.cpf,
    p.nome||' '||p.sobrenome nome_completo,
    s.nome setor,
    colab_2.nome cargo,
    colab_1.salario salario,
    colab_1.dt_admis,
    colab_1.ativo,
    c.cpf chefe
FROM tb_colaborador colab_1
INNER JOIN tb_pf p
    USING (cpf)
INNER JOIN tb_setor s
    ON (colab_1.setor = s.id)
INNER JOIN tb_cargo colab_2
    ON (colab_1.cargo = colab_2.id)
INNER JOIN tb_colaborador c
    ON (colab_1.chefe_direto = c.id)
LIMIT 5;
```

id	cpf	nome_completo	setor	cargo	salario	dt_admis	ativo	chefe
2	23625814788	Aldebarina Ferreira	Presidência	Presidente	10000.00	2015-06-22	t	111111111111
3	33344455511	Wolfrâmia Santos	RH	Gerente	4500.00	2015-06-22	t	111111111111
4	12345678901	Tungstênia Santana	Presidência	Secretária	5000.00	2015-06-22	t	111111111111
5	10236547895	Urânia Gomes	RH	Assistente	3500.00	2015-06-22	t	23625814788
6	14725836944	Estrôncio dos Santos	RH	Assistente	3500.00	2015-06-22	t	23625814788

Mas ainda não está lá muito amigável... Quem é o(a) dono(a) do CPF?

Descobrir o nome do chefe direto de cada funcionário:

```
> SELECT
  colab_1.id,
  colab_1.cpf,
  p1.nome||' '||p1.sobrenome nome_completo,
  s.nome setor,
  c.nome cargo,
  colab_1.salario salario,
  colab_1.dt_admis,
  colab_1.ativo,
  p2.nome||' '||p2.sobrenome "Chefe Direto"
FROM tb_colaborador colab_1
INNER JOIN tb_pf p1
  USING (cpf)
INNER JOIN tb_setor s
  ON (colab_1.setor = s.id)
INNER JOIN tb_cargo c
  ON (colab_1.cargo = c.id)
INNER JOIN tb_colaborador colab_2
  ON (colab_1.chefe_direto = colab_2.id)
INNER JOIN tb_pf p2
  ON (p2.cpf = colab_2.cpf)
LIMIT 5;
```

id	cpf	nome_completo	setor	cargo	salario	dt_admis	ativo	Chefe Direto
2	23625814788	Aldebarina Ferreira	Presidência	Presidente	10000.00	2015-06-22	t	Chiquinho da Silva
3	33344455511	Wolfrâmia Santos	RH	Gerente	4500.00	2015-06-22	t	Chiquinho da Silva
4	12345678901	Tungstênia Santana	Presidência	Secretária	5000.00	2015-06-22	t	Chiquinho da Silva
5	10236547895	Urânia Gomes	RH	Assistente	3500.00	2015-06-22	t	Aldebarina Ferreira
6	14725836944	Estrôncio dos Santos	RH	Assistente	3500.00	2015-06-22	t	Aldebarina Ferreira

Para conseguir exibir o nome do chefe, nesse caso, foi necessário criar um novo apelido pra tabela tb_pf (p2), pois é nessa tabela que estão os nomes das pessoas.

14.9 CROSS JOIN (Junção Cruzada)

Retorna um conjunto de informações o qual é resultante de todas as combinações possíveis entre os registros das tabelas envolvidas.

Como exemplo, criaremos 2 tabelas; uma de carros e outra de cores e depois de populá-las veremos as possíveis combinações entre carros e cores.

Criação das tabelas para teste, de carros e de cores:

```
> CREATE TEMP TABLE tb_carro(  
  id serial PRIMARY KEY,  
  nome VARCHAR(20));
```

```
> CREATE TEMP TABLE tb_cor(  
  id serial PRIMARY KEY,  
  nome VARCHAR(20));
```

Populando as tabelas:

```
> INSERT INTO tb_carro (nome) VALUES  
  ('Fiat 147'),  
  ('VW Fusca'),  
  ('Ford Corcel'),  
  ('GM Opala');
```

```
> INSERT INTO tb_cor (nome) VALUES  
  ('Verde'),  
  ('Azul'),  
  ('Amarelo'),  
  ('Branco'),  
  ('Preto'),  
  ('Vermelho'),  
  ('Laranja'),  
  ('Cinza');
```

Junção cruzada:

```
> SELECT c1.nome carro, c2.nome cor  
  FROM tb_carro c1  
  CROSS JOIN  
  tb_cor c2  
  LIMIT 5;
```

```
> SELECT c1.nome carro, c2.nome cor  
  FROM tb_carro c1, tb_cor c2  
  LIMIT 5;
```

```
carro | cor  
-----+-----  
Fiat 147 | Verde  
Fiat 147 | Azul  
Fiat 147 | Amarelo  
Fiat 147 | Branco  
Fiat 147 | Preto
```

Ambas as consultas são junções cruzadas mesmo que uma delas não tenha sido explicitada como com CROSS JOIN.

14.10 UPDATE com JOIN

O comando UPDATE também pode ser usado com junção.
É um recurso útil quando se deseja referir a algo sem ser pelo seu código.

O veículo com o id determinado terá como condutor na empresa a pessoa determinada:

```
> UPDATE tb_veiculo v
  SET colaborador_fk = c.id
  FROM tb_colaborador c
  INNER JOIN tb_pf p
    ON (c.cpf = p.cpf)
 WHERE v.id = 5 -- Linha para atualizar
    AND p.nome = 'Estriga'
    AND p.sobrenome = 'Souto';
```

Verificando o nome do condutor:

```
> SELECT p.nome||' '||p.sobrenome "Nome Completo"
  FROM tb_pf p
  INNER JOIN tb_colaborador c
    ON (p.cpf = c.cpf)
  INNER JOIN tb_veiculo v
    ON (c.id = v.colaborador_fk)
 WHERE v.id = 5;
```

```
Nome Completo
-----
Estriga Souto
```

14.11 DELETE com JOIN

É possível também fazer junção com DELETE, mas a sintaxe não tem a palavra JOIN.

Apagando um registro de veículo pelo nome de seu condutor:

```
> DELETE FROM tb_veiculo v
    USING tb_colaborador c, tb_pf p
    WHERE c.id = v.colaborador_fk -- Critério de junção
        AND c.cpf = p.cpf -- Critério de junção
        AND p.nome = 'Estriga'
        AND p.sobrenome = 'Souto'
    RETURNING v.marca || ' ' || v.modelo AS veiculo;

veiculo
-----
Simca Chambord
```

15 Funções Built-ins

- Sobre Funções
- Funções Matemáticas
- Funções de Data e Hora
- Funções de Strings
- Funções de Sistema

15.1 Sobre Funções

O PostgreSQL fornece um grande número de funções de diversas finalidades, as quais são executadas dentro de um comando SQL.

Funções podem ou não ter parâmetros, e mesmo as que não têm parâmetros, dependendo de qual for pode não ter os tradicionais parênteses.

Funções sejam elas *built-ins* ou criadas pelo usuário todas são executadas por um `SELECT`, conforme a sintaxe:

```
SELECT funcao;
```

15.2 Funções Matemáticas

São funções cujo objetivo é facilitar determinados tipos de cálculos. Algumas funções têm seu equivalente como operador.

Potenciação; 2 elevado a 3 (função):		Potenciação; 2 elevado a 3 (operador):
<pre>> SELECT pow(2, 3); pow ----- 8</pre>		<pre>> SELECT 2 ^ 3; ?column? ----- 8</pre>
Raiz quadrada de 49 (função):		Raiz quadrada de 49 (operador):
<pre>> SELECT sqrt(49); sqrt ----- 7</pre>		<pre>> SELECT / 49; ?column? ----- 7</pre>
Arredondamento:	Arredondamento para cima:	Arredondamento para baixo:
<pre>> SELECT round(4.5); round ----- 5</pre>	<pre>> SELECT ceil(7.0000001); ceil ----- 8</pre>	<pre>> SELECT floor(7.99999999); floor ----- 7</pre>
Multiplicando pi por 3:		Log de 10000:
<pre>> SELECT pi() * 3; ?column? ----- 9.42477796076938</pre>		<pre>> SELECT log(10000); log ----- 4</pre>
Resto da divisão de 13 por 5:		
<pre>> SELECT mod(13, 5); mod ----- 3</pre>	<pre>> SELECT 13 % 5; ?column? ----- 3</pre>	

15.3 Funções de Data e Hora

Data atual: <pre>> SELECT current_date;</pre> <p><i>date</i></p> <p>-----</p> <p>2015-08-17</p>	Data atual + 10 dias: <pre>> SELECT current_date + 10; /* Daqui a 10 dias*/</pre> <p><i>?column?</i></p> <p>-----</p> <p>2015-08-27</p>
------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------

Hora atual: <pre>> SELECT current_time;</pre> <p><i>timetz</i></p> <p>-----</p> <p>14:22:39.116069-03</p>	Data e hora atuais: <pre>> SELECT current_timestamp;</pre> <p><i>now</i></p> <p>-----</p> <p>2015-08-17 14:23:40.295841-03</p>	<pre>> SELECT now();</pre> <p><i>now</i></p> <p>-----</p> <p>2015-08-17 14:23:48.996225-03</p>
----------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------

A função `to_char` também pode ser usada para data e hora:

```
> SELECT to_char(now(), E'"Data: "DD/MM/YYYY \n"Hora:" HH24:MI:SS');

to_char
-----
Data: 17/08/2015 +
Hora: 14:40:29
```

15.3.1 extract(...)

Tem como objetivo pegar um dado de `current_timestamp` tais como:

Dia	<code>SELECT extract(day from now());</code>
Mês	<code>SELECT extract(month from now());</code>
Ano	<code>SELECT extract(year from now());</code>
Hora	<code>SELECT extract(hour from now());</code>
Minuto	<code>SELECT extract(minute from now());</code>
Segundos	<code>SELECT extract(secondfrom now());</code>
Década	<code>SELECT extract(decade from now());</code>
Dia da semana	<code>SELECT extract(dow from now());</code>
Dia do ano	<code>SELECT extract(doy from now());</code>
Época	<code>SELECT extract(epoch from now());</code>

15.4 Funções de Strings

Corta o caractere “x” da string:

```
> SELECT trim('xfoox', 'x');  
  
trim  
-----  
foo
```

```
> SELECT btrim('xfoox', 'x');  
  
btrim  
-----  
foo
```

Corta o caractere “x” da string à esquerda:

```
> SELECT ltrim('xfoox', 'x');  
  
ltrim  
-----  
foox
```

Corta o caractere “x” da string à direita:

```
> SELECT rtrim('xfoox', 'x');  
  
rtrim  
-----  
xfoo
```

Converte todos caracteres da string para letras minúsculas:

```
> SELECT lower('FOO');  
  
lower  
-----  
foo
```

Converte todos caracteres da string para letras maiúsculas:

```
> SELECT upper('foo');  
  
upper  
-----  
FOO
```

Primeira letra maiúscula:

```
> SELECT initcap('foo');  
  
initcap  
-----  
Foo
```

Tamanho da string:

```
> SELECT length('PostgreSQL');  
  
length  
-----  
10
```

Da string retornar do primeiro ao terceiro caractere:

```
> SELECT substr('foobar', 1, 3);  
  
substr  
-----  
foo
```

Substituir os caracteres “a” e “e”, respectivamente por “4” e “3”:

```
> SELECT translate('Hacker', 'ae', '43');  
  
translate  
-----  
H4ck3r
```

Retornar que posição está o caractere “i”:

```
> SELECT strpos('Linux', 'i');  
  
strpos  
-----  
2
```

Dada a string “23”, com 5 posições preencher com o caractere “0” à esquerda:

```
> SELECT lpad('23', 5, '0');  
  
lpad  
-----  
00023
```

<p>Dada a string “23”, com 5 posições preencher com o caractere “0” à direita:</p> <pre>> SELECT rpad('23', 5, '0');</pre> <pre> rpad ----- 23000 </pre>	<p>String em campos, separador um espaço em branco e retornar o segundo campo:</p> <pre>> SELECT split_part('Steve Harris', ' ', 2) AS "Sobrenome";</pre> <pre> Sobrenome ----- Harris </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

15.4.1 to_char()

Essa função é tão útil, merece até tratamento especial para falar dela. Como entrada pode-se colocar números ou datas e de acordo com a máscara inserida obtém-se interessantes resultados.

Detalhes da Função to_char (psql):

```
> \df to_char
```

Seu segundo argumento é do tipo text, uma máscara para modelar a conversão desejada.

<p>Algarismos Romanos:</p> <pre>> SELECT to_char(2009, 'RN');</pre> <pre> to_char ----- MMIX </pre>	<p>Saída de Data (Formato Brasileiro) e Hora (24H):</p> <pre>> SELECT to_char(now(), 'dd/mm/yyyy - HH24:MI');</pre> <pre> to_char ----- 18/08/2015 - 08:58 </pre>
--------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------

<p>Cinco Algarismos Preenchidos com Zeros à Esquerda:</p> <pre>> SELECT to_char(53, '00000');</pre> <pre> to_char ----- 00053 </pre>	<p>Até 9 (Nove) Algarismos + 2 (Duas) Casas Decimais, Preenchidos com Zeros e Separador de Milhar:</p> <pre>> SELECT to_char(29535.21, '000G000G000D00');</pre> <pre> to_char ----- 000.029.535,21 </pre>
-----------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Até 9 (nove) Algarismos + 2 (duas) Casas Decimais, não Preenchidos com Zeros e Separador de Milhar:

```
> SELECT to_char(29535.21, '999G999G999D99');  
  
      to_char  
-----  
      29.535,21
```

O caractere ponto (.) é especial para uma máscara, portanto devemos colocá-lo entre aspas para que ele seja interpretado literalmente:

```
> SELECT to_char(39684721495, '999"."999"."999-99') AS cpf;  
  
      cpf  
-----  
      396.847.214-95
```

15.5 Funções de Sistema

Banco de dados atual:	Qual é o usuário conectado na sessão atual:	
<pre>> SELECT current_database(); current_database ----- postgres</pre>	> <code>SELECT current_user;</code>	> <code>SELECT user;</code>
	<pre>current_user ----- postgres</pre>	

Qual é a versão do PostgreSQL?:

```
> SELECT version();

version
-----
PostgreSQL 9.4.4 on i686-pc-linux-gnu, compiled by gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3, 32-bit
```

Qual é o esquema atual:	Quais são os esquemas no caminho de procura, opcionalmente incluindo esquemas implícitos?:
<pre>> SELECT current_schema(); current_schema ----- public</pre>	<pre>> SELECT current_schemas(true); current_schemas ----- {pg_catalog,public}</pre>

Qual é o valor de determinada configuração?:	
> <code>SELECT current_setting('datestyle');</code>	> <code>SHOW datestyle;</code>
<pre>ISO, DMY</pre>	

Qual é endereço IP do servidor?:

```
> SELECT inet_server_addr();

inet_server_addr
-----
127.0.0.1
```

Qual é a porta de escuta do servidor?:	
> <code>SELECT inet_server_port();</code>	> <code>SHOW port;</code>
<pre>5432</pre>	

16 Agrupamentos de Dados

- Sobre Agrupamentos de Dados
- Funções de Agregação (ou de Grupos de Dados)
- A Cláusula GROUP BY
- A Cláusula HAVING
- A Cláusula FILTER

16.1 Sobre Agrupamentos de Dados

Em algumas situações é preciso ter dados por algum tipo de agrupamento e a partir desse agrupamento se fazer alguma coisa como for desejado.

16.2 Funções de Agregação (ou de Grupos de Dados)

Média salarial dos colaboradores:

```
> SELECT avg(salario)::numeric(7, 2) FROM tb_colaborador;
```

```
      avg
-----
 2633.57
```

Conta quantas linhas há na tabela colaboradores:

```
> SELECT count(*) FROM tb_colaborador;
```

```
      count
-----
        41
```

Conta quantas linhas em que o campo não é nulo:

```
> SELECT count(setor) FROM tb_colaborador;
```

```
      count
-----
        40
```

Exibir o CPF de quem tem o maior salário:

```
> SELECT cpf FROM tb_colaborador
   WHERE salario = (SELECT max(salario) FROM tb_colaborador);
```

```
      cpf
-----
1111111111
```

Exibir o CPF de quem tem o menor salário:

```
> SELECT cpf FROM tb_colaborador
   WHERE salario = (SELECT min(salario) FROM tb_colaborador);
```

```
      cpf
-----
96385274133
85274136944
```

Somatória da folha de pagamento:

```
> SELECT sum(salario) FROM tb_colaborador;
```

```
      sum  
-----  
105342.80
```

Desvio padrão do campo salario:

```
> SELECT stddev(salario) FROM tb_colaborador;
```

```
      stddev  
-----  
3338.113937755255
```

Variância do campo salario:

```
> SELECT variance(salario) FROM tb_colaborador;
```

```
      variance  
-----  
11143004.661435897436
```

16.3 GROUP BY

A cláusula `GROUP BY` agrupa resultados de acordo com funções de agrupamento.

É importante lembrar que todas as colunas relacionadas no `SELECT` que não estejam nas funções de grupo devem estar também na cláusula `GROUP BY`.

Quantos colaboradores há por setor:

```
> SELECT setor, count(setor) FROM tb_colaborador GROUP BY setor;
```

setor	count
	0
4	6
5	19
1	3
2	5
3	7

Quantos colaboradores ganham pelo menos R\$ 2.500,00 agrupando pelo chefe direto:

```
> SELECT chefe_direto, count(chefe_direto) AS n_subordinados
FROM tb_colaborador
WHERE salario >= 2500
GROUP BY chefe_direto;
```

chefe_direto	n_subordinados
8	3
1	7
2	2

16.4 A Cláusula HAVING

Tem por finalidade filtrar os dados agrupados impondo uma condição. Tal condição deve ter alguma das colunas do `SELECT`.

Quantos colaboradores ganham pelo menos R\$ 2.500,00 agrupando pelo chefe direto, somente agrupamentos com pelo Menos 3:

```
> SELECT chefe_direto, count(chefe_direto) AS n_subordinados
   FROM tb_colaborador
   WHERE salario >= 2500
   GROUP BY chefe_direto
   HAVING count(chefe_direto) >= 3;
```

chefe_direto	n_subordinados
8	3
1	7

16.5 A Cláusula FILTER

A cláusula FILTER foi introduzida como recurso no PostgreSQL na versão 9.4.

Estende funções de agregação (sum, avg, count, etc...) com uma cláusula WHERE adicional fazendo assim com que o resultado seja construído com apenas as linhas que satisfaçam a cláusula WHERE adicional.

De 1 a 900000 exibir quantos múltiplos de 3 tem (sem a cláusula FILTER):

```
> SELECT
  count(*) AS nao_filtrado,
  sum(CASE
    WHEN i % 3 = 0 THEN 1
    ELSE 0 END) AS filtrado
FROM generate_series(1, 900000) AS s(i);
```

De 1 a 900000 exibir quantos múltiplos de 3 tem (sem a cláusula FILTER):

```
> SELECT
  count(*) AS nao_filtrado,
  count(*) FILTER (WHERE i % 3 = 0) AS filtrado
FROM generate_series(1, 900000) AS s(i);
```

Duas formas diferentes de se obter o mesmo resultado:

```
nao_filtrado | filtrado
-----+-----
900000      | 300000
```

Qual dos dois jeitos será o mais eficiente?

Para isso devemos verificar o plano de execução (página seguinte).

Verificando o plano de execução de ambos os jeitos:

a) Sem a cláusula FILTER:

```
> EXPLAIN ANALYZE
SELECT
  count(*) AS nao_filtrado,
  sum(CASE
    WHEN i % 3 = 0 THEN 1
    ELSE 0 END) AS filtrado
FROM generate_series(1, 900000) AS s(i);
```

QUERY PLAN

```
Aggregate  (cost=20.00..20.01 rows=1 width=16) (actual time=256.618..256.618 rows=1 loops=1)
-> Function Scan on generate_series s  (cost=0.00..10.00 rows=1000 width=4) (actual
time=88.269..161.909 rows=900000 loops=1)
Planning time: 0.045 ms
Execution time: 258.582 ms
```

b) Com a cláusula FILTER:

```
> EXPLAIN ANALYZE
SELECT
  count(*) AS nao_filtrado,
  count(*) FILTER (WHERE i % 3 = 0) AS filtrado
FROM generate_series(1, 900000) AS s(i);
```

QUERY PLAN

```
Aggregate  (cost=20.00..20.01 rows=1 width=16) (actual time=250.518..250.518 rows=1 loops=1)
-> Function Scan on generate_series s  (cost=0.00..10.00 rows=1000 width=4) (actual
time=91.058..163.805 rows=900000 loops=1)
Planning time: 0.050 ms
Execution time: 252.477 ms
```

Foi observado que utilizando a cláusula FILTER tempo de planejamento é maior, porém a execução é mais rápida.

17 SEQUENCE – Sequência

- Sobre Sequência
- Funções de Manipulação de Sequência
- Usando Sequências em Tabelas
- Alterando uma Sequência
- Apagando uma Sequência
- Colunas Identity

17.1 Sobre Sequência

Uma sequência é usada para determinar valores automaticamente para um campo.

Sintaxe:

```
CREATE SEQUENCE nome_da_sequencia  
[INCREMENT incremento]  
[ MINVALUE valor_mínimo | NO MINVALUE ]  
[ MAXVALUE valor_máximo | NO MAXVALUE ]  
[ START [ WITH ] início ]  
[ CACHE cache ]  
[ [ NO ] CYCLE ];
```

Os parâmetros:

- **INCREMENT:** Valor de incremento.
- **MINVALUE:** Valor mínimo.
- **NO MINVALUE:** Não faz uso de valores mínimos padrões, sendo que 1 e $(-2^{63} - 1)$ para seqüências ascendentes e descendentes, respectivamente.
- **MAXVALUE:** Valor máximo.
- **NO MAXVALUE:** Será usado os valores máximos padrões: $(-2^{63} - 1)$ e -1 para seqüências ascendentes e descendentes, respectivamente.
- **START [WITH]:** Valor inicial.
- **CACHE:** Quantos números da sequência devem ser pré alocados e armazenados em memória para acesso mais rápido. O valor mínimo é 1 (somente um valor é gerado de cada vez, ou seja, sem cache), e este também é o valor padrão. Muito útil para ganho de performance. Os números pré alocados não utilizados são perdidos.
- **CYCLE:** Faz com que a sequência recomece quando for atingido o valor_máximo ou o valor_mínimo por uma sequência ascendente ou descendente, respectivamente. Se o limite for atingido, o próximo número gerado será o valor_mínimo ou o valor_máximo, respectivamente.
- **NO CYCLE:** Seu efeito se dá a toda chamada a nextval após a seqüência ter atingido seu valor máximo retornará um erro. Se não for especificado nem NO CYCLE é o padrão.

Criar uma sequência temporária que se dê de 5 em 5, valor mínimo 15, valor máximo 500 e sem ciclo:

```
> CREATE TEMP SEQUENCE sq_teste  
  INCREMENT 5  
  MINVALUE 15  
  MAXVALUE 500  
  NO CYCLE;
```

17.2 Funções de Manipulação de Sequência

- `nextval('sequencia')`: Próximo valor e incrementa.
- `currval('sequencia')`: Valor atual. Quando a sequência ainda não foi usada retornará erro.
- `setval('sequencia', valor)`: Determina um novo valor atual.

Damos início à sequência manualmente com `nextval` (repita três vezes):

```
> SELECT nextval('sq_teste');  
nextval  
-----  
      15  
  
nextval  
-----  
      20  
  
nextval  
-----  
      25
```

Observamos seu valor corrente com a função `currval`:

```
> SELECT currval('sq_teste');  
currval  
-----  
      25
```

Redefine o Valor Atual da Sequência:

```
> SELECT setval('sq_teste', 20);  
setval  
-----  
      20
```

17.3 Usando Sequências em Tabelas

Dentre os tipos de dados que foram vistos, serial nada mais é do que uma sequência criada na hora, de acordo com o nome da tabela.

Criação de Tabela:

```
> CREATE TEMP TABLE tb_teste_seq(  
    cod serial,  
    nome VARCHAR(15));
```

Verificando a Estrutura da tabela:

```
> \d tb_teste_seq
```

Column	Type	Table "pg_temp_3.tb_teste_seq"	Modifiers
cod	integer	not null default nextval('tb_teste_seq_cod_seq'::regclass)	
nome	character varying(15)		

Na descrição da estrutura da tabela, em “*Modifiers*” podemos observar que `cod` é um campo obrigatório (`not null`) e seu valor padrão (`default`) é o próximo valor da sequência (`nextval`) `tb_teste_seq_cod_seq`.

Obs.: Quando um campo de uma tabela é declarado como serial, na verdade se trata de um campo inteiro (`SMALLINT` → `SMALLSERIAL`, `INTEGER` → `SERIAL`, `BIGINT` → `BIGSERIAL`), para o qual é criada implicitamente uma sequência que o próximo valor dessa sequência é seu valor padrão.

O nome dessa sequência é composto da seguinte forma:
`nome_da_tabela_nome_do_campo_seq`.

Apagando a Tabela:

```
> DROP TABLE tb_teste_seq;
```

Criando a Tabela Usando a Sequência "sq_teste":

```
> CREATE TEMP TABLE tb_teste_seq(  
    cod int DEFAULT nextval('sq_teste'),  
    nome VARCHAR(15));
```

Inserções:

```
> INSERT INTO tb_teste_seq (nome)
  VALUES ('nome1'), ('nome2'), ('nome3'), ('nome4');
```

Verificando o Resultado:

```
> SELECT * FROM tb_teste_seq;
```

<i>cod</i>	<i>nome</i>
25	nome1
30	nome2
35	nome3
40	nome4

Como pôde ser observado, o próximo valor de `cod`, que é auto incremental.

17.4 Alterando uma Sequência

Ajuda para alterar uma sequência:

```
> \h ALTER SEQUENCE
```

Command: ALTER SEQUENCE

Description: change the definition of a sequence generator

Syntax:

```
ALTER SEQUENCE [ IF EXISTS ] name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

Voltar o Valor para 15:

```
> ALTER SEQUENCE sq_teste RESTART WITH 15;
```

De agora em diante os valores serão a partir de 15 em sq_teste.

17.5 Apagando uma Sequência

Ajuda para apagar uma sequência:

```
> \h DROP SEQUENCE
```

Command: DROP SEQUENCE

Description: remove a sequence

Syntax:

DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

Tentativa de Apagar uma Sequência:

```
> DROP SEQUENCE sq_teste;
```

ERROR: cannot drop sequence sq_teste because other objects depend on it

DETAIL: default for table tb_teste_seq column cod depends on sequence sq_teste

HINT: Use DROP ... CASCADE to drop the dependent objects too.

Erro de dependência, pois a sequência não pode ser removida porque o campo `cod` da tabela `tb_teste_seq` a tem como valor padrão conforme seu próximo retorno.

Em casos como esse é preciso fazer a remoção em cascata:

```
> DROP SEQUENCE sq_teste CASCADE;
```

NOTICE: drop cascades to default for table tb_teste_seq column cod
DROP SEQUENCE

A restrição `DEFAULT` foi removida do campo `cod`.

17.6 TRUNCATE para Reiniciar Sequências de uma Tabela

O comando `TRUNCATE`, como já foi visto redefine uma tabela como vazia.

O faz de uma forma muito mais inteligente e eficiente do que simplesmente um `DELETE` sem `WHERE`.

No entanto, se na tabela que é dado o `TRUNCATE`, por padrão ele não reinicia as sequências.

Isso é conseguido com a cláusula `RESTART IDENTITY`.

Criação de tabela de teste:

```
> CREATE TEMP TABLE tb_sequencia (  
    id serial primary key,  
    campo text);
```

Inserir valores:

```
> INSERT INTO tb_sequencia (campo) VALUES ('foo'), ('bar'), ('baz');
```

Verificar a tabela:

```
> TABLE tb_sequencia;
```

```
id | campo  
----+-----  
 1 | foo  
 2 | bar  
 3 | baz
```

Até aqui nada de fora do comum...

Verificando a estrutura da tabela:

```
> \d tb_sequencia
```

```
Table "pg_temp_2.tb_sequencia"  
Column | Type          | Modifiers  
-----+-----  
id      | integer       | not null default nextval('tb_sequencia_id_seq'::regclass)  
campo   | text          |  
Indexes:  
    "tb_sequencia_pkey" PRIMARY KEY, btree (id)
```

Podemos notar que há uma sequência criada implicitamente, devido ao pseudo-tipo `serial` na declaração de criação da tabela, cujo nome é `tb_sequencia_id_seq`.

Redefinir a tabela como vazia:

```
> TRUNCATE tb_sequencia;
```

Verificar o valor atual da sequência:

```
> SELECT currval('tb_sequencia_id_seq');

currval
-----
      3
```

Inserir valores:

```
> INSERT INTO tb_sequencia (campo) VALUES ('foo'), ('bar'), ('baz');
```

Verificar os valores da tabela:

```
> TABLE tb_sequencia;

id | campo
---+-----
 4 | foo
 5 | bar
 6 | baz
```

Apesar do TRUNCATE, a sequência continuou. Esse é o comportamento padrão (cláusula CONTINUE IDENTITY).

Apagar a tabela:

```
> DROP TABLE tb_sequencia;
```

Recriar a tabela:

```
> CREATE TEMP TABLE tb_sequencia (
  id serial primary key,
  campo text);
```

Inserir valores:

```
> INSERT INTO tb_sequencia (campo) VALUES ('foo'), ('bar'), ('baz');
```

Verificar valores da tabela:

```
> TABLE tb_sequencia;
```

id	campo
1	foo
2	bar
3	baz

Redefinir a tabela como vazia e também reiniciar todas as sequências que ela tiver:

```
> TRUNCATE tb_sequencia RESTART IDENTITY;
```

Verificar o valor atual da sequência:

```
> SELECT currval('tb_sequencia_id_seq');
```

currval
3

Isso faz surgir uma dúvida: será que a sequência não foi reiniciada e a situação continuará a mesma?

Inserir valores:

```
> INSERT INTO tb_sequencia (campo) VALUES ('foo'), ('bar'), ('baz');
```

Verificando os valores na tabela:

```
> TABLE tb_sequencia;
```

id	campo
1	foo
2	bar
3	baz

OK! Tabela com valores iniciais novamente!

17.7 Colunas Identity

Sintaxe:

```
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ]
```

Coluna identidade (identity column) tem uma sequência atrelada a ela implicitamente de forma similar a uma coluna serial.

As cláusulas `BY DEFAULT` e `ALWAYS` determinam respectivamente se permitirá ou não declarar o campo no `INSERT`.

`BY DEFAULT` permite que seja especificada a coluna identidade no `INSERT` e `ALWAYS` não permite a não ser que seja usada a cláusula `OVERRIDING SYSTEM VALUE` no `INSERT`.

A parte `sequence_options` é opcional e pode ser usada para sobrescrever as opções da sequência.

Tabela criada com serial:

```
> CREATE TABLE tb_serial (  
    id serial PRIMARY KEY,  
    campo text);
```

Criação de tabela como a tabela anterior:

```
> CREATE TABLE tb_serial2 (LIKE tb_serial INCLUDING ALL);
```

Verificando a estrutura de ambas as tabelas:

```
> \d tb_serial
```

```
Table "public.tb_serial"
Column | Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
id      | integer   |           | not null | nextval('tb_serial_id_seq'::regclass)
campo   | text      |           |          |
Indexes:
    "tb_serial_pkey" PRIMARY KEY, btree (id)
```

```
> \d tb_serial2
```

```
Table "public.tb_serial2"
Column | Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
id      | integer   |           | not null | nextval('tb_serial_id_seq'::regclass)
campo   | text      |           |          |
Indexes:
    "tb_serial2_pkey" PRIMARY KEY, btree (id)
```

Mesma sequence para ambas as tabelas...

Criação de uma tabela com coluna identity:

```
> CREATE TABLE tb_identity (  
    id int PRIMARY KEY  
        GENERATED BY DEFAULT AS IDENTITY,  
    campo text);
```

Verificando a estrutura:

```
> \d tb_identity
```

```
Table "public.tb_identity"  
Column | Type      | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
id      | integer   |           | not null | generated by default as identity  
campo   | text      |           |          |  
Indexes:  
    "tb_identity_pkey" PRIMARY KEY, btree (id)  
  
    Diferente de uma coluna com serial, aqui não vemos o nome da sequence atrelada.  
    No entanto a mesma existe e segue o padrão de nomenclatura do PostgreSQL, que  
tabela_coluna_seq.
```

Criação de uma nova tabela a partir de outra, em que há uma coluna identity:

```
> CREATE TABLE tb_identity2 (like tb_identity INCLUDING ALL);
```

Verificando a estrutura:

```
> \d tb_identity2
```

```
Table "public.tb_identity2"  
Column | Type      | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
id      | integer   |           | not null | generated by default as identity  
campo   | text      |           |          |  
Indexes:  
    "tb_identity2_pkey" PRIMARY KEY, btree (id)
```

Inserindo um valor em cada tabela:

```
> INSERT INTO tb_identity (campo) VALUES ('foo');  
  
> INSERT INTO tb_identity2 (campo) VALUES ('bar');
```

Verificando as tabelas;

```
> TABLE tb_identity;
```

id	campo
1	foo

```
> TABLE tb_identity2;
```

id	campo
1	bar

Verificando as sequences atreladas às tabelas:

```
> SELECT currval('tb_identity_id_seq');
```

currval
1

```
> SELECT currval('tb_identity2_id_seq');
```

currval
1

Se as tabelas usassem serial, teriam a mesma sequence.

17.7.1 GENERATED BY DEFAULT vs GENERATED ALWAYS

Criação de tabela como GENERATED BY DEFAULT:

```
> CREATE TABLE tb_identity_default (  
  id int PRIMARY KEY  
    GENERATED BY DEFAULT AS IDENTITY,  
  campo text);
```

Inserir valores:

```
> INSERT INTO tb_identity_default (campo) VALUES ('foo');
> INSERT INTO tb_identity_default (campo) VALUES ('bar');
> INSERT INTO tb_identity_default (campo) VALUES ('baz');
> INSERT INTO tb_identity_default (id, campo) VALUES (4, 'spam');
> INSERT INTO tb_identity_default (id, campo) VALUES (4, 'spam');

ERROR:  duplicate key value violates unique constraint "tb_identity_default_pkey"
DETAIL:  Key (id)=(4) already exists.
```

Houve uma colisão por causa do valor da sequence.

Criação de tabela como GENERATED ALWAYS:

```
> CREATE TABLE tb_identity_always (
    id int PRIMARY KEY
    GENERATED ALWAYS AS IDENTITY,
    campo text);
```

Inserir valores:

```
> INSERT INTO tb_identity_always (campo) VALUES ('foo');
> INSERT INTO tb_identity_always (campo) VALUES ('bar');
> INSERT INTO tb_identity_always (campo) VALUES ('baz');
> INSERT INTO tb_identity_always (id, campo) VALUES (4, 'spam');

ERROR:  cannot insert into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

Colunas GENERATED ALWAYS, por padrão não permitem ser declaradas a não ser que se use a cláusula OVERRIDING SYSTEM VALUE.

INSERT declarando a coluna GENERATED ALWAYS com a cláusula OVERRIDING SYSTEM VALUE:

```
> INSERT INTO tb_identity_always (id, campo) OVERRIDING SYSTEM VALUE VALUES (4, 'spam');
```

INSERT sem declarar a coluna:

```
> INSERT INTO tb_identity_always (campo) VALUES ('eggs');
```

```
ERROR: duplicate key value violates unique constraint "tb_identity_always_pkey"  
DETAIL: Key (id)=(4) already exists.
```

Por ter sido forçada a inserção de um valor que pode dar colisão, aconteceu o erro por causa de colisão de valores, que devem ser únicos.

Verificando o valor atual da sequence atrelada:

```
> SELECT currval('tb_identity_always_id_seq');
```

```
currval  
-----  
4
```

17.7.2 TRUNCATE em Tabelas com Coluna Identity

Simple TRUNCATE:

```
> TRUNCATE tb_identity_always;
```

Inserir um dado:

```
> INSERT INTO tb_identity_always (campo) VALUES ('eggs');
```

Verificar a tabela:

```
> TABLE tb_identity_always;
```

```
id | campo  
---+-----  
5 | eggs
```

Assim como acontece em colunas que não são identity, a sequence atrelada não foi reiniciada.

TRUNCATE com RESTART IDENTITY:

```
> TRUNCATE tb_identity_always RESTART IDENTITY;
```

Inserir um dado:

```
> INSERT INTO tb_identity_always (campo) VALUES ('eggs');
```

Verificando a tabela:

```
> TABLE tb_identity_always;
```

<i>id</i>	<i>campo</i>
1	eggs

TRUNCATE dado na tabela e sequence reiniciada.

18 UUID

- Sobre UUID
- O Módulo uuid-osp

18.1 Sobre UUID

UUID significa *Universally Unique Identifiers* (Identificadores Universais Únicos) e é um padrão definido pela RFC 4122, ISO/IEC 9834-8:2005 [1]. Alguns sistemas se referem a esse tipo de dado como GUID; *Globally Unique Identifier* (Identificador Único Globalmente).

É um identificador de 128 bits que é gerado por um algoritmo escolhido para fazê-lo de forma que seja muito improvável que o mesmo valor seja gerado por alguma outra pessoa no universo.

Sendo assim, para sistemas distribuídos, esses identificadores fornecem uma melhor garantia de unicidade do que geradores de sequência, que são únicos apenas em uma única base de dados.

Um UUID é uma sequência de dígitos hexadecimais (letras minúsculas), em grupos separados por hífens, especialmente um grupo de 8 (oito) dígitos seguidos de grupos de 4 (quatro) dígitos seguidos de um grupo de 12 (doze) dígitos, constituindo um total de 32 dígitos representando os 128 bits.

Exemplo:

94521790-02e7-4633-9bd6-41b29fd87dc4

O PostgreSQL também aceita as seguintes formas de entrada:

- Dígitos hexadecimais em letras maiúsculas:
9C4DD709-BDB2-4511-9922-1A5CB808BBF8;
- Formato padrão envolto por chaves:
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11};
- Omitindo algum ou todos hífens:
a0eebc999c0b4ef8bb6d6bb9bd380a11;
- Adicionando um hífen após qualquer grupo de quatro dígitos:
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11;
- Entre chaves com hífens fora do padrão:
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}.

A saída será sempre na forma padrão.

O PostgreSQL fornece armazenamento e comparações para o tipo uuid, mas no core (núcleo) não inclui nenhuma função para gerar UUIDs.

O módulo uuid-osp fornece funções que implementam vários algoritmos padrão.

O módulo pgcrypto também fornece função para gerar UUIDs aleatórios.

Alternativamente, UUIDs podem ser gerados pela aplicação cliente ou outras bibliotecas invocadas através de uma função do lado do servidor.

[1] <https://tools.ietf.org/html/rfc4122>

18.2 O Módulo uuid-oss

É um módulo contrib [2] que fornece funções para gerar UUIDs usando um ou vários algoritmos padrão.

Há também funções para produzir certas constantes UUID especiais.

Criação da tabela de teste:

```
> CREATE TABLE tb_uuid(  
    id uuid PRIMARY KEY,  
    campo text);
```

Habilitando o módulo na base de dados corrente:

```
> CREATE EXTENSION "uuid-oss";
```

Devido a uma má prática de nomenclatura, neste caso um hífen no nome, é necessário colocar o nome entre aspas.

Obs.: Para se instalar um módulo / extensão é preciso que o usuário tenha o atributo SUPERUSER.

Exibir informações da extensão:

```
> \dx+ "uuid-oss"
```

```
Objects in extension "uuid-oss"  
Object description
```

```
-----  
function uuid_generate_v1()  
function uuid_generate_v1mc()  
function uuid_generate_v3(uuid,text)  
function uuid_generate_v4()  
function uuid_generate_v5(uuid,text)  
function uuid_nil()  
function uuid_ns_dns()  
function uuid_ns_oid()  
function uuid_ns_url()  
function uuid_ns_x500()
```

[2] <https://www.postgresql.org/docs/current/static/contrib.html>

Executar a função de gerar UUID:

```
> SELECT uuid_generate_v4();
```

```
          uuid_generate_v4
-----
8e3734d9-7091-4ccf-8061-eedb999393ee
```

Inserir um registro na tabela declarando o campo uuid:

```
> INSERT INTO tb_uuid (id, campo) VALUES (uuid_generate_v4(), 'foo');
```

Verificando a tabela;

```
> TABLE tb_uuid;
```

```
          id                      | campo
-----+-----
65883db6-10e2-43d4-9172-b3f96c573301 | foo
```

Alterando a coluna id da tabela para ter um valor automático:

```
> ALTER TABLE tb_uuid ALTER COLUMN id SET DEFAULT uuid_generate_v4();
```

Verificando a estrutura da tabela:

```
> \d tb_uuid
```

```
          Table "public.tb_uuid"
  Column | Type  | Collation | Nullable | Default
-----+-----+-----+-----+-----
id       | uuid  |           | not null | uuid_generate_v4()
campo    | text  |           |          |
Indexes:
    "tb_uuid_pkey" PRIMARY KEY, btree (id)
```

Inserir um valor na tabela com o valor uuid gerado automaticamente:

```
> INSERT INTO tb_uuid (campo) VALUES ('bar');
```

Verificando a tabela;

> **TABLE** tb_uuid;

<i>id</i>	<i>campo</i>
65883db6-10e2-43d4-9172-b3f96c573301	foo
88fff5df-f359-4af6-9ebb-6410adcc4543	bar

19 VIEW – Visão

- Sobre Visão
- Visões Materializadas

19.1 Sobre Visão

View ou visão, nada mais é do que uma consulta armazenada como se fosse uma tabela virtual.

Sintaxe:

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name
[ ( column_name [, ...] ) ]
  [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
  AS query
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Criação da tabela de teste:

```
> CREATE TEMP TABLE tb_foo(
    id serial PRIMARY KEY,
    campo1 smallint,
    campo2 smallint);
```

Popular a tabela:

```
> INSERT INTO tb_foo (campo1, campo2)
    SELECT generate_series(1, 100, 3), generate_series(1, 100);
```

Criação da view:

```
> CREATE VIEW vw_foo AS
    SELECT campo1, campo2
    FROM tb_foo
    WHERE campo2 % 19 = 0;
```

Uma simples consulta limitada a cinco linhas:

```
> TABLE vw_foo LIMIT 5;
```

campo1	campo2
55	19
10	38
67	57
22	76
79	95

Criação de uma visão que visualiza de forma legível dados de funcionários:

```
> CREATE VIEW vw_ficha_colaborador AS
SELECT
    col.id "Matrícula",
    col.cpf "CPF",
    pfl.nome||' '||pfl.sobrenome "Nome completo",
    sel.nome "Setor",
    cal.nome "Cargo",
    pf2.nome||' '||pf2.sobrenome "Chefe direto",
    col.salario "Salario",
    col.dt_admis "Data de admissão"
FROM tb_colaborador col
INNER JOIN tb_pf pfl ON (col.cpf = pfl.cpf)
INNER JOIN tb_setor sel ON (col.setor = sel.id)
INNER JOIN tb_cargo cal ON (col.cargo = cal.id)
INNER JOIN tb_colaborador co2 ON (col.cheefe_direto = co2.id)
INNER JOIN tb_pf pf2 ON (co2.cpf = pf2.cpf)
WHERE col.ativo = true;
```

Uma simples consulta na nova view:

```
> SELECT "Nome completo", "Setor", "Cargo" FROM vw_ficha_colaborador LIMIT 3;
```

Nome completo	Setor	Cargo
Aldebarina Ferreira	Presidência	Presidente
Wolfrâmia Santos	RH	Gerente
Tungstênia Santana	Presidência	Secretária

Criação de uma visão que visualiza de forma legível dados de funcionários:

```
> CREATE VIEW vw_endereco_pf AS
SELECT
    p1.nome||' '||p1.sobrenome "Nome completo",
    cl.id "Matrícula",
    tl.abreviatura||' '||cep.logradouro||', '||e1.numero "Endereço",
    coalesce(e1.complemento, ' -- ') "Complemento",
    b1.nome "Bairro",
    m1.nome "Município",
    uf.uf "UF",
    to_char(cep.id, '00000-000') "CEP"
FROM tb_endereco_pf e1
INNER JOIN tb_pf p1 ON (e1.cpf = p1.cpf)
INNER JOIN tb_colaborador cl ON (e1.cpf = cl.cpf)
INNER JOIN tb_cep cep ON (e1.cep = cep.id)
INNER JOIN tb_tp_logradouro tl ON (cep.tp_logradouro = tl.id)
INNER JOIN tb_bairro b1 ON (cep.bairro = b1.id)
INNER JOIN tb_municipio m1 ON (b1.municipio = m1.id)
INNER JOIN tb_uf uf ON (m1.uf = uf.uf)
ORDER BY cl.id;
```


Uma simples consulta na nova view:

```
> SELECT "Nome completo", "Endereço" FROM vw_endereco_pf LIMIT 5;
```

Nome completo	Endereço
Chiquinho da Silva	AV Álvaro Otacilio, 2900
Chiquinho da Silva	R Breves, 3541
Aldebarina Ferreira	R Simão Lopes, 2003
Wolfrâmia Santos	R dos Jornalistas, 1592
Tungstênia Santana	AV Doutor Rudge Ramos, 648

19.2 Visões Materializadas

O comando `CREATE MATERIALIZED VIEW` define uma view materializada de uma consulta.

A consulta é executada e usada para popular a view na hora que o comando é dado (exceto se for usado `WITH NO DATA`) e pode ser atualizada posteriormente utilizando `REFRESH MATERIALIZED VIEW`.

`CREATE MATERIALIZED VIEW` é similar a `CREATE TABLE AS`, exceto que também lembra a consulta usada para iniciar a view, de modo que possa ser atualizada posteriormente sob demanda.

Uma view materializada tem muitas das mesmas propriedades de uma tabela, mas não há suporte para views materializadas temporárias ou geração automática de OIDs.

Sua grande vantagem com relação à uma view comum é o desempenho.

Uma visão materializada precisa do comando `REFRESH` para ter seus dados atualizados com relação à tabela de origem.

Sintaxe:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ...] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name

    [ WITH [ NO ] DATA ]
```

Criação da tabela de teste:

```
> CREATE TABLE tb_banda(
    id serial PRIMARY KEY,
    nome TEXT,
    origem TEXT);
```

Inserir dados:

```
> INSERT INTO tb_banda (nome, origem) VALUES
    ('Angra', 'Brasil'),
    ('Shaman', 'Brasil'),
    ('Sepultura', 'Brasil'),
    ('Helloween', 'Alemanha'),
    ('Blind Guardian', 'Alemanha'),
    ('Sanctuary', 'EUA'),
    ('Black Sabbath', 'Inglaterra'),
    ('Manowar', 'EUA'),
    ('Kamelot', 'EUA'),
    ('Epica', 'Holanda'),
    ('Gamma Ray', 'Alemanha');
```

Criação de uma view comum:

```
> CREATE VIEW vw_banda_br AS
    SELECT id, nome FROM tb_banda WHERE origem = 'Brasil';
```

Criação de view materializada:

```
> CREATE MATERIALIZED VIEW mv_banda_br AS
    SELECT id, nome FROM tb_banda WHERE origem = 'Brasil';
```

Selecionando todos os dados da view comum:

```
> TABLE vw_banda_br;
```

id	nome
1	Angra
2	Shaman
3	Sepultura

Selecionando todos os dados da view materializada:

```
> TABLE mv_banda_br;
```

id	nome
1	Angra
2	Shaman
3	Sepultura

Inserindo novos valores na tabela:

```
> INSERT INTO tb_banda (nome, origem) VALUES ('Viper', 'Brasil');
```

Selecionando todos os dados da view comum:

```
> TABLE vw_banda_br;
```

id	nome
1	Angra
2	Shaman
3	Sepultura
12	Viper

Podemos reparar que a nova linha é retornada na view comum.

Selecionando todos os dados da view materializada:

```
> TABLE mv_banda_br;
```

id	nome
1	Angra
2	Shaman
3	Sepultura

Atualizando a view materializada:

```
> REFRESH MATERIALIZED VIEW mv_banda_br;
```

Selecionando todos os dados da view materializada:

```
> TABLE mv_banda_br;
```

id	nome
1	Angra
2	Shaman
3	Sepultura
12	Viper

Exibindo o plano de execução na view comum:

```
> EXPLAIN ANALYZE TABLE vw_banda_br;
```

```
-----  
QUERY PLAN  
-----  
Seq Scan on tb_banda (cost=0.00..20.38 rows=4 width=36) (actual time=0.012..0.017 rows=4 loops=1)  
  Filter: (origem = 'Brasil'::text)  
    Rows Removed by Filter: 8  
Planning time: 0.093 ms  
Execution time: 0.045 ms
```

Exibindo o plano de execução na view materializada:

```
> EXPLAIN ANALYZE TABLE mv_banda_br;
```

QUERY PLAN

```
-----  
Seq Scan on mv_banda_br (cost=0.00..22.30 rows=1230 width=36) (actual time=0.008..0.011 rows=4  
loops=1)  
Planning time: 0.050 ms  
Execution time: 0.042 ms
```

20 Cursores

- Sobre Cursores
- FETCH – Recuperando Linhas de um Cursor
- MOVE – Movendo Cursores
- CLOSE – Fechando Cursores

20.1 Sobre Cursores

São variáveis que apontam para consultas, armazenam os resultados, economizam memória não retornando todos os dados de uma só vez em consultas que retornam muitas linhas.

Em PL/pgSQL não precisa se preocupar com isso, uma vez que *loops* `FOR` internamente utilizam um cursor automaticamente para evitar problemas com memória.

Uma interessante utilização é retornar a referência a um cursor criado pela função, que permite a quem executou ler as linhas. Assim proporciona uma maneira eficiente para a função retornar grandes conjuntos de linhas.

Retorno de dados no formato texto ou no formato binário pelo comando `SELECT`.

Sintaxe:

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]  
          CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

20.1.1 Parâmetros

- **BINARY**: Retorna os dados no formato binário ao invés do formato texto;
- **INSENSITIVE**: Indica que os dados retornados pelo cursor não devem ser afetados pelas atualizações feitas nas tabelas subjacentes ao cursor, enquanto o cursor existir. No PostgreSQL todos os cursores são `INSENSITIVE`. Atualmente esta palavra-chave não produz efeito, estando presente por motivo de compatibilidade com o padrão SQL;
- **SCROLL** (rolar): Especifica que o cursor pode ser utilizado para retornar linhas de uma maneira não sequencial (por exemplo, para trás). Dependendo da complexidade do plano de execução do comando, especificar `SCROLL` pode impor uma penalidade de desempenho no tempo de execução do comando;
- **NO SCROLL**: Faz com que o cursor não retorne linhas de uma maneira não sequencial. É recomendado usar essa opção para fins de desempenho e economia de memória quando é preciso que as linhas a serem retornadas sejam sequenciais;
- **WITH HOLD**: Especifica que o cursor pode continuar sendo utilizado após a transação que o criou ter sido efetivada com sucesso ou até mesmo criar um cursor sem estar dentro de uma transação;
- **WITHOUT HOLD**: Especifica que o cursor não pode ser utilizado fora da transação que o criou. Quando não é especificado nem `WITHOUT HOLD` nem `WITH HOLD`, o padrão é `WITHOUT HOLD`;
- **query**: A consulta feita para o cursor retornar;

Obs.:

Se `WITH HOLD` não for especificado, o cursor criado por este comando poderá ser utilizado somente dentro da transação corrente.

Cursor encerrados por transação são sempre fechados (efetivada ou não).

Se for especificado `NO SCROLL`, retornar linhas para trás não será permitido em nenhum caso.

O padrão SQL somente trata de cursores na linguagem SQL incorporada. O servidor PostgreSQL não implementa o comando `OPEN` para cursores; o cursor é considerado aberto ao ser declarado.

Entretanto o ECPG, o pré processador do PostgreSQL para a linguagem SQL incorporada, suporta as convenções de cursor do padrão SQL, incluindo as que envolvem os comandos `OPEN`.

Sessão 1	Sessão 2
<p>Criação de Cursor:</p> <pre>> BEGIN; DECLARE cursor1 CURSOR FOR SELECT * FROM tb_colaborador;</pre>	<p>Criação de Cursor Fora de uma Transação:</p> <pre>> DECLARE cursor2 CURSOR WITH HOLD FOR SELECT * FROM tb_colaborador;</pre>
	<p>Criação de Outro Cursor Fora de uma Transação:</p> <pre>> DECLARE cursor3 CURSOR WITH HOLD FOR SELECT * FROM tb_colaborador;</pre>

20.2 FETCH – Recuperando Linhas de um Cursor

Para tal tarefa é usado o comando `FETCH`, que em inglês significa buscar, trazer, alcançar...

Sintaxe:

```
FETCH [ direção { FROM | IN } ] nome_cursor;
```

Onde direção pode ser vazio ou:

- **NEXT**: Próxima linha. Este é o padrão quando a direção é omitida.
- **PRIOR**: Linha anterior.
- **ABSOLUTE n**: A n-ésima linha da consulta, ou a abs(n)-ésima linha a partir do fim se o n for negativo. Posiciona antes da primeira linha ou após a última linha se o n estiver fora do intervalo; em particular, `ABSOLUTE 0` posiciona antes da primeira linha.
- **RELATIVE n**: A n-ésima linha à frente, ou a abs(n)-ésima linha atrás se o n for negativo. `RELATIVE 0` retorna novamente a linha corrente, se houver.
- **FORWARD n**: É uma constante inteira, possivelmente com sinal, que determina a posição ou o número de linhas a serem retornadas. Para os casos `BACKWARD`.
- **ALL**: Todas as linhas restantes (o mesmo que `FORWARD ALL`).
- **FORWARD**: Próxima linha (o mesmo que `NEXT`).
- **FORWARD 0**: retorna novamente a linha corrente.
- **FORWARD ALL**: Todas as linhas restantes.
- **BACKWARD 0**: retorna novamente a linha corrente.
- **BACKWARD ALL**: Todas as linhas anteriores (varrendo para trás).

Sessão 2

Por padrão retorna a próxima linha:

```
> FETCH cursor2;
```

Retorna as próximas 2 linhas:

```
> FETCH 2 IN cursor2;
```

Retorna a linha anterior:

```
> FETCH -1 IN cursor2;
```

Próxima linha:

```
> FETCH FORWARD FROM cursor2;
```

Próximas 7 linhas:

```
> FETCH FORWARD 7 FROM cursor2;
```

Sempre retornará a primeira linha:

```
> FETCH FIRST FROM cursor2;
```

Sempre retornará a última linha:

```
> FETCH LAST FROM cursor2;
```

20.3 MOVE – Movendo Cursores

O comando MOVE faz a mudança na posição de um cursor sem retornar linhas. Sua sintaxe é similar à do FETCH:

Sintaxe:

```
MOVE [ direção { FROM | IN } ] nome_cursor
```

Sessão 2

Volta 7 posições do cursor:

```
> MOVE -7 FROM cursor2;
```

20.4 CLOSE – Fechando Cursores

Há duas maneiras de se fechar um cursor, de forma implícita; quando uma transação é encerrada ou não se concretiza (**COMMIT** ou **ROLLBACK**) e de forma explícita, com o comando **CLOSE**:

Sintaxe:

```
CLOSE { nome | ALL }
```

Sessão 2

Fecha Explicitamente cursor2:

```
> CLOSE cursor2;
```

Fecha Explicitamente Todos Cursores Abertos:

```
> CLOSE ALL;
```

Sessão 1

Fechamento Implícito por Finalização de Transação:

```
> ROLLBACK;
```

20.5 Apagando e Atualizando a Linha Onde o Cursor Aponta

Utilizando a cláusula CURRENT OF, pega-se a atual posição do cursor, ou seja, para qual linha ele aponta no momento e com esse apontamento podemos apagar ou alterar essa linha.

Se a tabela existir, apagar:

```
> DROP TABLE IF EXISTS tb_foo;
```

Criar tabela de teste: > SELECT generate_series(1, 5) campo INTO tb_foo;	Verificando a tabela: > TABLE tb_foo; <i>campo</i> ----- 1 2 3 4 5
------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------

Começar uma transação:

```
> BEGIN;
```

Criação de cursor:

```
> DECLARE c1 CURSOR FOR SELECT * FROM tb_foo;
```

Fazer o cursor apontar para a segunda posição:

```
> MOVE 2 c1;
```

Apagar a linha na posição atual do cursor: > DELETE FROM tb_foo WHERE CURRENT OF c1;	Verificando a tabela: > TABLE tb_foo; <i>campo</i> ----- 1 3 4 5
------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------

Posicionar o cursor na última linha da tabela:

```
> MOVE LAST c1;
```

Usar o cursor para atualizar a linha onde o cursor aponta atualmente:

```
> UPDATE tb_foo SET campo = 1000 WHERE CURRENT OF c1;
```

Verificando a tabela:

```
> TABLE tb_foo;
```

```
campo
-----
1
3
4
1000
```

Efetivando a transação:

```
> COMMIT;
```

21 BLOB

- Sobre BLOB
- Removendo BLOBs

21.1 Sobre BLOB

Sua utilidade está em armazenar arquivos dentro do banco de dados, sejam eles figuras, som, executáveis ou qualquer outro.

Os dados de um blob são do tipo `bytea`.

Para se inserir ou recuperar blobs no PostgreSQL é necessário utilizar as funções;

`lo_import(text)`: Seu parâmetro é a localização do arquivo a ser inserido no banco. Essa função retorna o identificador do arquivo inserido (oid).

`lo_export(oid, text)`: Seus parâmetros são respectivamente o oid do BLOB e a localização futura do arquivo.

Após importar um arquivo para a base de dados, o campo da tabela só terá o oid, que faz referência para os dados do BLOB que na verdade estão no catálogo `pg_largeobject`.

Comparando com um campo `bytea` não é algo vantajoso para se lidar com arquivos em bancos de dados, pois o arquivo precisa estar no servidor de banco de dados.

Descrição da Estrutura de "pg_largeobject" (via psql):

```
> \d pg_largeobject
```

```
pg_largeobject
```

```
Table "pg_catalog.pg_largeobject"
```

Column	Type	Modifiers
--------	------	-----------

loid	oid	not null
pageno	integer	not null
data	bytea	

```
Indexes:
```

```
"pg_largeobject_loid_pn_index" UNIQUE, btree (loid, pageno)
```

Criação de uma tabela de teste:

```
> CREATE TABLE tb_blob(  
    id_blob oid,  
    nome_blob varchar(15));
```

Seja o arquivo para nossos testes, a imagem `"/tmp/arquivo.png"`, importando o objeto (arquivo) para dentro do banco de dados:

```
> INSERT INTO tb_blob VALUES (lo_import('/tmp/arquivo.png'), 'Figura 1');
```


Recuperando o objeto no diretório /tmp:

```
> SELECT lo_export(id_blob, '/tmp/copia_blob.png')  
FROM tb_blob WHERE nome_blob = 'Figura 1';
```

Aviso:

Arquivos inseridos ou extraídos têm que estar no servidor.

21.2 Removendo BLOBs

Mesmo apagando uma linha de um BLOB com o comando `pg_largeobject`. Para tal deve-se usar a função `lo_unlink(oid)`, cujo parâmetro é a oid do BLOB.

Verificando a tabela:

```
> TABLE tb_blob;  
  
 id_blob | nome_blob  
-----+-----  
 16912 | Figura 1
```

A partir do id do blob (`lo_unlink`):

```
> SELECT lo_unlink(16912);
```

... ou também poderia ser `lo_unlink(id_blob)`:

```
> SELECT lo_unlink(id_blob) FROM tb_blob WHERE nome_blob = 'Figura 1';
```

Agora não existe mais o objeto binário, então pode-se apagar a referência a ele na tabela `tb_blob`.

Apagando a referência:

```
> DELETE FROM tb_blob WHERE nome_blob = 'Figura 1';
```

22 Bytea

- Sobre Bytea

22.1 Sobre Bytea

E se em vez de armazenarmos nossos arquivos binários em uma tabela de sistema pudéssemos fazer isso em uma tabela criada pelo próprio usuário?

Sim, isso é possível utilizando na(s) tabela(s) que for(em) usada(s) para guardá-los um ou mais campos do tipo `bytea`.

O campo `bytea` é utilizado para armazenar uma string, mas não uma string qualquer, uma string especial proveniente de um processo de “*dump*” de um arquivo binário.

Outra grande vantagem de se usar uma tabela com campo `bytea` é que os arquivos inseridos / extraídos não precisam estar no servidor.

Como exemplo utilizaremos a linguagem Python :)

Criação de tabela de teste:

```
> CREATE TABLE tb_bytea(  
    id SERIAL PRIMARY KEY,  
    arquivo VARCHAR(255),  
    dados bytea);
```

Em um terminal como root instale o driver PostgreSQL para Python:

```
# aptitude -y install python3-psycopg2 ipython3
```

Entrando no shell interativo de Python:

```
$ ipython3
```

Código em Python para Inserir o Arquivo na Tabela:

```
> # INSERIR UM BINÁRIO NO BANCO
# Exemplo em Python

# Importação de módulo - Driver PostgreSQL
import psycopg2

# Importação da função getpass do módulo getpass
from getpass import getpass

# Mensagem em tela pedindo endereço do servidor
pgserver = input('Digite o endereço do servidor de banco de dados: ')

# Mensagem em tela pedindo a senha do servidor
pgpass = getpass(prompt='Digite a senha do servidor de banco de dados: ')

# String de conexão
str_con = '''
    dbname=postgres
    user=postgres
    host={}
    password={}
    application_name=cliente_python'''.format(pgserver, pgpass)

# Conexão ao banco
conexao = psycopg2.connect(str_con)

# Criação de cursor para executar comandos SQL na base de dados
cur = conexao.cursor()

# Converte o arquivo aberto para bytes
arquivo = open('/tmp/arquivo.png', 'rb').read()

# Converte os bytes para o formato de armazenamento em banco
dados = psycopg2.Binary(arquivo)

# String SQL do comando a ser executado em banco
str_sql = """
    INSERT INTO tb_bytea (dados, arquivo) VALUES
        ({}, '/tmp/arquivo.png')""".format(dados)

# Execução do comando
cur.execute(str_sql)

# Efetivação da transação
conexao.commit()

# Antes de fechar a conexão Python (próximo statement Python), dê o seguinte
# comando no banco PostgreSQL:
#
# SELECT application_name FROM pg_stat_activity;

# Fechamento de conexão
conexao.close()
```

Apenas por curiosidade, vamos verificar as conexões estabelecidas no banco antes da conexão fechar:

```
> SELECT application_name FROM pg_stat_activity;
```

```
application_name
-----
. . .
cliente_python
```

Não selecionamos também o campo dados, pois nossa visualização ficaria muito bagunçada. Porém, vamos descobrir se realmente deu certo o que fizemos trazendo de volta os dados que estão no campo em forma de uma cópia do arquivo.

OK, para conferirmos inicialmente vamos selecionar o campo arquivo da tabela:

```
> SELECT arquivo FROM tb_bytea;
```

```
arquivo
-----
/tmp/arquivo.png
```

Não selecionamos também o campo dados, pois nossa visualização ficaria muito bagunçada. Porém, vamos descobrir se realmente deu certo o que fizemos trazendo de volta os dados que estão no campo em forma de uma cópia do arquivo.

Novamente no Shell Interativo de Python:

```
> # EXTRAIR DO BANCO PARA UM ARQUIVO
# Exemplo em Python

import psycopg2
from getpass import getpass

# Mensagem em tela pedindo endereço do servidor
pgserver = input('Digite o endereço do servidor de banco de dados: ')

# Mensagem em tela pedindo a senha do servidor
pgpass = getpass(prompt='Digite a senha do servidor de banco de dados: ')

# String de conexão
str_con = '''
    dbname=postgres
    user=postgres
    host={}
    password={}
    application_name=cliente_python'''.format(pgserver, pgpass)

# Conexão
conexao = psycopg2.connect(str_con)

# Criação de cursor
cur = conexao.cursor()

# String SQL da consulta em banco
str_sql = "SELECT dados FROM tb_bytea WHERE arquivo = '/tmp/arquivo.png';"

# Executar o comando no banco
cur.execute(str_sql)

# Dados extraídos
dados = cur.fetchone()[0]

# Fechamento de conexão com o banco
conexao.close()

# Criação do arquivo extraído do banco
open('/tmp/copia_bytea.png', 'wb').write(dados)
```

Para comprovarmos nossa experiência, vamos abrir o arquivo.
Como o arquivo é uma imagem, podemos utilizar o aplicativo eog:

Visualizando a Imagem:

```
$ eog /tmp/copia_bytea.png
```

23 COPY

- Sobre COPY
- Formato CSV
- COPY Remoto

23.1 Sobre COPY

O comando COPY faz a cópia de registros entre um arquivo e uma tabela, de uma tabela para *standard output* = saída padrão) ou de *standard input* = entrada padrão) para uma tabela.

Esse arquivo tem os campos da tabela com um delimitador conforme determinado no comando COPY.

Sintaxe:

```
COPY table_name [ ( column_name [, ...] ) ]  
  FROM { 'filename' | PROGRAM 'command' | STDIN }  
  [ [ WITH ] ( option [, ...] ) ]  
  
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }  
  TO { 'filename' | PROGRAM 'command' | STDOUT }  
  [ [ WITH ] ( option [, ...] ) ]
```

Onde option pode ser:

```
FORMAT format_name  
OIDS [ boolean ]  
FREEZE [ boolean ]  
DELIMITER 'delimiter_character'  
NULL 'null_string'  
HEADER [ boolean ]  
QUOTE 'quote_character'  
ESCAPE 'escape_character'  
FORCE_QUOTE { ( column_name [, ...] ) | * }  
FORCE_NOT_NULL ( column_name [, ...] )  
FORCE_NULL ( column_name [, ...] )  
ENCODING 'encoding_name'
```

Aviso:

Os arquivos tratados pelo comando COPY levam em conta a localização do *filesystem* do servidor e não localmente. Ou seja, caso esteja acessando um servidor remoto consulte a documentação da sua linguagem de programação.

Terminal shell do sistema operacional:

Criando o arquivo com os dados:

```
$ cat << EOF > /tmp/local.csv
São Paulo,SP
Rio de Janeiro,RJ
Belo Horizonte,MG
Cidade Fantasma,
EOF
```

Aviso:

Os campos são separados pelo delimitador e não pode ter espaço entre eles, pois o espaço também é considerado como um caractere.

Terminal psql:

Criação da tabela de teste:

```
> CREATE TABLE tb_copy(
    id serial PRIMARY KEY,
    cidade text,
    uf char(2));
```

Copiando os dados do arquivo para a tabela, especificando apenas dois campos:

```
> COPY tb_copy (cidade, uf)
    FROM '/tmp/local.csv' DELIMITER ',' NULL '';
```

No comando, o parâmetro “NULL” determina que uma string vazia será considerada como um valor nulo.

Visualizando os dados da tabela:

```
> TABLE tb_copy;
```

id	cidade	uf
1	São Paulo	SP
2	Rio de Janeiro	RJ
3	Belo Horizonte	MG
4	Cidade Fantasma	

Visualizando os dados da tabela com COPY para a tela (STDOUT):

```
> COPY tb_copy TO STDOUT;

1      São Paulo      SP
2      Rio de Janeiro RJ
3      Belo Horizonte MG
4      Cidade Fantasma      \N
```

Repare que o valor nulo é representado pelo “\N”.

Dados da tabela com COPY para a tela (STDOUT) com o pipe “|” como delimitador:

```
> COPY tb_copy TO STDOUT DELIMITER '|';

1|São Paulo|SP
2|Rio de Janeiro|RJ
3|Belo Horizonte|MG
4|Cidade Fantasma|\N
```

Primeira linha com o nome dos campos (cabeçalho / header):

```
> COPY tb_copy TO STDOUT DELIMITER ',' CSV HEADER NULL '--';

id,cidade,uf
1,São Paulo,SP
2,Rio de Janeiro,RJ
3,Belo Horizonte,MG
4,Cidade Fantasma,--
```

Copiar o conteúdo da tabela para o arquivo usando o pipe como delimitador:

```
> COPY tb_copy TO '/tmp/backup.txt' DELIMITER '|';
```

Terminal shell do sistema operacional:

Visualização do Conteúdo do Arquivo no Sistema Operacional:

```
$ cat /tmp/backup.txt

1|São Paulo|SP
2|Rio de Janeiro|RJ
3|Belo Horizonte|MG
4|Cidade Fantasma|\N
```

Terminal psql:

TRUNCATE na tabela, ou seja, redefinindo-a como vazia:

```
> TRUNCATE tb_copy;
```

E recuperamos os dados que nela estavam usando COPY:

```
> COPY tb_copy FROM '/tmp/backup.txt' DELIMITER '|';
```

Visualizando os dados da tabela:

```
> TABLE tb_copy;
```

id	cidade	uf
1	São Paulo	SP
2	Rio de Janeiro	RJ
3	Belo Horizonte	MG
4	Cidade Fantasma	

Criação de uma tabela temporária com dois campos de números inteiros:

```
> CREATE TEMP TABLE tb_copy_num_vazio(  
    campo_1 int2,  
    campo_2 int2);
```

Terminal shell do sistema operacional:

Criação do arquivo com valores não preenchidos:

```
$ cat << EOF > /tmp/local.csv  
1,7  
3,  
,9  
77,81  
EOF
```

Terminal psql:

Importando os dados do arquivo para a tabela:

```
> COPY tb_copy_num_vazio FROM '/tmp/local.csv' DELIMITER ',' NULL '';
```

Verificando a tabela:

```
> TABLE tb_copy_num_vazio;
```

campo_1	campo_2
1	7
3	
	9
77	81

Consultando onde tem valores nulos:

```
> SELECT campo_1, campo_2 FROM tb_copy_num_vazio  
WHERE campo_1 IS NULL  
OR campo_2 IS NULL;
```

campo_1	campo_2
3	
	9

Aviso:

Atentar a tabelas que utilizam **seqüências**, pois dependendo do caso é necessário alterar o valor de um objeto seqüência (SEQUENCE).

23.2 Formato CSV

CSV significa *Comma Separated Values* (Valores Separados por Vírgula), que no caso são os valores respectivos aos campos de uma tabela.

Com o parâmetro CSV o arquivo resultante é gerado automaticamente com vírgulas como delimitadores de colunas, mesmo as que estão nulas.

Gerando um Arquivo CSV:

```
> COPY tb_copy TO '/tmp/backup.csv' CSV HEADER;
```

Fazendo TRUNCATE na Tabela Novamente:

```
> TRUNCATE tb_copy;
```

De um Arquivo CSV para a Tabela:

```
> COPY tb_copy FROM '/tmp/backup.csv' CSV HEADER;
```

Visualizando os dados da tabela:

```
> TABLE tb_copy;
```

id	cidade	uf
1	São Paulo	SP
2	Rio de Janeiro	RJ
3	Belo Horizonte	MG
4	Cidade Fantasma	

Inserindo Dados com o COPY pelo Teclado:

```
> COPY tb_copy (cidade, uf) FROM stdin CSV;
```

```
Enter data to be copied followed by a newline.  
End with a backslash and a period on a line by itself.  
>> Uma cidade qualquer...,  
>> Maceió,AL  
>> Porto Velho,RO
```

Os dois sinais “maior que” (>>) indicam a linha corrente para se inserir dados.

No exemplo foi determinado o formato CSV o que implica na inserção dos valores separados por vírgula.

Para pular para a próxima linha, um novo registro, pressione <ENTER>.

Para finalizar as inserções pressione <CTRL+D>.

23.3 COPY Remoto

Quando o arquivo de origem ou destino estiver em uma máquina diferente do servidor PostgreSQL é necessário fazer uso de artifícios da linguagem de programação escolhida pelo usuário.

Aqui vamos ver como faríamos via shell de comandos no Linux, que também pode servir para outros sistemas operacionais da família Unix, como o FreeBSD e outros.

Com o comando `read`, exibe uma mensagem, pede uma entrada de teclado e a joga para a variável:

```
$ read -p 'Digite o IP do servidor PostgreSQL: ' PGSERVER
```

Digite o IP do servidor PostgreSQL:

Fazendo o truncate remotamente reiniciando a sequence atrelada à tabela:

```
$ psql -h ${PGSERVER} -U postgres -d postgres \  
-c 'TRUNCATE tb_copy RESTART IDENTITY;'
```

Criando o arquivo com os dados:

```
$ cat << EOF > /tmp/remoto.csv  
São Paulo,SP  
Rio de Janeiro,RJ  
Belo Horizonte,MG  
Cidade Fantasma,  
EOF
```

Enviando o conteúdo do arquivo local para o servidor remoto:

```
$ cat /tmp/remoto.csv | \  
psql -h ${PGSERVER} -U postgres -d postgres \  
-c 'COPY tb_copy (cidade, uf) FROM STDIN CSV;'
```

Visualizando os dados da tabela no servidor remoto:

```
$ psql -h ${PGSERVER} -U postgres -d postgres -c 'TABLE tb_copy;'
```

id	cidade	uf
1	São Paulo	SP
2	Rio de Janeiro	RJ
3	Belo Horizonte	MG
4	Cidade Fantasma	

Caminho reverso; do servidor remoto para um arquivo local:

```
$ psql -h ${PGSERVER} -U postgres -d postgres \  
-c 'COPY tb_copy TO STDOUT CSV;' > /tmp/backup.csv
```

Visualizando os dados do arquivo:

```
$ cat /tmp/backup.csv
```

```
1,São Paulo,SP  
2,Rio de Janeiro,RJ  
3,Belo Horizonte,MG  
4,Cidade Fantasma,
```


24 Range Types - Tipos de Intervalos

- Sobre Tipos de Intervalos

24.1 Sobre Tipos de Intervalos

Range Types são tipos de dados que representam uma faixa de valores de algum tipo de elemento (chamado de subtipo de faixa).

Por exemplo, faixas de `timestamp` que devem ser usadas para representar as faixas de tempo que uma sala está reservada.

Nesse caso o tipo de dados é `tsrange` (abreviação para "*timestamp range*"), e `timestamp` é o subtipo.

O subtipo deve ter uma ordem total para que seja bem definido se os elementos estão dentro, antes, ou depois de faixa de valores.

Range types são úteis porque representam muitos valores de elementos em uma única faixa de valores, e porque conceitos como sobreposição de intervalos podem ser expressos claramente.

O uso de tempo e intervalo de dados para propósitos de agendamentos é o mais claro exemplo; mas faixas de preços, intervalos de medidas de um instrumento, e assim por diante podem ser úteis.

24.1.1 Simbologia de Limites de Intervalos

Para simbolizarmos que limite temos:

- `()` → Parênteses: para simbolizar respectivamente limites inicial e final do tipo aberto;
- `[]` → Colchetes: representam respectivamente limites inicial e final do tipo fechado.

24.1.2 Built-in Range Types

O PostgreSQL nativamente vem com os seguintes range types:

- **int4range**: Inteiro de 4 bytes (`int4`, `int`, `integer`);
- **int8range**: Inteiro de 8 bytes (`int8`, `bigint`);
- **numrange**: Ponto flutuante (`numeric`);
- **tsrange**: timestamp sem time zone;
- **tstzrange**: timestamp com time zone;
- **daterange**: Data (`date`)

Em adição, você pode definir seus próprios range types; veja `CREATE TYPE` para mais informações.

Intervalo fechado de 2 a 9 (int4):	Representação Matemática:
<pre>> SELECT '[2, 9]':int4range; int4range ----- [2,10)</pre>	$\{x \in \mathbb{Z} \mid 2 \leq x \leq 9\}$

Intervalo fechado de 2 a 9 (numeric):	Representação Matemática:
<pre>> SELECT '[2, 9]':::numrange;</pre> <pre>numrange</pre> <pre>-----</pre> <pre>[2,9]</pre>	$\{x \in \mathbb{R} \mid 2 \leq x \leq 9\}$

Intervalo aberto em 2 e fechado em 9 (int4):	Representação Matemática:
<pre>> SELECT '(2, 9]':::int4range;</pre> <pre>int4range</pre> <pre>-----</pre> <pre>[3,10)</pre>	$\{x \in \mathbb{Z} \mid 2 < x \leq 9\}$

Intervalo aberto em 2 e fechado em 9 (numeric):	Representação Matemática:
<pre>> SELECT '(2, 9]':::numrange;</pre> <pre>numrange</pre> <pre>-----</pre> <pre>(2,9]</pre>	$\{x \in \mathbb{R} \mid 2 \leq x \leq 9\}$

Intervalo fechado em 2 e aberto em 9 (int4):	Representação Matemática:
<pre>> SELECT '[2, 9)':::int4range;</pre> <pre>int4range</pre> <pre>-----</pre> <pre>[2,9)</pre>	$\{x \in \mathbb{Z} \mid 2 \leq x < 9\}$

Intervalo fechado em 2 e aberto em 9 (numeric):	Representação Matemática:
<pre>> SELECT '[2, 9)':::numrange;</pre> <pre>numrange</pre> <pre>-----</pre> <pre>[2,9)</pre>	$\{x \in \mathbb{R} \mid 2 \leq x < 9\}$

Intervalos abertos em 2 e 9 (int4):	Representação Matemática:
<pre>> SELECT '(2, 9)':::int4range;</pre> <pre>int4range</pre> <pre>-----</pre> <pre>[3,9)</pre>	$\{x \in \mathbb{Z} \mid 2 < x < 9\}$

Intervalos abertos em 2 e 9 (numeric):	Representação Matemática:
<pre>> SELECT '(2, 9)::numrange;</pre> <pre> numrange ----- (2,9) </pre>	$\{x \in \mathbb{Z} \mid 2 < x < 9\}$

24.1.3 Extração de Limites Superior e Inferior

Extrai o limite superior (função upper):

```
> SELECT upper(int8range(15, 25));
```

```

upper
-----
    25

```

Extrai o limite inferior (função lower):

```
> SELECT lower(int8range(15, 25));
```

```

lower
-----
    15

```

24.1.4 Contenção: O Operador Contém @>

No intervalo de 10 a 20 contém 3 (função int4range)?

```
> SELECT int4range(10, 20) @> 3;
```

```

?column?
-----
f

```

24.1.5 Contenção: O Operador Contido <@

O valor 10 está contido entre 10 e 20 (função int4range)?

```
> SELECT 10 <@ int4range(10, 20);
```

```
?column?  
-----  
t
```

24.1.6 Overlaps - Sobreposições

Verifica se há sobreposição entre o primeiro e o segundo intervalo (função numrange):

```
> SELECT numrange(11.1, 20.2) && numrange(20.0, 30.0);
```

```
?column?  
-----  
t
```

Verifica se há sobreposição entre o primeiro e o segundo intervalo (função numrange):

```
> SELECT numrange(11.1, 20.0) && numrange(20.0, 30.0);
```

```
?column?  
-----  
f
```

24.1.7 Intersecção

Visualizar o intervalo de intersecção entre os dois intervalos de inteiros (função int4range):

```
> SELECT int4range(10, 20) * int4range(15, 25);
```

```
?column?  
-----  
[15,20)
```

Visualizar o intervalo de intersecção entre os dois intervalos de datas (função daterange):

```
> SELECT daterange('2014-09-01', '2014-09-14', '[]') *  
       daterange('2014-09-10', '2014-09-20', '[]');  
  
      ?column?  
-----  
[2014-09-10,2014-09-15)
```

24.1.8 Intervalo Vazio

Limite inferior e superior iguais a 7 aberto no início e fechado no final (função int4range):

```
> SELECT int4range(7, 7, '()');  
  
      int4range  
-----  
empty
```

Limite inferior e superior iguais a 7 aberto no início e fechado no final:

```
> SELECT '(7, 7]':int4range;  
  
      int4range  
-----  
empty
```

A faixa de 1 aberto a 5 aberto o limite é vazia? (função isempty):

```
> SELECT isempty(numrange(1, 5));  
  
      isempty  
-----  
f
```

Uma faixa que tem ambos os seus limites abertos e iguais a 5 é vazia? (função isempty):

```
> SELECT isempty(numrange(5, 5));  
  
      isempty  
-----  
t
```

24.1.9 Sem Limites (Mínimo, Máximo e Ambos)

Sem limite mínimo e limite máximo fechado em 7 (função int4range):

```
> SELECT int4range(null, 7, '[]');
```

```
int4range
-----
(,8)
```

Sem limite mínimo e limite máximo fechado em 7:

```
> SELECT '(', 7] '::int4range;
```

```
int4range
-----
(,8)
```

Sem limite máximo e limite mínimo fechado em 7:

```
> SELECT '[7,)'::int4range;
```

```
int4range
-----
[7,)
```

Sem limite máximo e limite mínimo fechado em 7 (função int4range):

```
> SELECT int4range(7, null, '[]');
```

```
int4range
-----
[7,)
```

Sem limites (função int4range):

```
> SELECT int4range(null, null, '()');
```

```
int4range
-----
(,)
```

```
> SELECT ' (, ) '::int4range;
```

24.1.10 Aplicação de Conceitos

```
> CREATE TABLE tb_reserva(
    sala int PRIMARY KEY,
    duracao tsrange);
```

```
> INSERT INTO tb_reserva VALUES
(1, '[2014-11-01 14:30, 2014-11-01 18:30)'),
(2, '[2014-11-02 11:00, 2014-11-02 15:00)'),
(3, '[2014-11-03 11:00, 2014-11-03 15:00)'),
(4, '[2014-11-04 17:00, 2014-11-04 19:00)'),
(5, tsrange(
    to_char(
        now(),
        'YYYY-MM-DD HH:mm'
    )::timestamp without time zone,
    to_char(
        now() + '1 week'::interval,
        'YYYY-MM-DD HH:mm'
    )::timestamp without time zone,
    '[]'
))
);
```

```
> TABLE tb reserva;
```

272

Verificando se há alguma sala cuja data e hora esteja contida em alguma duração de reserva:

```
> SELECT * FROM tb_reserva WHERE '2014-11-02 12:33'::timestamp <@ duracao;
```

sala	duracao
2	["2014-11-02 11:00:00", "2014-11-02 15:00:00")

Verificando se há alguma sala cuja duração contém a data e hora informada:

```
> SELECT * FROM tb_reserva WHERE duracao @> '2014-11-03 14:21'::timestamp;
```

sala	duracao
3	["2014-11-03 11:00:00", "2014-11-03 15:00:00")

25 Tipos de Dados Criados por Usuários

- Sobre Tipos de Dados Criados por Usuários
- Tipos Compostos (Composite Types)
- Tipos Enumerados (Enumerated Types)
- Tipos por Faixa (Range Types)
- Tipos Base (Base Types)

25.1 Sobre Tipos de Dados Criados por Usuários

O comando `CREATE TYPE` registra um novo tipo de dado para a base de dados atual.

O usuário que o criou passa a ser seu dono.

Há cinco formas do comando `CREATE TYPE`, que são os tipos de dados compostos, enumerados, de faixa, de base ou concha (*shell type*).

Sintaxe:

Command: `CREATE TYPE`

Description: define a new data type

Syntax:

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ]
    [, ... ] ] )
```

```
CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )
```

```
CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)
```

```
CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)
```

```
CREATE TYPE name
```

25.1.1 Tipo Concha (Shell Type)

Um tipo shell é apenas um espaço reservado para um tipo para ser definido mais tarde.

Um tipo concha (*shell type*) é criado ao dar o comando `CREATE TYPE` sem parâmetros (exceto pelo nome do tipo).

Tipos concha são necessários para referências futuras quando se cria tipos de faixa e tipos base.

25.2 Tipos Compostos (Composite Types)

É especificado por uma lista de atributos e seus tipos de dados.

Collation de atributos podem ser especificados também, se o tipo de dados permitir.

Um tipo composto é essencialmente o mesmo que um tipo de dados de linha de uma tabela, mas usando `CREATE TYPE` evita a necessidade de criar uma tabela atual quando tudo o que se quer é definir um tipo.

Um tipo de dados composto autônomo é útil por exemplo, como argumento ou tipo de retorno de uma função. Para se poder criar um tipo composto, o usuário deve ter o privilégio `USAGE` em todos os tipos de atributos.

Criação de um tipo composto:

```
> CREATE TYPE tp_uf_cidade AS (  
    uf char(2),  
    cidade varchar(50));
```

Criação da tabela que utilizará o tipo criado:

```
> CREATE TEMP TABLE tb_foo(  
    id serial PRIMARY KEY,  
    localidade tp_uf_cidade);
```

Populando a tabela:

```
> INSERT INTO tb_foo (localidade) VALUES  
    (('SP', 'São Paulo')),  
    (row('MG', 'Belo Horizonte')),  
    (('RO', 'Porto Velho')),  
    (('DF', 'Brasília')),  
    (row('SC', 'Florianópolis')),  
    (row('RJ', 'Rio de Janeiro')),  
    (('SP', 'Santo André'));
```

É opcional o uso de `row` para designar que é um tipo de linha (*row type*).

Consulta todos os registros da tabela:

```
> TABLE tb_foo;
```

id	localidade
1	(SP,"São Paulo")
2	(MG,"Belo Horizonte")
3	(RO,"Porto Velho")
4	(DF,"Brasília")
5	(SC,"Florianópolis")
6	(RJ,"Rio de Janeiro")
7	(SP,"Santo André")

Consulta especificando atributos do tipo enumerado para exibição e como critério de busca:

```
> SELECT id, (localidade).cidade FROM tb_foo WHERE (localidade).uf = 'SP';
```

id	cidade
1	São Paulo
7	Santo André

25.3 Tipos Enumerados (Enumerated Types)

Criação de um Tipo como Enumeração:

```
> CREATE TYPE tp_regiao AS ENUM ('Norte', 'Sul', 'Leste', 'Oeste');
```

Criação de uma Tabela para Teste:

```
> CREATE TEMP TABLE tb_base_militar(  
    id serial PRIMARY KEY,  
    num_soldados INT2,  
    regiao tp_regiao);
```

Inserção dos Dados:

```
> INSERT INTO tb_base_militar (num_soldados, regiao) VALUES  
    (35, 'Sul'), (21, 'Norte'), (15, 'Leste'), (22, 'Sul'), (33, 'Oeste'),  
    (19, 'Norte'), (19, 'Sul');
```

Aviso:

Tipos enumerados são *case sensitive*.

25.4 Tipos por Faixa (Range Types)

O subtipo de faixa pode ser qualquer tipo com um operador de classe b-tree associado (para determinar o ordenamento de valores para o tipo de faixa).

Criação do tipo de dado de faixa com inteiros de 2 bytes:

```
> CREATE TYPE tp_int2_range AS RANGE (SUBTYPE = int2);
```

Comparativo

Quantos bytes eu tenho em um intervalo fechado abaixo utilizando inteiro de 4 bytes?:

```
> SELECT pg_column_size('[2, 9]::int4range');

pg_column_size
-----
17
```

Quantos bytes eu tenho em um intervalo fechado abaixo utilizando inteiro de 2 bytes?:

```
> SELECT pg_column_size('[2, 9]::tp_int2_range');

pg_column_size
-----
13
```

Após esse comparativo podemos concluir o quanto nos vale a pena às vezes criar um tipo de dados personalizado não só para atender nossas necessidades, mas também para podermos ter um tipo de dado mais eficiente do que o que é oferecido por padrão.

Criação de tipo de dado de faixa de IPs, muito útil para administradores de redes:

```
> CREATE TYPE tp_ip_range AS RANGE (SUBTYPE=inet);
```

A faixa de IPs contém 192.168.0.17?:

```
> SELECT tp_ip_range('192.168.0.1', '192.168.0.10', '[') @> '192.168.0.17'::inet;

?column?
-----
f
```


25.5 Tipos Base (Base Types)

Tipo de base (*base type*) ou tipo escalar (*scalar type*) para ser criado é preciso ter o atributo `SUPERUSER`. Tal restrição é feita devido ao fato de definições errôneas podem confundir ou mesmo levar o servidor a uma falha (crash).

Os parâmetros de criação podem aparecer em qualquer ordem e a maioria é opcional.

É possível registrar duas ou mais funções usando `CREATE FUNCTION` antes de definir o tipo.

As funções de suporte `input_function` e `output_function` são obrigatórias, enquanto que as outras são opcionais e tem que ser codificadas em C ou outra linguagem de baixo nível.

Maiores informações no seguinte link:

<http://www.postgresql.org/docs/current/static/sql-createtype.html>

Há um exemplo no código-fonte do PostgreSQL, após descompactar o arquivo do mesmo, na subpasta `src/tutorial`, o qual ensina como criar um tipo escalar para números complexos, muito utilizado em engenharia elétrica.

Nos exercícios a seguir será visto na prática como se fazer isso, com algumas alterações em relação ao original. Vale lembrar também que esses exercícios não é o tutorial completo, mas o suficiente para entender como funciona e quem por ventura tiver os devidos conhecimentos pode se aprofundar no assunto e estender os tipos do PostgreSQL dessa forma.

26 DOMAIN (Domínio)

- Sobre Domínio
- DOMAIN vs TYPE

26.1 Sobre Domínio

Domínio é um tipo de dado personalizado em que se pode definir como os dados serão inseridos de acordo com restrições definidas opcionalmente.

Sintaxe:

```
CREATE DOMAIN name [ AS ] data_type  
    [ COLLATE collation ]  
    [ DEFAULT expression ]  
    [ constraint [ ... ] ]
```

Onde constraint é:

```
[ CONSTRAINT nome_restrição ]  
{ NOT NULL | NULL | CHECK (expressão) }
```

26.2 DOMAIN vs TYPE

Criar um tipo personalizado ou um domínio tem algumas diferenças.

Segue o quadro comparativo para explicar melhor a diferença entre ambos para ver qual é o melhor à sua necessidade:

	CREATE DOMAIN	CREATE TYPE
Scalar (Single Field) Type	Sim	Sim
Complex (Composite) Type	Não	Sim
Enumeration Type	Sim	Sim
DEFAULT Value	Sim	Não
NULL and NOT NULL Constraint	Sim	Não
CHECK Constraint	Sim	Não

Como exemplo vamos criar um domínio para validação de CEP.

Esse domínio só aceita entradas de inteiros de sete a oito dígitos.

Por que inteiro? Porque é mais fácil para o banco fazer buscas e indexar campos inteiros.

Mas e se o CEP começar com zero? A inserção será feita com sete dígitos e na exibição será utilizada a função `to_char` que retornará no formato `#####-###`.

Criação de um domínio, para validar CEPs que aceita inteiros com sete ou oito dígitos:

```
> CREATE DOMAIN dom_cep AS integer
    CONSTRAINT chk_cep
    CHECK (length(VALUE::text) = 7
           OR length(VALUE::text) = 8);
```

Criação de uma tabela que usará o domínio criado como tipo de dado para uma coluna:

```
> CREATE TEMP TABLE tb_endereco_tmp (
    id serial PRIMARY KEY,
    cep dom_cep,
    logradouro text,
    numero smallint,
    cidade varchar(50),
    uf char(2));
```

Inserções na Tabela com o domínio criado:

```
> INSERT INTO tb_endereco_tmp (cep, logradouro, numero, cidade, uf) VALUES
(1001000, 'Pça. da Sé', null, 'São Paulo', 'SP'),
(30130003, 'Av. Afonso Pena', 1212, 'Belo Horizonte', 'MG');
```

Selecionando os dados:

```
> SELECT
  to_char(cep, '00000-000') "CEP",
  logradouro "Logradouro",
  numero "Número",
  cidade "Cidade",
  uf "Estado"
FROM tb_endereco_tmp;
```

CEP	Logradouro	Número	Cidade	Estado
01001-000	Pça. da Sé		São Paulo	SP
30130-003	Av. Afonso Pena	1212	Belo Horizonte	MG

27 SCHEMA (Esquema)

- Sobre Esquemas
- Caminho de Busca de Esquema - Schema Search Path

27.1 Sobre Esquemas

É uma forma de organizar objetos dentro de um banco de dados, permitindo um mesmo tipo de objeto seja criado mais de uma vez com nome igual, mas em esquemas diferentes.

Quando criamos objetos, os mesmos pertencem ao esquema público (`public`). Sendo assim quando fazemos uma consulta, não precisamos especificar o esquema.

Todo esquema criado é registrado no catálogo de sistema `pg_namespace`.

Criação de base de dados de teste:

```
> CREATE DATABASE db_schema;
```

Conectando na base de dados:

```
> \c db_schema
```

Criação de um schema:

```
> CREATE SCHEMA sc_foo;
```

Exibindo schemas (exceto os de catálogo):

```
> SELECT nspname FROM pg_namespace
      WHERE nspname !~ '^pg_' AND nspname != 'information_schema';

 nspname 
-----
 public
sc_foo
```

Exibindo schemas (exceto os de catálogo) via comando de atalho do psql:

```
> \dn

List of schemas
Name | Owner
-----+-----
 public | postgres
sc_foo | postgres
```

Criação de tabela de teste no schema public:

```
> SELECT generate_series(1, 10) campo INTO tb_foo;
```

Criação de tabela de teste no schema sc_foo:

```
> SELECT generate_series(1, 20, 2) campo INTO sc_foo.tb_foo;
```

Consulta na tabela do schema public:

```
> TABLE tb_foo LIMIT 5;
```

```
campo
-----
 1
 2
 3
 4
 5
```

Consulta na tabela do schema sc_foo:

```
> TABLE sc_foo.tb_foo LIMIT 5;
```

```
campo
-----
 1
 3
 5
 7
 9
```

Estrutura da tabela do schema public:

```
> \dt tb_foo
          List of relations
 Schema | Name  | Type  | Owner
-----+-----+-----+-----
 public | tb_foo | table | postgres
```

Estrutura da tabela do schema sc_foo:

```
> \dt sc_foo.tb_foo
          List of relations
 Schema | Name  | Type  | Owner
-----+-----+-----+-----
 sc_foo | tb_foo | table | postgres
```


Lista tabelas do schema public:

```
> \dt
```

```
           List of relations
 Schema | Name   | Type  | Owner
-----+-----+-----+-----
 public | tb_foo | table | postgres
```

Lista tabelas do schema sc_foo:

```
> \dt sc_foo.*
```

```
           List of relations
 Schema | Name   | Type  | Owner
-----+-----+-----+-----
 sc_foo | tb_foo | table | postgres
```

Criação de tabela no schema sc_foo:

```
> SELECT generate_series(1, 10) campo INTO sc_foo.tb_bar;
```

Lista tabelas do schema sc_foo:

```
> \dt sc_foo.*
```

```
           List of relations
 Schema | Name   | Type  | Owner
-----+-----+-----+-----
 sc_foo | tb_bar | table | postgres
 sc_foo | tb_foo | table | postgres
```

Muda o schema da tabela para public:

```
> ALTER TABLE sc_foo.tb_bar SET SCHEMA public;
```

Lista tabelas do schema public:

```
> \dt
```

```
           List of relations
 Schema | Name   | Type  | Owner
-----+-----+-----+-----
 public | tb_bar | table | postgres
 public | tb_foo | table | postgres
```

27.2 Caminho de Busca de Esquema - Schema Search Path

Se quisermos acionar algum objeto que não esteja no esquema padrão é necessário mencionar o outro esquema explicitamente. Há como modificarmos esse comportamento de forma que o caminho de busca padrão seja o que desejarmos.

A configuração do caminho de busca de esquema é denominada `search_path`.

Exibe `search_path`:

```
> SHOW search_path;

 search_path
-----
"$user",public
```

Os elementos do caminho são separados por vírgula. O primeiro especifica um esquema como o mesmo nome do usuário corrente para ser procurado. Se não existir tal esquema, a entrada é ignorada. A posição determina a ordem de busca de esquemas.

Alterando o `search_path`:

```
> SET search_path = "$user",sc_foo,public;
```

Agora todos os objetos que estão em `sc_foo` também estarão incluídos nas buscas da sessão corrente.

Obs.:

A configuração `search_path` só é válida para a sessão atual.

Criação de uma tabela:

```
> CREATE TABLE tb_foobar();
```

Listando tabelas:

```
> \dt

          List of relations
 Schema | Name      | Type  | Owner
-----+-----+-----+-----
 public | tb_bar    | table | postgres
 sc_foo | tb_foo    | table | postgres
 sc_foo | tb_foobar | table | postgres
```

28 MVCC

- Sobre MVCC
- Transação
- Fenômenos Indesejados em Transações
- Níveis de Isolamento
- O Conceito de ACID
- Travas
- Savepoint – Ponto de Salvamento
- SELECT ... FOR UPDATE

28.1 Sobre MVCC

Diferente de outros SGBDs tradicionais, que usam bloqueios para controlar a simultaneidade, o PostgreSQL mantém a consistência dos dados utilizando o modelo multiversão (MVCC: *Multi Version Concurrency Control*). Significa que, ao consultar o banco de dados, cada transação enxerga um instantâneo (*snapshot*) dos dados (uma versão da base) como esses eram antes, sem considerar o atual estado dos dados subjacentes.

Esse modelo evita que a transação enxergue dados inconsistentes, o que poderia ser originado por atualizações feitas por transações simultâneas nos mesmos registros, fornecendo um isolamento da transação para cada sessão.

A principal vantagem de utilizar o modelo MVCC em bloqueios é pelo fato de os bloqueios obtidos para consultar os dados (leitura) não entram em conflito com os bloqueios de escrita, então, a leitura nunca bloqueia a escrita e a escrita nunca bloqueia a leitura.

28.2 Transação

Tudo ou nada! Uma frase que define bem o que é uma transação.

Transação é uma forma de se executar uma sequência de comandos indivisivelmente para fins de alterações, inserções ou remoções de registros.

Uma transação começa com `BEGIN` e termina ou com `COMMIT` (efetiva mudanças) ou com `ROLLBACK` (não efetiva mudanças, voltando ao estado anterior à transação).

Esquema geral de uma transação:

```
BEGIN;  
comando_1;  
comando2_;  
.  
.  
.  
comando_N;  
(COMMIT | ROLLBACK)
```

Sintaxe de `BEGIN`:

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

Onde `transaction_mode` pode ser:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED  
| READ UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

28.2.1 Utilidade de uma transação

Por padrão, o PostgreSQL é *auto commit*, que significa efetivar as alterações nos bancos de dados de forma automática, pois cada comando é executado e logo após a alteração é feita.

No entanto, há casos que exige-se uma maior segurança, como por exemplo, contas bancárias. Imagine uma transferência de uma conta “A” para uma conta “B” de um valor qualquer.

No banco de dados, a conta “A” terá de seu saldo subtraído o valor da transferência e na conta “B” adicionado. Ou seja, na primeira operação retira-se de uma e na segunda acrescenta-se a outra.

E se logo após o valor ser subtraído de “A” houver algum problema? A conta “B” ficaria sem o valor acrescentado e “A” ficaria com um saldo menor sem a transferência ter sido realmente feita... Seria um grande problema para a instituição financeira!

Se no caso citado tivesse sido feito de forma a usar transação não haveria tal transtorno, pois numa transação ou todos comandos são devidamente efetivados ou não haverá alteração alguma. Alterações feitas durante uma transação só podem ser vistas por quem está fazendo, os outros usuários só poderão ver depois da transação ser efetivada. Um banco de dados relacional deve ter um mecanismo eficaz para armazenar informações que estejam de acordo com o conceito ACID.

28.3 Fenômenos Indesejados em Transações

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações simultâneas.

Dirty Read (Leitura Suja): Também conhecida como dependência não efetivada, ocorre quando uma transação é permitida para ler dados de uma linha que foi modificada por outra transação concorrente, mesmo que essa transação concorrente não tenha sido efetivada. Esse fenômeno não acontece no PostgreSQL devido ao fato de não implementar efetivamente o nível de isolamento `READ UNCOMMITTED`;

Nonrepeatable Read (Leitura que não se repete): Ocorre quando em uma transação em curso uma linha buscada mais de uma vez poder retornar valores diferentes. Ou seja, a leitura pode ter diferença dentro da transação, ela pode não se repetir. Lê linhas dentro da transação atual que foram efetivadas com atualizações (`UPDATE`) por outra transação concorrente.;

Phantom Read (Leitura Fantasma): A transação executa novamente uma consulta e descobre que os registros mudaram devido a outra transação ter sido efetivada. Lê linhas dentro da transação atual que foram inseridas (`INSERT`) e / ou apagadas (`DELETE`) por outra transação concorrente.

28.4 Níveis de Isolamento

No PostgreSQL é possível solicitar qualquer um dos quatro níveis de isolamento de transação padrão. Porém, internamente eles são apenas 3 (três) níveis de isolamento distintos, que correspondem aos níveis: *Read Committed*, *Repeatable Read* e *Serializable*.

Quando é selecionado o nível *Read Uncommitted* o que se tem realmente é o *Read Committed*, e leituras fantasmas não são possíveis na implementação de *Repeatable Read* no PostgreSQL, então o nível de isolamento atual deve ser mais estrito do que foi selecionado. Isso é permitido pelo padrão SQL, pois os quatro níveis de isolamento apenas definem que fenômenos não podem acontecer, não definem que fenômenos devem acontecer.

O PostgreSQL só oferece três níveis de isolamento porque é a única maneira sensata de mapear os níveis de isolamento padrão para a arquitetura controle de concorrência multiversão.

Para configurar o nível de isolamento de uma transação, use o comando `SET TRANSACTION`.

READ COMMITTED: É o nível de isolamento padrão do PostgreSQL. Permite que sejam feitas leituras de linhas alteradas, inseridas ou apagadas por outras transações concorrentes.

READ UNCOMMITTED: Não implementado no PostgreSQL, vide `READ COMMITTED`.

REPEATABLE READ: Exerga apenas dados efetivados antes da transação atual começar. Nunca verá ou dados não efetivados ou mudanças feitas (por transações concorrentes) durante a execução da transação atual.

SERIALIZABLE: Fornece o nível de isolamento mais estrito, emula uma execução serial de transação para todas as transações efetivadas, como se fossem executadas uma após outra serialmente, em vez de concorrentemente. Porém, como o nível `REPEATABLE READ`, aplicações que usam este nível devem estar preparadas para tentar novamente as transações devido a falhas de serialização. Na verdade, `SERIALIZABLE` funciona da mesma forma que `REPEATABLE READ`, exceto pelo fato de monitorar condições que possam tornar a execução de um conjunto de transações serializáveis concorrentes se comportar de uma maneira inconsistente com tudo serial possível (um de cada vez) execuções dessas transações. Esse monitoramento não introduz qualquer bloqueio além do que se tem em `REPEATABLE READ`, mas há um certo overhead para o monitoramento, e detecção de condições que podem causar uma falha de serialização.

Nível de Isolamento	Fenômeno Indesejado		
	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	<u>Possível</u>	<u>Possível</u>	<u>Possível</u>
Read committed	Impossível	<u>Possível</u>	<u>Possível</u>
Repeatable read	Impossível	Impossível	<u>Possível</u>
Serializable	Impossível	Impossível	Impossível

28.5 O Conceito de ACID

Atomicidade: Relembrando as aulas de química, a palavra “átomo”, que vem do grego, significa indivisível. Quando se executa uma transação não há parcialidade. Ou todas alterações são efetivadas ou nenhuma.

Consistência: Certifica que o banco de dados permanecerá em um estado consistente antes do início da transação e depois que a transação for finalizada (bem-sucedida ou não).

Isolamento: Cada transação desconhece outras transações concorrentes no sistema.

Durabilidade: Após o término bem-sucedido de uma transação no banco de dados, as mudanças persistem.

Verificando qual é o nível de isolamento atual:

```
> SHOW TRANSACTION ISOLATION LEVEL;
```

ou

```
> SHOW transaction_isolation;
```

```
transaction_isolation
-----
read committed
```

Definindo para a Sessão Aberta:

```
> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

ou

```
> SET transaction_isolation = 'read committed';
```

Criação da tabela de teste:

```
> CREATE TABLE tb_mvcc(
  id serial PRIMARY KEY,
  nome text,
  idade int2);
```

Populando a tabela:

```
> INSERT INTO tb_mvcc (nome, idade) VALUES ('Joe', 20), ('Jill', 25);
```

Verificando os dados da tabela:

```
> TABLE tb_mvcc;

 id | nome | idade 
----+-----+-----
  1 | Joe  |    20 
  2 | Jill |    25
```

Criação de função para fazer TRUNCATE na tabela, reiniciar a sequência da tabela e inserir os dados novamente:

```
> CREATE OR REPLACE FUNCTION fc_zera_tb_mvcc() RETURNS void AS
$$
BEGIN
    TRUNCATE tb_mvcc RESTART IDENTITY;
    INSERT INTO tb_mvcc (nome, idade) VALUES ('Joe', 20), ('Jill', 25);
END;
$$ LANGUAGE PLPGSQL;
```

Início de uma transação:

```
> BEGIN;
```

Verificando dados da tabela:

```
> SELECT nome FROM tb_mvcc;

 nome 
-----
 Joe  
 Jill
```

Apagando um registro e retornando o campo nome do registro apagado:

```
> DELETE FROM tb_mvcc WHERE id = 1 RETURNING nome;

 nome 
-----
 Joe
```

Apagando um registro:

```
> SELECT nome FROM tb_mvcc;

 nome 
-----
 Jill
```

Desfazendo todos statements dentro da transação:

```
> ROLLBACK;
```


Verificando a tabela:

```
> SELECT nome FROM tb_mvcc;
```

```
nome
-----
Joe
Jill
```

Com o comando `ROLLBACK`, o registro apagado foi desfeito.

Situação em que a transação tem seu nível de isolamento `READ COMMITTED`:

Sessão 1	Sessão 2
> BEGIN;	
> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	
> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 20	
	> UPDATE tb_mvcc SET idade = 21 WHERE id = 1;
> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 20	> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 21
	> COMMIT;
> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 21	
> COMMIT;	

Repare que no exemplo ocorreu o fenômeno **Nonrepeatable Read**, pois o último `SELECT` da mesma linha, dentro da transação, retornou um resultado diferente dos anteriores.

Utilizando a função que prepara a tabela para novos testes:

```
> SELECT fc_zera_tb_mvcc();
```

Situação em que a transação tem seu nível de isolamento `SERIALIZABLE`:

Sessão 1	Sessão 2
> <code>BEGIN;</code>	
> <code>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</code>	
> <code>SELECT idade FROM tb_mvcc WHERE id = 1;</code> idade ----- 20	
	> <code>UPDATE tb_mvcc SET idade = 21 WHERE id = 1;</code>
> <code>SELECT idade FROM tb_mvcc WHERE id = 1;</code> idade ----- 20	> <code>SELECT idade FROM tb_mvcc WHERE id = 1;</code> idade ----- 21
	> <code>COMMIT;</code>
> <code>SELECT idade FROM tb_mvcc WHERE id = 1;</code> idade ----- 20	
> <code>COMMIT;</code>	
> <code>SELECT idade FROM tb_mvcc WHERE id = 1;</code> idade ----- 21	

Utilizando a função que prepara a tabela para novos testes:

> `SELECT fc_zera_tb_mvcc();`

Situação em que a transação tem seu nível de isolamento REPEATABLE READ:

Sessão 1	Sessão 2
> BEGIN;	
> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
> SELECT * FROM tb_mvcc WHERE idade BETWEEN 10 AND 30; id nome idade ----+----- 1 Joe 20 2 Jill 25	
	> INSERT INTO tb_mvcc (nome, idade) VALUES ('Maria', 27);
> SELECT * FROM tb_mvcc WHERE idade BETWEEN 10 AND 30; id nome idade ----+----- 1 Joe 20 2 Jill 25	id nome idade ----+----- 1 Joe 20 2 Jill 25 3 Maria 27
	> COMMIT;
> SELECT * FROM tb_mvcc WHERE idade BETWEEN 10 AND 30; id nome idade ----+----- 1 Joe 20 2 Jill 25	
> COMMIT;	

Utilizando a função que prepara a tabela para novos testes:

> SELECT fc_zera_tb_mvcc();

28.6 Travas

Travas exclusivas, também conhecidas como travas de escrita, evitam que outras sessões modifiquem um registro ou uma tabela inteira.

Linhas modificadas por remoção (DELETE) ou atualização (UPDATE) acabam sendo bloqueadas de forma automática e exclusiva durante a transação. Isso evita que outras sessões possam mudar a linha até que a transação seja ou efetivada (COMMIT) ou desfeita (ROLLBACK).

Demonstração de trava (READ COMMITTED):

Sessão 1	Sessão 2
> BEGIN;	
> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	
> UPDATE tb_mvcc SET idade = 21 WHERE id = 1;	
	> UPDATE tb_mvcc SET idade = 22 WHERE id = 1;
Podemos ver aqui que a transação da sessão 2 está travada. E assim vai permanecer enquanto a sessão 1 não finaliza sua transação.	
> COMMIT;	
> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 21	
	> COMMIT;
> SELECT idade FROM tb_mvcc WHERE id = 1; idade ----- 22	

Utilizando a função que prepara a tabela para novos testes:

```
> SELECT fc_zera_tb_mvcc();
```

Demonstração de trava (SERIALIZABLE):

Sessão 1	Sessão 2
> BEGIN;	
> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	
> UPDATE tb_mvcc SET idade = 21 WHERE id = 1;	
	> UPDATE tb_mvcc SET idade = 22 WHERE id = 1;
Podemos ver aqui que a transação da sessão 2 está travada. E assim vai permanecer enquanto a sessão 1 não finaliza sua transação.	
> COMMIT;	
	<i>ERROR: could not serialize access due to concurrent update</i>
Quando foi dado um COMMIT na sessão 1, na sessão 2 foi dada uma mensagem de erro devido à atualização concorrente no nível de isolamento serializável.	
> SELECT idade FROM tb_mvcc WHERE id = 1;	
idade ----- 21	
	> COMMIT;
	<i>ROLLBACK</i>
Em vão a sessão 2 tenta efetivar sua transação. Logo após o comando para efetivação lê-se a mensagem <i>ROLLBACK</i> , indicando que a mesma não foi efetivada.	
> SELECT idade FROM tb_mvcc WHERE id = 1;	
idade ----- 21	

Utilizando a função que prepara a tabela para novos testes:

```
> SELECT fc_zera_tb_mvcc();
```

28.7 SAVEPOINT – Ponto de Salvamento

Define um novo ponto de salvamento na transação atual.

De uma maneira simplória, poderíamos dizer que é uma transação dentro de outra transação.

O ponto de salvamento é uma marca especial dentro da transação que permite desfazer todos comandos executados após a criação do `SAVEPOINT`, dessa forma voltando ao estado anterior da criação do ponto de salvamento.

Sintaxe:

```
SAVEPOINT nome_savepoint;
```

Para desfazer até o ponto de salvamento:

```
ROLLBACK TO SAVEPOINT
```

Desfaz até um ponto de salvamento nomeado, sendo que a palavra `SAVEPOINT` pode ser omitida:

```
ROLLBACK TO [SAVEPOINT] nome_savepoint;
```

Obs.:

Os pontos de salvamento somente devem ser estabelecidos dentro de um bloco de transação.

Podem haver vários pontos de salvamento definidos dentro de uma transação.

Início de uma Nova Transação:

```
> BEGIN;
```

Remoção de Registro:

```
> DELETE FROM tb_mvcc WHERE id = 1 RETURNING nome, idade;
```

```
nome | idade
-----+-----
Joe  |    20
```

Verificando:

```
> SELECT nome, idade FROM tb_mvcc;
```

nome	idade
Jill	25

Criação do Ponto de Salvamento:

```
> SAVEPOINT sp1;
```

Alteração de um Registro:

```
> UPDATE tb_mvcc SET idade = (idade + 1) WHERE id = 2;
```

Verificando:

```
> SELECT nome, idade FROM tb_mvcc WHERE id = 2;
```

Voltando ao Ponto de Salvamento "sp1":

```
> ROLLBACK TO sp1;
```

Verificando:

```
> SELECT nome, idade FROM tb_mvcc WHERE id = 2;
```

nome	idade
Jill	25

Efetivar as alterações antes de "sp1" que não tiveram ROLLBACK TO:

```
> COMMIT;
```

Verificando:

```
> SELECT nome, idade FROM tb_mvcc;
```

nome	idade
Jill	25

28.7.1 RELEASE SAVEPOINT

Destrói um ponto de salvamento definido anteriormente, mas mantém os efeitos dos comandos executados após a criação do SAVEPOINT.

Sintaxe:

```
RELEASE [ SAVEPOINT ] nome_savepoint
```

Utilizando a função que prepara a tabela para novos testes:

```
> SELECT fc_zera_tb_mvcc();
```

Início de Transação:

```
> BEGIN;
```

Apagando um registro:

```
> DELETE FROM tb_mvcc WHERE id = 1 RETURNING nome, idade;
```

nome	idade
Joe	20

Verificando:

```
> SELECT nome, idade FROM tb_mvcc;
```

nome	idade
Jill	25

Criação do ponto de salvamento:

```
> SAVEPOINT sp1;
```

Alteração de Registro:

```
> UPDATE tb_mvcc SET idade = (idade + 2) WHERE id = 2 RETURNING nome, idade;
```

nome	idade
Jill	27

Destruição de Ponto de Salvamento:

```
> RELEASE sp1;
```

Efetivação:

```
> COMMIT;
```

Mesmo com o ponto de salvamento destruído, as alterações podem ser efetivadas.

Verificando:

```
> SELECT * FROM tb_mvcc;
```

<i>id</i>	<i>nome</i>	<i>idade</i>
2	Jill	27

Utilizando a função que prepara a tabela para novos testes:

```
> SELECT fc_zera_tb_mvcc();
```

28.8 SELECT ... FOR UPDATE

Dentro de uma transação, através dos registros selecionados bloqueia os mesmos para que outras sessões não os apaguem ou alterem enquanto a transação não for finalizada.

Sessão 1	Sessão 2
> BEGIN;	
> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
> SELECT * FROM tb_mvcc WHERE id = 1 FOR UPDATE;	
<pre> id nome idade ----+-----+----- 1 Joe 20 </pre>	
	Tentativa de Remoção de Registro: > DELETE FROM tb_mvcc WHERE id = 1;
O registro está bloqueado!	
> COMMIT;	
Após a efetivação dos dados pela Sessão 2 pôde finalmente fazer as alterações que desejava.	
	> COMMIT;

29 PREPARE

- Sobre PREPARE

29.1 Sobre PREPARE

Prepara um comando para execução: cria um comando preparado (*prepared statement*).

Um *prepared statement* é um objeto do lado do servidor que pode ser usado para otimizar performance. Quando `PREPARE statement` é executado, o comando (*statement*) é analisado, são feitas coletas de estatísticas (`ANALYZE`) e reescrito.

Quando é dado um comando `EXECUTE`, o *prepared statement* é planejado e executado. Essa divisão de trabalho evita repetitivos trabalhos de coleta de estatística, enquanto permite ao plano de execução de depender de parâmetros específicos que podem ser fornecidos.

Um comando preparado é um objeto de sessão.

Sintaxe:

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

Criação da tabela de testes:

```
> CREATE TABLE tb_banda(  
    id serial PRIMARY KEY,  
    nome text,  
    origem text);
```

Inserir dados na tabela:

```
> INSERT INTO tb_banda (nome, origem) VALUES  
    ('Angra', 'Brasil'),  
    ('Shaman', 'Brasil'),  
    ('Sepultura', 'Brasil'),  
    ('Helloween', 'Alemanha'),  
    ('Blind Guardian', 'Alemanha'),  
    ('Sanctuary', 'EUA'),  
    ('Black Sabbath', 'Inglaterra'),  
    ('Manowar', 'EUA'),  
    ('Kamelot', 'EUA'),  
    ('Epica', 'Holanda'),  
    ('Gamma Ray', 'Alemanha'),  
    ('Viper', 'Brasil'),  
    ('Paradise Lost', 'Inglaterra'),  
    ('Wizards', 'Brasil'),  
    ('Tierra Santa', 'Espanha'),  
    ('Saxon', 'Inglaterra'),  
    ('Testament', 'EUA'),  
    ('Dr. Sin', 'Brasil'),  
    ('Eterna', 'Brasil'),  
    ('Accept', 'Alemanha');
```

Criando um prepared statement de forma a selecionar somente bandas brasileiras:

```
> PREPARE q_banda_br AS SELECT * FROM tb_banda WHERE origem = 'Brasil';
```

Executando o prepared statement:

```
> EXECUTE q_banda_br;
```

id	nome	origem
1	Angra	Brasil
2	Shaman	Brasil
3	Sepultura	Brasil
12	Viper	Brasil
14	Wizards	Brasil
18	Dr. Sin	Brasil
19	Eterna	Brasil

Criando um prepared statement com um parâmetro inteiro:

```
> PREPARE q_banda_id (int) AS SELECT nome FROM tb_banda WHERE id = $1;
```

Executando o prepared statement:

```
> EXECUTE q_banda_id (7);
```

nome
Black Sabbath

Criando um prepared statement com dois parâmetros:

```
> PREPARE q_banda_id_origem (int, text) AS
  SELECT nome FROM tb_banda
  WHERE origem = $2 AND id > $1;
```

Executando o prepared statement:

```
> EXECUTE q_banda_id_origem (0, 'Brasil');
```

nome
Angra
Shaman
Sepultura
Viper
Wizards
Dr. Sin
Eterna

Visualizando todos os prepared statements da sessão:

```
> SELECT * FROM pg_prepared_statements;
```

```
-[ RECORD 1 ]---+-----  
name          | q_banda_id_origem  
statement      | PREPARE q_banda_id_origem (int, text) AS  
                | SELECT nome FROM tb_banda  
                | WHERE origem = $2 AND id > $1;  
prepare_time   | 2015-12-10 16:29:42.29059-02  
parameter_types | {integer,text}  
from_sql       | t  
-+-----  
-[ RECORD 2 ]---+-----  
name          | q_banda_id  
statement      | PREPARE q_banda_id (int) AS SELECT nome FROM tb_banda WHERE id = $1;  
prepare_time   | 2015-12-10 16:29:08.806482-02  
parameter_types | {integer}  
from_sql       | t  
-+-----  
-[ RECORD 3 ]---+-----  
name          | q_banda_br  
statement      | PREPARE q_banda_br AS SELECT * FROM tb_banda WHERE origem = 'Brasil';  
prepare_time   | 2015-12-10 16:27:50.151873-02  
parameter_types | {}  
from_sql       | t
```

Desalocando da memória (apagando) os prepared statements criados:

```
> DEALLOCATE q_banda_id;
```

```
> DEALLOCATE q_banda_id_origem;
```

```
> DEALLOCATE q_banda_br;
```

30 **PREPARE TRANSACTION**

- Sobre PREPARE TRANSACTION

30.1 Sobre PREPARE TRANSACTION

`PREPARE TRANSACTION` prepara a transação atual para efetivação em duas fases (*two-phase commit*).

Após esse comando a transação não é mais associada com a sessão atual; seu estado é armazenado em disco, e há uma grande probabilidade que possa haver uma efetivação com sucesso, mesmo se o banco de dados sofrer uma falha antes do `COMMIT` ser requisitado.

Uma vez preparada, a transação pode depois ser efetivada ou desfeita com `COMMIT PREPARED` ou `ROLLBACK PREPARED`, respectivamente. Esses comandos podem ser dados em qualquer sessão e não somente onde a transação original foi executada.

Do ponto de vista da sessão que preparou a transação, `PREPARE TRANSACTION` não é diferente de um comando `ROLLBACK`: após executá-lo, não há transação ativa atual, e os efeitos de `PREPARE TRANSACTION` não estará mais visível, pois os efeitos se tornarão visíveis novamente se a transação for efetivada.

Se o comando `PREPARE TRANSACTION` falhar por qualquer motivo, ele se tornará um `ROLLBACK` e a transação atual é cancelada.

Sintaxe:

```
PREPARE TRANSACTION transaction_id;
```

Aviso:

Para poder utilizar o recurso `PREPARE TRANSACTION` é preciso ter o valor do parâmetro de configuração `max_prepared_transactions` maior do que zero.

Sessão 1	Sessão 2
> BEGIN ;	
> PREPARE TRANSACTION 'foo';	
> INSERT INTO tb_banda (nome, origem) VALUES ('Riot', 'EUA');	
	> TABLE pg_prepared_xacts; -[RECORD 1]----- transaction 859 gid foo prepared 2015-09-04 16:49:53.272232-03 owner postgres database postgres
	> COMMIT PREPARED 'foo';

Início de transação:

```
> BEGIN;
```

Inserindo um novo valor na tabela:

```
> INSERT INTO tb_banda (nome, origem) VALUES ('Anthrax', 'EUA');
```

Preparando a transação:

```
> PREPARE TRANSACTION 'bar';
```

Saindo da sessão:

```
> \q
```

Matando o serviço simulando um crash:

```
$ kill -9 `pidof postgres`
```

Iniciando o serviço:

```
$ pg_ctl start
```

Cliente psql:

```
$ psql
```

Verificando o catálogo de transações preparadas:

```
> TABLE pg_prepared_xacts;
```

transaction	gid	prepared	owner	database
861	bar	2015-09-04 16:51:59.704561-03	postgres	postgres

Verificando a tabela, de forma a assegurar que o dado não tinha sido efetivamente inserido:

```
> SELECT * FROM tb_banda WHERE nome = 'Anthrax';
```

<i>id</i>	<i>nome</i>	<i>origem</i>
-----+	-----+	-----

Efetivando a transação preparada:

```
> COMMIT PREPARED 'bar';
```

Verificando se o dado foi efetivamente inserido:

```
> SELECT * FROM tb_banda WHERE nome = 'Anthrax';
```

<i>id</i>	<i>nome</i>	<i>origem</i>
-----+	-----+	-----
21	Anthrax	EUA

31 Arrays

- Sobre Arrays
- Arrays PostgreSQL
- Inserindo Valores de Array
- Consultas em Arrays
- Modificando Arrays
- Funções de Arrays
- Operadores de Arrays
- Alterando Tipos para Array

31.1 Sobre Arrays

Também conhecido como **vetor** é uma estrutura de dados que armazena uma coleção de valores, os quais são identificados por um índice.

Cada dimensão é delimitada por colchetes "[]".

Na declaração de um *array*, o número entre os colchetes indica a quantidade de elementos que esse *array* terá.

Quando um *array* tem mais de uma dimensão é chamado de **matriz**.

Para acessar um valor de um *array* ou matriz é pelo seu índice, que tradicionalmente inicia-se por zero, de maneira que se um *array* foi declarado com o tamanho 10, seus índices vão de zero a nove.

Declaração de um array com 3 (três) elementos:

v[3] = {27, 33, 98}	v[0] = 27 v[1] = 33 v[2] = 98
---------------------	-------------------------------------

Exemplo de uma matriz de 2 (duas) dimensões, 2 (duas) linhas e 4 (quatro) colunas:

m[2][4]	m = { {0, 1, 2, 3}, {4, 5, 6, 7} }			
	coluna 0	coluna 1	coluna 2	coluna 3
linha 0	m[0][0] = 0	m[0][1] = 1	m[0][2] = 2	m[0][3] = 3
linha 1	m[1][0] = 4	m[1][1] = 5	m[1][2] = 6	m[1][3] = 7

31.2 Arrays PostgreSQL

No PostgreSQL *arrays* são aplicados a campos de tabelas.

Esses *arrays* podem ser de qualquer tipo, porém domínios (*domains*) não são suportados ainda.

Para definirmos um campo como um *array* ou adicionamos colchetes ([]) logo após o nome do tipo ou a palavra “*array*”.

Mesmo se for declarado mais de uma dimensão e / ou tamanho de uma dimensão, em tempo de execução é ignorado.

Outra peculiaridade dos *arrays* e matrizes do PostgreSQL é que o primeiro índice é 1 (um) e não 0 (zero), como em C, por exemplo.

Sintaxes de criação:

<pre>CREATE TABLE tabela(campo tipo []);</pre>	<pre>CREATE TABLE tabela(campo tipo array);</pre>	<pre>CREATE TABLE tabela(campo tipo array []);</pre>
--------------------------------------------------------------	-----------------------------------------------------------------	--------------------------------------------------------------------

Como pôde ser notado, para criar uma coluna como *array* devemos adicionar os colchetes ([]) logo após o tipo de dado.

Mesmo se especificarmos um tamanho de vetor a implementação do PostgreSQL não força a obedecer. Nem mesmo se especificarmos que é multi-dimensional.

Criação de tabela de teste:

```
> CREATE TEMP TABLE tb_array(  
    campo_a text,  
    campo_b int2[],  
    campo_c text[][]  
);
```

Verificando a estrutura da tabela:

```
> \d tb_array
```

```
Table "pg_temp_2.tb_array"  
Column |      Type      | Modifiers  
-----+-----+-----  
campo_a | text           |  
campo_b | smallint[]     |  
campo_c | text[]         |
```

Repare que para campo_c mesmo tendo especificado mais de uma dimensão, na estrutura só consta uma.

31.3 Inserindo Valores de Array

Para inserirmos valores de um array, esse valor é da seguinte forma:

```
'{ val1 delim val2 delim ... }'
```

“delim” é o delimitador usado, o qual pode variar de um tipo para outro.

Verificando qual é o delimitador para smallint (int2):

```
> SELECT typdelim FROM pg_type WHERE typname = 'int2';  
  
typdelim  
-----  
,
```

São raríssimos os casos que o delimitador não seja a vírgula, para nossos exercícios vamos exemplificar na sintaxe a utilizando como delimitador.

Na dúvida faça a consulta anterior especificando o tipo que utilizará no lugar de int2.

Para arrays de uma dimensão:

```
'{num_1, num_2, ..., num_N}'  
{'string_1', 'string_2', ..., 'string_N'}
```

Com o construtor “array”:

```
array[num_1, num_2, ..., num_N]  
array['string_1', 'string_2', ..., 'string_N']
```

Para arrays de 2 dimensões:

```
'{  
{"string_1_1", "string_1_2", "string_1_3"},  
{"string_2_1", "string_2_2", "string_2_3"}  
}'
```

```
array[  
['string_1_1', 'string_1_2', 'string_1_3'],  
['string_2_1', 'string_2_2', 'string_2_3']  
]
```

O que difere de uma inserção comum é o fato de que a coluna determinada como array:

```
> INSERT INTO tb_array VALUES (  
    'bla bla bla',  
    '{2, 3, 4, 5}',  
    '{  
        {"texto1", "texto2", "texto3"},  
        {"foo", "bar", "baz"}  
    }'  
) RETURNING campo_a, campo_b, campo_c;
```

campo_a	campo_b	campo_c
bla bla bla	{2,3,4,5}	{{texto1,texto2,texto3},{foo,bar,baz}}

Inserção de valores com o construtor array para inteiros:

```
> INSERT INTO tb_array (campo_b) VALUES (  
    array[2, 3, 4, 5]  
) RETURNING campo_b;
```

campo_b
{2,3,4,5}

Inserção de valores com o construtor array para inteiros (duas dimensões):

```
> INSERT INTO tb_array (campo_b) VALUES (  
    array[[2, 3], [4, 5]]  
) RETURNING campo_b;
```

campo_b
{{2,3},{4,5}}

Inserção de valores com o construtor array para inteiros (duas dimensões):

```
> INSERT INTO tb_array (campo_b) VALUES (  
    array[[1, 2, 3], [4, 5]]  
) RETURNING campo_b;
```

ERROR: multidimensional arrays must have array expressions with matching dimensions

O erro foi provocado devido à quantidade diferente de elementos entre uma dimensão e outra.

Inserção de valores com o construtor array para inteiros (duas dimensões):

```
> INSERT INTO tb_array (campo_b) VALUES (  
    array[[1, 2, 3], [4, 5, 7]]  
) RETURNING campo_b;
```

```
      campo_b  
-----  
{ {1,2,3}, {4,5,7} }
```

Inserção de valores de texto:

```
> INSERT INTO tb_array (campo_c) VALUES (  
    '{"string 1", "string 2"}'  
) RETURNING campo_c;
```

```
      campo_c  
-----  
{"string 1","string 2"}
```

Inserção de valores para texto (duas dimensões):

```
> INSERT INTO tb_array (campo_c) VALUES (  
    '{  
        {"string 1_1", "string 1_2"},  
        {"string 2_1", "string 2_2"}  
    }'  
) RETURNING campo_c;
```

```
      campo_c  
-----  
{ {"string 1_1","string 1_2"}, {"string 2_1","string 2_2"} }
```

Inserção de valores para texto com construtor array:

```
> INSERT INTO tb_array (campo_c) VALUES (  
    array['string 1', 'string 2']  
) RETURNING campo_c;
```

```
      campo_c  
-----  
{"string 1","string 2"}
```


Inserção de valores para texto com construtor array (duas dimensões):

```
> INSERT INTO tb_array (campo_c) VALUES (
    array[
        ['string 1_1', 'string 1_2'],
        ['string 2_1', 'string 2_2']
    ]
) RETURNING campo_c;

-----
      campo_c
-----
{{"string 1_1","string 1_2"},"string 2_1","string 2_2"}}
```

Verificando a tabela após todos valores inseridos:

```
> TABLE tb_array;
```

campo_a	campo_b	campo_c
bla bla bla	{2,3,4,5}	{{texto1,texto2,texto3},{foo,bar,baz}}
	{2,3,4,5}	
	{{2,3},{4,5}}	
	{{1,2,3},{4,5,7}}	
		{"string 1","string 2"}
		{{"string 1_1","string 1_2"},"string 2_1","string 2_2"}}
		{"string 1","string 2"}
		{{"string 1_1","string 1_2"},"string 2_1","string 2_2"}}

31.3.1 Implementando Limites em Arrays

Como já foi mencionado neste capítulo, o PostgreSQL simplesmente ignora a quantidade de dimensões e o tamanho declarado de cada uma.

Caso necessite implementar limites para uniformizar, é necessário fazê-lo via criação de *constraints* CHECK junto com as funções `array_ndims` e `array_length`, que respectivamente retornam a quantidade de dimensões e o tamanho da dimensão (quantidade de elementos).

Apagando a tabela de testes:

```
> DROP TABLE IF EXISTS tb_array;
```

Criação da tabela de testes:

```
> CREATE TEMP TABLE tb_array(
    id serial PRIMARY KEY,
    vetor int2 [3], -- vetor de 3 elementos
    matriz int2 [3][2] -- matriz de 2 dimensões com 3 linhas e 2 colunas
);
```

Descrição da estrutura da tabela:

```
> \d tb_array
```

```
Table "pg_temp_2.tb_array"
Column |      Type      | Modifiers
-----+-----+-----
id      | integer        | not null default nextval('tb_array_id_seq'::regclass)
vetor   | smallint[]     |
matriz  | smallint[]     |
Indexes:
    "tb_array_pkey" PRIMARY KEY, btree (id)
```

Criação de constraint que determina o número de dimensões para 1 (campo vetor):

```
> ALTER TABLE tb_array ADD CONSTRAINT ck_vetor_dim_1
CHECK (array_ndims(vetor) = 1);
```

Criação de constraint que determina o tamanho da dimensão para 3 elementos (campo vetor):

```
> ALTER TABLE tb_array ADD CONSTRAINT ck_vetor_sz_3
CHECK (array_length(vetor, 1) = 3);
```

Tentativa de inserir um valor com duas dimensões:

```
> INSERT INTO tb_array (vetor) VALUES (array[[1], [2], [3]]);
```

```
ERROR: new row for relation "tb_array" violates check constraint "ck_vetor_dim_1"
DETAIL: Failing row contains (2, {{1},{2},{3}}, null).
```

Tentativa de inserir um valor com apenas dois elementos:

```
> INSERT INTO tb_array (vetor) VALUES (array[1, 2]);
```

```
ERROR: new row for relation "tb_array" violates check constraint "ck_vetor_sz_3"
DETAIL: Failing row contains (2, {1,2}, null).
```

Inserindo valores corretamente no campo, com uma dimensão e três elementos:

```
> INSERT INTO tb_array (vetor) VALUES (array[1, 2, 3]) RETURNING *;
```

```
id | vetor | matriz
---+-----+-----
3 | {1,2,3} |
```

Criação de constraint que determina o número de dimensões para 2 (campo matriz):

```
> ALTER TABLE tb_array ADD CONSTRAINT ck_matriz_dim_2
    CHECK (array_ndims(matriz) = 2);
```

Criação de constraint que determina a quantidade de linhas para 3 (campo matriz):

```
> ALTER TABLE tb_array ADD CONSTRAINT ck_matriz_lin_3
    CHECK (array_length(matriz, 1) = 3);
```

Criação de constraint que determina a quantidade de colunas para 2 (campo matriz):

```
> ALTER TABLE tb_array ADD CONSTRAINT ck_matriz_col_2
    CHECK (array_length(matriz, 2) = 2);
```

Tentativa de inserir dados na coluna matriz com 1 dimensão:

```
> INSERT INTO tb_array (matriz) VALUES (array[0, 1, 2]);
```

```
ERROR: new row for relation "tb_array" violates check constraint "ck_matriz_dim_2"
DETAIL: Failing row contains (19, null, {0,1,2}).
```

Tentativa de inserir dados na coluna matriz com 3 colunas:

```
> INSERT INTO tb_array (matriz) VALUES (array[
    [0, 1, null],
    [2, 3, null],
    [4, 5, null]
]);
```

```
ERROR: new row for relation "tb_array" violates check constraint "ck_matriz_col_2"
DETAIL: Failing row contains (20, null, {{0,1,NULL},{2,3,NULL},{4,5,NULL}}).
```

Tentativa de inserir dados na coluna matriz com 2 linhas:

```
> INSERT INTO tb_array (matriz) VALUES (array[
    [0, 1],
    [2, 3]
]);
```

```
ERROR: new row for relation "tb_array" violates check constraint "ck_matriz_lin_3"
DETAIL: Failing row contains (21, null, {{0,1},{2,3}}).
```

Inserção de dados corretamente, com 2 dimensões, 3 linhas e 2 colunas:

```
> INSERT INTO tb_array (matriz) VALUES (array[
  [0, 1], -- linha 1
  [2, 3], -- linha 2
  [4, 5] -- linha 3
]) RETURNING *;
```

<i>id</i>	<i>vetor</i>	<i>matriz</i>
7		{{0,1},{2,3},{4,5}}

31.4 Consultas em Arrays

Agora com uma tabela com campos de *array*, a qual foi preenchida, podemos fazer seleções nela.

Apagando a tabela de exemplo:

```
> DROP TABLE IF EXISTS tb_array;
```

Criando a tabela novamente:

```
> CREATE TEMP TABLE tb_array(  
    id SERIAL PRIMARY KEY,  
    campo TEXT []);
```

Inserindo valores na tabela:

```
> INSERT INTO tb_array (campo) VALUES  
    (array['I 1']),  
    (array['I 1', 'I 2']),  
    (array[  
        ['I 1', 'I 2', 'I 3', 'I 4'],  
        ['II 1', 'II 2', 'II 3', 'II 4'],  
        ['III 1', 'III 2', 'III 3', 'III 4']  
    ]) RETURNING campo;
```

campo

```
-----  
{ "I 1" }  
{ "I 1", "I 2" }  
{ "I 1", "I 2", "I 3", "I 4", {"II 1", "II 2", "II 3", "II 4"}, {"III 1", "III 2", "III 3", "III 4" }
```

Verificando o primeiro elemento do vetor:

```
> SELECT COALESCE(campo[1], 'NULO') FROM tb_array;
```

```
coalesce  
-----  
I 1  
I 1  
NULO
```

Verificando o segundo elemento do vetor:

```
> SELECT COALESCE(campo[2], 'NULO') FROM tb_array;
```

```
coalesce
-----
NULO
I 2
NULO
```

Elemento da linha 1 e coluna 2 da matriz:

```
> SELECT COALESCE(campo[1][2], 'NULO') FROM tb_array;
```

```
coalesce
-----
NULO
NULO
I 2
```

Elemento da linha 3 e coluna 2 da matriz:

```
> SELECT COALESCE(campo[3][2], 'NULO') FROM tb_array;
```

```
coalesce
-----
NULO
NULO
III 2
```

31.4.1 Slicing – Fatiamento de Vetores

Selecionar a primeira linha e da segunda à terceira coluna:

```
> SELECT campo[1][2:3] FROM tb_array;
```

ou

```
> SELECT campo[1:1][2:3] FROM tb_array;
```

```
campo
-----
{}
{}
{"I 2", "I 3"}
```

Elementos que estejam da primeira à segunda linha, da segunda à terceira coluna:

```
> SELECT campo[1:2][2:3] FROM tb_array;
```

ou

```
> SELECT campo[2][2:3] FROM tb_array;
```

```
-----  
campo  
-----  
{ }  
{ }  
{ "I 2", "I 3"}, {"II 2", "II 3"}
```

Elementos que estejam na segunda linha, da segunda à terceira coluna:

```
> SELECT campo[2:2][2:3] FROM tb_array;
```

```
-----  
campo  
-----  
{ }  
{ }  
{ "II 2", "II 3"}
```

Elementos que estejam da primeira à segunda linha, e na segunda coluna:

```
> SELECT campo[1:2][2:2] FROM tb_array;
```

```
-----  
campo  
-----  
{ }  
{ }  
{ "I 2"}, {"II 2"}
```

31.5 Modificando Arrays

Modificando todos elementos de um array:

```
> UPDATE tb_array SET campo = array[1, 2, 3] WHERE id = 1 RETURNING *;
```

```
id | campo
----+-----
 1 | {1,2,3}
```

Alterando um único elemento do array:

```
> UPDATE tb_array SET campo[1] = 0 WHERE id = 1 RETURNING *;
```

```
id | campo
----+-----
 1 | {0,2,3}
```

Atualizando a linha em um único elemento da matriz:

```
> UPDATE tb_array SET campo[2][3] = 23 WHERE id = 3 RETURNING *;
```

```
id | campo
----+-----
 3 | {{ "I 1", "I 2", "I 3", "I 4"}, {"II 1", "II 2", 23, "II 4"}, {"III 1", "III 2", "III 3", "III 4"}}
```

Fazendo modificação por slicing:

```
> UPDATE tb_array SET campo[2:2][1:3] = array['Um', 'dois', 'três']
WHERE id = 3 RETURNING *;
```

```
id | campo
----+-----
 3 | {{ "I 1", "I 2", "I 3", "I 4"}, {Um, dois, três, "II 4"}, {"III 1", "III 2", "III 3", "III 4"}}
```


31.6 Funções de Arrays

Alguns exemplos de algumas funções para manipulação de arrays.

Maiores detalhes na documentação oficial:

<http://www.postgresql.org/docs/current/static/functions-array.html>

Converte os elementos de um vetor ou matriz em linhas:

```
> SELECT unnest(array[[1, 2], [3, 4]]);
```

```
unnest
-----
1
2
3
4
```

Tamanho de uma dimensão:

```
> SELECT array_length(array[[1, 2, 3, 4], [5, 6, 7, 8]], 2);
```

```
array_length
-----
4
```

Tamanho de uma dimensão:

```
> SELECT array_length(array[
    [1, 2, 3, 4],
    [5, 6, 7, 8]],
    1);
```

```
array_length
-----
2
```

Retorna o array passado no primeiro argumento junto com o elemento dado no segundo parâmetro:

```
> SELECT array_append(array[1, 2], 3);
```

```
array_append
-----
{1,2,3}
```

Concatenação de arrays:

```
> SELECT array_cat(array[1, 2, 3], ARRAY[4, 5]);
```

```
array_cat
-----
{1,2,3,4,5}
```

Quantos elementos há na matriz?:

```
> SELECT
  cardinality(
    array[
      ['I 1', 'I 2', 'I 3', 'I 4'],
      ['II 1', 'II 2', 'II 3', 'II 4'],
      ['III 1', 'III 2', 'III 3', 'III 4']
    ]
  );
```

```
cardinality
-----
12
```

31.7 Operadores de Arrays

Operador	Descrição	Exemplo	Resultado
=	igual	ARRAY[1.1, 2.1, 3.1]::int[] = ARRAY[1, 2, 3]	t
<>, !=	diferente	ARRAY[1, 2, 3] <> ARRAY[1, 2, 4]	t
<	menor que	ARRAY[1, 2, 3] < ARRAY[1, 2, 4]	t
>	maior que	ARRAY[1, 4, 3] > ARRAY[1, 2, 4]	t
<=	menor ou igual	ARRAY[1, 2, 3] <= ARRAY[1, 2, 3]	t
>=	maior ou igual	ARRAY[1, 4, 3] >= ARRAY[1, 4, 3]	t
@>	contém	ARRAY[1, 4, 3] @> ARRAY[3, 1]	t
<@	contido	ARRAY[2, 7] <@ ARRAY[1, 7, 4, 2, 6]	t
&&	sobreposição (elementos comuns?)	ARRAY[1, 4, 3] && ARRAY[2, 1]	t
	concatenação array-to-array	ARRAY[1, 2, 3] ARRAY[4, 5, 6]	{1, 2, 3, 4, 5, 6}
		ARRAY[1, 2, 3] ARRAY[[4, 5, 6], [7, 8, 9]]	{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
	concatenação element-to-array	3 ARRAY[4, 5, 6]	{3, 4, 5, 6}
	concatenação array-to-element	ARRAY[4, 5, 6] 7	{4, 5, 6, 7}

31.8 Alterando Tipos para Array

Imagine que após modelar uma base descobre-se que certos campos deveriam ser arrays... Como proceder?

Caso I

Criação da tabela de teste de livros:

```
> CREATE TEMP TABLE tb_livro (  
    id serial primary key,  
    titulo text not null,  
    autor text);
```

Inserir dados:

```
> INSERT INTO tb_livro (titulo, autor) VALUES ('O Hobbit', 'Tolkien');
```

Tentativa de alteração do tipo de dado para o campo autor, para que o mesmo seja um array:

```
> ALTER TABLE tb_livro ALTER COLUMN autor TYPE text[];  
  
ERROR: column "autor" cannot be cast automatically to type text[]  
HINT: You might need to specify "USING autor::text[]".
```

Seguindo as orientações da dica, alterando devidamente o tipo de dados para array textual:

```
> ALTER TABLE tb_livro ALTER COLUMN autor TYPE text[] USING array[autor];
```

Inserindo novos registros (com arrays):

```
> INSERT INTO tb_livro (titulo, autor) VALUES  
('Database System Concepts', array['Silberschatz', 'Korth', 'Sudarshan']);
```

Verificando os dados na tabela:

```
> TABLE tb_livro;

id |          titulo          |          autor
---+-----+-----
 1 | O Hobbit                 | {Tolkien}
 2 | Database System Concepts | {Silberschatz,Korth,Sudarshan}
```

Consultando todos os títulos que tenham como autor Korth:

```
> SELECT titulo FROM tb_livro WHERE autor && array['Korth'];

          titulo
-----
Database System Concepts
```

Consultando todos os títulos que tenham como autor Korth ou Lewis:

```
> SELECT titulo FROM tb_livro WHERE autor && array['Korth', 'Lewis'];

          titulo
-----
Database System Concepts
```

Consultando todos os títulos que tenham como autor Korth ou Tolkien:

```
> SELECT titulo FROM tb_livro WHERE autor && array['Korth', 'Tolkien'];

          titulo
-----
O Hobbit
Database System Concepts
```

Caso II

Criação da tabela de teste de servidor:

```
> CREATE TEMP TABLE tb_servidor(
    id smallserial PRIMARY KEY,
    hostname text,
    ip inet);
```

Inserir um registro sem arrays:

```
> INSERT INTO tb_servidor (hostname, ip) VALUES ('srv00001', '192.168.7.1');
```

Tentativa de alteração do tipo de dado para o campo ip, para que o mesmo seja um array:

```
> ALTER TABLE tb_servidor ALTER COLUMN ip TYPE inet[];

ERROR: column "ip" cannot be cast automatically to type inet[]
HINT: You might need to specify "USING ip::inet[]".
```

Seguindo as orientações da dica, alterando devidamente o tipo de dados para array de endereço de IP:

```
> ALTER TABLE tb_servidor ALTER COLUMN ip TYPE inet[] USING array[ip]::inet[];
```

Tentativa de inserir um novo registro com array de IPs:

```
> INSERT INTO tb_servidor (hostname, ip)
  VALUES ('srv00002', array['192.168.7.2', '10.0.0.2']);

ERROR: column "ip" is of type inet[] but expression is of type text[]
LINE 1: ...TO tb_servidor (hostname, ip) VALUES ('srv00001', array['192...
                                     ^
HINT: You will need to rewrite or cast the expression.
```

Comando INSERT com cast para o tipo do array:

```
> INSERT INTO tb_servidor (hostname, ip)
  VALUES ('srv00002', array['192.168.7.2', '10.0.0.2']::inet[]);
```

Verificando os dados da tabela:

```
> TABLE tb_servidor;

 id | hostname | ip
-----+-----+-----
  1 | srv00001 | {192.168.7.1}
  2 | srv00002 | {192.168.7.2,10.0.0.2}
```

Buscar na tabela de servidores, exibindo o hostname onde um dos IPs é 10.0.0.2:

```
> SELECT hostname FROM tb_servidor WHERE ip @> array['10.0.0.2']::inet[];

 hostname
-----
  srv00002
```

32 INDEX - Índice

- Sobre Índice
- Índices B-tree
- Índices Hash
- Índices GiST
- Índices SP-GiST
- Índices GIN
- Índices BRIN
- Índices Compostos
- Índices Parciais
- Fillfactor - Fator de Preenchimento
- Reconstrução de Índices – REINDEX
- CLUSTER – Índices Clusterizados
- Excluindo um Índice

32.1 Sobre Índice

Um índice (INDEX) é um recurso que agiliza buscas de informações em tabelas.

Imagine que você está em uma biblioteca e gostaria de procurar “O Senhor dos Anéis”, de Tolkien. O que seria mais fácil? Começar a vasculhar a biblioteca inteira até achar o livro desejado (busca sequencial) ou buscar no arquivo da biblioteca (busca indexada), nas fichas que estão ordenados por autor? Logicamente se for escolhido ir buscar nas fichas a busca será muito mais rápida, pois não será necessário vasculhar livro por livro na biblioteca, porque haverá uma ficha do autor e daquele livro que mostrará exatamente onde está o livro desejado. É um apontamento para a localização do livro. Um índice de banco de dados funciona de forma semelhante.

Seu funcionamento consiste em criar ponteiros para dados gravados em campos específicos. Quando não existe índice num campo usado como critério de filtragem, é feita uma varredura em toda a tabela, de maneira que haverá execuções de entrada e saída (I/O) de disco desnecessárias, além de também desperdiçar processamento.

Ao se criar chaves primárias, automaticamente um índice é criado.

32.1.1 Tipos de Índices

O PostgreSQL em seu core dispõe dos seguintes tipos de índices: Btree (padrão), Hash, GiST, SP-GiST, GIN e BRIN, que serão vistos aqui.

Há também outros tipos de índices que são oferecidos como extensões.

32.1.2 Dicas Gerais

- Crie índices para campos cujas consultas o envolvam em uma das seguintes cláusulas: `WHERE`, `DISTINCT`, `ORDER BY`, `GROUP BY` e `LIKE`;
- Crie índices para campos de chaves estrangeiras e em campos envolvidos como critérios de junção (`JOIN`);
- Se houver uma consulta frequente utilize índices parciais com sua condição conforme a consulta;
- Para consultas que buscam faixas de valores é bom ter um índice clusterizado para isso;
- O PostgreSQL oferece diferentes tipos de índices, o que nos dá a possibilidade de adotar diferentes tipos de indexação conforme a consulta e seus tipos de dados, teste e adote o mais adequado;
- Após criar um índice atualize as estatísticas da tabela com `ANALYZE` [1] ou `VACUUM ANALYZE` [2];
- Antes de colocar uma base em produção faça testes com massas de dados considerável e utilize o comando `EXPLAIN ANALYZE` [3] para verificar o plano de execução.

[1] <https://www.postgresql.org/docs/current/static/sql-analyze.html>

[2] <https://www.postgresql.org/docs/current/static/sql-vacuum.html>

[3] <https://www.postgresql.org/docs/current/static/sql-explain.html>

32.2 Índices B-tree

Pode lidar com consultas de igualdade ou de faixa de valores cujos dados podem ser ordenados.

O planejador de consultas do PostgreSQL considerará usar um índice B-tree sempre que um campo indexado estiver envolvido em uma comparação usando um dos seguintes operadores: <, <=, =, >=, > ou BETWEEN, IN, IS [NOT] NULL.

O otimizador pode também usar índices B-tree para consultas envolvendo operadores de padrão de combinação LIKE e ~ se o padrão for uma constante e estiver no início da string, por exemplo: coluna LIKE 'foo%' ou coluna ~ '^foo', mas não coluna LIKE '%bar'. Porém, se sua base não usa locale C será necessário criar o índice com uma classe de operador especial para suportar indexação de consultas de casamento de padrões.

É possível também usar índices B-tree para ILIKE e ~*, mas apenas se o padrão começar com caracteres não-alfabéticos, i. e., caracteres que não são afetados pela conversão upper/lower case (letras maiúsculas/minúsculas).

Índices B-tree podem também ser usados para buscar dados ordenadamente. Isso não é sempre mais rápido do que uma busca simples e ordenação, mas é sempre útil.

Criação de tabela de teste:

```
> SELECT
generate_series(1, 20000)::int2 AS campo1, -- 20 mil registros
round((random()*10000))::int2 AS campo2,
round((random()*10000))::int2 AS campo3 INTO tb_index;
```

Verificando o plano de execução:

```
> EXPLAIN ANALYZE SELECT campo1 FROM tb_index WHERE campo2 BETWEEN 235 AND 587;
```

```

                                QUERY PLAN
-----
Seq Scan on tb_index  (cost=0.00..390.71 rows=101 width=2) (actual time=0.046..8.204 rows=684
loops=1)
  Filter: ((campo2 >= 235) AND (campo2 <= 587))
  Rows Removed by Filter: 19316
  Planning time: 0.161 ms
  Execution time: 8.456 ms
```

A tabela ainda não tem índice, portanto a busca vai ser sequencial.

Criação de índice para a tabela:

```
> CREATE INDEX idx_tb_index_campo2 ON tb_index (campo2);
```

Na criação do índice não foi especificado seu tipo, portanto esse é um índice btree, que é o tipo padrão do PostgreSQL.

Verificando o plano de execução:

```
> EXPLAIN ANALYZE SELECT campo1 FROM tb_index WHERE campo2 BETWEEN 235 AND 587;
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on tb_index  (cost=15.39..114.79 rows=693 width=2) (actual time=0.075..0.346 rows=684 loops=1)  
  Recheck Cond: ((campo2 >= 235) AND (campo2 <= 587))  
  Heap Blocks: exact=89  
-> Bitmap Index Scan on idx_tb_index_campo2  (cost=0.00..15.22 rows=693 width=0) (actual time=0.064..0.064  
rows=684 loops=1)  
    Index Cond: ((campo2 >= 235) AND (campo2 <= 587))  
Planning time: 0.171 ms  
Execution time: 0.437 ms
```

Comparando antes e depois da criação do índice, vemos que antes foi usada a busca sequencial (*Seq Scan*) e levou **8.456 ms**. Após criar o índice, sendo que o mesmo foi criado para o campo utilizado na condição do `WHERE`, vemos que foi utilizada busca via índice (*Index Scan*) e levou **0.437 ms** para executar.

Por esse experimento podemos ver o quão útil é ter um índice, devido à forma como ele agiliza uma busca.

32.3 Índices Hash

Índices hash podem apenas lidar com comparações de igualdade. O planejador de consulta considerará usar um índice hash sempre que uma coluna indexada estiver envolvida em uma comparação com o operador =.

Se a tabela existir, deve ser removida:

```
> DROP TABLE IF EXISTS tb_foo;
```

Criação de uma tabela de teste que tenha dez milhões de registros:

```
> SELECT generate_series(1, 10000000) AS numero INTO tb_foo;
```

Criação de dois índices, hash e btree, respectivamente:

```
> CREATE INDEX idx_hash ON tb_foo USING hash (numero);
```

```
> CREATE INDEX idx_btree ON tb_foo USING btree (numero);
```

Verificando o tamanho de cada índice criado:

```
> SELECT pg_size_pretty(pg_relation_size('idx_hash'));
```

```
pg_size_pretty
-----
320 MB
```

```
> SELECT pg_size_pretty(pg_relation_size('idx_btree'));
```

```
pg_size_pretty
-----
214 MB
```

Podemos notar que o índice btree é 33% menor do que o índice hash.

Verificando do plano de execução de uma consulta de igualdade.

```
> EXPLAIN ANALYZE SELECT numero FROM tb_foo WHERE numero = 95773;
```

QUERY PLAN

```
-----  
Index Scan using idx_hash on tb_foo (cost=0.00..8.02 rows=1 width=4) (actual time=0.014..0.016  
rows=1 loops=1)  
  Index Cond: (numero = 95773)  
Planning time: 0.097 ms  
Execution time: 0.039 ms
```

Verificando o plano de execução de consulta com o operador BETWEEN:

```
> EXPLAIN ANALYZE SELECT numero FROM tb_foo WHERE numero BETWEEN 95773 AND 100000;
```

QUERY PLAN

```
-----  
Index Only Scan using idx_btree on tb_foo (cost=0.43..142.95 rows=3876 width=4) (actual  
time=0.063..1.518 rows=4228 loops=1)  
  Index Cond: ((numero >= 95773) AND (numero <= 100000))  
  Heap Fetches: 4228  
Planning time: 0.204 ms  
Execution time: 1.799 ms
```

32.4 Índices GiST

Índices GiST não são um único tipo de índice, mas sim uma infraestrutura interna que muitas estratégias diferentes de indexação podem ser implementadas. Portanto, os operadores em particular com que um índice GiST pode ser usado varia dependendo da estratégia de indexação (a classe de operador). Como um exemplo, a distribuição padrão do PostgreSQL inclui classes de operador GiST para vários tipos de dados geométricos de duas dimensões, que suporte consultas indexadas usando os seguintes operadores: <<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~=, &&. Muitas outras classes de operadores GiST estão disponíveis na coleção contrib ou como projetos separados.

Índices GiST são também capazes de otimizar buscas “*nearest-neighbor*” (vizinho mais próximo) como:

```
SELECT * FROM lugares
ORDER BY localizacao <-> ponto '(101, 456)'
LIMIT 10;
```

Essa consulta encontra os dez lugares mais perto de um dado ponto alvo. A capacidade de fazer isso é novamente dependente de classe de operador particular a ser usada.

Classes de Operadores GiST Built-in			
Classe	Tipo	Operadores	Operadores de Ordenação
box_ops	box	&& &> &< &< >> << << <@ @> @ &> >> ~ ~=	
circle_ops	circle	&& &> &< &< >> << << <@ @> @ &> >> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <=< < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &< >> << << <@ @> @ &> >> ~ ~=	<->
range_ops	any range type	&& &> &< >> << <@ - = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

A classe `inet_ops` não é a classe padrão para tipos `cidr` e `inet`. Para utilizá-la, mencione o nome da classe na criação do índice:

```
CREATE INDEX ON tabela USING GIST (campo_inet inet_ops);
```

Criação de função para gerar um dado do tipo `int4range`:

```
> CREATE OR REPLACE FUNCTION fc_gera_int4range(n int4)
RETURNS int4range AS $$
DECLARE
    limite_superior int4 := round(random() * n);
    limite_inferior int4 := round(random() * limite_superior);
BEGIN
    RETURN ('[||limite_inferior||', '||limite_superior||']')::int4range;
END; $$ LANGUAGE PLPGSQL;
```

Se a tabela existir, deve ser apagada:

```
> DROP TABLE IF EXISTS tb_foo;
```

Criação da tabela de teste:

```
> CREATE TABLE tb_foo(  
    id int,  
    faixa int4range);
```

Popular a tabela utilizando a função criada:

```
> INSERT INTO tb_foo (id, faixa)  
    SELECT generate_series(1, 1000000), fc_gera_int4range(1000000);
```

Criação de índices GiST sem declarar a classe de operador e declarando, respectivamente:

```
> CREATE INDEX idx_gist ON tb_foo USING gist (faixa);
```

```
> CREATE INDEX idx_gist_op_class ON tb_foo USING gist (faixa range_ops);
```

Verificando o plano de execução:

```
> EXPLAIN ANALYZE SELECT id, faixa FROM tb_foo WHERE faixa @> 777;
```

```
-----  
QUERY PLAN  
-----  
Bitmap Heap Scan on tb_foo (cost=1959.92..65333.64 rows=50000 width=36) (actual  
time=19.713..25.031 rows=7340 loops=1)  
  Recheck Cond: (faixa @> 777)  
  Heap Blocks: exact=6919  
    -> Bitmap Index Scan on idx_gist_op_class (cost=0.00..1947.42 rows=50000 width=0) (actual  
time=18.861..18.861 rows=7340 loops=1)  
      Index Cond: (faixa @> 777)  
Planning time: 0.254 ms  
Execution time: 25.301 ms
```

O planejador de consultas escolheu o índice cuja classe de operador foi declarada.

32.5 Índices SP-GiST

Como índices GiST, índices SP-GiST, oferecem uma infraestrutura que suporta vários tipos de buscas. Permite implementar uma vasta faixa de diferentes estruturas de dados baseadas em disco não balanceadas, como *quadtrees*, *k-d trees* e *radix trees* (tentativas).

Como um exemplo, a distribuição padrão do PostgreSQL inclui classes de operadores SP-GiST para pontos de duas dimensões, que suportem consultas usando estes operadores: <<, >>, ~=, <@, <^, >^.

Classes de Operadores SP-GiST Built-in		
Classe	Tipo	Operadores
kd_point_ops	point	<< <@ <^ >> >^ ~=
quad_point_ops	point	<< <@ <^ >> >^ ~=
range_ops	Qualquer tipo de faixa	&& &< &> - - << <@ = >> @>
box_ops	box	<< &< && &> >> ~= @> <@ &< << >> &>
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~
inet_ops	inet, cidr	&& >> >>= > >= <> << <= < <= =

Das duas classes de operadores para o tipo de ponto, `quad_point_ops` é o padrão. `kd_point_ops` suporta os mesmos operadores, mas usa uma estrutura diferente de dados de índice que pode oferecer um melhor desempenho em algumas aplicações.

Criação de índices SP-GiST sem declarar a classe de operador e declarando, respectivamente:

```
> CREATE INDEX idx_spgist ON tb_foo USING spgist (faixa);
```

```
> CREATE INDEX idx_spgist_op_class ON tb_foo USING spgist (faixa range_ops);
```

Verificando o plano de execução:

```
> EXPLAIN ANALYZE SELECT id, faixa FROM tb_foo WHERE faixa @> 777;
```

```
-----  
QUERY PLAN  
-----  
Bitmap Heap Scan on tb_foo (cost=200.50..17275.59 rows=5688 width=18) (actual time=20.704..25.929  
rows=7340 loops=1)  
  Recheck Cond: (faixa @> 777)  
  Heap Blocks: exact=6919  
    -> Bitmap Index Scan on idx_spgist_op_class (cost=0.00..199.08 rows=5688 width=0) (actual  
time=19.837..19.837 rows=7340 loops=1)  
      Index Cond: (faixa @> 777)  
Planning time: 104.493 ms  
Execution time: 28.205 ms
```

Como aconteceu com os índices GiST, aqui também o planejador de consultas escolheu o índice cuja classe de operador foi declarada.

Verificando o tamanho de todos os índices da tabela:

```
> SELECT  
  indexname indice,  
  pg_size_pretty(pg_relation_size(indexname::text)) tamanho  
FROM pg_indexes  
WHERE tablename = 'tb_foo';
```

indice	tamanho
idx_gist	615 MB
idx_gist_op_class	613 MB
idx_spgist	526 MB
idx_spgist_op_class	526 MB

Os índices SP-GiST têm um tamanho menor do que os índices GiST e o planejador de consultas utilizou o índice GiST de classe de operador declarada. Neste caso foi a melhor escolha de indexação de acordo com a consulta.

32.6 Índices GIN

Índices GIN são “índices invertidos” que são apropriados para valores de dados que contêm valores de componentes múltiplos, como *arrays*. Um índice invertido contém uma entrada para cada valor componente, e pode eficientemente lidar com consultas que testam a presença de valores componentes específicos.

Como GiST e SP-GiST, GIN pode suportar muitas diferentes estratégias de indexação definidas por usuário e os operadores com que um índice GIN podem ser usados variam de acordo com a estratégia de indexação.

Como um exemplo, a distribuição padrão de PostgreSQL inclui uma classe de operador GIN para *arrays*, que suporte consultas indexadas usando estes operadores: `<@`, `@>`, `=`, `&&`.

Muitos outros operadores de classes GIN estão disponíveis na coleção contrib como projetos separados.

Classes de Operadores GIN Built-in		
Classe	Tipo	Operadores
array_ops	anyarray	<code>&&</code> <code><@</code> <code>=</code> <code>@></code>
jsonb_ops	jsonb	<code>? ?& ? </code> <code>@></code>
jsonb_path_ops	jsonb	<code>@></code>
tsvector_ops	tsvector	<code>@@</code> <code>@@@</code>

Se a tabela existir, deve ser apagada:

```
> DROP TABLE IF EXISTS tb_foo;
```

Criação da tabela de teste:

```
> CREATE TABLE tb_foo (  
    id int,  
    vetor int[]);
```

Popular a tabela:

```
> INSERT INTO tb_foo (id, vetor)  
    SELECT  
        generate_series(1, 10000000),  
        ARRAY[  
            round(random() * 10000000),  
            round(random() * 10000000),  
            round(random() * 10000000)];
```

Criação dos índices, sem e com declaração de classe de operador, respectivamente:

```
> CREATE INDEX idx_gin ON tb_foo USING gin (vetor);  
  
> CREATE INDEX idx_gin_op_class ON tb_foo USING gin (vetor array_ops);
```

Verificando o plano de execução:

```
> EXPLAIN ANALYZE SELECT vetor FROM tb_foo WHERE vetor @> ARRAY[754532];
```

```
-----  
QUERY PLAN  
-----  
Bitmap Heap Scan on tb_foo (cost=483.50..76566.99 rows=50000 width=33) (actual time=47.920..47.937  
rows=7 loops=1)  
  Recheck Cond: (vetor @> '{754532}'::integer[])  
  Heap Blocks: exact=7  
    -> Bitmap Index Scan on idx_gin_op_class (cost=0.00..471.00 rows=50000 width=0) (actual  
time=46.489..46.489 rows=7 loops=1)  
      Index Cond: (vetor @> '{754532}'::integer[])  
Planning time: 0.294 ms  
Execution time: 47.970 ms
```

O índice com classe de operador declarada foi escolhido pelo planejador de consultas.

32.7 Índices BRIN

Índices BRIN (uma abreviação para *Block Range INdices*) armazena sumários sobre os valores armazenados em faixas de blocos físicos consecutivos de uma tabela. Assim como GiST, SP-GiST e GIN, BRIN pode suportar muitas diferentes estratégias e operadores com que um índice BRIN pode ser usado variam de acordo com a estratégia de indexação.

Para tipos de dados que tem uma ordem de classificação linear, os dados indexados correspondem ao mínimo e máximo de valores na coluna para cada faixa de bloco. Isso suporta consultas indexadas usando estes operadores: <, <=, =, >=, >.

Um índice BRIN para uma busca de uma consulta é uma mistura de busca sequencial e busca indexada porque o que essa busca indexada está armazenando é uma faixa de dados dado um número fixo de blocos de dados.

Criação de tabela de testes:

```
> CREATE TABLE tb_temperatura_log (  
    id serial,  
    dt timestamp without time zone,  
    temperatura int);
```

Popular tabela:

```
> INSERT INTO tb_temperatura_log (dt, temperatura) VALUES  
    (generate_series(  
        '2017-01-01'::timestamp,  
        '2017-12-31'::timestamp,  
        '1 second'),  
        round(random() * 100)::int);
```

Criação de índice btree:

```
> CREATE INDEX idx_btree ON tb_temperatura_log USING btree (dt);
```

Verificar o plano de execução:

```
> EXPLAIN ANALYZE
SELECT avg(temperatura)
  FROM tb_temperatura_log
 WHERE dt >='2017-03-01' AND dt < '2017-07-07';
```

QUERY PLAN

```
-----
Finalize Aggregate  (cost=358139.28..358139.29 rows=1 width=32) (actual time=991.406..991.406 rows=1 loops=1)
-> Gather  (cost=358139.06..358139.27 rows=2 width=32) (actual time=991.395..991.398 rows=3 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial Aggregate  (cost=357139.06..357139.07 rows=1 width=32) (actual time=949.103..949.103 rows=1
loops=3)
        -> Parallel Index Scan using idx_btree on tb_temperatura_log  (cost=0.56..345708.55 rows=4572205
width=4) (actual time=0.055..696.472 rows=368
6400 loops=3)
            Index Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt < '2017-07-07
00:00:00'::timestamp without time zone))
            Planning time: 9.103 ms
            Execution time: 1005.805 ms
```

Classe de Operadores BRIN Built-in		
Classe	Tipo	Operadores
abstime_minmax_ops	abstime	< <= = >= >
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &< << >> &>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
macaddr8_minmax_ops	macaddr8	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >
range_inclusion_ops	any range type	<< &< && &> >> @> <@ - = < <= = >= >
float4_minmax_ops	real	< <= = >= >
reltime_minmax_ops	reltime	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >

Classe de Operadores BRIN Built-in		
timestamp_tz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >
timetz_minmax_ops	time with time zone	< <= = >= >
uuid_minmax_ops	uuid	< <= = >= >

Criação de índice BRIN sem classe de operador:

```
> CREATE INDEX idx_brin ON tb_temperatura_log USING brin (dt);
```

É de se notar que a criação do índice BRIN é muito mais rápida do que a do índice B-tree.

Verificar o plano de execução:

```
> EXPLAIN ANALYZE
SELECT avg(temperatura)
FROM tb_temperatura_log
WHERE dt >='2017-03-01' AND dt < '2017-07-07';
```

QUERY PLAN

```
-----
Finalize Aggregate (cost=284242.36..284242.37 rows=1 width=32) (actual time=740.252..740.252 rows=1 loops=1)
  -> Gather (cost=284242.14..284242.35 rows=2 width=32) (actual time=740.217..740.244 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (cost=283242.14..283242.15 rows=1 width=32) (actual time=737.556..737.556 rows=1
loops=3)
              -> Parallel Bitmap Heap Scan on tb_temperatura_log (cost=2793.83..271811.63 rows=4572205 width=4)
(actual time=3.276..492.919 rows=3686400 loops=3)
                    Recheck Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt < '2017-07-
07 00:00:00'::timestamp without time zone))
                    Rows Removed by Index Recheck: 4565
                    Heap Blocks: lossy=23492
                    -> Bitmap Index Scan on idx_brin (cost=0.00..50.50 rows=10992288 width=0) (actual
time=4.692..4.692 rows=705280 loops=1)
                          Index Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt <
'2017-07-07 00:00:00'::timestamp without time zone))
Planning time: 0.251 ms
Execution time: 740.991 ms
```

Criação de índice BRIN com classe de operador:

```
> CREATE INDEX idx_brin_op_class ON tb_temperatura_log
USING brin (dt timestamp_minmax_ops);
```

Verificar o plano de execução:

```
> EXPLAIN ANALYZE
SELECT avg(temperatura)
FROM tb_temperatura_log
WHERE dt >='2017-03-01' AND dt < '2017-07-07';

QUERY PLAN
-----
Finalize Aggregate (cost=284242.36..284242.37 rows=1 width=32) (actual time=738.905..738.905 rows=1 loops=1)
-> Gather (cost=284242.14..284242.35 rows=2 width=32) (actual time=738.853..738.895 rows=3 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial Aggregate (cost=283242.14..283242.15 rows=1 width=32) (actual time=736.258..736.258 rows=1
loops=3)
        -> Parallel Bitmap Heap Scan on tb_temperatura_log (cost=2793.83..271811.63 rows=4572205 width=4)
(actual time=3.297..490.951 rows=3686400 lo
ops=3)
            Recheck Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt < '2017-07-
07 00:00:00'::timestamp without time zone))
            Rows Removed by Index Recheck: 4565
            Heap Blocks: lossy=23390
            -> Bitmap Index Scan on idx_brin_op_class (cost=0.00..50.50 rows=10992288 width=0) (actual
time=4.652..4.652 rows=705280 loops=1)
                Index Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt <
'2017-07-07 00:00:00'::timestamp without time zon
e))
            Planning time: 0.246 ms
            Execution time: 739.612 ms
```

32.7.1 Páginas por Faixa

Em índices BRIN tem um parâmetro que controla a quantidade de páginas por faixa, `pages_per_range`, cujo valor padrão é 128.

Quanto mais páginas por faixa menor o índice será, no entanto a parte de busca sequencial será maior.

Criação de índices com diferentes configurações para páginas por faixa:

```
> CREATE INDEX idx_brin_op_class_64 ON tb_temperatura_log
    USING brin (dt timestamp_minmax_ops)
    WITH (pages_per_range = 64);

> CREATE INDEX idx_brin_op_class_256 ON tb_temperatura_log
    USING brin (dt timestamp_minmax_ops)
    WITH (pages_per_range = 256);

> CREATE INDEX idx_brin_op_class_512 ON tb_temperatura_log
    USING brin (dt timestamp_minmax_ops)
    WITH (pages_per_range = 512);
```

Verificar o plano de execução:

```
> EXPLAIN ANALYZE
SELECT avg(temperatura)
  FROM tb_temperatura_log
 WHERE dt >='2017-03-01' AND dt < '2017-07-07';
```

QUERY PLAN

```
Finalize Aggregate (cost=284219.30..284219.31 rows=1 width=32) (actual time=739.260..739.260 rows=1 loops=1)
  -> Gather (cost=284219.08..284219.29 rows=2 width=32) (actual time=739.224..739.252 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (cost=283219.08..283219.09 rows=1 width=32) (actual time=737.219..737.219 rows=1
loops=3)
              -> Parallel Bitmap Heap Scan on tb_temperatura_log (cost=2776.86..271788.57 rows=4572205 width=4)
(actual time=1.737..492.086 rows=3686400 loops=3)
                    Recheck Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt < '2017-07-
07 00:00:00'::timestamp without time zone))
                    Rows Removed by Index Recheck: 11264
                    Heap Blocks: lossy=23488
                    -> Bitmap Index Scan on idx_brin_op_class_512 (cost=0.00..33.54 rows=10991314 width=0)
(actual time=2.076..2.076 rows=706560 loops=1)
                        Index Cond: ((dt >= '2017-03-01 00:00:00'::timestamp without time zone) AND (dt <
'2017-07-07 00:00:00'::timestamp without time zone))
Planning time: 21.801 ms
Execution time: 739.944 ms
```

Verificar o plano de execução com sinal de igualdade:

```
> EXPLAIN ANALYZE
SELECT avg(temperatura)
  FROM tb_temperatura_log
 WHERE dt = '2017-03-01';
```

QUERY PLAN

```
Aggregate (cost=8.46..8.47 rows=1 width=32) (actual time=0.015..0.015 rows=1 loops=1)
  -> Index Scan using idx_btree on tb_temperatura_log (cost=0.44..8.46 rows=1 width=4) (actual
time=0.008..0.008 rows=1 loops=1)
        Index Cond: (dt = '2017-03-01 00:00:00'::timestamp without time zone)
Planning time: 0.243 ms
Execution time: 0.040 ms
```

Verificando o tamanho dos índices na tabela de teste:

```
> SELECT
  indexname indice,
  pg_size_pretty(pg_relation_size(indexname::text)) tamanho
 FROM pg_indexes
 WHERE tablename = 'tb_temperatura_log'
 ORDER BY pg_relation_size(indexname::text);
```

indice	tamanho
idx_brin_op_class_512	32 kB
idx_brin_op_class_256	40 kB
idx_brin	72 kB
idx_brin_op_class	72 kB
idx_brin_op_class_64	120 kB
idx_btree	674 MB

32.8 Índices Compostos

São aqueles que contêm em sua composição mais de um campo.

Sintaxe:

```
CREATE INDEX nome_do_index  
ON nome_da_tabela (campoX, campoY, campoZ);
```

Verificando o plano de consulta:

```
> EXPLAIN ANALYZE SELECT campol FROM tb_index  
WHERE (campo2 BETWEEN 235 AND 587) AND campo3 = 1000;
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on tb_index (cost=15.22..116.35 rows=1 width=2) (actual time=0.475..0.475 rows=0 loops=1)  
  Recheck Cond: ((campo2 >= 235) AND (campo2 <= 587))  
  Filter: (campo3 = 1000)  
  Rows Removed by Filter: 684  
  Heap Blocks: exact=89  
-> Bitmap Index Scan on idx_tb_index_campo2 (cost=0.00..15.22 rows=693 width=0) (actual time=0.104..0.104  
rows=684 loops=1)  
    Index Cond: ((campo2 >= 235) AND (campo2 <= 587))  
Planning time: 0.158 ms  
Execution time: 0.500 ms
```

Criação de índice composto:

```
> CREATE INDEX idx_tb_index_campo2_campo3 ON tb_index (campo2, campo3);
```

Verificando o plano de consulta:

```
> EXPLAIN ANALYZE SELECT campol FROM tb_index  
WHERE (campo2 BETWEEN 235 AND 587) AND campo3 = 1000;
```

QUERY PLAN

```
-----  
Index Scan using idx_tb_index_campo2_campo3 on tb_index (cost=0.29..20.96 rows=1 width=2) (actual  
time=0.124..0.124 rows=0 loops=1)  
  Index Cond: ((campo2 >= 235) AND (campo2 <= 587) AND (campo3 = 1000))  
Planning time: 0.350 ms  
Execution time: 0.160 ms
```


32.9 Índices Parciais

Índice parcial é um índice que aponta para registros de acordo com uma condição.

Sintaxe:

```
CREATE INDEX nome_do_index ON nome_da_tabela (campo)
WHERE condição;
```

Apagando a tabela do exercício anterior:

```
> DROP TABLE tb_index;
```

Criação da Tabela de Teste (Não Temporária):

```
> CREATE TABLE tb_index(campo1 int);
```

Inserção de 1 Milhão de Registros:

```
> INSERT INTO tb_index SELECT generate_series(1, 1000000);
```

Análise sem Índices de Valores Múltiplos de 19:

```
> EXPLAIN ANALYZE SELECT * FROM tb_index WHERE campo1 % 19 = 0;
```

```
-----
                        QUERY PLAN
-----
Seq Scan on tb_index  (cost=0.00..61065.00 rows=15930 width=4) (actual time=0.021..499.281 rows=157894 loops=1)
  Filter: ((campo1 % 19) = 0)
  Rows Removed by Filter: 2842106
Planning time: 0.080 ms
Execution time: 521.719 ms
```

Por não ter índices foi usada uma busca sequencial (Seq Scan).

Criação de Índice Total :

```
> CREATE INDEX idx_teste_index_total ON tb_index (campo1);
```

Verifica o plano de execução:

```
> EXPLAIN ANALYZE SELECT * FROM tb_index WHERE campol % 19 = 0;
```

```
-----  
QUERY PLAN  
-----  
Seq Scan on tb_index (cost=0.00..58275.00 rows=15000 width=4) (actual time=0.019..392.215 rows=157894 loops=1)  
  Filter: ((campol % 19) = 0)  
    Rows Removed by Filter: 2842106  
  Planning time: 0.337 ms  
  Execution time: 413.200 ms
```

Criação de índice parcial múltiplos de 19:

```
> CREATE INDEX idx_teste_index_19 ON tb_index (campol) WHERE campol % 19 = 0;
```

Análise com valores múltiplos de 19:

```
> EXPLAIN ANALYZE SELECT * FROM tb_index WHERE campol % 19 = 0;
```

```
-----  
QUERY PLAN  
-----  
Index Only Scan using idx_teste_index_19 on tb_index (cost=0.42..538.42 rows=15000 width=4) (actual  
time=0.063..63.894 rows=157894 loops=1)  
  Heap Fetches: 157894  
  Planning time: 0.247 ms  
  Execution time: 87.157 ms
```

Análise com uma consulta de condição diferente de números divisíveis por 19:

```
> EXPLAIN ANALYZE SELECT * FROM tb_index WHERE campol BETWEEN 241 AND 875;
```

```
-----  
QUERY PLAN  
-----  
Index Only Scan using idx_teste_index_total on tb_index (cost=0.43..27.07 rows=632 width=4) (actual  
time=0.018..0.450 rows=635 loops=1)  
  Index Cond: ((campol >= 241) AND (campol <= 875))  
  Heap Fetches: 635  
  Planning time: 0.251 ms  
  Execution time: 0.638 ms
```

Conclusão

Pudemos constatar o que foi dito na teoria; antes da criação dos índices a busca foi sequencial, demorando *521.719 ms*.

Após a criação dos índices, o planejador de consultas já podia contar com eles, optando por usar o índice com maior restrição de valores levando *87.157 ms*, usufruindo de uma busca agora indexada por um índice parcial.

32.10 Fillfactor – Fator de Preenchimento

Fillfactor (fator de preenchimento) para um índice é a porcentagem que determina como o método de indexação encherá as páginas de índices, o quão cheias essas páginas ficarão em porcentagem.

Quanto menor for o valor de *fillfactor*, maior será o tamanho do índice.

Para tabelas estáticas é melhor deixar em 100 (representando 100%) de maneira a manter o tamanho de índice enxuto.

Para tabelas que sofrem muitas alterações um valor de 80 ou menos pode ser mais adequado, mas quanto menor for o fator de preenchimento, mais espaço em disco ocupará.

Para índices B-tree, páginas no nível das folhas são preenchidas com essa porcentagem durante a construção inicial do índice, e também quando estendem o índice à direita (adicionando novos grandes valores). Se as páginas ficarem cheias posteriormente, elas serão divididas, levando a uma degradação aos poucos da eficiência do índice. Os índices B-tree têm como padrão o valor de 90 para fillfactor, mas qualquer inteiro entre 10 e 100 pode ser definido.

Outros métodos de indexação utilizam fillfactor de jeitos diferentes, mas similar entre si. O valor de fillfactor padrão pode variar entre os métodos.

Habilita o temporizador de comandos do psql:

```
> \timing
```

Parte I → Fillfactor = 100

Criação da tabela de testes para fillfactor de índice de 100%:

```
> CREATE TABLE tb_ff100(campo int);
```

Criação do índice com fillfactor igual a 100:

```
> CREATE INDEX idx_ff100 ON tb_ff100 (campo) WITH (fillfactor = 100);
```

Verificando nas informações de estrutura da tabela as informações sobre o índice criado para ela:

```
> \d tb_ff100

      Table "public.tb_ff100"
  Column | Type      | Modifiers
-----+-----+-----
  campo  | integer   |
Indexes:
    "idx_ff100" btree (campo) WITH (fillfactor=100)
```

Inserindo um milhão de registros:

```
> INSERT INTO tb_ff100 (campo) SELECT generate_series(1, 1000000);
```

Média: 3854.282 ms

Verificando o tamanho do índice após o INSERT:

```
> SELECT pg_size_pretty(pg_relation_size('idx_ff100'));

pg_size_pretty
-----
19 MB
```

Atualização de todos os registros:

```
> UPDATE tb_ff100 SET campo = campo + 1;
```

Média: 6826.0742 ms

Verificando o tamanho do índice após o UPDATE:

```
> SELECT pg_size_pretty(pg_relation_size('idx_ff100'));

pg_size_pretty
-----
58 MB
```

Parte II → Fillfactor = 50

Apagando a tabela de fillfactor 100:

```
> DROP TABLE tb_ff100;
```

Criação da tabela de testes para fillfactor de índice de 50%:

```
> CREATE TABLE tb_ff50 (campo int);
```

Criação do índice com fillfactor igual a 50:

```
> CREATE INDEX idx_ff50 ON tb_ff50 (campo) WITH (fillfactor = 50);
```

Verificando nas informações de estrutura da tabela as informações sobre o índice criado para ela:

```
> \d tb_ff50

Table "public.tb_ff50"
Column | Type      | Modifiers
-----+-----+-----
campo  | integer   |
Indexes:
    "idx_ff50" btree (campo) WITH (fillfactor=50)
```

Inserindo um milhão de registros:

```
> INSERT INTO tb_ff50 (campo) SELECT generate_series(1, 1000000);
```

Média: 4256.008 ms

Verificando o tamanho do índice após o INSERT:

```
> SELECT pg_size_pretty(pg_relation_size('idx_ff50'));

pg_size_pretty
-----
39 MB
```

Atualização de todos os registros:

```
> UPDATE tb_ff50 SET campo = campo + 1;
```

Média: 6232.7034 ms

Verificando o tamanho do índice após o UPDATE:

```
> SELECT pg_size_pretty(pg_relation_size('idx_ff50'));
```

```
pg_size_pretty
-----
39 MB
```

Resumo dos Testes		
Fillfactor	100	50
INSERT Inicial (ms)	3854.282	4256.008
Tamanho do Índice após INSERT (MB)	19	39
UPDATE (ms)	6826.0742	6232.7034
Tamanho do Índice após UPDATE (MB)	58	39

32.11 Reconstrução de Índices – REINDEX

O comando `REINDEX` faz a reconstrução de um índice utilizando os dados guardados na tabela do mesmo e substitui a cópia antiga do índice.

É utilizado nas seguintes situações:

- Um índice que acabou ficando “inchado”, que é conter muitas páginas vazias ou quase vazias. Isso pode ocorrer com índices B-tree sob certos padrões incomuns de acesso. `REINDEX` fornece uma maneira de reduzir o consumo de espaço do índice escrevendo uma nova versão do índice sem as páginas mortas;
- Se for alterado algum parâmetro de armazenamento (como `fillfactor`) para um índice, e deseja assegurar que a mudança tenha o efeito completo;
- Um índice feito com a opção `CONCURRENTLY` falhou, deixando um índice “inválido”. O `REINDEX` não vai refazer o índice com a opção `CONCURRENTLY`. Para fazer o índice sem interferir na produção, deve-se apagar o índice (`DROP`) e criar o índice novamente como o comando `CREATE INDEX CONCURRENTLY`.

Sintaxe:

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nome
```

Onde:

INDEX: Índice específico;

TABLE: Todos os índices da tabela específica;

DATABASE: Todos os índices do banco de dados específico;

SYSTEM: Todos os índices em todos os catálogos no âmbito do atual banco de dados;

Apagando a tabela:

```
> DROP TABLE tb_index;
```

Recriando a tabela:

```
> CREATE TABLE tb_index (campo int);
```

Criando índice para a tabela:

```
> CREATE INDEX idx_teste ON tb_index (campo);
```

Inserindo valores na tabela:

```
> INSERT INTO tb_index (campo) SELECT generate_series(1, 1000000);
```

Alterando um padrão de armazenamento do índice; fillfactor:

```
> ALTER INDEX idx_teste SET (fillfactor = 70);
```

Reconstruindo o índice:

```
> REINDEX INDEX idx_teste;
```


32.12 CLUSTER – Índices Clusterizados

O comando `CLUSTER` agrupa uma tabela de acordo com um índice de modo a aumentar a performance no banco de dados.

A tabela é fisicamente reordenada baseando-se na informação do índice. O agrupamento é feito somente uma vez: após ser atualizada, as atualizações feitas nas linhas da tabela não seguirão o agrupamento, ou seja, não será feita nenhuma tentativa para armazenar as linhas novas ou atualizadas na ordem do índice. Se for desejado, a tabela pode ser reagrupada periodicamente executando este comando novamente.

Sintaxe:

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

Apagando a tabela de teste:

```
> DROP TABLE tb_index;
```

Recriando a tabela:

```
> CREATE TEMP TABLE tb_index(
    id serial PRIMARY KEY,
    cor text);
```

Inserindo registros:

```
> INSERT INTO tb_index (cor) VALUES
    ('Laranja'),
    ('Preto'),
    ('Branco'),
    ('Azul'),
    ('Amarelo'),
    ('Vermelho'),
    ('Verde'),
    ('Cinza');
```

Verificando os registros da tabela:

```
> TABLE tb_index;
```

id	cor
1	Laranja
2	Preto
3	Branco
4	Azul
5	Amarelo
6	Vermelho
7	Verde
8	Cinza

Fazendo uma atualização:

```
> UPDATE tb_index SET cor = 'Vinho' WHERE id = 6;
```

Verificando os registros da tabela:

```
> TABLE tb_index;
```

id	cor
1	Laranja
2	Preto
3	Branco
4	Azul
5	Amarelo
7	Verde
8	Cinza
6	Vinho

Verificando na estrutura da tabela o nome do índice para clusterização:

```
> \d tb_index
          Table "pg_temp_2.tb_index"
  Column | Type          | Modifiers
-----+-----+-----
 id      | integer       | not null default nextval('tb_index_id_seq'::regclass)
 cor     | text          |
Indexes:
    "tb_index_pkey" PRIMARY KEY, btree (id)
```

Clusterização:

```
> CLUSTER tb_index USING tb_index_pkey;
```

Verificando os dados da tabela após a clusterização:

```
> TABLE tb_index;
```

<i>id</i>	<i>cor</i>
1	Laranja
2	Preto
3	Branco
4	Azul
5	Amarelo
6	Vinho
7	Verde
8	Cinza

32.13 Excluindo um Índice

Sintaxe:

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...]
[ CASCADE | RESTRICT ]
```

32.13.1 Antes de Excluir um Índice...

Não remova um índice de seu banco sem antes saber o quão útil ele é.

Tabela de Estatísticas de Uso de Índices:

```
> SELECT indexrelname, relname, idx_scan, idx_tup_read, idx_tup_fetch
FROM pg_stat_user_indexes;
```

<i>indexrelname</i>	<i>relname</i>	<i>idx_scan</i>	<i>idx_tup_read</i>	<i>idx_tup_fetch</i>
<i>tb_index_pkey</i>	<i>tb_index</i>	2	9	10

indexrelname: Nome do índice;
relname: Nome da tabela à qual o índice pertence;
idx_scan: Quantas vezes o índice foi usado;
idx_tup_read: Quantas tuplas o índice leu;
idx_tup_fetch: Quantas tuplas o índice recuperou.

Ou seja, se um índice já existe há um certo tempo e não tem sido usado, será necessário replanejar o mesmo, devido à sua inutilidade.

É possível notar também que o índice parcial que foi criado em comandos anteriores aparece como o mais usado de nossos exercícios.

Exclusão de Índice:

```
> DROP INDEX idx_ff50;
```

33 Modelos de Banco de Dados – Templates

- Sobre Templates

33.1 Sobre Templates

O comando `CREATE DATABASE` funciona copiando um banco de dados existente.

Por padrão, ele copia o banco de dados de sistema chamado `template1` para criação de outros bancos de dados.

Se forem adicionados objetos no `template1`, esses objetos serão copiados para bancos criados posteriormente.

Por exemplo, se for instalada uma linguagem procedural como a PL/pgSQL no `template1`, estará automaticamente disponível em bases de dados criadas posteriormente.

Há um outro banco de dados padrão do sistema chamado `template0`, que são, apenas os objetos padrões pré definidos pela versão do PostgreSQL.

`template0` é uma base de dados imutável e que não aceita conexões.

Sintaxe:

```
CREATE DATABASE nome_do_banco_de_dados TEMPLATE banco_modelo;
```

Conexão ao banco padrão `template1`:

```
> \c template1
```

Criação de uma simples tabela dentro de `template1`:

```
> CREATE TABLE tb_exemplo (campo int);
```

Inserção de valores na tabela criada dentro de `template1`:

```
> INSERT INTO tb_exemplo VALUES (1), (2), (3), (4), (5);
```

Criação de um novo banco de dados:

```
> CREATE DATABASE xyz;
```

Conectando ao novo banco de dados:

```
> \c xyz
```

Visualizando a existência de tabelas dentro do banco recém-criado:

```
> \d
```

```

      List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | tb_exemplo     | table | postgres
```

Consulta na tabela existente:

```
> SELECT * FROM tb_exemplo;
```

```

 campo
-----
      1
      2
      3
      4
      5
```

A partir de agora todos os bancos novos criados no sistema terão a tabela `template1`.

Como pôde ser comprovado, ao se criar objetos dentro de `template1`, esses objetos serão transmitidos para bancos de dados que forem criados posteriormente, a não ser que se queira um outro banco de dados como modelo, ou mesmo um banco de dados puro, como no exemplo a seguir:

Criação de um banco de dados puro:

```
> CREATE DATABASE novo_banco TEMPLATE template0;
```

O banco de dados criado é um banco de dados puro, pois não há objetos no mesmo, pois seu modelo foi o `template0`.

Também podemos copiar uma base de dados tomando outra como modelo:

```
> CREATE DATABASE abc TEMPLATE xyz;
```

Ao se conectar nesse banco de dados poderá ser constatado que possui os mesmos objetos e registros que seu modelo.

Para prosseguir com o aprendizado do PostgreSQL, após as devidas verificações de resultados até aqui apresentados serem comprovadas, é recomendável fazer uma “limpeza” do que foi feito para fins de exemplos:

Conectando novamente a template1:

```
> \c template1
```

Apagando a tabela criada:

```
> DROP TABLE tb_exemplo;
```

Apagando os bancos criados:

```
> DROP DATABASE xyz;
```

```
> DROP DATABASE abc;
```


34 Herança

- Sobre Herança de Tabelas
- Herança Múltipla

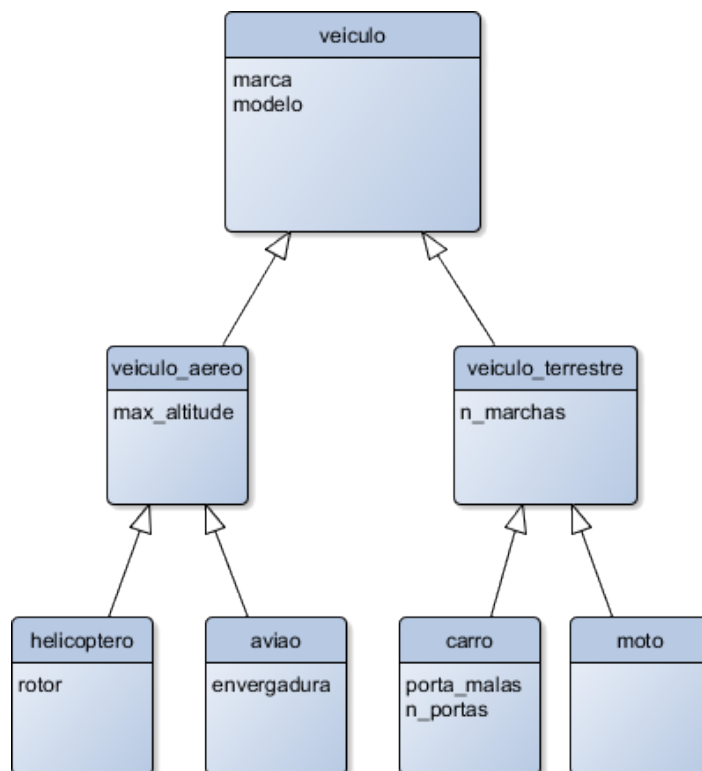
34.1 Sobre Herança de Tabelas

O PostgreSQL implementa herança de tabela, que pode ser muito útil para um projeto de banco de dados.

O padrão SQL:1999 define um tipo de recurso de herança que difere em muitos aspectos aos descritos aqui.

Caso I

Uma empresa administra um cadastro de tipos diferentes de veículos.
As tabelas mãe não contêm dados, só servem de modelo para suas filhas.



Criação da tabela que dá origem a todas outras:

```
> CREATE TEMP TABLE tb_veiculo(  
    marca VARCHAR(30),  
    modelo VARCHAR(40));
```

Criação da tabela filha que dará origem a outras tabelas filhas para veículos aéreos:

```
> CREATE TEMP TABLE tb_veiculo_aereo(  
    max_altitude REAL)  
    INHERITS (tb_veiculo);
```

Criação da tabela filha que dará origem a outras tabelas filhas para veículos terrestres:

```
> CREATE TEMP TABLE tb_veiculo_terrestre(  
    n_marchas int2  
    INHERITS (tb_veiculo);
```

Criação da tabela de cadastro de helicópteros:

```
> CREATE TEMP TABLE tb_helicoptero(  
    rotor varchar(10),  
    id serial PRIMARY KEY  
    INHERITS (tb_veiculo_aereo);
```

Criação da tabela de cadastro de aviões:

```
> CREATE TEMP TABLE tb_aviao(  
    envergadura real,  
    id serial PRIMARY KEY  
    INHERITS (tb_veiculo_aereo);
```

Criação da tabela de cadastro de carros:

```
> CREATE TEMP TABLE tb_carro(  
    porta_malas real,  
    n_portas int2,  
    id serial PRIMARY KEY  
    INHERITS (tb_veiculo_terrestre);
```

Criação da tabela de cadastro de motos:

```
> CREATE TEMP TABLE tb_moto(  
    id serial PRIMARY KEY  
    ) INHERITS (tb_veiculo_terrestre);
```

Descrição da tabela de origem:

```
> \d tb_veiculo  
  
      Table "pg_temp_2.tb_veiculo"  
  Column |          Type          | Modifiers  
-----+-----+-----  
  marca  | character varying(30) |  
  modelo | character varying(40) |  
Number of child tables: 2 (Use \d+ to list them.)
```

Na própria descrição há uma dica para utilizarmos a opção de mais detalhes acrescentando o sinal de positivo para verificarmos quais são as tabelas filhas.

Descrição com maiores detalhes de tb_veiculo:

```
> \d+ tb_veiculo
```

```
Table "pg_temp_2.tb_veiculo"
Column |          Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
marca  | character varying(30) |           | extended |               |
modelo | character varying(40) |           | extended |               |
Child tables: tb_veiculo_aereo,
               tb_veiculo_terrestre
```

Descrição com maiores detalhes de tb_veiculo_terrestre:

```
> \d+ tb_veiculo_terrestre
```

```
Table "pg_temp_2.tb_veiculo_terrestre"
Column |          Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
marca  | character varying(30) |           | extended |               |
modelo | character varying(40) |           | extended |               |
n_marchas | smallint              |           | plain   |               |
Inherits: tb_veiculo
Child tables: tb_carro,
               tb_moto
```

Descrição com maiores detalhes de tb_veiculo_aéreo:

```
> \d+ tb_veiculo_aereo
```

```
Table "pg_temp_2.tb_veiculo_aereo"
Column |          Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
marca  | character varying(30) |           | extended |               |
modelo | character varying(40) |           | extended |               |
max_altitude | real                |           | plain   |               |
Inherits: tb_veiculo
Child tables: tb_aviao,
               tb_helicoptero
```

Descrição com maiores detalhes de tb_helicoptero:

```
> \d+ tb_helicoptero
```

```
...
```

```
Inherits: tb_veiculo_aereo
```

Descrição com maiores detalhes de tb_aviao:

```
> \d+ tb_aviao
```

```
...
```

```
Inherits: tb_veiculo_aereo
```

Descrição com maiores detalhes de tb_carro:

```
> \d+ tb_carro
. . .
Inherits: tb_veiculo_terrestre
```

Descrição com maiores detalhes de tb_moto:

```
> \d+ tb_moto
. . .
Inherits: tb_veiculo_terrestre
```

Inserindo valores para a tabela de helicópteros:

```
> INSERT INTO tb_helicoptero (marca, modelo, max_altitude, rotor) VALUES
  ('Bell Helicopter', 'OH-58 Kiowa', 15000, 'coaxial'),
  ('Boeing Rotorcraft Systems', 'CH-47 Chinook', 18500, 'tandem');
```

Inserindo valores para a tabela de aviões:

```
> INSERT INTO tb_aviao (marca, modelo, max_altitude, envergadura) VALUES
  ('Northrop Grumman/Boeing', 'B-2 Spirit', 50000, 170),
  ('Airbus Defence and Space', 'Eurofighter Typhoon', 65000, 35.9);
```

Inserindo valores para a tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo, n_marchas, porta_malas, n_portas) VALUES
  ('Fiat', '147', 5, 350, 2),
  ('Ford', 'Corcel', 4, 380, 4);
```

Inserindo valores para a tabela de motos:

```
> INSERT INTO tb_moto (marca, modelo, n_marchas) VALUES
  ('Kawasaki', 'Z300', 6),
  ('Harley Davidson', 'V-Rod', 5);
```

Selecionando os dados da tabela de origem:

```
> SELECT marca, modelo FROM tb_veiculo;
```

marca	modelo
Bell Helicopter	OH-58 Kiowa
Boeing Rotorcraft Systems	CH-47 Chinook
Northrop Grumman/Boeing	B-2 Spirit
Airbus Defence and Space	Eurofighter Typhoon
Fiat	147
Ford	Corcel
Kawasaki	Z300
Harley Davidson	V-Rod

Todos os dados exibidos são de tabelas filhas das filhas, nenhum é da própria tabela.

Para exibir somente os dados da própria tabela utilizamos a cláusula ONLY:

```
> SELECT marca, modelo FROM ONLY tb_veiculo;
```

marca	modelo
-------	--------

Caso II

Um cadastro de cidades, sendo que para uma maior agilidade na busca foi criada uma tabela filha somente de capitais.

A tabela de origem vai ter registros próprios.

Criação da tabela de cidades:

```
> CREATE TEMP TABLE tb_cidade(  
    id SERIAL PRIMARY KEY,  
    nome TEXT,  
    uf CHAR(2),  
    populacao INT);
```

Criação da tabela de capitais:

```
> CREATE TEMP TABLE tb_capital() INHERITS (tb_cidade);
```

Inserindo valores na tabela de cidades:

```
> INSERT INTO tb_cidade (nome, uf, populacao) VALUES  
    ('Santo André', 'SP', 707613),  
    ('São José dos Campos', 'SP', 688597);
```

Inserindo valores na tabela de capitais:

```
> INSERT INTO tb_capital (nome, uf, população)
  VALUES ('São Paulo', 'SP', 11895893);
```

Selecionando todas as cidades (inclusive as capitais):

```
> SELECT * FROM tb_cidade;
```

id	nome	uf	populacao
1	Santo André	SP	707613
2	São José dos Campos	SP	688597
3	São Paulo	SP	11895893

Selecionando apenas os registros da tabela tb_cidade:

```
> SELECT * FROM ONLY tb_cidade;
```

id	nome	uf	populacao
1	Santo André	SP	707613
2	São José dos Campos	SP	688597

Selecionando os dados da tabela de capitais:

```
> SELECT * FROM tb_capital;
```

id	nome	uf	populacao
3	São Paulo	SP	11895893

Ao alterarmos o parâmetro de sessão `sql_inheritance` para `off`, temos que explicitar quando queremos também os dados das tabelas filhas:

```
> SET sql_inheritance = off;
```

Na tabela de cidade já não aparece mais os dados da tabela de capitais:

```
> TABLE tb_cidade;
```

id	nome	uf	populacao
1	Santo André	SP	707613
2	São José dos Campos	SP	688597

Adicionando um asterisco ao nome da tabela explicitamos que desejamos também os dados das tabelas filhas:

```
> TABLE tb_cidade*;
```

<i>id</i>	<i>nome</i>	<i>uf</i>	<i>populacao</i>
1	Santo André	SP	707613
2	São José dos Campos	SP	688597
3	São Paulo	SP	11895893

Habilitando o padrão novamente, como é seu comportamento padrão:

```
> SET sql_inheritance = on;
```


34.2 Herança Múltipla

Algumas linguagens de programação orientadas a objeto como Python e C++ implementam de forma nativa e fácil o recurso de herança múltipla de forma a facilitar a modelagem de acordo com um determinado conjunto de entidades em que se estabelece a relação entre entidades primárias, as quais são herdadas de entidades secundárias em forma de classes.

O PostgreSQL não somente implementa a herança de tabelas de forma similar à herança de classes, como também permite herança múltipla.

Primeira tabela primária:

```
> CREATE TABLE tb_a(  
    a_1 INT,  
    a_2 INT,  
    a_3 INT);
```

Segunda tabela primária:

```
> CREATE TABLE tb_b(  
    b_1 INT,  
    b_2 INT);
```

Tabela derivada de ambas as tabelas primárias:

```
> CREATE TABLE tb_ab () INHERITS (tb_a, tb_b);
```

A tabela que herda os campos não implementa nenhum campo próprio.

Note que as tabelas que ela se deriva são especificadas entre parênteses e delimitadas por vírgula.

Verificando a estrutura da nova tabela:

```
> \d tb_ab
```

```
Table "public.tb_ab"  
Column | Type      | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
a_1    | integer   |           |          |  
a_2    | integer   |           |          |  
a_3    | integer   |           |          |  
b_1    | integer   |           |          |  
b_2    | integer   |           |          |  
Inherits: tb_a,  
          tb_b
```

Observa-se logo na parte de baixo da descrição as tabelas que derivaram a nova tabela.

Consultando as colunas e tipos de dados no catálogo:

```
> SELECT column_name, data_type
   FROM information_schema.columns
   WHERE table_name = 'tb_ab';
```

<i>column_name</i>	<i>data_type</i>
<i>a_1</i>	<i>integer</i>
<i>a_2</i>	<i>integer</i>
<i>a_3</i>	<i>integer</i>
<i>b_1</i>	<i>integer</i>
<i>b_2</i>	<i>integer</i>

Campos e seus respectivos tipos originados das tabelas primárias.

35 Relacionamentos

- Cardinalidade
- Relacionamento 1:1 – Um para Um
- Relacionamento 1:n – Um para Muitos
- Relacionamento n:n – Muitos para Muitos
- Relacionamento Ternário

35.1 Cardinalidade

Cardinalidade em bancos de dados é sobre o grau de relacionamento entre duas entidades.

Esse grau de relacionamento pode ser um dos três: **1:1**, **1:n** e **n:n**.

Exemplos:



Uma pessoa dirige um carro.



Uma pessoa dirige vários carros.

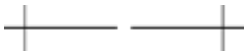
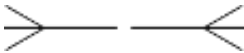
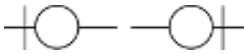
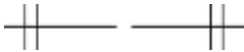
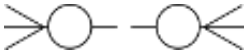
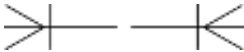


Várias pessoas dirigem vários carros.

Nos exemplos dados, as entidades “Pessoa” e “Carro” tem o relacionamento “Dirigir”. Ou seja, Pessoa Dirige Carro. A cardinalidade trata de quantos para quantos se dá esse relacionamento.

35.1.1 Simbologia

Para determinar a cardinalidade em um relacionamento representado por um diagrama existem alguns símbolos para facilitar o entendimento.

Símbolo	Descrição
	1: Um.
	n: Vários.
	(0, 1): Zero ou um.
	(1, 1): Um somente.
	(0, n): Zero ou vários.
	(1, n): Um ou vários.

35.1.2 Cardinalidade Mínima e Cardinalidade Máxima

Dado o relacionamento Pessoa (1, 1) Dirige (0, n) Carro, do lado de Pessoa a cardinalidade mínima é um e sua cardinalidade máxima também. Do lado de Carro a cardinalidade mínima é zero e a máxima é n. Em outras palavras, uma pessoa pode dirigir nenhum ou vários carros.

Quando só se representa a cardinalidade máxima, significa que a cardinalidade mínima é zero. Isso faz o relacionamento ser opcional para ambos os lados.

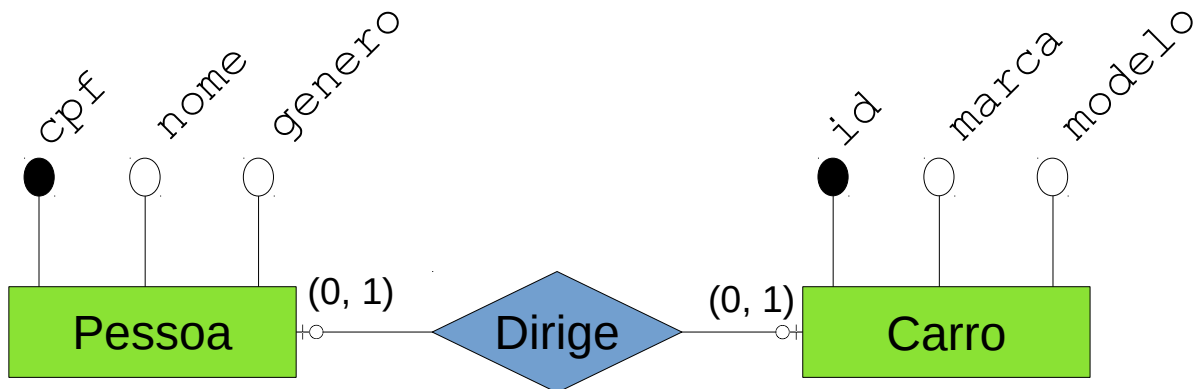
35.2 Relacionamento 1:1 – Um para Um

35.2.1 Relacionamento (0, 1):(0, 1)

Relacionamento opcional para ambos os lados.

A chave estrangeira deve estar em qualquer um dos lados.

Modelo Conceitual



Modelo Lógico

Pessoa			
cpf	nome	genero	carro_fk
11111111111	Chiquinho da Silva	m	1
22222222222	Maria Santos	f	2
33333333333	Zé das Coves	m	3
44444444444	Bertolina Chaves	f	NULL
55555555555	Crivélío Almeida	m	NULL

Carro		
id	marca	modelo
1	Fiat	147
2	Volkswagen	Variant
3	Ford	Corcel I
4	Chevrolet	Chevette
5	Simca	Chambord

Relacionamento não obrigatório e chave estrangeira na tabela Pessoa fazendo referência à tabela Carro.

Modelo Físico

Criação de tipo personalizado:

```
> CREATE TYPE tp_genero AS ENUM ('f', 'm');
```

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text);
```

Criação da tabela de pessoas:

```
> CREATE TABLE tb_pessoa(  
    id int8 primary key, -- cpf  
    nome text,  
    genero tp_genero,  
    carro_fk int  
        REFERENCES tb_carro (id) -- Chave estrangeira  
        UNIQUE -- Evita que outra pessoa utilize um carro já alocado  
);
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo) VALUES  
    ('Fiat', '147'),  
    ('Volkswagen', 'Variant'),  
    ('Ford', 'Corcel I'),  
    ('Chevrolet', 'Chevette'),  
    ('Simca', 'Chambord');
```

Inserir dados na tabela de pessoas:

```
> INSERT INTO tb_pessoa (id, nome, genero, carro_fk) VALUES  
    (11111111111, 'Chiquinho da Silva', 'm', 1),  
    (22222222222, 'Maria Santos', 'f', 2),  
    (33333333333, 'Zé das Coves', 'm', 3),  
    (44444444444, 'Bertolina Chaves', 'f', NULL),  
    (55555555555, 'Crivélcio Almeida', 'm', NULL);
```

Selecionando os dados:

```
> SELECT
  p.id "CPF",
  p.nome "Nome",
  p.genero "Gênero",
  c.id "ID Carro",
  c.marca "Marca",
  c.modelo "Modelo"
FROM tb_pessoa p
LEFT JOIN tb_carro c
  ON (p.carro_fk = c.id);
```

CPF	Nome	Gênero	ID Carro	Marca	Modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
44444444444	Bertolina Chaves	f			
55555555555	Crivélío Almeida	m			

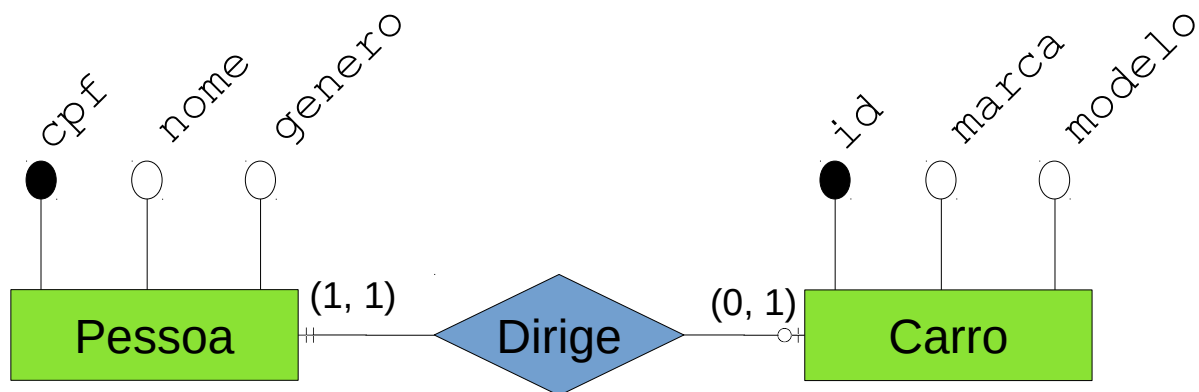
Apagar as tabelas criadas:

```
> DROP TABLE tb_carro, tb_pessoa;
```


35.2.2 Relacionamento (1, 1):(0, 1)

Relacionamento obrigatório para a entidade à direita.

Modelo Conceitual



Modelo Lógico

Pessoa		
cpf	nome	genero
11111111111	Chiquinho da Silva	m
22222222222	Maria Santos	f
33333333333	Zé das Coves	m
44444444444	Bertolina Chaves	f
55555555555	Crivélío Almeida	m

Carro			
id	marca	modelo	pessoa_fk
1	Fiat	147	11111111111
2	Volkswagen	Variant	22222222222
3	Ford	Corcel I	33333333333

Não é aconselhável, mas ambas as tabelas poderiam ser uma só da seguinte forma:

PessoaCarro					
cpf	nome	genero	carro_id	marca	modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
44444444444	Bertolina Chaves	f	NULL	NULL	NULL
55555555555	Crivélío Almeida	m	NULL	NULL	NULL

O relacionamento obrigatório é à esquerda ← direita (Carro), na tabela aglutinada mostra como ficaria para Bertolina e Crivélío que não têm um carro relacionado. Portanto, em uma única tabela esses valores seriam nulos.

Modelo Físico

Criação da tabela de pessoas:

```
> CREATE TABLE tb_pessoa(  
    id int8 primary key, -- cpf  
    nome text,  
    genero tp_genero);
```

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text,  
    pessoa_fk int8  
        REFERENCES tb_pessoa (id) -- Chave estrangeira  
        NOT NULL -- Preenchimento obrigatório  
        UNIQUE -- Único  
);
```

Inserir dados na tabela de pessoas:

```
> INSERT INTO tb_pessoa (id, nome, genero) VALUES  
    (11111111111, 'Chiquinho da Silva', 'm'),  
    (22222222222, 'Maria Santos', 'f'),  
    (33333333333, 'Zé das Coves', 'm'),  
    (44444444444, 'Bertolina Chaves', 'f'),  
    (55555555555, 'Crivélcio Almeida', 'm');
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo, pessoa_fk) VALUES  
    ('Fiat', '147', 11111111111),  
    ('Volkswagen', 'Variant', 22222222222),  
    ('Ford', 'Corcel I', 33333333333);
```

Selecionando os dados:

```
> SELECT
  p.id "CPF",
  p.nome "Nome",
  p.genero "Gênero",
  c.id "ID Carro",
  c.marca "Marca",
  c.modelo "Modelo"
FROM tb_pessoa p
LEFT JOIN tb_carro c
  ON (p.id = c.pessoa_fk);
```

CPF	Nome	Gênero	ID Carro	Marca	Modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
44444444444	Bertolina Chaves	f			
55555555555	Crivélcio Almeida	m			

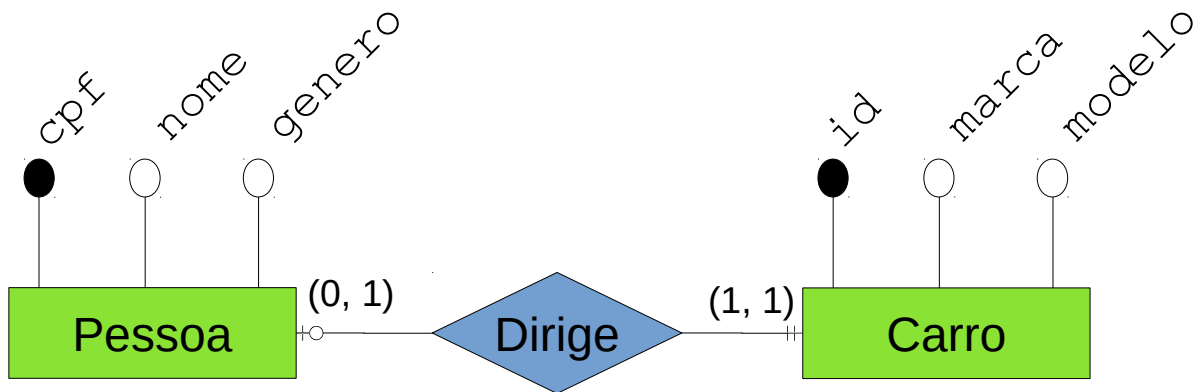
Apagar as tabelas criadas:

```
> DROP TABLE tb_carro, tb_pessoa;
```

35.2.3 Relacionamento (0, 1):(1, 1)

Relacionamento obrigatório para a entidade à esquerda.

Modelo Conceitual



Modelo Lógico

Pessoa			
<u>cpf</u>	nome	genero	carro_fk
11111111111	Chiquinho da Silva	m	1
22222222222	Maria Santos	f	2
33333333333	Zé das Coves	m	3

Carro		
<u>id</u>	marca	modelo
1	Fiat	147
2	Volkswagen	Variant
3	Ford	Corcel I
4	Chevrolet	Chevette
5	Simca	Chambord

Se fosse uma tabela só:

CarroPessoa					
<u>id</u>	marca	modelo	cpf	nome	genero
1	Fiat	147	11111111111	Chiquinho da Silva	m
2	Volkswagen	Variant	22222222222	Maria Santos	f
3	Ford	Corcel I	33333333333	Zé das Coves	m
4	Chevrolet	Chevette	NULL	NULL	NULL
5	Simca	Chambord	NULL	NULL	NULL

Assim como no exemplo anterior, para registros que não têm correspondência na outra tabela, se fosse uma única tabela teriam campos nulos.

Modelo Físico

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text);
```

Criação da tabela de pessoas:

```
> CREATE TABLE tb_pessoa(  
    id int8 primary key, -- cpf  
    nome text,  
    genero tp_genero,  
    carro_fk int  
        REFERENCES tb_carro (id) -- Chave estrangeira  
        NOT NULL -- Preenchimento obrigatório  
        UNIQUE -- Evita que um carro seja associado a mais de uma pessoa  
);
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo) VALUES  
    ('Fiat', '147'),  
    ('Volkswagen', 'Variant'),  
    ('Ford', 'Corcel I'),  
    ('Chevrolet', 'Chevette'),  
    ('Simca', 'Chambord');
```

Inserir dados na tabela de pessoas:

```
> INSERT INTO tb_pessoa (id, nome, genero, carro_fk) VALUES  
    (1111111111, 'Chiquinho da Silva', 'm', 1),  
    (2222222222, 'Maria Santos', 'f', 2),  
    (3333333333, 'Zé das Coves', 'm', 3);
```

Selecionando os dados:

```
> SELECT
  c.id "ID Carro",
  c.marca "Marca",
  c.modelo "Modelo",
  p.id "CPF",
  p.nome "Nome",
  p.genero "Gênero"
FROM tb_carro c
LEFT JOIN tb_pessoa p
  ON (c.id = p.carro_fk);
```

ID Carro	Marca	Modelo	CPF	Nome	Gênero
1	Fiat	147	11111111111	Chiquinho da Silva	m
2	Volkswagen	Variant	22222222222	Maria Santos	f
3	Ford	Corcel I	33333333333	Zé das Coves	m
5	Simca	Chambord			
4	Chevrolet	Chevette			

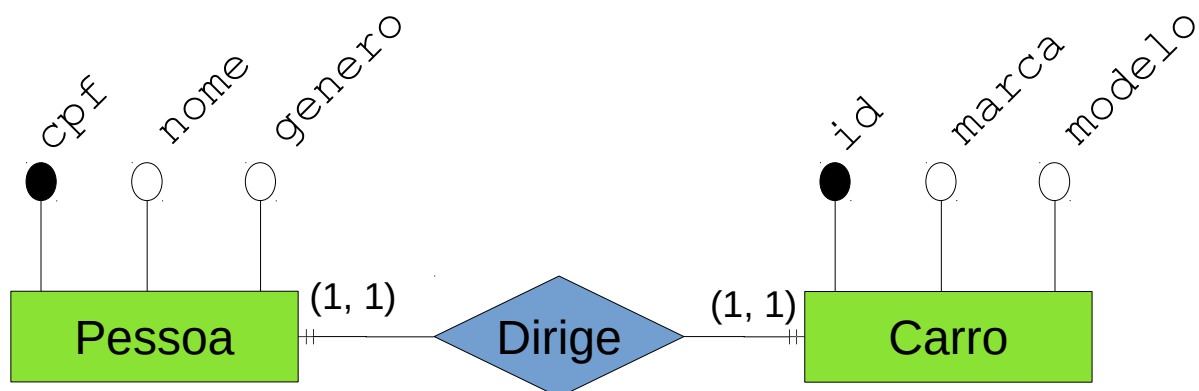
Apagar as tabelas criadas:

```
> DROP TABLE tb_carro, tb_pessoa;
```

35.2.4 Relacionamento (1, 1):(1, 1)

Relacionamento obrigatório para ambos os lados.
A chave estrangeira deve estar em qualquer um dos lados.

Modelo Conceitual



Modelo Lógico

Pessoa			
cpf	nome	genero	carro_fk
11111111111	Chiquinho da Silva	m	1
22222222222	Maria Santos	f	2
33333333333	Zé das Coves	m	3
44444444444	Bertolina Chaves	f	NULL
55555555555	Crivélío Almeida	m	NULL

Carro		
id	marca	modelo
1	Fiat	147
2	Volkswagen	Variant
3	Ford	Corcel I
4	Chevrolet	Chevette
5	Simca	Chambord

Se fosse uma tabela só:

CarroPessoa					
id	marca	modelo	cpf	nome	genero
1	Fiat	147	11111111111	Chiquinho da Silva	m
2	Volkswagen	Variant	22222222222	Maria Santos	f
3	Ford	Corcel I	33333333333	Zé das Coves	m
4	Chevrolet	Chevette	44444444444	Bertolina Chaves	f
5	Simca	Chambord	55555555555	Crivélío Almeida	m

Quando o relacionamento é obrigatório para ambos os lados, se fosse uma única tabela não haveriam campos nulos.

Modelo Físico

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text);
```

Criação da tabela de pessoas:

```
> CREATE TABLE tb_pessoa(  
    id int8 primary key, -- cpf  
    nome text,  
    genero tp_genero,  
    carro_fk int  
        REFERENCES tb_carro (id) -- Chave estrangeira  
        NOT NULL -- Preenchimento obrigatório  
        UNIQUE -- Evita que um carro seja associado a mais de uma pessoa  
);
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo) VALUES  
    ('Fiat', '147'),  
    ('Volkswagen', 'Variant'),  
    ('Ford', 'Corcel I'),  
    ('Chevrolet', 'Chevette'),  
    ('Simca', 'Chambord');
```

Inserir dados na tabela de pessoas:

```
> INSERT INTO tb_pessoa (id, nome, genero, carro_fk) VALUES  
    (11111111111, 'Chiquinho da Silva', 'm', 1),  
    (22222222222, 'Maria Santos', 'f', 2),  
    (33333333333, 'Zé das Coves', 'm', 3),  
    (44444444444, 'Bertolina Chaves', 'f', 4),  
    (55555555555, 'Crivélcio Almeida', 'm', 5);
```


Selecionando os dados:

```
> SELECT
  c.id "ID Carro",
  c.marca "Marca",
  c.modelo "Modelo",
  p.id "CPF",
  p.nome "Nome",
  p.genero "Gênero"
FROM tb_carro c
INNER JOIN tb_pessoa p
  ON (c.id = p.carro_fk);
```

ID Carro	Marca	Modelo	CPF	Nome	Gênero
1	Fiat	147	11111111111	Chiquinho da Silva	m
2	Volkswagen	Variant	22222222222	Maria Santos	f
3	Ford	Corcel I	33333333333	Zé das Coves	m
4	Chevrolet	Chevette	44444444444	Bertolina Chaves	f
5	Simca	Chambord	55555555555	Crivélío Almeida	m

Apagar as tabelas criadas:

```
> DROP TABLE tb_carro, tb_pessoa;
```

35.2.5 Cardinalidade 1:1 Como Estratégia de Particionamento

Há determinadas entidades que tem atributos que são muito mais utilizados do que outros.

Então divide-se a tabela em duas ou mais separando seus atributos como se fossem grupos de forma a resultar em um particionamento vertical.

Se uma tabela que representa essa entidade puder ser dividida pode trazer benefícios:

- Manutenção: Para vacuum / autovacuum e atualização de estatísticas do banco, acaba sendo facilitados por processar tabelas menores;
- Redução de IO de disco;
- Redução de *locks* (travas) permitindo operações concorrentes com mais facilidade.

Criação da tabela de pessoa física e documentos comuns:

```
> CREATE TABLE tb_pf_doc_comuns(  
  id int8 primary key, -- cpf  
  nome varchar(40) not null,  
  sobrenome varchar(200) not null,  
  rg int4 not null,  
  rg_digito char(1) not null,  
  titulo_eleitor int8 not null,  
  data_nascto date not null,  
  genero tp_genero not null);
```

Criação da tabela de pessoa física e documentos extras:

```
> CREATE TABLE tb_pf_doc_extras(  
  id int8 primary key -- cpf  
    references tb_pf_doc_comuns(id),  
  reservista int4,  
  ctps int4,  
  ctps_serie int2,  
  habilitacao int8);
```

Inserir dados na tabela de pessoa física e documentos comuns:

```
> INSERT INTO tb_pf_doc_comuns (id, nome, sobrenome, rg, rg_digito, titulo_eleitor,  
data_nascto, genero)  
VALUES  
(111111111111, 'Chiquinho', 'da Silva', 11111111, '1', 111111111111, '05/05/1950', 'm');
```

Inserir dados na tabela de pessoa física e documentos extras:

```
> INSERT INTO tb_pf_doc_extras (id, reservista) VALUES (111111111111, 111111);
```

Selecionando os dados:

```
> SELECT
  c.id "CPF",
  c.nome "Nome",
  c.sobrenome "Sobrenome",
  c.rg||'-'||c.rg_digito "RG",
  c.titulo_eleitor "Título de Eleitor",
  c.data_nascto "Data de Nascimento",
  c.genero "Gênero",
  e.reservista "Reservista",
  e.ctps "CTPS",
  e.ctpsSerie "Série CTPS",
  e.habilitacao "Habilitação"
FROM tb_pf_doc_comuns c
INNER JOIN tb_pf_doc_extras e
  ON (c.id = e.id);
```

```
-[ RECORD 1 ]-----+-----
CPF           | 11111111111
Nome          | Chiquinho
Sobrenome     | da Silva
RG            | 11111111-1
Título de Eleitor | 1111111111111
Data de Nascimento | 1950-05-05
Gênero        | m
Reservista    | 111111
CTPS          |
Série CTPS    |
Habilitação   |
```

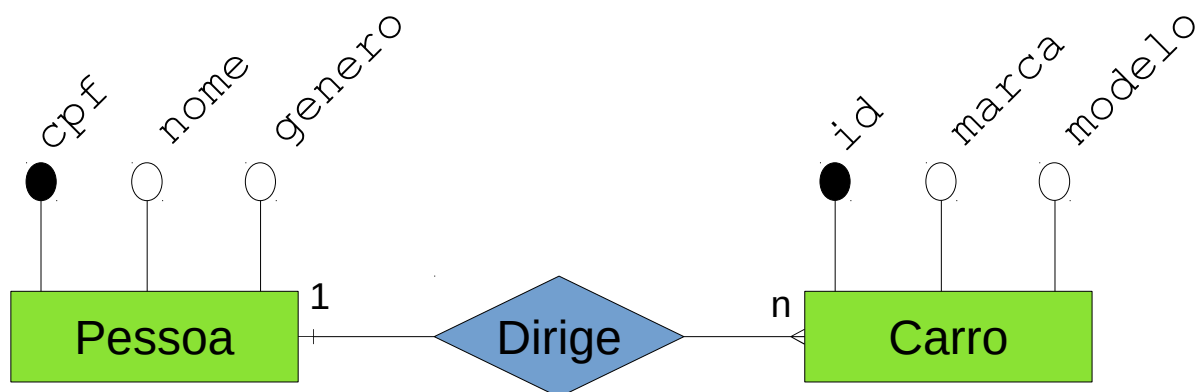
35.3 Relacionamento 1:n – Um para Muitos

Uma instância de uma entidade se relaciona com várias instâncias de outra entidade.

35.3.1 Relacionamento (0, 1):(0, n)

Relacionamento opcional para ambos os lados.

Modelo Conceitual



Modelo Lógico

Pessoa		
<u>cpf</u>	nome	genero
11111111111	Chiquinho da Silva	m
22222222222	Maria Santos	f
33333333333	Zé das Coves	m
44444444444	Bertolina Chaves	f
55555555555	Crivélío Almeida	m

Carro			
<u>id</u>	marca	modelo	pessoa_fk
1	Fiat	147	11111111111
2	Volkswagen	Variant	22222222222
3	Ford	Corcel I	33333333333
4	Chevrolet	Chevette	11111111111
5	Simca	Chambord	NULL

Pessoa é a entidade referenciada e Carro é a entidade referenciadora.

Em uma tabela de uma entidade referenciadora, em sua chave estrangeira podem haver valores repetidos.

PessoaCarro					
<u>cpf</u>	nome	genero	carro_id	marca	modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
11111111111	Chiquinho da Silva	m	4	Chevrolet	Chevette
NULL	NULL	NULL	5	Simca	Chambord

Modelo Físico

Criação da tabela de pessoas:

```
> CREATE TABLE tb_pessoa(  
    id int8 primary key, -- cpf  
    nome text,  
    genero tp_genero);
```

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text,  
    pessoa_fk int8  
        REFERENCES tb_pessoa (id) -- Chave estrangeira  
);
```

Inserir dados na tabela de pessoas:

```
> INSERT INTO tb_pessoa (id, nome, genero) VALUES  
    (11111111111, 'Chiquinho da Silva', 'm'),  
    (22222222222, 'Maria Santos', 'f'),  
    (33333333333, 'Zé das Coves', 'm'),  
    (44444444444, 'Bertolina Chaves', 'f'),  
    (55555555555, 'Crivélío Almeida', 'm');
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo, pessoa_fk) VALUES  
    ('Fiat', '147', 11111111111),  
    ('Volkswagen', 'Variant', 22222222222),  
    ('Ford', 'Corcel I', 33333333333),  
    ('Chevrolet', 'Chevette', 11111111111),  
    ('Simca', 'Chambord', NULL);
```

Selecionando os dados:

```
> SELECT
  p.id "CPF",
  p.nome "Nome",
  p.genero "Gênero",
  c.id "ID Carro",
  c.marca "Marca",
  c.modelo "Modelo"
FROM tb_carro c
LEFT JOIN tb_pessoa p
  ON (p.id = c.pessoa_fk);
```

CPF	Nome	Gênero	ID Carro	Marca	Modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
11111111111	Chiquinho da Silva	m	4	Chevrolet	Chevette
			5	Simca	Chambord

Apagar a tabela de carros:

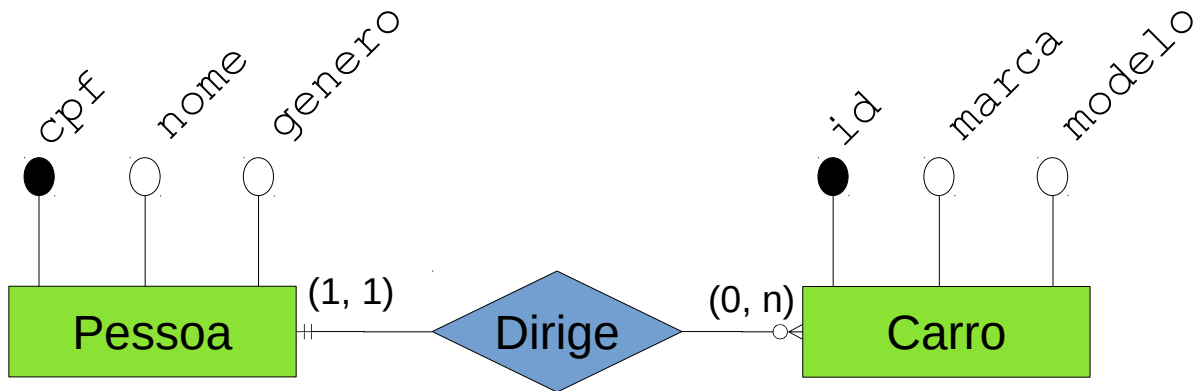
```
> DROP TABLE tb_carro;
```

35.3.2 Relacionamento (1, 1):(0, n)

Relacionamento obrigatório para a entidade da direita.

Não são permitidos valores nulos (fk) para a entidade relacionadora.

Modelo Conceitual



Modelo Lógico

Pessoa		
cpf	nome	genero
11111111111	Chiquinho da Silva	m
22222222222	Maria Santos	f
33333333333	Zé das Coves	m
44444444444	Bertolina Chaves	f
55555555555	Crivélío Almeida	m

Carro			
id	marca	modelo	pessoa_fk
1	Fiat	147	11111111111
2	Volkswagen	Variant	22222222222
3	Ford	Corcel I	33333333333
4	Chevrolet	Chevette	44444444444
5	Simca	Chambord	55555555555

Nenhum valor nulo na tabela da entidade Carro.

PessoaCarro					
cpf	nome	genero	carro_id	marca	modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
11111111111	Chiquinho da Silva	m	4	Chevrolet	Chevette
44444444444	Bertolina Chaves	f	5	Simca	Chambord
55555555555	Crivélío Almeida	m	NULL	NULL	NULL

Modelo Físico

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text,  
    pessoa_fk int8  
        REFERENCES tb_pessoa (id) -- Chave estrangeira  
        NOT NULL -- Preenchimento obrigatório  
);
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo, pessoa_fk) VALUES  
    ('Fiat', '147', 11111111111),  
    ('Volkswagen', 'Variant', 22222222222),  
    ('Ford', 'Corcel I', 33333333333),  
    ('Chevrolet', 'Chevette', 11111111111),  
    ('Simca', 'Chambord', 44444444444);
```

Selecionando os dados:

```
> SELECT  
    p.id "CPF",  
    p.nome "Nome",  
    p.genero "Gênero",  
    c.id "ID Carro",  
    c.marca "Marca",  
    c.modelo "Modelo"  
FROM tb_pessoa p  
LEFT JOIN tb_carro c  
    ON (p.id = c.pessoa_fk);
```

CPF	Nome	Gênero	ID Carro	Marca	Modelo
11111111111	Chiquinho da Silva	m	1	Fiat	147
22222222222	Maria Santos	f	2	Volkswagen	Variant
33333333333	Zé das Coves	m	3	Ford	Corcel I
11111111111	Chiquinho da Silva	m	4	Chevrolet	Chevette
44444444444	Bertolina Chaves	f	5	Simca	Chambord
55555555555	Crivélío Almeida	m			

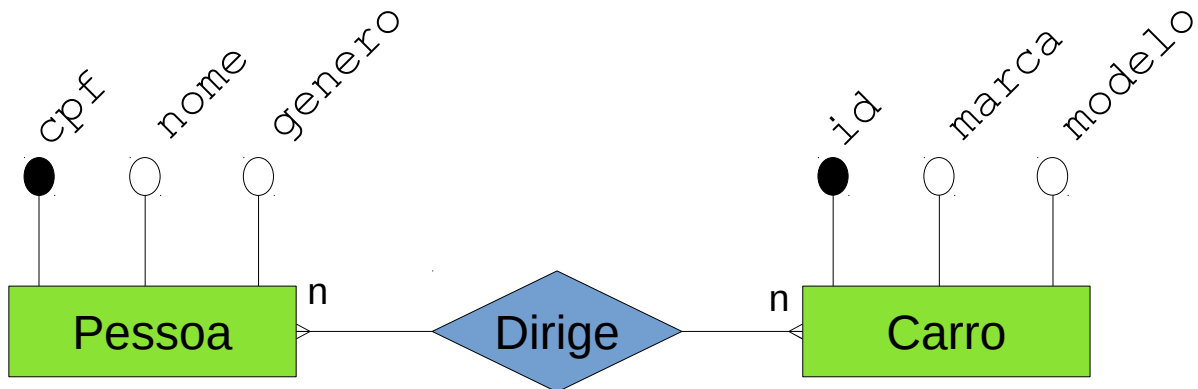
Apagar a tabelas de carros:

```
> DROP TABLE tb_carro;
```


35.4 Relacionamento n:n – Muitos para Muitos

Neste tipo de relacionamento é preciso uma tabela adicional para intermediar o relacionamento. Esse tipo de tabela é conhecido como **tabela associativa**.

Modelo Conceitual



Modelo Lógico

Pessoa		
<u>cpf</u>	nome	genero
11111111111	Chiquinho da Silva	m
22222222222	Maria Santos	f
33333333333	Zé das Coves	m
44444444444	Bertolina Chaves	f
55555555555	Crivélío Almeida	m

Carro		
<u>id</u>	marca	modelo
1	Fiat	147
2	Volkswagen	Variant
3	Ford	Corcel I
4	Chevrolet	Chevette
5	Simca	Chambord

Dirige	
<u>pessoa_cpf</u>	<u>carro_id</u>
11111111111	1
11111111111	2
33333333333	3
44444444444	4
55555555555	3

A tabela associativa (Dirige) fará o relacionamento entre as duas entidades anteriores.

Modelo Físico

Criação da tabela de carros:

```
> CREATE TABLE tb_carro(  
    id serial primary key,  
    marca text,  
    modelo text  
);
```

Criação da tabela associativa:

```
> CREATE TABLE tb_dirige(  
    pessoa_id int8 REFERENCES tb_pessoa(id),  
    carro_id int REFERENCES tb_carro (id),  
    PRIMARY KEY (pessoa_id, carro_id));
```

Inserir dados na tabela de carros:

```
> INSERT INTO tb_carro (marca, modelo) VALUES  
    ('Fiat', '147'),  
    ('Volkswagen', 'Variant'),  
    ('Ford', 'Corcel I'),  
    ('Chevrolet', 'Chevette'),  
    ('Simca', 'Chambord');
```

Valores a serem inseridos na tabela associativa:

```
> INSERT INTO tb_dirige (pessoa_id, carro_id) VALUES  
    (11111111111, 1),  
    (11111111111, 2),  
    (33333333333, 3),  
    (44444444444, 4),  
    (55555555555, 3);
```

Selecionando os dados:

```
> SELECT p.nome, c.marca, c.modelo FROM tb_dirige d
      INNER JOIN tb_pessoa p
        ON (d.pessoa_id = p.id)
      INNER JOIN tb_carro c
        ON (d.carro_id = c.id);
```

nome	marca	modelo
Chiquinho da Silva	Fiat	147
Chiquinho da Silva	Volkswagen	Variant
Zé das Coves	Ford	Corcel I
Bertolina Chaves	Chevrolet	Chevette
Crivélcio Almeida	Ford	Corcel I

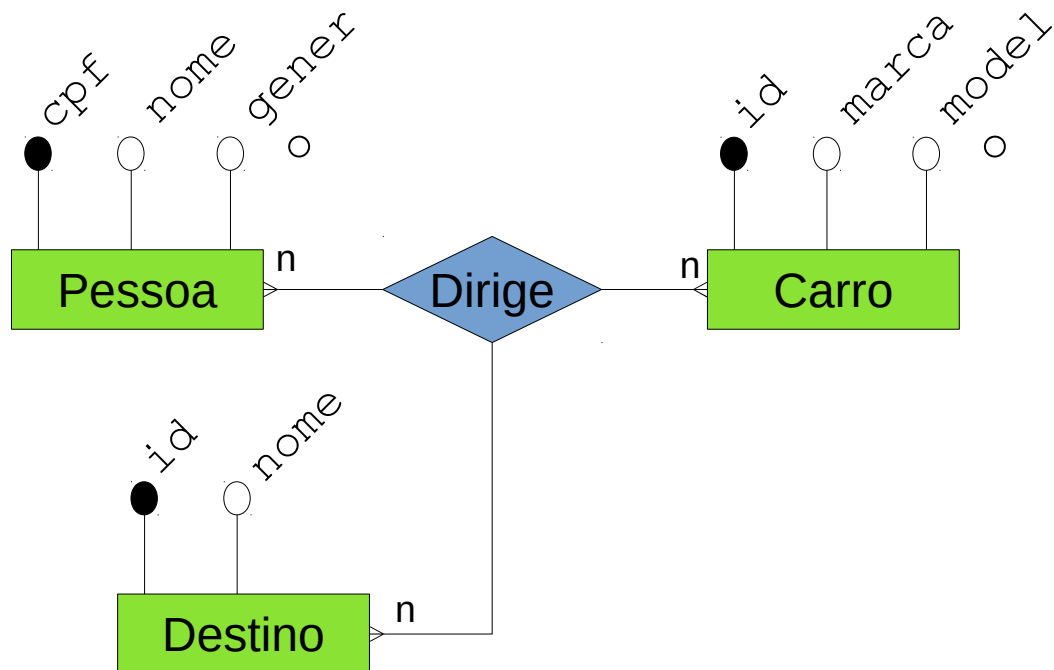
Apagar a tabela associativa:

```
> DROP TABLE tb_dirige;
```

35.5 Relacionamento Ternário

Tipo de relacionamento que é estabelecido entre três entidades.
Similar ao n:n, também precisa de uma tabela associativa.

Modelo Conceitual



Modelo Lógico

Pessoa		
<u>cpf</u>	nome	genero
11111111111	Chiquinho da Silva	m
22222222222	Maria Santos	f
33333333333	Zé das Coves	m
44444444444	Bertolina Chaves	f
55555555555	Crivélío Almeida	m

Carro		
<u>id</u>	marca	modelo
1	Fiat	147
2	Volkswage n	Variant
3	Ford	Corcel I
4	Chevrolet	Chevette
5	Simca	Chambord

Destino	
<u>id</u>	nome
1	Sede
2	Filial 1
3	Filial 2

Dirige		
<u>pessoa_cpf</u>	<u>carro_id</u>	<u>destino_id</u>
11111111111	1	1
11111111111	2	1
33333333333	3	3
44444444444	4	2
55555555555	3	2

Uma Pessoa Dirige um Carro para um Destino.

Modelo Físico

Criação da tabela para destinos:

```
> CREATE TABLE tb_destino(  
    id serial PRIMARY KEY,  
    nome varchar(20));
```

Criação da tabela associativa:

```
> CREATE TABLE tb_dirige(  
    pessoa_id int8 REFERENCES tb_pessoa(id),  
    carro_id int REFERENCES tb_carro(id),  
    destino_id int REFERENCES tb_destino(id),  
    PRIMARY KEY (pessoa_id, carro_id, destino_id));
```

Inserir valores de destino:

```
> INSERT INTO tb_destino (nome) VALUES ('Sede'), ('Filial 1'), ('Filial 2');
```

Inserir valores na tabela associativa:

```
> INSERT INTO tb_dirige (pessoa_id, carro_id, destino_id) VALUES  
    (11111111111, 1, 1),  
    (11111111111, 2, 1),  
    (33333333333, 3, 3),  
    (44444444444, 4, 2),  
    (55555555555, 3, 2);
```

Selecionando os dados:

```
> SELECT p.nome, c.marca, c.modelo, d.nome FROM tb_dirige a  
    INNER JOIN tb_pessoa p  
        ON (a.pessoa_id = p.id)  
    INNER JOIN tb_carro c  
        ON (a.carro_id = c.id)  
    INNER JOIN tb_destino d  
        ON (a.destino_id = d.id);
```

nome	marca	modelo	nome
Chiquinho da Silva	Fiat	147	Sede
Chiquinho da Silva	Volkswagen	Variant	Sede
Zé das Coves	Ford	Corcel I	Filial 2
Bertolina Chaves	Chevrolet	Chevette	Filial 1
Crivélcio Almeida	Ford	Corcel I	Filial 1

36 Particionamento de Tabelas

- O que é Particionamento de Tabelas
- Tipos de Particionamento

36.1 O que é Particionamento de Tabelas

Particionar uma tabela significa dividi-la em subtabelas (partições), de forma que o que seria uma tabela enorme, seus dados serão distribuídos entre suas partições.

A tabela particionada funciona como um roteador, recebe os dados ou requisições e redireciona para a partição pertinente.

Particionamos uma tabela para resolver problemas relativos a grandes volumes de dados concentrados em uma única tabela.

Dividir para conquistar!

<https://www.postgresql.org/docs/current/ddl-partitioning.html>

36.1.1 Benefícios de Particionamento de Tabelas

- **Desempenho**

Tabelas são representadas fisicamente por data files, se uma tabela é particionada não será mais somente um arquivo, assim em consultas cujos dados estejam em uma determinada partição sua leitura ou escrita serão mais rápidas por não precisar ler blocos de partes que não sejam pertinentes.

Os índices também serão menores, pois ao criar um índice para uma tabela particionada, esse índice na verdade serão vários índices variando de quantidade conforme o número de partições, ou seja, cada partição terá seu próprio índice.

- **Remoções de Dados**

A operação de DELETE é algo de alto custo para um SGBD, produz tuplas mortas e continua ocupando espaço em disco.

Se uma certa quantidade de dados for exatamente de acordo com o critério de uma partição, pode-se utilizar `ALTER TABLE DETACH PARTITION` ou simplesmente dando um `DROP TABLE` na(s) partição(es), o que faz com que se evite um *overhead* por conta de uma operação de `VACUUM`.

- **Redução de Custo**

Dados usados raramente podem ser migrados para mídias de armazenamento mais baratas e mais lentas.

- **Manutenção**

Para operações de `VACUUM` ou `ANALYZE` em vez de ser executado em uma grande tabela será em suas partições, dependendo da quantidade de memória o tamanho de uma partição pode caber dentro dela, assim agilizando todo o processo.

- **Locks**

Diminuição de locks graças a um certo grau de autonomia que as partições têm.

36.1.2 Partição Padrão

É a partição definida para pegar valores que não combinam com quaisquer outras partições.

36.1.3 Dicas e Boas Práticas de Particionamento

Padronização

Crie as partições seguindo um padrão de forma que futuras partições possam ser criadas dinamicamente obedecendo um critério.

Por exemplo, uma tabela que se chama `tb_vendas` e que será particionada por faixas de meses:

```
tb_vendas_201901  
tb_vendas_201902  
tb_vendas_201903  
tb_vendas_201904  
tb_vendas_201905
```

Nota-se que no exemplo o nome é formado pelo nome da tabela principal seguido de *underscore*, ano (com quatro dígitos) e mês (com dois dígitos).

Namespaces (Schemas)

Crie schemas próprios para as partições, pois assim facilitará a visualização das tabelas existentes.

É sempre bom organizar objetos e conforme dito anteriormente, faça isso de forma padronizada.

Tablespaces

Pode ser interessante, para fins de performance ter um disco somente para uma determinada tabela devido a I/O de disco.

Para uma tabela particionada isso pode melhorar muito o desempenho.

Tendo um disco disponível para esse fim, crie uma partição no mesmo, depois crie um ponto de montagem apontando para essa partição e enfim crie um *tablespace* apontando para esse diretório ponto de montagem da partição. Então, se a tabela e suas partições já existem, mova-as para esse *tablespace*. Caso ainda não existam crie-as nesse *tablespace*.

36.2 Tipos de Particionamento

36.2.1 Range

A tabela é particionada por intervalos de valores, sendo que nesse intervalo é no formato “[)”; fechado no início e aberto no final.

Tabela particionada por intervalo:

```
> CREATE TABLE tb_intervalo (  
    id serial,  
    dt date)  
    PARTITION BY RANGE (dt);
```

Partições para receber dados somente do ano de 2019:

```
> CREATE TABLE tb_intervalo_2019_01  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');  
  
> CREATE TABLE tb_intervalo_2019_02  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-02-01') TO ('2019-03-01');  
  
> CREATE TABLE tb_intervalo_2019_03  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-03-01') TO ('2019-04-01');  
  
> CREATE TABLE tb_intervalo_2019_04  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-04-01') TO ('2019-05-01');  
  
> CREATE TABLE tb_intervalo_2019_05  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-05-01') TO ('2019-06-01');  
  
> CREATE TABLE tb_intervalo_2019_06  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-06-01') TO ('2019-07-01');  
  
> CREATE TABLE tb_intervalo_2019_07  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-07-01') TO ('2019-08-01');  
  
> CREATE TABLE tb_intervalo_2019_08  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-08-01') TO ('2019-09-01');  
  
> CREATE TABLE tb_intervalo_2019_09  
    PARTITION OF tb_intervalo  
    FOR VALUES FROM ('2019-09-01') TO ('2019-10-01');
```

```
> CREATE TABLE tb_intervalo_2019_10
  PARTITION OF tb_intervalo
  FOR VALUES FROM ('2019-10-01') TO ('2019-11-01');

> CREATE TABLE tb_intervalo_2019_11
  PARTITION OF tb_intervalo
  FOR VALUES FROM ('2019-11-01') TO ('2019-12-01');

> CREATE TABLE tb_intervalo_2019_12
  PARTITION OF tb_intervalo
  FOR VALUES FROM ('2019-12-01') TO ('2020-01-01');
```

E se tentássemos um valor fora do que foi previsto?

```
> INSERT INTO tb_intervalo (dt) VALUES ('2020-01-01');
```

```
ERROR: no partition of relation "tb_intervalo" found for row
DETAIL: Partition key of the failing row contains (dt) = (2020-01-01).
```

Isso aconteceu porque não foi definida uma partição padrão (DEFAULT);

Criação de uma partição padrão:

```
> CREATE TABLE tb_intervalo_default
  PARTITION OF tb_intervalo DEFAULT;
```

Nova tentativa do INSERT que não funcionou:

```
> INSERT INTO tb_intervalo (dt) VALUES ('2020-01-01');
```

OK! Sem problemas agora :)

Vamos dar um TRUNCATE na tabela para iniciarmos os testes do zero:

```
> TRUNCATE tb_intervalo RESTART IDENTITY;
```

Utilizando o recurso de CTE, gerar uma massa de dados com dez milhões de registros utilizando datas aleatórias com um intervalo entre 01/01/2019 a 31/03/2020:

```
> WITH t (id_, random_date) AS (  
    SELECT  
        generate_series(1, 10000000),  
        '2019-01-01'::date +  
        (round(random() * ('2020-03-31'::date -  
            '2019-01-01'::date)))::int2  
    )  
    INSERT INTO tb_intervalo (dt)  
    SELECT random_date FROM t;
```

Atualizar as estatísticas da tabela:

```
> ANALYZE VERBOSE tb_intervalo;
```

Verificando o tamanho de cada partição e sua respectiva quantidade de registros:

```
> SELECT  
    relname AS tabela,  
    pg_size_pretty(pg_relation_size(relname::regclass)) AS tamanho,  
    reltuples::int8 AS registros  
FROM pg_class  
WHERE relname ~ 'tb_intervalo'  
    AND relkind = 'r'  
ORDER BY relname;
```

tabela	tamanho	registros
tb_intervalo_2019_01	23 MB	670065
tb_intervalo_2019_02	21 MB	616045
tb_intervalo_2019_03	24 MB	679929
tb_intervalo_2019_04	23 MB	658630
tb_intervalo_2019_05	24 MB	680965
tb_intervalo_2019_06	23 MB	659367
tb_intervalo_2019_07	24 MB	683019
tb_intervalo_2019_08	24 MB	680747
tb_intervalo_2019_09	23 MB	658873
tb_intervalo_2019_10	24 MB	681525
tb_intervalo_2019_11	23 MB	659522
tb_intervalo_2019_12	24 MB	681605
tb_intervalo_default	69 MB	1989708

Nota-se que ao colocar uma parte da faixa de datas fora do intervalo foram parar na tabela default.

Desanexar uma partição:

```
> ALTER TABLE tb_intervalo DETACH PARTITION tb_intervalo_2019_01;
```

Se novos dados pertinentes ao seu critério forem inseridos na tabela particionada, essa partição não receberá mais os registros.

Reanexando a partição:

```
> ALTER TABLE tb_intervalo ATTACH PARTITION tb_intervalo_2019_01
    FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');
```

Ao anexar uma partição, que pode ser qualquer tabela que tenha a mesma estrutura da tabela particionada, precisamos determinar o critério de dados válidos da partição.

E se realmente for necessário apagar uma partição?

```
> DROP TABLE tb_intervalo_2019_02;
```

Um simples `DROP TABLE` resolve, muito útil quando é preciso liberar espaço em casos que não há mais necessidade dos dados ali presentes. Muito melhor para a base de dados do que dar um `DELETE`.

36.2.1.1 List

Particionamento baseado em lista de valores válidos.

Tabela particionada por lista:

```
> CREATE TABLE tb_cidade (
    uf CHAR(2),
    nome VARCHAR(50)
) PARTITION BY LIST (uf);
```

Partições:

```
> -- Região Sul
CREATE TABLE tb_cidade_sul
    PARTITION OF tb_cidade
    FOR VALUES IN ('RS', 'SC', 'PR');

> -- Região Sudeste
CREATE TABLE tb_cidade_sudeste
    PARTITION OF tb_cidade
    FOR VALUES IN ('SP', 'RJ', 'MG', 'ES');

> -- Região Centro Oeste
CREATE TABLE tb_cidade_centro_oeste
    PARTITION OF tb_cidade
    FOR VALUES IN ('DF', 'GO', 'MS', 'MT');
```

```

> -- Região Nordeste
CREATE TABLE tb_cidade_nordeste
    PARTITION OF tb_cidade
    FOR VALUES IN ('MA', 'PI', 'CE', 'RN', 'PB', 'PE', 'AL', 'SE', 'BA');

> -- Região Norte
CREATE TABLE tb_cidade_norte
    PARTITION OF tb_cidade
    FOR VALUES IN ('AC', 'AM', 'RO', 'RR', 'AP', 'PA', 'TO');

```

Dados:

```

> INSERT INTO tb_cidade (uf, nome) VALUES
    ('SP', 'São Paulo'),
    ('SP', 'Santo André'),
    ('SP', 'São Bernardo do Campo'),
    ('RJ', 'Niterói'),
    ('MG', 'Belo Horizonte'),
    ('MG', 'Varginha'),
    ('RN', 'Natal'),
    ('RO', 'Porto Velho'),
    ('RS', 'Porto Alegre'),
    ('PR', 'Curitiba');

```

Selecionando os dados:

```

> -- Todas cidades
SELECT uf, nome FROM tb_cidade;

```

```

uf |           nome
----+-----
RO | Porto Velho
RN | Natal
SP | São Paulo
SP | Santo André
SP | São Bernardo do Campo
RJ | Niterói
MG | Belo Horizonte
MG | Varginha
RS | Porto Alegre
PR | Curitiba

```

```

> -- Cidades do Sul
SELECT uf, nome FROM tb_cidade_sul;

```

```

uf |           nome
----+-----
RS | Porto Alegre
PR | Curitiba

```

```
> -- Cidades do Sudeste
SELECT uf, nome FROM tb_cidade_sudeste;
```

```
uf | nome
---+-----
SP | São Paulo
SP | Santo André
SP | São Bernardo do Campo
RJ | Niterói
MG | Belo Horizonte
MG | Varginha
```

36.2.1.2 Hash

É o mais novo tipo de abordagem de particionamento de tabelas, que foi introduzido na versão 11 do PostgreSQL.

Seu comportamento se baseia balancear a carga de dados entre as partições baseando se em um campo que passamos um módulo (MODULUS) e um resto (REMAINDER).

Para a parte prática, vão ser criadas 5 (cinco) partições, o módulo será 5 e para cada uma terá um resto variando de 0 (zero) a 4 (quatro).

Criação da tabela particionada:

```
> CREATE TABLE tb_hash (
    id INT,
    campo TEXT)
    PARTITION BY HASH (id);
```

Criação das partições:

```
> CREATE TABLE tb_hash_0
    PARTITION OF tb_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 0);

> CREATE TABLE tb_hash_1
    PARTITION OF tb_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 1);

> CREATE TABLE tb_hash_2
    PARTITION OF tb_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 2);

> CREATE TABLE tb_hash_3
    PARTITION OF tb_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 3);
```

```
> CREATE TABLE tb_hash_4
PARTITION OF tb_hash
FOR VALUES WITH (MODULUS 5, REMAINDER 4);
```

Dados:

```
>
INSERT INTO tb_hash (id, campo)
SELECT
    generate_series(1, 10000000), -- Dez milhões de registros
    sha512((random() * 1000)::text::bytea); -- Geração de texto aleatório
```

Atualize as estatísticas:

```
> ANALYZE VERBOSE tb_hash;
```

Tamanho e quantidade de registros das tabelas:

```
> SELECT
    relname AS tabela,
    pg_size_pretty(pg_relation_size(relname::regclass)) AS tamanho,
    reltuples::int8 AS registros
FROM pg_class
WHERE relname ~ 'tb_hash' ORDER BY relname;
```

tabela	tamanho	registros
tb_hash	0 bytes	0
tb_hash_0	332 MB	1998236
tb_hash_1	333 MB	2000790
tb_hash_2	332 MB	1999944
tb_hash_3	332 MB	1999521
tb_hash_4	333 MB	2001636

Verificando a média de registros entre as partições:

```
> WITH t AS (
SELECT
    relname AS tabela,
    pg_size_pretty(pg_relation_size(relname::regclass)) AS tamanho,
    reltuples::int8 AS registros
FROM pg_class
WHERE relname ~ 'tb_hash_' ORDER BY relname)
SELECT round(avg(registros)) AS media FROM t;
```

media
2000025