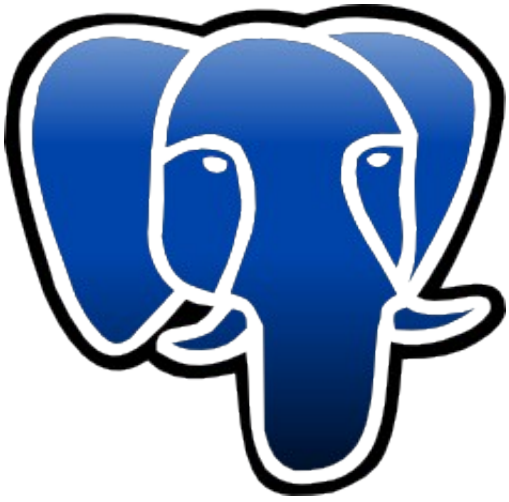
Tomando as Rédeas do Grande Elefante dos Dados



PostgreSQL

Rev. 2

Juliano Atanazio

Índice

0 - Sobre a Apostila

Esta apostila tem como objetivo explicar de forma simples e objetiva conceitos do banco de dados open source PosrgreSQL.

Pode ser baixada pelo site http://excelsis.com.br/downloads/postgresql.pdf e redistribuída livremente.

0.1 - Padrões Usados

Para facilitar a leitura e fazer uma melhor organização, de acordo com o tipo de texto são adotados os seguintes padrões:

- `Códigos SQL`
- *`Mensagens geradas por comandos SQL`*
- `Comandos do sistema operacional (shell)`
- *`Mensagens geradas por comandos do sistema operacional`*
- `Nomes de objetos`
- **Palavras em destaque e novos termos**

1 - 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 California 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.

1.1 - Como se Fala e como se Escreve

Uma dúvida comum ao PostgreSQL é seu nome. As formas corretas são as duas 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 “postgrí”. Infelizmente ainda há fontes na Internet com o nome do Postgres escrito erroneamente, o que leva muita gente também a falar errado.

1.2 - 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 do linhas por tabela	Ilimitado
Número máximo de colunas por tabela	250 - 1600 depende do tipo de coluna
Número máximo de índices por tabela	Ilimitado

1.3 - SGBD – Sistemas Gerenciadores de Bancos de Dados

É o conjunto de softwares (server) que gerenciam um banco de dados, o qual disponibiliza interface para gerenciamento através de aplicações clientes para manipulação dos dados. Exemplos de SGBDs: PostgreSQL, Oracle, MySQL, DB2, Firebird, etc.

Aplicações clientes → SGBD → Base de Dados

1.4 - Cluster de Banco de Dados

Ou simplesmente “cluster” é onde estão os arquivos físicos de um sistema de banco de dados, os quais contém os objetos, tais como bancos de dados, tabelas e etc.

1.5 - Instalação do PostgreSQL

Abra algum terminal de sua preferência e torne-se root, no Linux, dê o *comando:

```
sudo su
```

ou

```
su
```

*Depende como o sistema foi configurado

Mande instalar o PostgreSQL:

```
apt-get -y install postgresql
```

1.6 - Primeiro contato

Para fins de aprendizagem, nosso usuário do sistema operacional será “aluno” e então, como root dê o seguinte comando:

```
su postgres
```

“postgres” é o usuário administrador do banco de dados e agora o prompt do console mostra que “viramos” esse usuário. E como postgres dê o comando:

```
psql
```

Entramos na ferramenta cliente do PostgreSQL, o psql e por ele faremos todo o gerenciamento de nosso banco de dados.

Seu prompt mudará para: “postgres=#”, o que indica:
“postgres” → nome do banco de dados (quando não se declara explicitamente, o psql entende que deve se conectar a um banco de dados de mesmo nome do usuário);
“=” (igual) → indica que está pronto para se receber algum comando novo;
“#” (sustenido) → indica que o usuário que está operando o banco de dados no momento é um “superuser”, no caso de ser um usuário comum é “>” (maior que).

Obs.: Muitas vezes no lugar do “=” poderá conter algum outro caractere, tal como um “(“ (parênteses aberto), o qual indica que deverá ser devidamente fechado.

1.7 - O psql

É o terminal interativo do PostgreSQL, um aplicativo cliente, que permite a conexão a um banco de dados.

Uso: `psql [OPÇÕES]... [NOMEBD [USUÁRIO]]`

Dentre suas opções mais usadas:

-h	host	máquina do servidor de banco de dados ou diretório do soquete (padrão: "/var/run/postgresql")
-P	port	porta do servidor de banco de dados (padrão: "5432")
-U	user	nome de usuário do banco de dados (padrão: "postgresql")
-W		pergunta senha (pode ocorrer automaticamente)

1.7.1 - Comandos internos do psql

Dê o seguinte comando para exibir ajuda dos comandos internos: **\?**

Comando	**Parâmetros**	**Descrição**
\c[onnect]	[NOMEBD|- USUÁRIO|- MÁQUINA|- PORTA|-]	conecta a um outro banco de dados (atual "postgres")
\cd	[DIRETÓRIO]	muda o diretório de trabalho atual
\copyright		mostra termos de uso e distribuição do PostgreSQL
\encoding	[CODIFICAÇÃO]	mostra ou define codificação do cliente
\h	[COMANDO SQL]	mostra sintaxe dos comandos SQL, * para todos os comandos
\prompt	[TEXTO] NOME	pergunta o usuário ao definir uma variável interna
\password	[USUÁRIO]	muda a senha de um usuário
\q		sair do psql
\set	[NOME [VALOR]]	define variável interna ou lista todos caso não tenha parâmetros
\timing		alterna para duração da execução de comandos (atualmente desabilitado)
\unset	[NOME]	apaga (exclui) variável interna
\!	[COMANDO]	executa comando na shell ou inicia shell iterativa
Buffer de consulta		
\e	[ARQUIVO]	edita o buffer de consulta (ou arquivo) com um editor externo
\g	[ARQUIVO]	envia o buffer de consulta para o servidor (e os resultados para arquivo ou “|” (pipe))
\p		mostra o conteúdo do buffer de consulta
\r		reinicia (apaga) o buffer de consulta
\s	[ARQUIVO]	mostra histórico ou grava-o em um arquivo
\w	[ARQUIVO]	escreve o buffer de consulta para arquivo
Entrada/Saída		
\echo	[STRING]	escreve cadeia de caracteres na saída padrão
\i	[ARQUIVO]	executa comandos de um arquivo
\o	[ARQUIVO]	envia todos os resultados da consulta para arquivo ou |pipe
\qecho	[STRING]	escreve cadeia de caracteres para saída da consulta (veja \o)
Informativo		
\d	[NOME]	descreve tabela, índice, sequência ou visão
\d	{t|i|s|v|S} [MODELO]	(adicione "+" para obter mais detalhe) lista tabelas/índices/sequências/visões/tabelas do sistema
\da	[MODELO]	lista funções de agregação
\db	[MODELO]	lista tablespaces (adicione "+" para obter mais detalhe)
\dc	[MODELO]	lista conversões
\dC		lista conversões de tipos
\dd	[MODELO]	mostra comentário do objeto
\dD	[MODELO]	lista domínios
\df	[MODELO]	lista funções (adicione "+" para obter mais detalhe)
\dF	[MODELO]	lista configurações de busca textual (adicione "+" para obter mais detalhes)
\dFd	[MODELO]	lista dicionários de busca textual (adicione "+" para obter mais detalhes)
\dFt	[MODELO]	lista modelos de busca textual

Comando	**Parâmetros**	**Descrição**
\dFp	[MODELO]	lista analisadores de busca textual (adicione "+" para obter mais detalhe)
\dg	[MODELO]	lista grupos
\dn	[MODELO]	lista esquemas (adicione "+" para obter mais detalhe)
\do	[NOME]	lista operadores
\dl		lista objetos grandes, mesmo que \lo_list
\dp	[MODELO]	lista privilégios de acesso de tabelas, visões e sequências
\dT	[MODELO]	lista tipos de dado (adicione "+" para obter mais detalhe)
\du	[MODELO]	lista usuários
\l		lista todos os bancos de dados (adicione "+" para obter mais detalhe)
\z	[MODELO]	lista privilégios de acesso de tabelas, visões e sequências (mesmo que \dp)
Formatação		
\a		alterna entre modo de saída desalinhado e alinhado
\C	[STRING]	define o título da tabela, ou apaga caso nada seja especificado
\f	[STRING]	mostra ou define separador de campos para saída de consulta desalinhada
\H		alterna para modo de saída em HTML (atual desabilitado)
\pset	NOME [VALOR]	define opção de saída da tabela (NOME := {format|border|expanded|fieldsep|footer|null| numericlocale|recordsep|tuples_only|title|tableattr|pager})
\t		mostra somente registros (atual desabilitado)
\T	[STRING]	define atributos do marcador HTML <table> ou apaga caso nada seja especificado
\x		alterna para saída expandida (atual desabilitado)
Cópia, Objetos Grandes		
\copy		realiza comando SQL COPY dos dados para máquina cliente
\lo_export	OIDLOB ARQUIVO	
\lo_import	ARQUIVO [COMENTÁRIO]	
\lo_list		
\lo_unlink	OIDLOB	operações com objetos grandes

1.7.2 - Variáveis do psql

Dentro do psql podemos acessar variáveis de sistema e também criar nossas próprias variáveis. Para exibir todas as variáveis e seus respectivos conteúdos:

```
\set
```

```
AUTOCOMMIT = 'on'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
VERBOSITY = 'default'
VERSION = 'PostgreSQL 8.4.2 on i486-pc-linux-gnu, compiled by GCC gcc-4.3.real (Ubuntu 4.3.2-
1ubuntu11) 4.3.2'
DBNAME = 'postgres'
USER = 'user'
HOST = '/var/run/postgresql'
PORT = '5432'
ENCODING = 'UTF8'
```

Acessando o conteúdo de uma determinada variável:

```
\echo :USER
```

1.7.2.1 - Criando uma nova variável

```
\set nova_variavel valor
```

Se dermos o comando `\set` novamente poderemos ver que a variável criada se encontrará na última linha.

1.7.2.2 - Destruindo uma variável

```
\unset nova_variavel
```

Digitando o comando `\set` verificamos que ela deixou de existir.

2 - SQL (Structured Query Language – Linguagem Estruturada de Consulta)

É a linguagem usada nos SGBDs 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.

No PostgreSQL os comandos SQL são finalizados com ponto e vírgula (;) e seus comandos não são *case sensitive*.

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 - 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.

2.3 - 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.

2.4 - Criação de Usuário

```
CREATE USER aluno
```
<aperte a tecla tab 2X>

O psql tem uma característica muito interessante, a qual te ajuda a completar um comando ou mostrar outras opções possíveis apertando a tecla tab.

No caso do comando acima, exibirá na sua tela:

ADMIN	CREATEROLE	IN	NOCREATEDB	NOINHERIT	ROLE	UNENCRYPTED
CONNECTION LIMIT	CREATEUSER	INHERIT	NOCREATEROLE	NOLOGIN	SUPERUSER	CREATEDB
ENCRYPTED	LOGIN	NOCREATEUSER	NOSUPERUSER	SYSID		

Essas são opções que são exibidas para completarmos nosso comando. No caso o faremos assim:

```
CREATE USER aluno NOSUPERUSER;
```

Criamos o usuário que irá mexer no nosso banco de dados, que por conveniência é o mesmo nome do usuário do sistema (Linux). Mas o mesmo, por enquanto tem ações muito limitadas, pois não é um “superuser”.

Para sair do psql:

```
\q
```

Voltamos ao prompt do console, como o usuário postgres e então voltaremos a ser root:

```
exit
```

Agora devemos voltar a ser o usuário aluno, apenas repita o comando acima:

```
exit
```

O usuário root tem o poder de se “mutar” para qualquer usuário do sistema, nosso usuário aluno não tem tais poderes e precisava se transformar no usuário postgres, que como já foi dito é o super usuário do PostgreSQL para adicionar usuários novos ao SGBD, no caso adicionamos o usuário “aluno”.

Com o usuário aluno, para fins de teste, tente conectar-se ao banco de dados postgres:

```
psql postgres
```

Caso receber a seguinte mensagem:

psql: FATAL: autenticação do tipo Ident falhou para usuário "aluno", o arquivo de configuração de autenticação de cliente PostgreSQL deverá ser configurado.

O próximo tópico trata do assunto.

3 - Autenticação de cliente e configuração geral do PostgreSQL (pg_hba.conf e postgresql.conf)

Para podermos conectar ao PostgreSQL, nosso usuário tem que estar autorizado.

Como root, editaremos o arquivo `/etc/postgresql/8.4/main/pg_hba.conf` e devem ser adicionadas as seguintes linhas em seu final:

```
local   all     aluno                   ident sameuser
host    all     aluno  192.168.0.1/24       md5
```

Obs.: No lugar de “192.168.0.0/24” adapte conforme sua máquina na rede.

Ao editarmos o arquivo `pg_hba.conf` demos a devida permissão para que o usuário possa acessar. No entanto, seria bom também fazer com que o servidor abra a porta do PostgreSQL (5432) em todas as interfaces.

Edite também o arquivo `/etc/postgresql/8.4/main/postgresql.conf`:

Mude a linha

```
#listen_addresses = 'localhost'
```

para

```
listen_addresses = '*'
```

Efetive as alterações, após salvar dando o comando no prompt do sistema:

```
/etc/init.d/postgresql restart
```

Levando-se em conta que agora somos o usuário root, mudemos para postgres:

```
su postgres
```

Entre no psql e nele crie uma senha para aluno:

```
ALTER USER aluno password '123456';
```

Com o usuário aluno, digite no prompt:

```
psql -h 192.168.0.1 -d postgres
```

Foi solicitada uma senha, aquela que foi criada pelo administrador do postgresql.

Após a senha ter sido aceita com sucesso o prompt do psql ficou da seguinte forma:

postgres=>

Como já foi comentado no início, o sinal “>” indica que o usuário “aluno” é limitado.

Então, por fim, mudaremos seu status para “superuser”. Faremos isso através do até então, único administrador do banco de dados, o usuário “postgres”, dentro psql:

```
ALTER USER aluno SUPERUSER;
```

De agora em diante só usaremos o usuário “aluno”, pois até aqui esse rodízio de usuários foi necessário, mas agora que tem-se o poder de “superuser”, só ele fará os comandos.

4 - Importando um banco de dados já pronto de um arquivo

Nos extras você encontra-se o código do arquivo “curso.sql”, copie e salve-o no diretório “/tmp”.
Agora vamos rodá-lo:

```
\i /tmp/curso.sql
```

Apesar de estarmos criando um novo banco de dados chamado “curso”, devemos temporariamente nos conectar ao servidor e pra isso é necessário especificar algum banco de dados pré existente, no caso o “postgres” que assim como o usuário de mesmo nome é criado na instalação do SGBD.
Poderíamos ter feito de maneira diferente, no prompt do sistema operacional, com o psql:

```
psql -f /tmp/curso.sql postgres
```

Agora, pra entrar no banco de dados usaremos no shell:

```
psql curso -h 192.168.7.2
```

Como definimos no arquivo “pg_hba.conf”, será pedida uma senha.

Saia do psql:

```
\q
```

Dê o comando:

```
psql curso
```

Dessa vez não foi pedida uma senha. Pedimos a conexão na máquina local (localhost).

Sugestão de estudo: arquivos `pg_hba.conf` e `postgresql.conf`.

5 - Identificadores

Para nomearmos identificadores devemos usar letras ou “_” (underline):

Certo:

```
CREATE DATABASE abc;
```

Errado:

```
CREATE DATASE 123;
```

Se por acaso fosse criado um banco usando letras maiúsculas:

```
CREATE DATABASE ABC;
```

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

```
DROP DATABASE abc;
```

Há uma maneira de fazer com 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:

```
CREATE DATABASE “ABc”;

CREATE DATABASE “123tEste”;
```

Liste os bancos de dados com:

`\l` ← Letra “L” minúscula

Essa não é uma boa prática e faz com que toda vez que for preciso manipulá-los, deverão ser referenciados também entre aspas:

```
DROP DATABASE “Abc”;
DROP DATABASE ”123tEste”;
```

6 - Modelos de Banco de Dados - Templates

O comando "CREATE DATABASE" funciona copiando um banco de dados existente. Por padrão, ele copia o banco de dados de sistema chamado "template1", deste modo é modelo para criação de outros banco 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". Esse banco contém os mesmos dados que o conteúdo inicial de "template1", que é, apenas os objetos padrões pré definidos pela versão do PostgreSQL. É uma base de dados imutável e que não aceita conexões. Quando se cria um banco copiando "template0" ao invés de "template1" é criado um banco de dados "virgem" que não contém nada do que foi adicionado a "template1".

Sintaxe:

```
CREATE DATABASE nome_do_banco_de_dados TEMPLATE banco_modelo;
```

Exemplos:

```
/* 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

/* Consulta na tabela existente */
SELECT * FROM tb_exemplo;
```

A partir de agora todos os bancos novos criados no sistema terão a tabela `tb_exemplo` com seus valores que foram inseridos dentro do banco “template1”. Como pôde ser comprovado, ao se criar objetos dentro de `template1`, esses objetos serão retransmitidos 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 virgem, como no exemplo a seguir:

```
CREATE DATABASE novo_banco TEMPLATE template0;
```

O banco de dados criado acima é um banco de dados “virgem”, pois não há qualquer objeto no mesmo, pois seu modelo foi o “template0”.

Se for preciso criar um novo banco de dados tomando outro como modelo também é posssível:

```
CRATE DATABASE curso2 TEMPLATE curso;
```

Ao se conectar nesse banco de dados poderá ser constatado que possui os mesmos objetos e registros que seu modelo (“curso”).

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 o banco criado */
DROP DATABASE xyz;
```

7 - Tipos de Dados

7.1 - Numéricos

Nome	Alias	Faixa	Tamanho	Descrição
smallint	int2	-32768 até +32767	2 bytes	integer de 2 bytes com sinal
integer	Int, int4	-2147483648 até +2147483647	4 bytes	integer de 4 bytes com sinal
bigint	int8	-9223372036854775808 até +9223372036854775807	8 bytes	integer de 8 bytes com sinal
* 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	precisão dupla de ponto flutuante
serial	serial4	1 até 2147483647	4 bytes	auto incremento inteiro de 4 bytes
bigserial	serial8	1 até 9223372036854775807	8 bytes	auto incremento inteiro de 8 bytes

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

7.1.1 - Máscaras para formatos numéricos

Modelo	Descrição
9	valor com o número especificado de dígitos
0	valor com zeros à esquerda
. (ponto)	ponto decimal
, (vírgula)	separador de grupo (milhares)
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)
Notas: a. RN — roman numerals.	

7.2 - De Caracteres

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

7.3 - Valor nulo (null)

Um valor nulo é simplesmente um valor não preenchido.

7.4 - Lógico ou “Booleano”

Nome	Alias	Descrição
boolean	bool	logical Boolean (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”, os seguintes valores podem ser usados: `FALSE`, `'f'`, `'false'`, `'n'`, `'no'`, `'0'`.

7.5 - Data e Hora

Nome	Alias	Descrição
date		calendar date (year, month, day)
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.

Além de todos os tipos de dados apresentados aqui existem outros, porém não são tão usados. A variedade é muito grande.
Para maiores informações, no psql digite o comando:

```
\dT+
```

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

7.5.1 - Máscaras de Data e Hora

Formato	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)
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)

8 - Obtendo ajuda de comandos SQL dentro do psql

O utilitário cliente do PostgreSQL, o psql, dentre suas funções, possui um sistema de auto ajuda para o usuário para comandos SQL.

Para ajuda geral:

```
\h
```

São mostrados os comandos SQL.

Mas se por acaso necessitar de uma explicação maior sobre um determinado comando:

```
\h <comando> [opções]
```

Exemplos:

```
\h CREATE
```

São mostradas todas as formas do comando SQL “CREATE”.

```
\h CREATE TABLE
```

Exibe apenas informações de criação de tabelas.

```
\h CREATE DATABASE
```

Exibe apenas informações de criação de banco de dados.

9 - Comentários do PostgreSQL

Assim como os comandos SQL, devem terminar com ponto e vírgula, podem ser:

```
CREATE DATABASE db_teste; --única linha
```

ou

```
CREATE DATABASE db_teste; /*
múltiplas linhas
 */
```

10 - Conectando-se a outro banco de dados dentro do psql

Estamos no banco de dados “curso” agora, para nos conectarmos ao banco de dados “postgres”:

```
\c postgres
```

Para voltar ao “curso”:

```
\c curso
```

11 - Criando, Modificando e Apagando Objetos

De acordo com o título, temos respectivamente os seguintes comandos: CREATE, ALTER e DROP.

Vejamos agora alguns exemplos dos mesmos com o tipo de objeto DATABASE (banco de dados):

```
CREATE DATABASE db_teste; — Banco de dados “db_teste” foi criado!;

ALTER DATABASE db_teste RENAME TO db_teste2; /* Banco de dados “db_teste” foi alterado,
seu novo nome agora é “db_teste2” */ ;

DROP DATABASE db_teste2; — Banco de dados “db_teste2” foi apagado;
```

É importante lembrar que quando o objeto em questão está sendo acessado no momento, não se pode alterá-lo.

12 - Tabelas

É a estrutura 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.

12.1 - Criação de Tabelas

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

12.2 - 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” após `CREATE`:

```
CREATE TEMPORARY TABLE tabela_tmp(
       …
);
```

12.3 - Exibindo tabelas criadas

```
\d
```

12.4 - Exibindo a estrutura de uma tabela

```
\d nome_da_tabela
```

Com detalhes:

```
\d+ nome_da_tabela
```

12.5 - Alterando Tabelas

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

O quadro abaixo contém de maneira resumida e objetiva os meios para se fazer as alterações de acordo com o que for preciso:

ALTER TABLE nome_tabela	ADD	[COLUMN] coluna tipo [restrição_coluna [...]]		
		restrição_tabela		
	ALTER [COLUMN] coluna		TYPE tipo [USING expressão]	
			SET DEFAULT expressão	
			DROP DEFAULT	
			{ SET | DROP } NOT NULL	
			SET STATISTICS inteiro	
			SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }	
	CLUSTER ON nome_índice			
	DROP	[COLUMN] coluna [RESTRICT | CASCADE]		
		CONSTRAINT nome_restrição [RESTRICT | CASCADE]		
	DISABLE	TRIGGER [nome_gatilho | ALL | USER]		
		RULE nome_regra_reescrita		
	ENABLE	TRIGGER [nome_gatilho | ALL | USER]		
		REPLICA TRIGGER nome_gatilho		
		ALWAYS TRIGGER nome_gatilho		
		RULE nome_regra_reescrita		
		REPLICA RULE nome_regra_reescrita		
		ALWAYS RULE nome_regra_reescrita		
	INHERIT tabela_ancestral			
	NO INHERIT tabela_ancestral			
	OWNER TO novo_dono			
	RENAME	TO novo_nome_da_tabela		
		COLUMN nome_da_coluna TO novo_nome_da_coluna		
	RESET (parâmetro_armazenamento [, ...])			
	SET	WITHOUT	CLUSTER	
			OIDS	
		(parâmetro_armazenamento = valor [, ...])		
		TABLESPACE nome_tablespace		
		SCHEMA nome_esquema		

Exemplos:

Antes e após cada comando SQL abaixo dê o comando “\d colaboradores” para verificar as alterações:

- Adicionando um novo campo:

```
ALTER TABLE colaboradores ADD COLUMN nova_coluna int2;
```

- Mudando o tipo de dados de um campo:

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

- Alterando o nome de um campo:

```
ALTER TABLE colaboradores RENAME COLUMN nova_coluna TO ultima_coluna;
```

- Alterando o nome de uma tabela:

```
ALTER TABLE pedido RENAME TO pedidos;
```

- Apagando um campo de uma tabela:

```
ALTER TABLE colaboradores DROP COLUMN ultima_coluna;
```

12.6 - Apagando uma tabela

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

O comando mais simples pra se deletar uma:

```
DROP TABLE nome_da_tabela;
```

Se a mesma for referenciada por outra tabela, não poderá ser apagada.
A não ser que:

```
DROP TABLE nome_da_tabela CASCADE;
```

13 - Tablespaces

Tablespaces no PostgreSQl permitem a administradores de bancos de dados definirem onde os arquivos físicos que representam os objetos do banco de dados serão gravados. Ou seja, é possível definir o diretório onde os objetos serão armazenados. Após a criação de um tablespace, o mesmo pode ser referenciado pelo nome ao criar os objetos que serão gravados no local desejado.

Sintaxe:

```
CREATE TABLESPACE nome [ OWNER usuário ] LOCATION 'diretório';
```

Exemplos:

Primeiro vamos criar um diretório para ser o tablespace (como root):

```
mkdir /tbs
```

Dando a propriedade do diretório para o usuário postgres e o grupo de usuários postgres:

```
chown -R postgres:postgres /tbs
```

Criando o tablespace:

```
CREATE TABLESPACE ts_tmp LOCATION '/tbs';
```

Criando uma tabela no tablespace “ts_tmp”:

```
CREATE TABLE tb_ts(
cod serial PRIMARY KEY,
nome varchar(15)
) TABLESPACE ts_tmp;
```

Obs.: No diretório do cluster, há um subdiretório chamado “pg_tblspc”, o qual contém links que apontam para o local de cada tablespace criado.

14 - Restrições (Constraints)

São regras que inserimos para que uma coluna não receba dados indesejados. Por exemplo: imagine uma tabela de produtos que tem dentre seus campos um que se refere ao preço. Se não for colocada uma restrição, serão aceitos valores negativos (<0).

14.1 - Tipos de Restrições

14.1.1 - CHECK

A mais genéria das restrições, permite especificar que valor, em uma determinada coluna, deve satisfazer uma expressão booleana.

Exemplo:

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

Nesse exemplo evita que no campo “preco” seja inserido qualquer valor que não seja maior que zero.

14.1.2 - NOT NULL

Faz com que a coluna não aceite valores nulos, ou seja, a partir do momento em que for inserido um novo registro a(s) coluna(s) com a restrição NOT NULL deverá receber algum valor.

14.1.3 - UNIQUE

Assegura que todos os registros, na coluna especificada não terão valores repetidos.

14.1.4 - PRIMARY KEY

Também conhecida como **Chave Primária**. 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.

Exemplos:

```
CREATE TABLE produtos(
       cod_prod int2 UNIQUE NOT NULL,
       preco numeric(7,2) CHECK(preco>0)
);
```

Tem o mesmo efeito de:

```
CREATE TABLE produtos(
       cod_prod int2 PRIMARY KEY,
       preco numeric(7,2) CHECK(preco>0)
);
```

Pode também ser da seguinte maneira:

```
CREATE TABLE produtos(
       cod_prod int2,
       preco numeric(7,2) CHECK(preco>0),
       PRIMARY KEY(cod_prod)
);
```

Mais esta:

```
CREATE TABLE produtos(
       cod_prod int2 CONSTRAINT pk_produtos PRIMARY KEY,
       preco numeric(7,2) CHECK(preco>0)
);
```

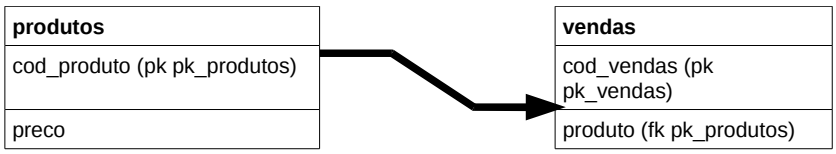
E esta:

```
CREATE TABLE produtos(
    cod_prod int2,
    preco numeric(7,2) CHECK(preco>0),
    CONSTRAINT pk_produtos PRIMARY KEY(cod_prod)
);
```

Nestas duas últimas demos um nome à chave primária.

14.1.5 - FOREIGN KEY

Também conhecida como **Chave Estrangeira**. Especifica que os valores em uma coluna (ou grupo de colunas) deve corresponder aos valores que estão em um campo ou outra tabela.
Quando é entre tabelas chamamos de **integridade relacional entre tabelas**.
Os valores dos campos referenciados devem ser únicos.



Acima está representado o relacionamento de “vendas” com “produtos”, vejamos como a tabela “vendas” foi criada:

```
CREATE TABLE vendas(
       cod_vendas int2 PRIMARY KEY,
       produto int2 REFERENCES produtos (cod_prod)
);
```

O campo “produto”, faz referência à chave primária da tabela “produtos”.

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

```
CREATE TABLE vendas(
       cod_vendas int2 PRIMARY KEY,
       produto int2,
       CONSTRAINT fk_prod FOREIGN KEY (produto) REFERENCES produtos (cod_prod)
);
```

14.1.6 - O Modificador “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:

```
CREATE 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”.

15 - Descrição de Objetos

Também conhecido como “Comentário de Objetos”.

Exemplo de como se insere uma descrição em uma tabela:

```
COMMENT ON TABLE colaboradores IS 'Funcionários da empresa';
```

Para visualizar:

```
 \dt+ colaboradores
```

Para excluir:

```
COMMENT ON TABLE colaboradores IS NULL;
```

16 - Operadores

Operador/Elemento		**Descrição/Uso**
.		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	IS NULL	teste para nulo
NOTNULL	NOT NULL	teste para não nulo
IN		“em”
BETWEEN		“entre”
OVERLAPS		sobreposição de intervalo de tempo
LIKE	~~	comparação de string *case sensitive*
ILIKE	~~*	comparação de string *case insensitive*
SIMILAR		comparação de string por similaridade
<		menor que
>		maior que
=		igualdade, atribuição
<>	!=	diferente
>=		maior ou igual que
<=		menor ou igual que
NOT		“NÃO” lógico
AND		“E” lógico
OR		“OU” lógico
||		concatena strings

17 - SELECT - Consultas (Queries)

Consultar ou fazer uma “query” em um banco de dados, pode ser simplesmente dito como **selecionar** os dados.

Um simples SELECT:

```
SELECT 5+2;

 ?column?
----------
        7
(1 registro)
```

Podemos melhorar um pouco:

```
SELECT 5+2 AS "Resultado";

 Resultado
----------
        7
(1 registro)
```

Uma operação matemática que retorna o valor absoluto (não negativo), usando o operador “@”:

```
SELECT @ 3-10 AS "Resultado";

 Resultado
----------
        7
(1 registro)
```

O parâmetro “AS” é usado para dar um rótulo à coluna, na ausência dele é usado o nome do campo da tabela. No caso acima, o resultado exibido não foi retirado de uma tabela e sim de uma simples operação matemática.

Foram inseridas as aspas para que a palavra “Resultado” não aparecesse apenas com letras minúsculas.

Geralmente puxamos dados de tabelas e a forma mais simples pra isso é a seguinte:

```
SELECT campoX,campoY,...,campoN FROM nome_da_tabela;
```

Em nosso banco de dados de exemplo, há uma tabela chamada “colaboradores”. Vamos fazer uma query selecionando todos os dados da mesma:

```
SELECT * FROM colaboradores;
```

Obtivemos então todos os registros de todos os campos. No comando acima o asterisco “*” representa todos os campos da tabela.

Agora vamos somente selecionar alguns campos:

```
SELECT nome,snome FROM colaboradores;
```

Dessa vez selecionamos apenas os campos `nome` e `snome`.

Modificando um pouco o comando:

```
SELECT nome||' '||snome AS "Nome completo" FROM colaboradores;
```

Com o uso operador de concatenação de strings, o pipe duplo “||”, juntamos os valores de duas colunas em uma só intercalados com um espaço (um espaço entre as aspas simples).

Podemos misturar concatenações também:

```
SELECT
       nome||' '||snome AS "Nome completo",
       cidade||' - '||uf AS "Cidade - UF"
FROM colaboradores;
```

Fazer operações matemáticas com os valores dos campos é um recurso interessante também.

Suponhamos que é preciso calcular o valor do Vale Transporte (que é de 6% do salário)para cada empregado:

```
SELECT
       nome||' '||snome AS "Nome completo",
       salario*0.06 AS "Vale Transporte"
FROM colaboradores;
```

Se for necessário saber quais são as cidades que se encontram na tabela, de forma que não haja repetição:

```
SELECT DISTINCT cidade FROM colaboradores;
```

A palavra chave “DISTINCT” faz com que seja exibida apenas uma ocorrência dos valores da tabela selecionada. Mas é possível fazer a distinção avaliando mais de um campo:

```
SELECT DISTINCT cidade,uf FROM colaboradores;
```

Se tiver cidades com nomes iguais, mas de diferentes estados, as duas seriam constadas, o campo “uf” seria o diferencial.

O recurso de paginação é muito útil quando não queremos ver todos os resultados. Se for optado por exibir apenas os 7 primeiros registros:

```
SELECT * FROM colaboradores LIMIT 7;
```

Ou ainda exibir por faixa de registros. Do décimo primeiro ao décimo sétimo, por exemplo:

```
SELECT * from colaboradores LIMIT 7 OFFSET 10;
```

Uma tradução para a query acima: selecionar todos os campos da tabela “colaboradores”, limitando a exibição a 7 registros e dispensando os 10 primeiros.

Sem limites, mas “cortando” os 6 primeiros:

```
SELECT * from colaboradores OFFSET 6;
```

17.1 - O Comando TABLE

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

```
TABLE = SELECT * FROM
```

Logo, se fizermos:

```
TABLE colaboradores;
```

é a mesma coisa que:

```
SELECT * FROM colaboradores;
```

17.2 - SELECT sem FROM

Há uma forma de se fazer consultas usando o SELECT sem o FROM da seguinte forma:

```
SELECT tabela.coluna;
```

Exemplo:

```
SELECT colaboradores.*;
```

Erro! Para que a omissão da palavra “FROM” dê certo é necessário habilitar. Há duas formas:

Por seção:

```
SET add_missing_from TO ON;
```

De forma definitiva:

Edite o arquivo postgresql.conf (`/etc/postgresql/8.4/main/postgresql.conf`) e mude a linha:

```
#add_missing_from = off
```

para

```
add_missing_from = on
```

E por fim mande o serviço carregar as novas configurações:

```
/etc/init.d/postgresql reload
```

Não é preciso reiniciar (restart)

17.3 - Type Casts

Faz uma conversão de um tipo de dado para outro:

```
CAST ( expressão AS tipo )
```

ou

```
expressão::tipo
```

Exemplos:

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

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

Obs.: `CAST` está conforme o padrão SQL e o operador “::” é próprio do PostgreSQL.

17.4 - Case - Um “IF” disfarçado

O comando CASE, para quem conhece linguagens de programação, pode ser comparado ao “IF”. Ao invés de usarmos a sequência “IF, THEN, ELSE“, usamos “CASE, WHEN, THEN, ELSE” e finalizamos com um “END”.

```
SELECT nome||' '||snome AS "Nome Completo",
       CASE uf
              WHEN 'SP' THEN 'São Paulo'
              WHEN 'MG' THEN 'Minas Gerais'
              ELSE 'Rio de Janeiro'
       END
       AS "Origem"
FROM colaboradores;
```

Caso uf for igual a SP; retorna *São Paulo*, igual a MG; retorna *Minas Gerais*, senão; retorna *Rio de Janeiro*.

17.5 - COALESCE

Em certas situações, quando há valores nulos em uma tabela é necessário preenchê-los provisoriamente com algum valor. O comando COALESCE faz essa função.
Como exemplo, a tabela “colaboradores” será usada. A mesma não tem valores nulos.
O comando INSERT será explicado mais adiante, mas por hora nos será útil:

```
INSERT INTO colaboradores (mat_id,nome,snome) VALUES (00020,'Zé','Ninguém');
```

Quando for feita uma query na tabela, constará apenas valores para os campos mat_id, nome, snome e uf.

Os observadores mais atentos notarão que no INSERT não consta uma referência ao campo uf, no entanto, se olharmos o anexo que contém todo o código SQL de como foi feita a estrutura da tabela, verá que há um valor padrão DEFAULT associado.

Uma query na tabela “colaboradores”, mostra que o último registro agora tem valores nulos. Para exemplo, um dos campos nulos será usado com o comando “COALESCE”, o campo “setor”:

```
SELECT nome,snome,COALESCE(setor,'inexistente') as setor
FROM colaboradores;
```

17.6 - NULLIF

É uma função que retorna um valor nulo se o primeiro valor for igual ao segundo valor, caso contrário retornará o primeiro valor.
Isso pode ser usado para fazer a operação inversa de COALESCE:

Exemplos:

```
SELECT NULLIF(1,1); --Retorna Null
SELECT NULLIF(7,7); --Retorna Null
SELECT NULLIF(9,1); --Retorna 9
SELECT NULLIF(3,3); --Retorna Null
```

17.7 - 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.

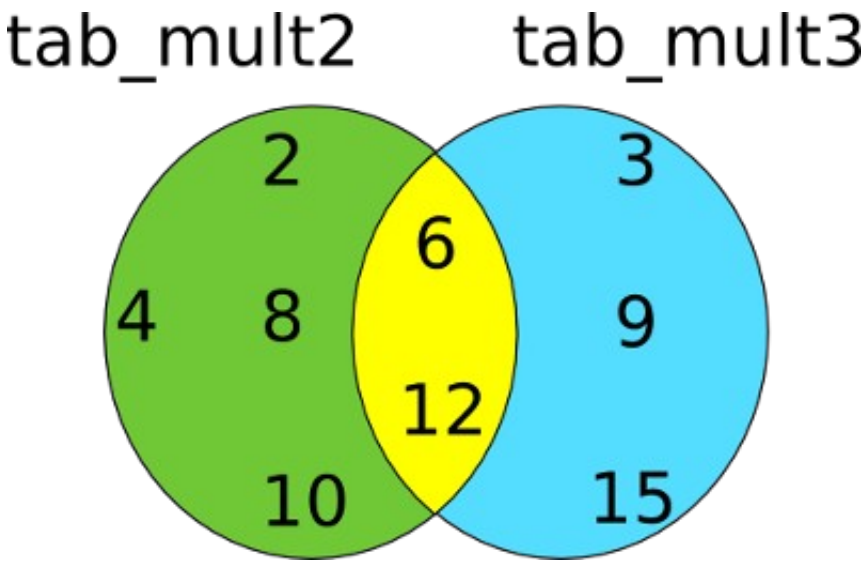
Essas funções não fazem do padrão SQL, mas são uma extensão comum. Alguns outros bancos de dados farão o retorno nulo se qualquer argumento for nulo, ao invés de somente quanto todos forem nulos.

Exemplos:

```
SELECT GREATEST(2,1,-9,55,20); --Retorna 55
SELECT LEAST(2,1,-9,55,20); --Retorna -9
SELECT GREATEST(NULL,5); --Retorna 5
SELECT LEAST(NULL,3); --Retorna 5
```

17.8 - Conjuntos

Relembrando a teoria de conjuntos de matemática:



Os dois conjuntos representam, respectivamente as tabelas tab_mult2 e tab_mult3.

No PostgreSQL funcionam de acordo com a correspondência ou não de colunas selecionadas entre uma tabela e outra.

Os comandos abaixo criarão as tabelas citadas e as preencherão:

```
CREATE TEMP TABLE tab_mult2( valor int2 );

CREATE TEMP TABLE tab_mult3( valor int2 );

INSERT INTO tab_mult2 VALUES (2),(4),(6),(8),(10),(12);

INSERT INTO tab_mult3 VALUES (3),(6),(9),(12),(15);
```

17.8.1 - Fazendo operações de conjunto

Temos União (UNION), Intersecção (INTERSECT) e Exceção (EXCEPT).

17.8.1.1 - UNION

É a união de todos os valores das colunas das tabelas envolvidas. Por padrão cada valor é exibido somente uma vez, mas se for usado o parâmetro “ALL”, valores que existem em mais de uma tabela é exibido de acordo com a ocorrência.

```
SELECT campo1,campo2,...,campoN FROM tabelaX
UNION [ALL]
SELECT campo1,campo2,...,campoN FROM tabelaY;
```

Exemplo:

```
SELECT valor FROM tab_mult2 UNION SELECT valor FROM tab_mult3;
```

17.8.1.2 - INTERSECT

Só retorna valores em comum entre as tabelas:

```
SELECT valor FROM tab_mult2 INTERSECT SELECT valor FROM tab_mult3;
```

17.8.1.3 - EXCEPT

É a diferença da tabela da esquerda menos a da direita:

```
SELECT valor FROM tab_mult2 EXCEPT SELECT valor FROM tab_mult3;
```

Só foram retornados os valores que só existem em “tab_mult2”.

17.9 - Expressões Regulares

Tem grande utilidade em operações que requerem manipulação complexa de strings.

Operador	Sintaxe	Descrição
~	~ 'string'	Retorna a string exata
~*	~* 'string'	Retorna a string de modo insensitive
!~	!~ 'string'	Retorna o que não for a string exata
!~*	!~* 'string'	Retorna o que não for a string exata de modo insensitive
^	~ '^string'	Retorna o que começa com a string
$	~ 'string$'	Retorna o que acaba com a string
[]	~ [xyz]	Retorna o que contiver algum dos caracteres declarados entre os colchetes
[-]	~ [0-5]	Retorna o que contiver o intervalo de caracteres que é separado pelo hífen
^[-]	~ ^[a-z]	Retorna o que começa com o algum dos caracteres do intervalo
|	string1 |string2	Retorna se alguma das strings coincide

Exemplos:

```
SELECT * from colaboradores where nome ~ 'Chiquinho'; /* Procura pelo nome “Chiquinho” */;

SELECT * from colaboradores where nome ~* 'chiquinho'; /* Procura pelo nome “chiquinho” (insensitive) */;

SELECT * from colaboradores where uf !~ 'SP'; /* Retorna o que não tem “SP” em uf */;
```

17.10 - WHERE

A cláusula WHERE é usada para fazer filtragem nas consultas. A palavra “where” traduzida significa “onde”. Na linguagem SQL ela pode ser interpretada como por exemplo “selecionar as colunas de uma determinada tabela, **onde** atenda a esta(s) condição(ões): <condição(ões)>”.

Para determinar uma condição usamos operadores, exemplos:

Selecionar todos os campos da tabela colaboradores, onde o setor é “Marketing” e que não seja do estado de SP:

```
SELECT * FROM colaboradores WHERE setor='Marketing' AND uf!='SP';
```

Buscar dentre os registros aqueles em que o campo “cargo” não foi preenchido:

```
SELECT * FROM colaboradores WHERE cargo IS NULL;
```

17.10.1 - OR, AND, IN, agrupamento de comparações e BETWEEN

Exemplos:

Consultar os colaboradores que são do setor de “Marketing” **ou** de “Publicidade”:

```
SELECT * FROM colaboradores WHERE setor = 'Marketing' OR setor = 'Publicidade';
```

O comando acima não ficou de uma forma inteligente e elegante, soou repetitivo. Podemos melhorar isso com o operador `IN`:

```
SELECT * FROM colaboradores WHERE setor IN ('Marketing','Publicidade');
```

Descobrir quem tem o cargo de diretor **e** é de SP **e** do setor de “Marketing”:

```
SELECT * FROM colaboradores WHERE cargo = 'Diretor' AND uf = 'SP' AND setor = 'Marketing';
```

O comando anterior é outro exemplo que podemos melhorar a sintaxe, através do **agrupamento de comparações**, em casos que o operador é o mesmo (no caso exposto, de igualdade), em que de um lado se colocam os campos e no outro seus respectivos valores:

```
SELECT * FROM colaboradores WHERE (cargo,uf,setor) = ('Diretor','SP','Marketing');
```

Como fazer uma consulta por faixa salarial? Quem ganha **entre** R$ 1000,00 e R$ 2000,00?

```
SELECT * from colaboradores WHERE salario BETWEEN 1000 AND 2000;
```

O comando acima é o equivalente a:

```
SELECT * from colaboradores WHERE salario >=1000 AND salario<=2000;
```

No entanto, usando **`BETWEEN`** é mais fácil de se entender e dependendo do caso pode economizar digitação de forma inteligente.

17.10.2 - O Operador LIKE ou ~~

Tanto faz usar a palavra “LIKE” ou “~~”, pois o efeito será o mesmo. A finalidade é fazer consultas *case sensitive* tomando como base uma string ou parte dela através de caracteres curingas:

Símbolo	Função
%	Porcentagem: simboliza um, vários ou nenhum caractere na posição.
_	Underline: simboliza um úncio caractere na posição.
\\	Barra invertida dupla: Escape, faz com que se interprete literalmente os caracteres “%” e “_”.

Exemplos:

Buscar colaboradores que seu nome inicia com a letra “A”:

```
SELECT * FROM colaboradores WHERE nome ~~ 'A%';
```

Selecionar colaboradores que a segunda letra de seu nome é “a”:

```
SELECT * FROM colaboradores WHERE nome LIKE '_a%';
```

17.10.3 - O Operador ILIKE ou ~~*

Sua única diferença com o LIKE é 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.

17.10.4 - SIMILAR TO

Recurso parecido com o operador LIKE, porém possui mais recursos de pesquisa de strings através do uso de caracteres curinga.

Curinga		Descrição
%	_	São os mesmos curingas usados no LIKE.
[A-J]		Qualquer caractere entre “A” e “J” na posição
[XYZ]		“X”, “Y” ou “Z” na posição
[^XYZ]		Qualquer caractere, exceto “X”, “Y” ou “Z”
|		Operador OR
||		Concatenação

Exemplos:

Todos os nomes de colaboradores que a segunda letra é uma vogal sem acentuação:

```
SELECT * FROM colaboradores WHERE nome SIMILAR TO '_[aeiou]%';
```

Todos os nomes de colaboradores que a segunda letra não é uma vogal sem acentuação:

```
SELECT * FROM colaboradores WHERE nome SIMILAR TO '_[^aeiou]%';
```

Busca de registros cuja uf não seja “SP” ou “MG”:

```
SELECT * FROM colaboradores WHERE uf NOT SIMILAR TO '(SP|MG)';
```

Nomes cuja primeira letra é de “S” à “Z”:

```
SELECT * FROM colaboradores WHERE nome SIMILAR TO '[S-Z]%';
```

17.10.5 - O operador IS NULL

Valores nulos significam não preenchimento. Muitas vezes, equivocadamente para fazer uma consulta em que se buscam valores nulos usa-se o sinal de igualdade “=”. Ao invés disso, o correto é usar operador `IS`:

Exemplos:

Fazer uma consulta para achar registros onde o campo “cargo” não foi preenchido:

Forma errada:

```
SELECT * FROM colaboradores WHERE cargo = NULL;
```

Forma correta:

```
SELECT * FROM colaboradores WHERE cargo IS NULL;
```

17.10.6 - O operador NOT

Significa negação de algo.

Exemplo:

Filtrando resultados sem valores nulos em “cargo”:

```
SELECT * FROM colaboradores WHERE cargo IS NULL;
```

17.11 - 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.

Exemplos:

Consultar todos os funcionários ordenando pela data de admissão (`campo dt_admis`):

```
SELECT * FROM colaboradores ORDER BY dt_admis;
```

Por padrão consultas com a cláusula `ORDER BY` são exibidas em ordem ascendente (`ASC`). O comando anterior também poderia ser dado desta maneira, especificando que a ordem é crescente:

```
SELECT * FROM colaboradores ORDER BY dt_admis ASC;
```

Ou colocar em ordem decrescente (`DESC`):

```
SELECT * FROM colaboradores ORDER BY dt_admis DESC;
```

É possível também especificarmos mais de uma coluna e tipos de ordem diferentes pra cada, como por exemplo consultar primeiramente pela data de admissão decrescente e nome crescente:

```
SELECT * FROM colaboradores ORDER BY dt_admis DESC, nome ASC ;
```

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 `nome=2` e `snome=3`.

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

```
SELECT * FROM colaboradores ORDER BY 3,2;
```

Classificando por apelido de coluna:

```
SELECT nome||' '||snome AS "Nome Completo",setor,cidade||' - '||uf AS "Localidade" FROM colaboradores ORDER BY "Nome Completo";
```

Usando colunas invisíveis... Colunas invisíveis são assim chamadas aquelas que são usadas como critério de classificação, mas não aparecem no resultado:

```
SELECT nome||' '||snome AS "Nome Completo",setor,cidade||' - '||uf AS "Localidade" FROM colaboradores ORDER BY dt_admis DESC;
```

A coluna `dt_admis` não aparece no resultado.

17.12 - SELECT INTO

Cria uma nova tabela com os resultados da consulta.
Diferente de um `SELECT` convencional, os resultados não são mostrados e sim inseridos na nova tabela.

Exemplo:

Criar uma tabela idêntica à tabela “colaboradores”, apenas quando `uf = “SP”`:

```
SELECT * INTO colaboradores_sp FROM colaboradores WHERE uf='SP';
```

Após uma tabela ser criada no PostgreSQL, não se pode mais alterar a ordem das colunas com o comando `ALTER TABLE`. Uma solução para isso é usar o `SELECT INTO` para criar uma nova tabela, com os mesmos dados da antiga, apagar a tabela cuja ordem não é interessante e renomear a nova tabela para o nome da antiga.
Na tabela `colaboradores` , a coluna `cargo` vem antes de `setor`, então vamos trocá-las de lugar:

```
SELECT mat_id,nome,snome,setor,cargo,uf,cidade,salario,dt_admis INTO colaboradores_tmp FROM colaboradores; — Nova tabela criada, na ordem de colunas desejada;

DROP TABLE colaboradores; — Tabela antiga apagada;

ALTER TABLE colaboradores_tmp RENAME TO colaboradores; — Tabela nova renomeada;
```

17.13 - CREATE TABLE AS

Assim como `SELECT INTO`, também cria uma nova tabela com os resultados da consulta.

Exemplo:

```
CREATE TABLE diretores AS SELECT * FROM colaboradores WHERE cargo='Diretor';
--Nova tabela “diretores” criada a partir de uma consulta em “colaboradores”;
```

17.14 - Selecionando dados de diferentes tabelas

17.14.1 - Aliases de Tabelas (apelidos de tabelas)

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.

Uma saída pra isso é usar um apelido para a tabela antes do ponto e do nome da coluna e dizer a quem pertence o apelido:

Pose se usar “AS”:

```
SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM colaboradores AS c, funcionarios_premiados AS p
WHERE c.mat_id = p.mat_id;
```

Ou omitir “AS”:

```
SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM colaboradores c, funcionarios_premiados p
WHERE c.mat_id = p.mat_id;
```

17.14.2 - Joins

Em português, junção, uma JOIN reúne registros de tabelas diferentes em uma mesma consulta.

17.14.2.1 - Tipos de Joins

17.14.2.1.1 - CROSS JOIN (Junção cruzada)

Retorna uma conjunto de informações o qual é resultante de todas as combinações possíveis entre os registros das tabelas envolvidas:

```
SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM colaboradores c
CROSS JOIN funcionarios_premiados p;
```

17.14.2.1.2 - NATURAL JOIN (Junção natural)

Faz a junção tomando como base as colunas de mesmo nome nas tabelas envolvidas:

```
SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM funcionarios_premiados p
       NATURAL JOIN colaboradores c;
```

17.14.2.1.3 - INNER JOIN (Junção interna)

Retorna as informações apenas de acordo com as linhas que obedeçam as definições de relacionamento. Existe uma ligação lógica para se fazer a junção:

```
SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM funcionarios_premiados p
       INNER JOIN colaboradores c ON c.mat_id = p.mat_id;

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
       FROM funcionarios_premiados p
       INNER JOIN colaboradores c USING (mat_id);
```

17.14.2.1.4 - OUTER JOIN (Junção externa)

Assim como na `INNER JOIN`, existe uma ligação lógica, mas não retorna apenas as informações que satisfaçam a regra da junção. `OUTER JOIN`s 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.

Obs.: É de uso opcional a palavra `OUTER`.

```
        /* LEFT OUTER JOIN */;

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        LEFT OUTER JOIN colaboradores c USING (mat_id);

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        LEFT OUTER JOIN colaboradores c ON c.mat_id = p.mat_id;

        /* RIGHT OUTER JOIN */;

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        RIGHT OUTER JOIN colaboradores c USING (mat_id);

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        RIGHT OUTER JOIN colaboradores c ON c.mat_id = p.mat_id;

        /* FULL OUTER JOIN */;

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        FULL OUTER JOIN colaboradores c USING (mat_id);

SELECT c.nome||' '||c.snome AS "Funcionário contemplado",p.premio_id AS "Prêmio"
        FROM funcionarios_premiados p
        FULL OUTER JOIN colaboradores c ON c.mat_id = p.mat_id;
```

17.14.2.1.5 - SELF JOIN

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 aliases.

Exemplo:

Descobrir o nome e sobrenome do chefe direto de cada funcionário:

```
SELECT c2.nome||' '||c2.snome AS "Colaborador",c1.nome||' '||c1.snome AS "Chefe Direto"
        FROM colaboradores AS c1
        JOIN colaboradores c2 ON (c1.mat_id = c2.chefe_direto);
```

17.14.3 - Sub-consultas

Também conhecidas como *subquerries*, são SELECTs embutidos dentro de outro SELECT têm por finalidade flexibilizar consultas.

Esse recurso está disponível no PostgreSQL desde a versão 6.3.

17.14.3.1 - Sub-consulta no WHERE

```
SELECT nome,snome,salario FROM colaboradores
WHERE
salario=(SELECT min(salario) FROM colaboradores);
```

17.14.3.2 - Sub-consulta no SELECT

```
SELECT nome,snome,salario-(SELECT salario*0.06) AS "salario bruto"
FROM colaboradores WHERE salario < 3000;
```

17.14.3.3 - Sub-consulta no FROM

Subconsultas no FROM requerem um alias:

```
SELECT * FROM (SELECT * FROM colaboradores WHERE cidade='São Paulo') AS colab_cid_sp;
```

17.14.3.4 - EXISTS

Se a sub-consulta que ele antecede retornar pelo menos uma linha seu resultado é "*true*":

Exemplos:

Forçando um retorno verdadeiro:

```
SELECT nome,snome,salario FROM colaboradores
 WHERE EXISTS
(SELECT * FROM colaboradores WHERE salario<9000);

Forçando um retorno falso:

SELECT nome,snome,salario FROM colaboradores
 WHERE EXISTS
(SELECT * FROM colaboradores WHERE salario=9000);
```

17.14.3.5 - IN e NOT IN

Retorna verdadeiro para os valores que casem com o requisito.

Exemplo:

Mostrar todos os colaboradores que não têm o salário máximo:

```
SELECT nome,snome,salario FROM colaboradores
WHERE salario NOT IN
(SELECT max(salario) FROM colaboradores);
```

17.14.3.6 - ANY ou SOME

Tanto faz escrever um ou outro, pois o efeito é o mesmo. Retorna verdadeiro para qualquer valor da sub-consulta que satisfaça a condição:

Exemplo:

Retornar apenas os salários que estiverem entre R$ 2000,00 e R$ 5000,00:

```
SELECT nome,snome,salario FROM colaboradores
WHERE salario = SOME
(SELECT salario FROM colaboradores WHERE salario BETWEEN 2000 AND 5000);
```

17.14.3.7 - ALL

Retorna somente se todos valores casarem com a sub-consulta.

Exemplo:

Retornar todos os salários que estiverem fora da faixa R$ 2000,00 a R$ 5000,00:

```
SELECT nome,snome,salario FROM colaboradores
 WHERE salario!=ALL

(SELECT salario FROM colaboradores WHERE salario BETWEEN 2000 AND 5000);
```

18 - VIEW

View (ou visão) é uma consulta armazenada, a qual é acessada como uma tabela virtual.

Usa-se visões para facilitar o gerenciamento e deixar o design do banco de dados com um aspecto melhor e assim encapsulando os detalhes estruturais de tabelas.

Como o próprio nome diz, uma visão apenas exibe informações, nela não são inseridos dados.

Pode-se inclusive construir views baseando-se em outras views.

Exemplo:

```
CREATE OR REPLACE VIEW vw_teste AS
       SELECT * FROM colaboradores WHERE uf='SP' AND nome ~* '^c';
```

19 - Esquemas (Schemas)

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.

Por padrão, quando criamos objetos, os mesmos pertencem ao esquema público (public). Sendo assim quando fazemos uma consulta, não precisamos especificar o esquema:

```
SELECT * FROM colaboradores;
```

Isto também produz o mesmo resultado até então:

```
SELECT * FROM public.colaboradores;
```

```
esquema.nome_do_objeto
```

Para um melhor entendimento, criaremos um esquema chamado paralelo:

```
CREATE SCHEMA paralelo;
```

Criando um objeto dentro do novo esquema:

```
CREATE TABLE paralelo.tabela_teste(
       campo1 int,
       campo2 int
);
```

```
CREATE tipo_de_objeto nome_do_esquema.nome_do_objeto …;
```

Acessando dados de uma tabela criada em um determinado esquema:

```
SELECT * FROM paralelo.tabela_teste;
```

Listando todas as tabelas dentro do esquema paralelo:

```
\dt paralelo.*
```

Obs.: Isso vale pra outros tipos de objetos, basta trocar o “t” pela letra correspondente ao tipo de objeto, conforme pode ser conferido na ajuda de comandos internos do psql (\?).

Alterando o esquema de uma tabela:

```
ALTER TABLE colaboradores SET schema paralelo;
```

Como a tabela colaboradores fazia parte do esquema público, o mesmo não precisou ser mencionado, mas para fazê-la voltar para o esquema público será preciso mencionar o esquema:

```
ALTER TABLE paralelo.colaboradores SET schema public;
```

20 - Funções

O PostgreSQL fornece um grande número de funções nativas para várias finalidades, as quais são executadas dentro de um comando SQL.

20.1 - Funções Matemáticas

São funções cujo objetivo é facilitar determinados tipos de cálculos.

Função	Descrição	Exemplo	Resultado
pow(b,e)	Potência: Eleva um número de uma base (b) a um expoente (e)	SELECT pow(2,3);	8
log(x)	Logaritmo de “x”	SELECT log(100);	2
mod(x,y)	Resto de divisão: Retorna o resto da divisão de “x” por “y”	SELECT mod(19,5);	4
sqrt(x)	Raiz quadrada de “x”	SELECT sqrt(49);	7
ceil(x)	Arredondamento para cima de “x”	SELECT ceil(9.1);	10
floor(x)	Arredondamento para baixo de “x”	SELECT floor(9.999);	9
round(x)	Arredondamento padrão de “x”	SELECT round(9.5);	10
pi()	Pi	SELECT pi();	3.14159265358979

20.2 - Funções de Data e Hora

Função	Descrição	Exemplo
current_date	Data atual	SELECT current_date + 10; /* Daqui a 10 dias*/
current_time	Hora atual	SELECT current_time;
current_timestamp	Data e hora	SELECT current_timestamp; ou SELECT now();
to_char(dt,mask)	Converte a data de acordo com a máscara definida	SELECT to_char(now(),'"Data: " DD/MM/YYYY "Hora:" HH24:MI:SS');

20.2.1 - extract(...)

Tem como objetivo pegar um dado de `current_timestamp` tais como:

Descrição	Exemplo
Dia	SELECT extract(day from now());
Mês	SELECT extract(month from now());
Ano	SELECT extract(year from now());
Hora	SELECT extract(hour from now());
Minuto	SELECT extract(minute from now());
Segundos	SELECT extract(secondfrom now());
Década	SELECT extract(decade from now());
Dia da semana	SELECT extract(dow from now());
Dia do ano	SELECT extract(doy from now());
Época	SELECT extract(epoch from now());

20.3 - Funções de Strings

Função	Descrição	Exemplo
trim(string,'x')	Retira o caractere “x” da stringsel	SELECT trim('xxxxABCxxxx','x');
ltrim(string,'x')	Retira o caractere “x” da string à esquerda	SELECT ltrim('xxxxABCxxxx','x');
rtrim(string,'x')	Retira o caractere “x” da string à direita	SELECT rtrim('xxxxABCxxxx','x');
lower(string)	Exibe a string em letras minúsculas	SELECT lower('tEsTe');
upper(string)	Exibe a string em letras maiúsculas	SELECT upper('tEsTe');
initcap(string)	Exibe a string com a primeira letra maiúscula	SELECT initcap('tEsTe');
length(string)	Informa a quantidade de caracteres da string	SELECT length('tEsTe');
substr(string,n,x)	Retorna uma sub-string da posição “n” de comprimento “x”	SELECT substr('tEsTe',1,3);
translate(string,o,n)	Substitui os caracteres originais (“o”) para os novos (“n”), de acordo com a posição	SELECT translate('hacker','ae','43');
strpos(string,c)	Informa a posição da primeira ocorrência do caractere “c” na string	SELECT strpos('goiaba','a');
lpad(string,x,c)	Define o tamanho (“x”) máximo que a string será exibida e preenche com o caractere “c” à esquerda quando for menor que “x”.	SELECT lpad('Linux rulz!!',20,'-');
rpad(string,x,c)	Define o tamanho (“x”) máximo que a string será exibida e preenche com o caractere “c” à direita quando for menor que “x”.	SELECT rpad('Linux rulz!!',20,'-');
split_part(string,d,n)	Retorna o campo “n” tomando como referência o delimitador “d”	SELECT split_part('Steve Harris',' ',2) AS "Sobrenome";

20.3.1 - to_char()

Essa função 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.

Para ver a ajuda de sintaxe da função digite: \df to_char

Sendo que do que foi apresentado como segundo argumento é tudo *text*, uma máscara para modelar a conversão desejada.

Exemplos:

Converter o número 2009 para algarismos romanos:

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

Apresentar a data e hora atual no formato dd/mm/yyyy – hh:mm:

```
SELECT to_char(now(),'dd/mm/yyyy – hh:mm');
```

Determinar que um código será composto por 5 algarismos, preenchidos por zeros à esquerda:

```
SELECT to_char(53,'00000');
```

Até 9 (nove) algarismos + 2 (duas) casas decimais, preenchidos com zeros e separador de milhar:

```
SELECT to_char(29535.21,'000G000G000D00');
```

Número de até 9 (nove) algarismos + 2 (duas) casas decimais, não preenchidos com zeros e separador de milhar:

```
SELECT to_char(29535.21,'999G999G999D99');
```

20.4 - Funções de Sistema

Função	Descrição	Exemplo
current_database()	Banco de dados em uso	SELECT current_database();
current_user	Informa qual é o atual usuário	SELECT current_user; ou SELECT user;
version()	Versão do PostgreSQL	SELECT version();
current_schema()	Esquema corrente	SELECT current_schema();
current_schemas(bool)	Esquemas no caminho de procura incluindo, opcionalmente (true), os esquemas implícitos	SELECT current_schemas(true);
current_setting(string)	Informa uma configuração de sistema	SELECT current_setting('datestyle');
inet_server_addr()	Retorna o endereço IP do servidor	SELECT inet_server_addr();
inet_server_port()	Retorna a porta do servidor PostgreSQL	SELECT inet_server_port();

20.5 - Funções de Agregação (ou de Grupos de Dados)

Função	Descrição
avg(expressão)	Média aritmética dos valores de entrada
`SELECT avg(salario) FROM colaboradores;`	
count(*)	A quantidade de valores de entrada
`SELECT cont(*) FROM colaboradores;`	
count(expressão)	A quantidade de valores de entrada em que o valor da expressão não é nulo
`SELECT count(setor) FROM colaboradores;`	
max(expressão)	Retorna o valor máximo dentre todos os valores de entrada

`SELECT nome,snome,salario FROM colaboradores`
` WHERE`
`salario=(SELECT max(salario) FROM colaboradores);`	

min(expressão)	Retorna o valor mínimo dentre todos os valores de entrada

`SELECT nome,snome,salario FROM colaboradores`
` WHERE`
`salario=(SELECT min(salario) FROM colaboradores);`	

sum(expressão)	Informa a soma dos valores de entrada da expressão
`SELECT sum(salario) FROM colaboradores;`	
stddev(expressão)	Desvio padrão
`SELECT stddev(salario) FROM colaboradores;`	
variance(expressão)	Variância estatística
`SELECT variance(salario) FROM colaboradores;`	

20.6 - A Cláusula GROUP BY

A cláusula GROUP BY concentra em apenas uma linha todos os registros cujo valor é o mesmo da coluna GROUP BY. É importante lembrar que todas as colunas relacionadas no SELECT que não estejam nas funções de grupo devem estar também na cláusula GROUP BY.

Exemplos:

Selecionar quantos colaboradores há por UF:

```
SELECT uf,count(uf) FROM colaboradores GROUP BY uf;
```

Consultar quantos colaboradores ganham pelo menos R$ 2.500,00 agrupando pelo estado de origem:

```
SELECT uf,count(uf) FROM colaboradores WHERE salario >= 2500 GROUP BY uf;
```

20.7 - A Cláusula HAVING

Tem por finalidade filtrar os dados agrupados impondo uma condição. Tal condição deve ter alguma das colunas do `SELECT`.

Exemplo:

Consultar quantos colaboradores ganham pelo menos R$ 2.500,00 agrupando pelo estado de origem, mas apenas quando o estado possuir mais que 3 (três) colaboradores que atendam ao requisito:

```
SELECT uf,count(uf) FROM colaboradores WHERE salario >= 2500 GROUP BY uf HAVING count(uf)>3;
```

21 - INSERT – inserção de Dados

O comando INSERT cria novos registros em uma tabela.

Sintaxe geral:

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

Como exemplo, vamos inserir um novo funcionário na tabela colaboradores, mencionando todos os campos:

```
INSERT INTO colaboradores
(mat_id,nome,snome,cargo,setor,uf,cidade,salario,dt_admis,chefe_direto)
VALUES
('00036','Teobaldo','Melo','Diretor','Financeiro',DEFAULT,'São
Paulo',4523.75,'09/06/2009','00001');
```

Analisando o código acima podemos notar que logo após o nome da tabela abre-se parênteses e vem a sequência de nomes de colunas separados por vírgulas e após o nome da última há o fechamento de parênteses.

A palavra chave `VALUES` anuncia os valores que serão inseridos de acordo com a posição do nome de coluna.

Para valores de tipos não numéricos, datas ou tempo usa-se aspas simples para delimitar. Tipos numéricos, palavras chave, chamada de funções ou identificadores não são envolvidos por aspas simples.

Um dos valores a ser inserido usa a palavra chave “DEFAULT”, cujo, na estrutura da tabela “colaboradores” foi declarado como valor padrão igual a “SP”.

Quando vai inserir valores para todos os campos de uma tabela, não é preciso declarar os nomes das colunas, podendo assim ganhar agilidade. Então o comando acima poderia ser feito da seguinte forma:

```
INSERT INTO colaboradores VALUES
('00036','Teobaldo','Melo','Diretor','Financeiro',DEFAULT,'São
Paulo',4523.75,'09/06/2009','00001');
```

Adicionando apenas alguns valores:

```
INSERT INTO colaboradores (mat_id,nome,snome,dt_admis)
       VALUES ('00037','Hugo','Nóbrega',now());
```

Foram usados apenas os campos referentes à matrícula, ao nome, ao sobrenome e à data de admissão do funcionário inserido, os outros campos (os não declarados), ou ficaram nulos ou com algum valor padrão.

Nesse caso, para o campo `dt_admis`, usamos a função `now()` que preencheu com a data atual.

Vale lembrar que para colunas do tipo `NOT NULL` o preenchimento é obrigatório.

21.1 - Inserindo Registros com SELECT

Método similar ao `SELECT INTO` e `CREATE TABLE AS`, é uma forma de passarmos dados de uma tabela para outra.

Para exemplo, vamos primeiro criar uma tabela temporária, com poucos campos. cuja finalidade será conter apenas os funcionários amazonenses:

```
CREATE TEMP TABLE "colaboradores_AM" (
       mat_id CHAR(5) PRIMARY KEY,
       nome VARCHAR(15),
       snome VARCHAR(15)
);
```

Inserindo os dados na nova tabela através de uma consulta filtrada em “colaboradores”:

```
INSERT INTO "colaboradores_AM"
       SELECT mat_id,nome,snome FROM colaboradores WHERE uf='AM';
```

21.2 - INSERT Agrupado

Se for necessário inserir mais de um registro, não é necessário fazer vários INSERTs.

Pode ser feito separando cada conjunto de valores dos registros (delimitados por parênteses), separados por vírgulas e após o último ao invés da vírgula, ponto e vírgula:

```
INSERT INTO colaboradores (mat_id,nome,snome) VALUES
       ('00038','teste','insert_1'),
       ('00039','tEsTe','insert_2'),
       ('00040','TEste','insert_3');
```

22 - Dollar Quoting

Dollar Quoting é uma forma alternativa de inserir strings sem usar as aspas simpes 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 dollar quoting se aplica a strings, sendo que pode ser feito de duas formas:

```
$$sua string$$
```

ou

```
$marcador$sua string$marcador$
```

Sendo que “marcador” pode ser qualquer string que você quiser escolher para denominar.

Exemplo:

```
INSERT INTO tabela (campo_string) VALUES ($dfghjkliu$teste teste teste$dfghjkliu$);
```

O valor inserido foi “teste teste teste” (sem as aspas), usamos como marcador “$dfghjkliu$”.

O uso de dollar quoting nos permite inserir strings com caracteres especias, tais como a própria aspas simples:

```
INSERT INTO tabela (campo_string) VALUES ($outro_marcador$'teste'$outro_marcador$);
```

Nesse caso, o valor inserido foi:

```
'teste'
```

Sim! Com o uso de dollar quoting o insert considerou as aspas simples como um caracter comum.

Obs.: Apesar dos exemplo dos exemplos terem sido com INSERT, poderiam ser de acordo com o contexto usando outros comandos DML (`SELECT`, `UPDATE` e `DELETE`).

22 - 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 “colaboradores”, levando-se em conta que os trabalhadores que ganham menos ou igual a R$ 1.200,00 e são do estado de São Paulo receberam um aumento de 10% no salário:

```
UPDATE colaboradores SET salario = salario*1.1 WHERE salario<=1200 AND uf ILIKE 'sp';
```

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.

23.1 - Atualizando Mais de Um Campo

Podemos também fazer atualização de registros referenciando mais de um campo.
Vamos supor que por um motivo qualquer a unidade de Manaus teve que mudar para Belém:

```
UPDATE colaboradores SET (uf,cidade) = ('PA','Belém') WHERE uf='AM';
```

Para voltar ao normal:

```
UPDATE colaboradores SET (uf,cidade) = ('AM','Manaus') WHERE uf='PA';
```

Poderia ser feito também da seguinte forma:

```
UPDATE colaboradores SET uf='RN',cidade='Natal' WHERE (uf,cidade) = ('SP','São Paulo');
```

Voltando ao que era antes:

```
UPDATE colaboradores SET uf='SP',cidade='São Paulo' WHERE uf='RN' AND cidade='Natal';
```

23.2 - Atualizando com Sub-consulta

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 colaboradores SET salario =
(SELECT avg(salario) FROM colaboradores WHERE salario<2000)
WHERE salario<=1050;
```

24 - DELETE – Removendo Registros

Para apagar linhas de uma tabela usa-se o comando DELETE.

Sintaxe geral:

```
DELETE FROM tabela WHERE coluna condição;
```

Apagando todos os registros cuja coluna “cargo” tem valor nulo:

```
DELETE FROM colaboradores WHERE cargo IS NULL;
```

Assim como em outros exemplos, pode-se usar sub-consultas para também deletar dados.

25 - 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: **DDL**.

Sintaxe geral:

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

Vamos tentar fazer o comando `TRUNCATE` na tabela “colaboradores”:

```
TRUNCATE colaboradores;
```

Erro! Não foi possível, pois a mesma é referenciada por outra tabela, sendo então necessário fazer a remoção em **cascata**, usando o parâmetro **`CASCADE`**:

```
TRUNCATE colaboradores CASCADE;
```

Todos os dados foram perdidos! Mas para continuarmos o curso simplesmente copie a parte de INSERTS do arquivo curso.sql e cole no prompt do psql.

26 - COPY

O comando COPY copia registros entre um arquivo e uma tabela, ou de uma tabela para STDOUT (standard output → saída padrão) ou de STDIN (standard input → entrada padrão) para uma tabela.

Para se ter uma idéia de como funciona o comando COPY, vamos primeiro copiar os dados para a tela:

```
COPY colaboradores TO stdout;
```

Podemos observar que o resultado obtido é similar a mandar selecionar todos os dados. Cada campo é separado por espaço, mas podemos usar outro delimitador como a vírgula, por exemplo:

```
COPY colaboradores TO stdout DELIMITER ',';
```

O delimitador é determinado pelo caractere entre as aspas simples.

Para prosseguirmos nossos exemplos com COPY vamos criar uma tabela temporária contendo apenas os funcionários de São Paulo:

```
CREATE TEMP TABLE colab_sp AS SELECT * FROM colaboradores WHERE uf='SP';
```

Com o comando COPY enviaremos os dados dessa tabela para o arquivo “/tmp/backup_sp.txt”, usando como delimitador o caractere pipe (“|”):

```
COPY colab_sp TO '/tmp/backup_sp.txt' DELIMITER '|';
```

Usando o comando TRUNCATE para redefinir a tabela “colab_sp” como uma tabela vazia:

```
TRUNCATE colab_sp ;
```

E recuperamos os dados que nela estavam usando COPY:

```
COPY colab_sp FROM '/tmp/backup_sp.txt' DELIMITER '|';
```

26.1 - Formato CSV

CSV significa Comma Separeted Values (Valores Separados por Vírgula), que no caso são os valores respectivos aos campos de uma tabela.

Para gerarmos um CSV:

```
COPY colab_sp TO '/tmp/backup_sp.csv' CSV;
```

Com o parâmetro CSV o arquivo resultante é gerado automaticamente com vírgulas como delimitadores de colunas, mesmo as que estão nulas.

Para recuperar dados de um arquivo CSV para uma tabela:

```
COPY colab_sp FROM '/tmp/backup_sp.csv' CSV;
```

26.2 - Inserindo Dados com o COPY pelo Teclado

Uma alternativa para inserir dados diferente do comando “INSERT”:

```
COPY colab_sp FROM stdin CSV;
```

Aparecerá uma mensagem na tela:

```
Informe os dados a serem copiados seguido pelo caractere de nova linha.
Finalize com uma barra invertida e um ponto na linha.
>>
```

Os dois sinais “maior que” indicam a linha corrente para se inserir dados. No exemplo foi determinado o formato `csv` e então como exemplo digitaremos:

```
00032,Paula,Franco,Programador,TI,SP,Jundiaí,2100,2007-09-30,00003
<ENTER>
00030,Martina,Santos,An. de Sistemas,TI,SP,Jundiaí,4200,2007-09-30,00003
<ENTER>
\.
```

27 - SEQUENCE – Sequência

Uma sequência é usada para determinar valores automaticamente para um campo.

Vejamos como se cria uma:

```
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: Neste caso é usado os 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 seqüência 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.
CYCLE: Faz com que a seqüência recomece quando for atingido o valor_máximo ou o valor_mínimo por uma seqüência 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 CYCLE nem NO CYCLE, NO CYCLE é o padrão.

Exemplo:

Criar uma sequencia que se dê de 5 em 5, valor mínimo 15, valor máximo 500, com 20 números da sequência pré alocados na memória e sem ciclo:

```
CREATE SEQUENCE sq_teste
       INCREMENT 5
       MINVALUE 15
       MAXVALUE 500
       CACHE 20
       NO CYCLE;
```

27.1 - 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:

```
SELECT nextval('sq_teste');
```

(Repita três vezes)

Observamos seu valor corrente:

```
SELECT currval('sq_teste');
```

Mudamos seu valor para 20:

```
SELECT setval('sq_teste',20);
```

27.2 - Usando Sequencias em Tabelas

Dentre os tipos de dados que foram vistos, o “serial” nada mais é do que uma sequência criada na hora, de acordo com o nome da tabela. Vejamos este exemplo:

```
CREATE TEMP TABLE tb_teste(
       cod serial,
       nome VARCHAR(15)
);
```

É exibida uma nota:

```
CREATE TABLE criará sequência implícita "tb_teste_cod_seq" para coluna serial "tb_teste.cod"
```

Vamos fazer novamente a tabela, mas com a sequência “sq_teste”:

```
CREATE TABLE tb_teste(
       cod int DEFAULT nextval('sq_teste'),
       nome VARCHAR(15)
);
```

Testando:

```
INSERT INTO tb_teste (nome) VALUES ('nome1'),('nome2'),('nome3'),('nome4');

SELECT * FROM tb_teste ;
```

Como pôde ser observado, o próximo valor de “sq_teste” foi tomado como um valor padrão para o campo “cod”. Um campo auto incremental.

27.3 - Alterando uma Sequência

Usando o comando “\h ALTER SEQUENCE” podemos ver o que pode ser mudado na sequência desejada, quase igual a criar uma. Como exemplo vamos voltar o valor para 15:

```
ALTER SEQUENCE sq_teste RESTART WITH 15;
```

De agora em diante os valores serão a partir do 15 em “sq_teste”.

27.4 - Deletando uma Sequência

Podemos apagar uma ou mais sequências usando a seguinte sintaxe:

```
DROP SEQUENCE [ IF EXISTS ] nome [, ...] [ CASCADE | RESTRICT ]
```

Exemplo:

```
DROP SEQUENCE sq_teste ;
```

Erro! Pois “sq_teste” está vinculada à tabela “tb_teste”, então devemos fazer a remoção em cascata:

```
DROP SEQUENCE sq_teste CASCADE;
```

O valor padrão da tabela “tb_teste” foi removido.

28 - INDEX - Índice

Um índice (INDEX) é um recurso que agiliza buscas de informações em tabelas.
Indexamos campos usados como critérios de filtragem numa consulta (cláusula WHERE, por exemplo) e aqueles cujos valores são mais restritivos comparados a outros valores da tabela.
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.

Obs.: Ao se criar chaves primárias, automaticamente um índice é criado.

Sintaxe geral:

```
CREATE [UNIQUE] INDEX nome_do_index ON tabela [USING método] (campo);
```

O parâmetro “UNIQUE” faz com que a coluna indexada só aceite valores únicos. No entanto, se a tabela já estiver com valores duplicados na coluna desejada não será possível criar o índice como “UNIQUE”.

Exemplo:

```
CREATE INDEX idx_colab_nome ON colaboradores USING BTREE (nome);
```

28.1 - Métodos de Indexação

O PostgreSQL fornece quatro tipos de índices: BTREE (padrão), GIST, GIN e HASH.
Cada tipo de índice usa um algoritmo diferente, cada qual se encaixa melhor em algum tipo de consulta.

28.1.1 - BTREE

Podem lidar com consultas de faixas ou igualdades em dados que podem ser organizados ordenadamente.
Este tipo de índice deve será considerado pelo query planner quando a coluna indexada envolve comparações usando algum desses operadores: **<**, **<=**, **=**, **=>** e **>**.

28.1.2 - GIST

Não são tipos singulares de índices, mas sim dentro de uma infra-estrutura quem muitas diferentes

estratégias de indexação podem ser implementadas. Assim, os operadores com os quais um índice GIST pode ser usado varia dependendo da estratégia de indexação (a classe de operador). Por exemplo, a distribuição padrão do PostgreSQL inclui classes de operadores GIST para vários tipos de dados geométricos bi-dimensionais, que suporta consultas usando estes operadores: **<<**, **&<**, **&>**, **>>**, **<<|**, **&<|** , **|&>**, **|>>** , **@>** , **<@** , **~=** e **&&**.

28.1.3 - GIN

São índices invertidos que podem lidar com valores que contenham mais de uma chave,arrays por exemplo. Como o GIST, pode suportar muitas estratégias diferentes de indexação definidas por usuários e os operadores particulares com que um índice GIN pode ser usado dependendo da estratégia de indexação. O PostgreSQL por padrão inclui classes de operadores GIN para arrays uni-dimensionais, que suportam consultas usando estes operadores: **<@**, **@>**, **=**, **&&**.

28.1.4 - HASH

Lidam apenas com comparações simples. O query planner considerará o índice hash sempre que uma coluna indexada estiver envolvida em uma comparação com o operador **=**.

Índices do tipo hash não suportam consultas `IS NULL`**.**

Obs.: Testes demonstram que índices hash do PostgreSQL não tem performance melhor que índices do tipo BTREE e o tamanho e tempo para construir é bem pior. O uso do índice hash é desencorajado.

28.2 - Índices Compostos

São aqueles que contém em sua composição mais de um campo, sendo que para um melhor desempenho deve-se declarar na criação do índice os campos com valores mais distribuídos no início.

Exemplo:

CREATE INDEX nome_do_index ON nome_da_tabela (campoX,campoY,campoZ);

28.3 - Índices Parciais

Quando se tem uma tabela com muitos registros, um índice comum pode não oferecer um desempenho não tão bom. Felizmente o Postgres tem um recurso que permite que sejam criados índices de acordo com uma condição estabelecida na sua criação.

Exemplo:

```
--Criação da tabela de teste
CREATE TABLE tb_teste_index(
       campo1 INT
);

--Inserção de 2 milhões de registros
INSERT INTO tb_teste_index VALUES (generate_series(1,2000000));

--Análise sem índices de valores múltiplos de 19 (sem índice)
EXPLAIN ANALYZE SELECT * FROM tb_teste_index WHERE campo1%19=0;
```

```
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------
--------------------
 Seq Scan on tb_teste_index  (cost=0.00..40710.00 rows=10620 width=4) (actual time=0.132..1009.375
```

```
rows=105263 loops=1)
   Filter: ((campo1 % 19) = 0)
 Total runtime: 1144.964 ms
(3 rows)

--Criação de índice total
CREATE INDEX idx_teste_index_total ON tb_teste_index (campo1);

--Criação de índice parcial múltiplos de 19
CREATE INDEX idx_teste_index_19 ON tb_teste_index (campo1) WHERE campo1%19=0;



--Análise sem índices de valores múltiplos de 19 (COM ÍNDICE)
EXPLAIN ANALYZE SELECT * FROM tb_teste_index WHERE campo1%19=0;

                                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------
-------------------------------------------
 Index Scan using idx_teste_index_19 on tb_teste_index  (cost=0.00..1892.58 rows=10000 width=4)
(actual time=0.286..292.026 rows=105263 loops=1)
 Total runtime: 430.361 ms
(2 rows)
```

Pelo exemplo acima pudemos constatar o que foi dito na teoria; antes da criação dos índices a busca foi sequencial, demorando 1144.964 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 430.361 ms, usufruindo de uma busca agora indexada por um índice parcial.

28.4 - Antes de Excluir um Índice

Não remova um índice de seu banco sem antes saber o quão útil ele é.

```
SELECT indexrelname,relname,idx_scan,idx_tup_read,idx_tup_fetch FROM pg_stat_user_indexes ;
```

Sendo que cada campo significa:

- **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.

28.5 - Excluindo um Índice

Síntaxe;

```
DROP INDEX [ IF EXISTS ] nome [, ...] [ CASCADE | RESTRICT ]
```

Exemplo;

```
DROP INDEX idx_colab_nome;
```

28.6 - 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. O comando REINDEX é utilizado nas seguintes situações:

- Índice corrompido e não contém mais dados válidos. Embora teoricamente isso nunca deva acontecer, na prática os índices podem se corromper devido a erros de programação ou falhas de hardware. O comando REINDEX fornece um método de recuperação.
- Índice “dilatado”. Contém muitas páginas vazias ou quase vazias. Tal situação pode acontecer com índices B-Tree sob usos fora do comum. O comando REINDEX fornece uma maneira para diminuir o consumo de espaço do índice através da escrita de uma nova versão sem as páginas mortas.

Sintaxe:

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nome
```

- 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;
- nome: Identificador de INDEX, TABLE, DATABASE ou SYSTEM

Exemplo:

```
REINDEX TABLE colaboradores ;
```

O comando acima reindexou todos índices da tabela “colaboradores”.

28.7 - CLUSTER

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 nome_tabela [ USING nome_índice ]
```

Exemplo:

```
CLUSTER colaboradores USING colaboradores_pkey ;
```

29 - DOMAIN (Domíno)

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 nome [ AS ] tipo_dado
       [ DEFAULT expressão ]
       [ restrição [ ... ] ]
```

onde restrição é:

```
[ CONSTRAINT nome_restrição ]
{ NOT NULL | NULL | CHECK (expressão) }
```

Exemplo:

Criação de um domínio, cuja finalidade é validar CEPs no formato “#####-###”:

```
CREATE DOMAIN dom_cep AS char(9)
       CONSTRAINT chk_cep
       CHECK (VALUE ~ E'^\\d{5}-\\d{3}$');
```

O domínio criado de nome “dom_cep” é do tipo char(9), pois recebe 5 (cinco) dígitos, um “-” (hífen) e por último mais 3 (três) dígitos.

Possui uma restrição denominada “chk_cep”, a qual verifica se o valor inserido estará de acordo com a expressão regular se o valor inserido está no valor determinado (5 dígitos + “-” + 3 dígitos).

Criação de uma tabela que usará o domínio criado como tipo de dado para uma coluna:

```
CREATE TEMP TABLE endereco(
       cep dom_cep,
       logradouro text,
       numero varchar(6),
       cidade varchar(20),
       uf char(2)
);
```

Um exemplo de inserção na tabela com o domíno criado:

```
INSERT INTO endereco VALUES ('01001-000','Pça. da Sé','s/n','São Paulo','SP');
```

30 - Transações / Controle de Simultaneidade

Transação é uma forma de se executar uma sequência de comandos indivisivelmente para fins de alterações, inserções ou remoções de registros.

30.1 - MVCC – Multi Version Concurrency Control (Controle de Concorrência Multi Versão)

De maneira 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 do banco de dados) 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.

30.2 - 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.

30.3 - 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.

30.4 - Níveis de Isolamento

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações simultâneas. Os **fenômenos não desejados** são:

- **Dirty Read (Leitura Suja)**: Uma transação lê dados gravados por outra transação concorrente não efetivada;
- **Nonrepeatable Read (Leitura que não se repete)**: A transação relê dados que foram previamente lidos e descobre que os dados foram modificados por outra transação (efetivados desde a primeira leitura);
- **Phantom Read (Leitura Fantasma)**: A transação executa novamente uma consulta e descobre que os registros mudaram devido a outra transação ter sido efetivada;

Nível de Isolamento	Dirty Read	Nonrepeatble Read	Phantom Read
Read Uncommited	Possível	Possível	Possível
Read Commited (padrão PostgreSQL)	Impossível	Possível	Possível
Repeatable Read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

No PostgreSQL qualquer um dos níveis de isolamento padrão pode ser escolhido, mas internamente só há dois níveis de isolamento distintos que são *Read Commited* e *Serializable.* Quando é selecionado *Read Uncommited* na verdade obtém-se *Read Commited* e quando se escolhe *Repeteatable Read* obtém-se *Serializable*. Então, o nível de isolamento real pode ser mais estrito do que o escoolhido. **O padrão SQL permite que os quatro níveis de isolamento apenas definem quais fenômenos não podem acontecer**, não definem quais fenômenos devem acontecer.

Podemos definir o nível de isolamento da transação usando o comando SET TRANSACTION.

Sintaxe:

```
SET TRANSACTION modo_transação [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION modo_transação [, ...]
```

onde modo_transação é um dos:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

Exemplo:

Definindo para a transaçao:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Verificando:

```
SHOW TRANSACTION ISOLATION LEVEL;
```

Definindo para a sessão aberta:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

30.4.1 - Nível de Isolamento Read Commited (Lê Efetivado)

É o nível de isolamento padrão do PostgreSQL. Em uma transação sob este nível, o comando SELECT enxerga apenas os dados efetivados antes da consulta e nunca dados não efetivados ou as alterações efetivadas por transações simultâneas durante a execução da consulta (porém, o SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). O comando SELECT, na verdade, enxerga um instantâneo do banco de dados, como era no momento em que a consulta começou. Deve ser notado que dois comandos SELECT sucessivos podem enxergar dados diferentes mesmo dentro da mesma transação, se outras transações efetivarem alterações durante a execução do primeiro SELECT.

30.4.2 - Nível de Isolamento Serializable (Serializável)

É o nível de isolamento mais rigoroso, o qual emula a execução em série das transações, como se fossem executadas uma após a outra, ao invés de ao mesmo tempo. Porém, os aplicativos que utilizemeste nível de isolamento devem estar preparados para tentar executar as transações novamente, devido à falhas de serialização.

Em uma transação contida nesse nível de isolamento, o comando SELECT enxerga apenas os dados efetivados antes da transação começar e nunca os não efetivados ou alterações efetivadas durante a execução da transação por transações simultâneas (Porém, o SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas).

Se difere do Read pelo fato de o SELECT enxergar um instantâneo do momento de início da transação, não do momento de início do comando corrente dentro da transação. Portanto, SELECTs sucessivos dentro de uma mesma transação sempre enxergam os mesmos dados.

30.5 - Fazendo uma transação

Uma transação deve começar com o comando BEGIN:

Sintaxe:

```
BEGIN [ WORK | TRANSACTION ] [ modo_transação [, ...] ]
```

Onde modo_transação é um dos:

```
   ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

As palavras `WORK` e `TRANSACTION` podem ser omitidas.

Após o `BEGIN`, que declara que a transação foi iniciada tem se a sequencia de comandos desejada:

```
comando_SQL_1;
…
comando_SQL_n;
```

E depois dos comandos, se por algum motivo quiser desfazê-los usa-se o comando ROLLBACK:

```
ROLLBACK;
```

Nos exemplos a seguir sobre transação, mesmo que os comandos sejam copiados e colados, devem ser feitos um por um de modo que se observe seu comportamento e suas relativas alterações.

É também aconselhável abrir uma outra sessão no mesmo banco de dados para observar o que acontece quando sessões diferentes tentam alterar o mesmo registro.

Exemplo:

```
BEGIN ; -- Início da transação;
        DELETE FROM colaboradores WHERE mat_id = '00035'; -- Remoção de um registro dentro da
transação;
        SELECT * FROM colaboradores ; -- Em uma transação podemos ver dados ainda não
efetivados;
ROLLBACK ; -- Agora vamos fazer o banco de dados voltar ao estado anterior;
```

O comando `ROLLBACK` desfaz todas as alterações feitas pelos comandos após o BEGIN. Mas se as alterações que os comandos fizeram devem ser efetivadas, usa-se o comando COMMIT:

```
COMMIT;
```

Exemplo:

```
BEGIN ;
        DELETE FROM colaboradores WHERE mat_id = '00035';
        SELECT * FROM colaboradores ;
COMMIT ; -- Agora efetivar as alterações;
```

Colocando de forma geral uma transação tem o seguinte corpo:

```
BEGIN;

comando_SQL_1;
…
comando_SQL_n;

[COMMIT | ROLLBACK]
```

30.6 - SAVEPOINT - Ponto de Salvamento

Define um novo ponto de salvamento na transação atual.

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 antes da criação do ponto de salvamento.

Sintaxe:

```
SAVEPOINT nome_savepoint;
```

30.6.1 - ROLLBACK TO SAVEPOINT

É usado para desfazer até o ponto de salvamento.

Sintaxe;

```
ROLLBACK TO [SAVEPOINT] nome_savepoint;
```

Obs.: A palavra “SAVEPOINT” pode ser omitida.

Exemplo:

```
BEGIN ;
       DELETE FROM colaboradores WHERE mat_id = '00034';
       SELECT * FROM colaboradores ;
       SAVEPOINT sp1; -- Criação do ponto de salvamento;
       UPDATE colaboradores SET salario = 600 WHERE mat_id = '00033';
       SELECT * FROM colaboradores ;
       ROLLBACK TO sp1;
       SELECT * FROM colaboradores ;
COMMIT ; -- Agora efetivar as alterações antes de “sp1” que não tiveram “ROLLBACK TO”;
```

30.6.2 - 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
```

Exemplo:

```
BEGIN ;
       DELETE FROM colaboradores WHERE mat_id = '00034';
       SELECT * FROM colaboradores ;
       SAVEPOINT sp1; -- Criação do ponto de salvamento;
       UPDATE colaboradores SET salario = 600 WHERE mat_id = '00033';
       SELECT * FROM colaboradores ;
       RELEASE sp1; -- O ponto de salvamento foi destruído;
       SELECT * FROM colaboradores ;
COMMIT; -- Mesmo com o ponto de salvamento destruído, as alterações podem ser efetivadas;
```

Os pontos de salvamento somente podem ser estabelecidos dentro de um bloco de transação. Podem haver vários pontos de salvamento definidos dentro de uma transação.

30.7 - 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.

Exemplo:

Sessão 1:

```
BEGIN;
	SELECT * FROM colaboradores WHERE cargo IS NULL FOR UPDATE;
	-- Todos os registros exibidos estão travados enquanto a transação não acabar;
```

Sessão 2:

```
	DELETE FROM colaboradores WHERE cargo IS NULL;
```

Sessão 1:

```
COMMIT;
```

Após a efetivação dos dados pela sessão 1 a sessão 2 pôde finalmente fazer as alterações que desejava.

31 - Cursores

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. Para usuários da linguagem PL/pgSQL, normalmente não há necessidade de se preocuparem com isso, já que os laços "FOR" utilizam internamente um cursor 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.

Cursores podem retornar dados no formato texto ou no formato binário pelo comando FETCH. Por padrão retornam dados no formato texto, assim como os produzidos pelo comando SELECT.

Sintaxe de criação de um cursor:

```
DECLARE nome [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
        CURSOR [ { WITH | WITHOUT } HOLD ] FOR comando
        [ FOR { READ ONLY | UPDATE [ OF coluna [, ...] ] } ]
```

Parâmetros:

nome	Nome do cursor;
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 seqüencial. 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.
comando	A consulta feita para o cursor retornar.
FOR READ ONLY	Indica que o cursor será utilizado no modo somente para leitura.
FOR UPDATE	Indica que o cursor será utilizado para atualizar tabelas. Uma vez que as atualizações por cursor não são suportadas atualmente pelo PostgreSQL, especificar FOR UPDATE provoca uma mensagem de erro, e especificar FOR READ ONLY não produz efeito.
coluna	As colunas a serem atualizadas pelo cursor. Uma vez que as atualizações por cursor não são suportadas atualmente pelo PostgreSQL, a cláusula FOR UPDATE provoca uma mensagem de erro.

Observações:

As palavras chave BINARY, INSENSITIVE e SCROLL podem estar em qualquer ordem.

Se WITH HOLD não for especificado, o cursor criado por este comando poderá ser utilizado somente dentro da transação corrente. E se a efetivação da transação que criou o cursor for bem-sucedida, o cursor poderá continuar sendo acessado pelas transações seguintes na mesma sessão (Mas se a transação que criou o cursor for interrompida, o cursor será removido). O cursor criado com WITH HOLD é fechado quando é submetido um comando CLOSE explícito para o cursor, ou quando a sessão termina.

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 DECLARE e OPEN.

Exemplos:

Criando um cursor dentro de uma transação:

```
BEGIN;
DECLARE cursor1 CURSOR FOR SELECT * FROM colaboradores;
…
COMMIT;
```

Criando um cursor fora de uma transação:

```
DECLARE cursor2 CURSOR WITH HOLD FOR SELECT * FROM colaboradores;
```

31.1 - 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 um dos:

NEXT	Próxima linha. Este é o padrão quando a direção é omitida.
PRIOR	Linha anterior.
FIRST	Primeira linha da consulta (o mesmo que ABSOLUTE 1).
LAST	Última linha da consulta (o mesmo que ABSOLUTE -1).
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.
n	As próximas n linhas (o mesmo que FORWARD n). É uma constante inteira, possivelmente com sinal, que determina a posição ou o número de linhas a serem retornadas. Para os casos FORWARD e BACKWARD, especificar um n negativo é equivalente a mudar o sentido de FORWARD e BACKWARD.
ALL	Todas as linhas restantes (o mesmo que FORWARD ALL).
FORWARD	Próxima linha (o mesmo que NEXT).
FORWARD n	As próximas n linhas. FORWARD 0 retorna novamente a linha corrente.
FORWARD ALL	Todas as linhas restantes.
BACKWARD	Linha anterior (o mesmo que PRIOR).
BACKWARD n	As n linhas anteriores (varrendo para trás). BACKWARD 0 retorna novamente a linha corrente.
BACKWARD ALL	Todas as linhas anteriores (varrendo para trás).

Exemplos:

```
FETCH cursor2; --Por padrão retorna a próxima linha
FETCH 2 IN cursor2; --Retorna as próximas 2 linhas
FETCH -1 IN cursor2; Retorna a linha anterior
FETCH FORWARD FROM cursor2; --Próxima linha
FETCH FORWARD 7 FROM cursor2; --Próximas 7 linhas
FETCH FIRST FROM cursor2; --Sempre retornará a primeira linha
FETCH LAST FROM cursor2; --Sempre retornará a última linha
```

31.2 - MOVE – Movendo Cursores

O comando MOVE faz a mudança na posição de um cursor sem retornar linhas.
Sua sintaxe é similar à do FETCH:

```
MOVE [ direção { FROM | IN } ] nome_cursor
```

Exemplo:

```
MOVE -7 FROM cursor2; --Volta 7 posições do cursor
```

31.3 - 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 comndo CLOSE:

Sintaxe:

```
CLOSE { nome | ALL }
```

Exemplos:

```
CLOSE cursor2; --Fecha o cursor2
CLOSE ALL; --Fecha todos cursores abertos
```

32 - BLOB – Binary Large OBject (Objeto Grande Binário)

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 se encontram na tabela de sistema “pg_largeobject”.

Exemplo:

```
/* Tabela do BLOB */
CREATE TABLE figuras(
       id_figura oid,
       nome_figura varchar(15)
);

/* Importando o objeto para dentro do banco de dados */
INSERT INTO figuras VALUES (lo_import('/tmp/figura1.jpg'),'Figura 1');

/* Recuperando o objeto no diretório /tmp */
SELECT lo_export(id_figura,'/tmp/copia_figura1.jpg') FROM figuras WHERE nome_figura = 'Figura 1';
```

32.1 - Removendo BLOBs

Mesmo apagando uma linha de um BLOB com o comando “DELETE” na tabela “figuras”, o objeto binário relativo não será removido da tabela “pg_largeobject”. Para tal deve-se usar a função “lo_unlink(oid)”, cujo parâmetro é o oid do BLOB.

Exemplo:

```
SELECT lo_unlink(16484);
```

ou também poderia ser:

```
SELECT lo_unlink(id_figura) FROM figuras WHERE nome_figura = 'Figura 1';
```

Agora não existe mais o objeto binário, então pode-se apagar a referência a ele na tabela “figuras”:

```
DELETE FROM figuras WHERE nome_figura = 'Figura 1';
```

33 - Funções Criadas Pelo Usuário

O PostgreSQL tem quatro tipos de função:

- Escritas em SQL;
- Internas;
- Escritas na linguagem C.
- Procedurais, escritas em PL/pgSQL ou PL/Tcl, por exemplo;

Todos os tipos de função aceitam tipos base ou compostos ou combinados, como parâmetros. E também podem retornar um conjunto de valores base ou composto.

Sintaxe:

```
        CREATE [ OR REPLACE ] FUNCTION
                nome ( [ [ modo ] [ nome ] tipo [, ...] ] )
    [ RETURNS tipo_retorno ]
  { LANGUAGE nome_linguagem
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST custo_execução
    | ROWS registros_retornados
    | SET parâmetro_configuração { TO valor | = valor | FROM CURRENT }
    | AS 'definição'
    | AS 'arquivo_objeto', 'link_simbólico'
  } ...
    [ WITH ( atributo [, ...] ) ]
```

Corpo de uma função simplificado:

```
CREATE [OR REPLACE] FUNCTION nome_funcao(parametros)
               RETURNS tipo AS delimitador
               comando1;
               …
               comandoN;
               delimitador LANGUAGE linguagem;
```

Obs.: Quando se cria uma função de uma forma geral temos: `CREATE FUNCTION nome …`
Se quisermos sobrescrevê-la usamos `CREATE OR REPLACE FUNCTION nome...`
No entanto só é válido sobrescrever uma função se o tipo de retorno for o mesmo da original.
Caso contrário terá que primeiro apagar a função.

33.1 - Funções SQL

Executam uma lista de declarações SQL e retornam o resultado da última consulta dessa lista.

33.1.1 - Funções com Retorno Nulo

A última declaração de uma função deve ser um `SELECT`, o qual servirá como retorno, a não ser que seu retorno seja declarado do tipo void.

Exemplo:

```
        CREATE FUNCTION func_teste()
               RETURNS void AS $$
               DELETE FROM colaboradores WHERE cargo IS NULL;
        $$ LANGUAGE SQL;

        SELECT func_teste(); -- Usando a função;
```

33.1.2 - Funções com Retorno Não Nulo

Funções podem retornar uma linha ou um conjunto de linhas. Nos casos em que o retorno não é um conjunto, a primeira linha do resultado da última consulta passa a ser o retorno da função mesmo que tenham mais linhas.

Se a última consulta não retornar nenhuma linha o retorno é nulo.

Exemplo:

```
CREATE OR REPLACE FUNCTION func_teste2()
       RETURNS varchar(15) AS $$
       SELECT nome FROM colaboradores ORDER BY nome;
       $$ LANGUAGE SQL;
```

33.1.3 - Apagando uma Função

Como se faz com a maior parte dos tipos de objetos...

Exemplo:

```
DROP FUNCTION func_teste();
```

33.1.4 - Funções que Retornam Conjuntos

Para que uma função retorne um conjunto deve-se especificar o retorno como `SETOF algum_tipo`, sendo que “algum_tipo” pode ser uma tabela.

E então todas as linhas do resultado são retornadas.

Exemplo:

```
CREATE OR REPLACE FUNCTION func_teste()
       RETURNS SETOF colaboradores AS $$
       SELECT * FROM colaboradores WHERE uf='RJ';
       $$ LANGUAGE SQL;
```

Podemos inclusive usar funções que retornam várias linhas de uma tabela como se a mesma fosse uma tabela.

Exemplos:

```
SELECT nome FROM func_teste();
SELECT * FROM func_teste();
```

33.1.5 - Funções com Parâmetros

Uma função pode ser criada com ou sem parâmetros. Os parâmetros **não são nomeados**, devem ser colocados apenas os tipos entre os parênteses e no caso de ser mais de um devem ser separados por vírgulas.

Nos comandos, cada parâmetro é chamado pela referência “$n”, onde “n” é a posição que corresponde ao parâmetro.

Exemplos:

```
-- Função de apenas um parâmetro;
CREATE OR REPLACE FUNCTION func_teste(char(5))
       RETURNS text AS $$
       SELECT nome||' : '||salario FROM colaboradores WHERE mat_id=$1;
       $$ LANGUAGE SQL;
```

```
SELECT func_teste('00001'); -- O parâmetro de ver inserido de acordo com o tipo;

CREATE OR REPLACE FUNCTION func_teste(real,char(2))
       RETURNS SETOF text AS $$
       SELECT nome||' '||snome FROM colaboradores WHERE salario<=$1 AND uf=$2;
       $$ LANGUAGE SQL;

SELECT func_teste(3000,'SP'); -- Dois parâmetros inseridos na função

CREATE OR REPLACE FUNCTION daqui_uma_semana(date)
       RETURNS date AS '
       SELECT $1+7;
       ' LANGUAGE SQL;

SELECT daqui_uma_semana('01/07/2009');
```

33.1.6 - Sobrecarga de Função

Um recurso muito usado em linguagens de programação Orientadas a Objeto (Java, por exemplo), possibilita criar mais de uma função com o mesmo nome, desde que os parâmetros sejam diferentes. Ou seja, nomes de funções podem ser sobrecarregados.

Quando se executa um comando, o servidor determina qual função exatamente deve ser chamada tomando como base os tipos de dados e os parâmetros.

Pode-se utilizar sobrecarga de funções para simular funções com número variável de parâmetros (numero máximo finito).

Exemplos:

```
-- Função com parâmetros do tipo real;
CREATE OR REPLACE FUNCTION soma_parametros(real,real)
       RETURNS real AS $$
       SELECT $1+$2;
       $$ LANGUAGE SQL;

-- Função com parâmetros do tipo varchar;
CREATE OR REPLACE FUNCTION soma_parametros(varchar,varchar)
       RETURNS varchar AS $$
       SELECT $1||' '||$2;
       $$ LANGUAGE SQL;

SELECT soma_parametros(333,444); -- Parâmetros do tipo real;
SELECT soma_parametros('PostgreSQL','rulz!!!'); -- Parâmetros do tipo varchar;
```

Para apagar funções sobrecarregadas é preciso especificar os tipos de parâmetros. Portando se desejarmos apagar as duas variações de “soma_parametros()” devemos fazer da seguinte maneira:

```
DROP FUNCTION soma_parametros (real,real);
DROP FUNCTION soma_parametros (varchar,varchar);
```

33.2 - Funções internas

São funções escritas na Linguagem C que foram estaticamente ligadas ao servidor PostgreSQL.
Todas as funções internas que estão no servidor, normalmente, são declaradas na inicialização do cluster, porém o usuário pode criar aliases para uma função interna através do comando CREATE FUNCTION.

Exemplo:

```
CREATE FUNCTION raizquad(double precision) RETURNS double precision
       AS 'dsqrt'
       LANGUAGE internal STRICT;

SELECT raizquad(49);
```

Obs.: Nem todas as funções “pré-definidas” são “internas” no sentido acima. Algumas funções pré-definidas são escritas em SQL.

33.3 - Funções na linguagem C

O usuário pode definir funções em C, as quais devem ser compiladas como bibliotecas e são carregadas conforme são requisitadas pelo servidor.
Funções compiladas ganham maior desempenho do que funções escritas em linguagens compiladas.

33.4 - Linguagens procedurais

No PostgreSQL é possível definir funções pelo usuário além de SQL e C. Essas linguagens são genericamente chamadas de linguagens procedurais (PL: Precedural Language).
São exemplos de PLs: PL/pgSQL, PL/Tcl, PL/Perl, PL/Python e outras linguagens definidas pelo usuário.
Há outras PLs adicionais, porém não são incluídas na distribuição núcleo.

33.4.1 - Instalação de linguagem procedural

A linguagem procedural deve ser instalada em cada banco onde será utilizada.
Se instalar no banco de dados template1, automaticamente estará disponível para todos os bancos dados que forem criados após a instalação da linguagem, sendo que suas entradas em template1 são copiadas. Então o administrador pode definir quais linguagens ficarão disponíveis de acordo com o banco de dados, inclusive, se quiser tornar algumas padrão.

Para instalar a linguagem PL/pgSQL no banco de dados atual use a sintaxe:

```
CREATE LANGUAGE plpgsql;
```

Ou se preferir, use o aplicativo createlang no shell:

```
createlang plpgsql banco_de_dados
```

33.4.2 - PL/pgSQL

PL/pgSQL é uma linguagem procedural carregável feita para o PostgreSQL, a qual é similar à PL/SQL do Oracle.
Enquanto declarações SQL são executadas individualmente pelo servidor, a linguagem PL/pgSQL agrupa um bloco de processamento e comandos em série dentro do servidor, somando o poder da linguagem procedural com a facilidade do SQL.

Economiza-se tempo, devido ao fato de não ser necessário sobrecarregar a comunicação entre cliente e servidor, aumentando o desempenho.

Em PL/pgSQL todos os tipos de dados, operadores e funções SQL podem ser utilizados.

A utilização de um editor de texto preferido pelo usuário é aconselhável para se desenvolver em PL/pgSQL.

Em uma janela se escreve o código (pelo editor) e em outra utiliza-se o psql para carregar as funções escritas. Utilizando "CREATE OR REPLACE FUNCTION" é recomendável para fins de atualização do que já foi criado.

33.4.2.1 - Estrutura PL/pgSQL

A linguagem PL/pgSQL é estruturada em blocos. O texto completo da definição da função deve ser um bloco. Um bloco é
definido como:

```
[ <<rótulo>> ]
[ DECLARE
        declarações ]
BEGIN
        instruções
END;
```

Declarações e instruções dentro do bloco devem ser terminadas com ponto-e-vírgula. Um bloco dentro de
outro bloco deve ter um ponto-e-vírgula após o END, conforme exibido acima, entretanto, o END final que conclui o corpo da função não requer o ponto-e-vírgula.

Palavras chaves e identificadores podem ser escritos mesclando letras maiúsculas e minúsculas. As letras dos identificadores são convertidas implicitamente em minúsculas, a menos que estejam entre aspas.

Há dois tipos de comentários no PL/pgSQL. O hífen duplo (--) comentário de linha. O /* inicia um bloco de comentário vai até a próxima ocorrência de */. Os blocos de comentário não podem ser aninhados, mas comentários de linha podem estar dentro de blocos de comentário, e os hífens duplos escondem os delimitadores de bloco de comentário /* e */.

Variáveis declaradas na seção de declaração que antecede um bloco são inicializadas com seu valor padrão toda vez que o bloco é executado, e não somente uma vez a cada chamada da função. Por exemplo:

```
CREATE FUNCTION teste_escopo() RETURNS integer AS $$
<<bloco_externo>>
DECLARE
        quantidade integer := 30;
BEGIN
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30
        quantidade := 50;

-- Sub-bloco---------------------------------------------------------------------------
        <<bloco_interno>>
        DECLARE
                quantidade integer := 80;
        BEGIN
                RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 80
        END;

-- Fim do sub-bloco--------------------------------------------------------------------

RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50
RETURN quantidade;
END;
$$ LANGUAGE plpgsql;
```

Obs.: Não confundir o uso BEGIN/END de agrupamento de instruções PL/pgSQL com controle de transação. BEGIN/END de PL/pgSQL é só para agrupamento, não iniciam nem terminam transações.

33.4.2.2 - Declarações

Variáveis utilizadas em um bloco devem ser declaradas em DECLARE (exceto a variável de laço FOR que interage sobre um intervalo de valores inteiros, que é automaticamente declarada como do tipo inteiro).

As variáveis PL/pgSQL podem ser de qualquer tipo SQL.

A sintaxe geral para declaração de variáveis é:

```
nome [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expressão ];
```

DEFAULT	Se for inserida, especifica o valor padrão à variável quando o bloco for processado. Caso contrário, a variável terá seu valor nulo.
CONSTANT	Não permite que seja atribuído outro valor à variável, mantendo o valor constante enquanto o bloco durar.
NOT NULL	Faz com que seja obrigatória uma atribuição de valor não nulo à variável

Exemplos:

```
num_peca integer CONSTANT integer := 97;
qtd numeric(7);
url varchar:= 'http://meu-site.com';
minha_linha nome_da_tabela%ROWTYPE;
meu_campo nome_da_tabela.nome_da_coluna%TYPE;
uma_linha RECORD;
quantidade integer DEFAULT 32;
```

33.4.2.3 - Aliases de Parâmetros de Função

Parâmetros de uma função tem como identificadores a simbologia $n, sendo que n é o número relativo à posição do parâmetro. No entanto, podemos melhorar a legibilidade do código atribuindo aliases, nomeando os parâmetros.

Exemplo:

```
CREATE OR REPLACE FUNCTION calcula_imposto(preco real,porcent_imp real) RETURNS real AS $$
        BEGIN
                RETURN preco*(porcent_imp/100);
        END;
        $$ LANGUAGE plpgsql;
```

A outra forma, que era a única disponível antes da versão 8.0 do PostgreSQL, é declarar explicitamente um alias utilizando a sintaxe de declaração:

```
nome ALIAS FOR $n;
```

O exemplo acima escrito utilizando este estilo fica da seguinte maneira:

```
CREATE OR REPLACE FUNCTION calcula_imposto(real,real) RETURNS real AS $$
        DECLARE
                preco ALIAS FOR $1;
                porcent_imp ALIAS FOR $2;

        BEGIN
                RETURN preco*(porcent_imp/100);
        END;
        $$ LANGUAGE plpgsql;
```

33.4.2.4 - SELECT INTO

Quando uma consulta retorna várias colunas (de apenas uma linha), o resultado pode ser atribuído a uma variável registro (`RECORD`), ou do tipo linha (`%ROWTYPE`) ou uma lista de variáveis escalares (lista de expressões). Da seguinte forma:

```
SELECT INTO destino expressoes_de_consulta FROM ...;
```

Sendo que "destino" pode ser uma variável do tipo `RECORD` ou `ROWTYPE` ou uma lista separada por vírgulas de variáveis simples
e campos de registro/linha, "expressoes_de-consulta" e o restante do docmando são como no puro SQL.

A interpretação de `SELECT INTO` em PL/pgSQL é diferente da interpretação padrão do PostgreSQL. Pois na forma padrão uma nova tabela é criada com o nome referente a "destino", conforme mostrado acima. Se for necessário criar uma tabela dentro de uma função PL/pgSQL a partir do resultado de uma consulta de ser usada a sintaxe `CREATE TABLE nova_tabela AS consulta.`

33.4.2.5 - Atributos

A linguagem PL/pgSQL disponibiliza atributos para auxiliar a lidar tipos de dados. São eles: `TYPE%` e `%ROWTYPE`.

Atributos são usados para declarar variáveis que coincidam com o tipo de dado de uma coluna (`%TYPE`) ou para corresponder à estrutura de uma linha (`%ROWTYPE`). Não é preciso saber o tipo de dado quando se usa atributos para declarar variáveis. Se um objeto for alterado futuramente, o tipo daa variável mudará automaticamente para o tipo de dado sem alterar o código.

33.4.2.5.1 - Tipo de Variável Copiado – variável %TYPE

A expressão `%TYPE` fornece o tipo da variável ou da coluna da tabela. Pode ser utilizada para declarar variáveis que guardam valores do banco de dados. Por exemplo, supondo que exista uma coluna chamada `id_produto` na tabela produtos, para declarar uma variável com o mesmo tipo de dado de produtos.id_produto:

```
id_produto produtos.id_produto%TYPE;
```

Com %TYPE não é preciso saber o tipo de dado da estrutura referenciada e, ainda mais importante, se o tipo de dado do item referenciado mudar no futuro (por exemplo: o tipo de dado de id_produto for mudado de integer para real), não precisará mudar a definição na função.

33.4.2.5.2 - Variável Tipo Linha - %ROWTYPE

Pode armazenar uma linha inteira do resultado de uma consulta se o conjunto de colunas do comando for do mesmo tipo declarado para a variável.

Cada coluna pode ser referenciado individualmente usando a notação `var.coluna`.

Ao declarar uma variável-linha pode ter o mesmo tipo das linhas de uma tabela com a notação tabela`%ROWTYPE` ou sua declaração especificando um tipo composto (tabelas possuem um tipo composto associado, sendo o próprio nome da tabela).

Por questões de portabilidade convém-se escrever `%ROWTYPE`, mas para o PostgreSQL não faz diferença.

Sintaxe:

```
variavel tabela%ROWTYPE;
variavel tipo_composto;
```

Exemplo:

```
CREATE OR REPLACE FUNCTION matricula2nome(matricula char(5)) RETURNS text AS $$
DECLARE
	/*
	A variável “nome_colaborador” aqui já recebe a estrutura da
	tabela “colaboradores”
	*/
	nome_colaborador colaboradores%ROWTYPE;

BEGIN
	-- A consulta para dar o retorno da função
	SELECT INTO nome_colaborador * FROM colaboradores WHERE mat_id = matricula;
	-- Retorna o primeiro
	RETURN nome_colaborador.nome||' '||nome_colaborador.snome;
END;
$$ LANGUAGE 'plpgsql';

SELECT matricula2nome('00007') AS "Nome Inteiro";
```

33.4.2.6 - Tipo Registro – RECORD

Variáveis do tipo RECORD são similares a variáveis do tipo `ROWTYPE`, porém `RECORD` não tem uma estrutura pré definida.

Elas assumem a estrutura da linha atual que foi atribuída por uma consulta. Essa atribuição, que é o assinalamento se dá pelo comando `SELECT INTO` ou `FOR`.

A sub-estrutura de uma variável `RECORD` pode mudar cada vez que for usada em uma atribuição.

Se uma variável do tipo `RECORD` for acessada antes de ser assinalada, haverá um erro.

`RECORD` na verdade não é um tipo de dados real e sim um espaço reservado.

Sintaxe:

```
variavel RECORD;
```

Exemplo:

```
CREATE OR REPLACE FUNCTION matricula2nome(matricula char(5)) RETURNS text AS $$
DECLARE
	nome_colaborador RECORD;
BEGIN
	SELECT INTO nome_colaborador * FROM colaboradores WHERE mat_id = matricula;
	-- Retorna o primeiro
	RETURN nome_colaborador.nome||' '||nome_colaborador.snome;
END;
$$ LANGUAGE 'plpgsql';

SELECT matricula2nome('00001');
```

33.4.2.7 - RENAME

Renomeia variáveis, registros ou linhas. Tem como mair utilidade quando NEW ou OLD devem ser referenciados por outro nome dentro da função de um trigger.

OBS.: Desde a versção 7.3 do PostgreSQL, RENAME parece estar com problemas. A correção tem baixa prioridade, devido ao fato de ALIAS atender a maior parte dos usos práticos de RENAME.

Sintaxe:

```
RENAME antigo_nome TO novo_nome;
```

Exemplo:

```
RENAME nome TO nome_funcionario;
```

33.4.2.8 - Atribuições

Para atribuir valor a uma variável, ou a um campo de linha ou de registro, se faz da seguinte maneira:

```
identificador := valor;
```

O valor pode ser também uma expressão que seja equivalente a um valor.

Exemplos:

```
quantidade := $1;
i INT:=0; -- Quando a variável é criada, também pode atribuir valor a ela;
preco := 5.27;
total := quantidaade * preco;
```

33.4.2.9 - PERFORM - Execução sem Resultado

Em certos casos, deseja-se avaliar um comando e desprezar seu resultado (especialmente quando uma função produz efeitos indesejados e/ou não tem um valor útil de resultado.
Na linguagem PL/pgSQL é usado a instrução `PERFORM`.

Sintaxe:

```
PERFORM comando;
```

A instrução deve ser excrita da mesma forma que uma consulta SQL, porém substituindo a palavra chave `SELECT` por `PERFORM`.
A variável especial `FOUND` é "true" se caso o comando resultar pelo menos uma linha ou "false" se não produzir nenhuma.

Exemplo:

Uma função que apenas diz se uma matrícula corresponde a um colaborador sem dizer qual:

```
CREATE OR REPLACE FUNCTION colab_teste(char(5)) RETURNS text AS $$
        BEGIN

                --Uso do PERFORM para desprezar o resultado
                PERFORM * FROM colaboradores WHERE mat_id=$1;

                --Se PERFORM "Achar" (FOUND) pelo menos uma linha é verdadeiro (true)
                IF FOUND THEN
                        RETURN 'Existe o colaborador!';
                ELSE
                        --Uma Nota com mensagem personalizada
                        RAISE NOTICE 'Colaborador de matricula % nao foi encontrado',$1;
                        RETURN 'Erro!!!';
                END IF;
END; $$ LANGUAGE plpgsql;
```

33.4.2.10 - NULL - Não Fazer Nada

Às vezes conforme a situação, ao invés de executar comandos, pode-se simplesmente ignorar e executar nada. Para isso utiliza-se a instrução `NULL`.

Exemplo:

As duas funções abaixo são equivalentes:

```
CREATE OR REPLACE FUNCTION teste_div(int,int) RETURNS int AS $$
        BEGIN
                RETURN $1/$2;

                EXCEPTION
                        WHEN division_by_zero THEN
                                NULL; -- ignora o erro com a instrução NULL
                                RETURN NULL;
        END;
$$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION teste_div(int,int) RETURNS int AS $$
        BEGIN
                RETURN $1/$2;

                EXCEPTION
                        WHEN division_by_zero THEN -- ignora o erro omitindo a instrução NULL
                                RETURN NULL;
        END;
$$ LANGUAGE plpgsql;
```

Qualquer uma das formas no PostgreSQL é válida.

Obs.: Na linguagem PL/SQL do Oracle instruções vazias não são permitidas, a instrução `NULL` é obrigatória em situações como essas.

33.4.2.11 - EXECUTE - Execução Dinâmica de Comandos

Em certos casos é necessário executar dinamicamente comandos dentro de PL/pgSQL que envolvam diferentes tabelas e / ou tipos de dados.

A instrução EXECUTE é fornecida para este fim executando uma string que contém um comando SQL.

Substituições de variáveis em PL/pgSQL não são feitas na string do comando. Seus valores devem ser inseridos quando a string é elaborada.

Resultados de SELECT são desprezados pelo EXECUTE e até o momento SELECT INTO não é suportado. Portanto, não há como extrair o resultado de um SELECT criado dinamicamente com EXECUTE puro. Porém, há dois jeitos de se fazer isso: com o laço FOR-IN-EXECUTE ou um cursor com OPEN-FOR-EXECUTE, que ambos serão vistos mais adiante.

Exemplos:

```
CREATE OR REPLACE FUNCTION testaExecute()
RETURNS text AS $$
BEGIN
	//*-----------------------------------------------------------------
	O comando EXECUTE e a string.
	Devido ao fato da própria string estar entre aspas simples, os valores string
	contidos nela devem estar entre dois pares de aspas simples (''string''),
	como se vê abaixo:
	------------------------------------------------------------------- -*/

	EXECUTE 'UPDATE colaboradores SET setor=''Higiene'' WHERE setor=''Limpeza''';
	RETURN 'Ok!';
END;
$$ LANGUAGE plpgsql;

SELECT testaExecute();


CREATE OR REPLACE FUNCTION testaExecute(tabela text,coluna text,valor_novo text,valor_antigo
text)
RETURNS text AS $marcador$
BEGIN
	EXECUTE 'UPDATE '||tabela||' SET '||coluna||'='||''''||valor_novo||''''||' WHERE '||
coluna||'='||''''||valor_antigo||'''';

	RETURN 'OK!';
END;
$marcador$ LANGUAGE plpgsql;

SELECT testaExecute('colaboradores','setor','Limpeza','Higiene');
```

33.4.2.12 - Status de um Resultado

Há várias maneiras para determinar o efeito de um comando através de dois métodos:

33.4.2.12.1 - 1º Método - GET DIAGNOSTICS

```
GET DIAGNOSTICS variável = item;
```

Permite obter os indicadores de status do sistema. "item" é uma palavra chave que identifica o valor de estado para a variável especificada.

Os itens de status podem ser:

-ROW_COUNT: Quantidade de linhas processadas pelo último comando SQL;
-RESULT_OID: O OID da última linha inserida pelo comando SQL mais recente, ou seja; só tem utilidade após um comando INSERT.

Exemplo:

```
CREATE OR REPLACE FUNCTION result_status(char(2))
	RETURNS integer AS  $$
		DECLARE
			linhas integer;
		BEGIN
			PERFORM * FROM colaboradores WHERE uf=$1;

			/* Atribui à variável "linhas" o nº de registros
			referidos pelo comando anterior;  */

			GET DIAGNOSTICS linhas = ROW_COUNT;
			RETURN linhas;
		END;
	$$ LANGUAGE plpgsql;
```

33.4.2.12.2 - 2º Método - FOUND

FOUND é uma variável booleana que é inicialmente "false" dentro de cada chamada de função PL/pgSQL.

Sua definição é dada por cada um dos seguintes tipos de instrução: SELECT INTO, PERFORM, UPDATE, INSERT, DELETE, FETCH e FOR. Cada instrução pode definir "True" se retornar / interagir com pelo menos uma linha ou "False" se não retornar / interagir com nenhuma linha.

FOUND é uma variável local dentro de cada função PL/pgSQL; qualquer alteração feita na mesma afeta somente a função corrente.

33.4.2.13 - Retorno de Uma Função

Dois comandos retornam dados de uma função: RETURN e RETURN NEXT.

33.4.2.13.1 - RETURN

O comando RETURN com uma expressão (que pode ser um valor, uma variável ou até mesmo void), finaliza a função e retorna o valor da expressão pra quem executa.

Para que o retorno seja um valor composto (linha), deve ser declarada uma variável registro ou linha como a expressão.

Sintaxe:

```
RETURN expressão;
```

O valor retornado pela função não pode ser deixado indefinido. Se o controle atingir o final do bloco de nível mais alto da função sem atingir uma instrução RETURN, ocorrerá um erro em tempo de execução. Se a função for declarada como retornando void, ainda assim deve ser especificada uma instrução RETURN; mas neste caso a expressão após o comando RETURN é opcional, sendo ignorada caso esteja presente.

33.4.2.13.2 - RETURN NEXT

Quando o retorno de uma função PL/pgSQL é declarado como sendo "SETOF tipo", há um procedimento diferente a ser seguido. Os itens individuais a serem retornados são especificados em comandos RETURN NEXT, e um comando RETURN final, sem argumento indica que a função chegou ao fim.

Sintaxe:

```
	RETURN NEXT expressão;
```

Exemplo:

```
CREATE OR REPLACE FUNCTION funcao_next()
       RETURNS SETOF varchar(10) AS $funcao_next$

       BEGIN
              RETURN NEXT 'Saturno';
              RETURN NEXT 'Netuno';
              RETURN NEXT 'Urano';
              RETURN;
       END;
       $funcao_next$ LANGUAGE plpgsql;
```

Funções que utilizam `RETURN NEXT` devem ser chamadas como uma fonte de tabela na cláusula `FROM`:

```
SELECT * from funcao_next();
```

O comando `RETURN NEXT` salva o valor da expressão e em seguida a execução continua na próxima instrução da função PL/pgSQL.

O conjunto de resultados (result set) é construído se executando sucessivos `RETURN NEXT` e o `RETURN` final, que não deve ter argumentos, faz o controle sair da função.

33.4.2.14 - Condicionais

Instruções IF (se) executam ou não comandos dependendo de condições. PL/pgSQL tem cinco formas de IF:

```
· IF ... THEN
· IF ... THEN ... ELSE
· IF ... THEN ... ELSE IF
· IF ... THEN ... ELSIF ... THEN ... ELSE
· IF ... THEN ... ELSEIF ... THEN ... ELSE
```

33.4.2.14.1 - IF-THEN

Os comandos entre `THEN` e `END IF` são executados se a condição for verdadeira, caso contrário são ignorados.

```
IF expressão_booleana THEN
       comando 1;
       comando 2;
       comando ...;
       comando N;
END IF;
```

33.4.2.14.2 - IF-THEN-ELSE

Instruções `IF-THEN-ELSE` ampliam o `IF-THEN` permitindo especificar alternativamente um conjunto de comandos a serem executadas se a condição for avaliada como falsa.

```
IF expressão_booleana THEN
     instruções
ELSE
     instruções
END IF;
```

33.4.2.14.3 - IF-THEN-ELSE IF

Aninhamentos de instruções podem ser feitos como segue o exemplo:

```
IF uf = 'SP' THEN
     pessoa_uf := 'paulista';
ELSE
     IF uf = 'RJ' THEN
          pessoa_uf := 'fluminense';
     END IF;
END IF;
```

Quando essa forma é utilizada, na verdade, uma instrução IF está sendo aninhada dentro da parte ELSE da instrução IF principal. Então, é preciso uma instrução END para cada IF aninhado e também para o "IF-ELSE" pai. Apesar de funcionar, é feito de forma esteticamente desagradável quando existem muitas alternativas para serem avaliadas. A próxima forma seria mais conveniente:

33.4.2.14.4 - IF-THEN-ELSIF-ELSE

```
IF expressão_booleana THEN
      comandos
[ ELSIF expressão_booleana THEN
      comandos
[ ELSIF expressão_booleana THEN
     comandos
      ...]]
[ ELSE
      instruções ]
END IF;
```

Usando o modo `IF-THEN-ELSIF-ELSE` é mais adequado para verificar muitas alternativas em uma instrução.

É equivalente aos comandos `IF-THEN-ELSE-IF-THEN` aninhados, porém só precisa de um `END IF`.

Exemplo:

```
IF numero = 0 THEN
       resultado := 'zero';
       ELSIF numero > 0 THEN
              resultado := 'positivo';
       ELSIF numero < 0 THEN
              resultado := 'negativo';
       ELSE -- Aqui a única outra possibilidade é que o número seja nulo
              resultado := 'NULL';
END IF;
```

33.4.2.14.5 - IF-THEN-ELSEIF-ELSE

`ELSEIF` é um alias para `ELSIF`, ou seja, o resultado será o mesmo usando qualquer um dos dois.

33.4.2.15 - Laços

Com os comandos `LOOP`, `EXIT`, `WHILE`, e `FOR` é possível fazer uma função repetir uma série de comandos.

33.4.2.15.1 - LOOP

Define um laço incondicional, repetindo indefinidamente até que seja finalizado por um EXIT ou "RETURN".

No caso de laços aninhados pode ser utilizado um rótulo opcional em EXIT para definir o nível de aninhamento que deve ser
finalizado.

```
[<<rótulo>>]
LOOP
        instruções
END LOOP;
```

33.4.2.15.2 - EXIT

Se nenhum rótulo for determinado, o laço mais interno é finalizado e o comando após END LOOP é executado em seguida.

Se o rótulo for determinado, esse deve ser o rótulo do nível atual ou de algum nível externo do laço ou do bloco aninhado.
E então o laço ou bloco é finalizado, e o controle continua na instrução após o END do laço ou do bloco.

Quando se tem o parâmetro WHEN, a saída do laço se faz apenas se a condição especificada for verdadeira, caso contrário, o controle passa para a instrução após o EXIT.

O EXIT pode interromper prematuramente qualquer tipo de laço, não se limitando a apenas laços incondicionais.

Sintaxe:

```
EXIT [ rótulo ] [ WHEN expressão ];
```

Exemplos:

```
CREATE OR REPLACE FUNCTION teste_loop() RETURNS int AS $$

DECLARE
        i int;

BEGIN

        i := 0;

        LOOP
                IF i = 10 THEN
                        RAISE NOTICE 'O indexador vale 10 agora!!!';
                        EXIT; -- Faz sair do laço
                END IF;

                RAISE NOTICE 'Ainda não... :(';
                i := i+1;
        END LOOP;

        RETURN i;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION teste_loop()
RETURNS int AS $$

DECLARE
	i int;

BEGIN

	i := 0;

	LOOP
		IF i < 10 THEN
			RAISE NOTICE 'Ainda não... :(';
		END IF;
		i := i+1;
		EXIT WHEN i=10; -- Faz sair do laço
	END LOOP;

	RAISE NOTICE 'O indexador vale 10 agora!!!';

	RETURN i;
END;
$$ LANGUAGE plpgsql;
```

33.4.2.15.3 - WHILE

Repete uma seqüência de comandos enquanto a expressão de condição for verdadeira. A condição é avaliada antes de cada entrada no corpo do laço.

Sintaxe:

```
[<<rótulo>>]
WHILE expressão LOOP
	comandos
END LOOP;
```

Exemplos:

```
CREATE OR REPLACE FUNCTION teste_loop()
RETURNS int AS $$

DECLARE
	i int;

BEGIN
	i := 0;

	WHILE i < 10 LOOP
		RAISE NOTICE 'Ainda não... :(';
		i := i+1;
	END LOOP;

	RAISE NOTICE 'O indexador vale 10 agora!!!';

	RETURN i;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION teste_loop()
        RETURNS int AS $$

        DECLARE
                i int;

        BEGIN

                i := 0;

                WHILE NOT i=10 LOOP
                        RAISE NOTICE 'Ainda não... :(';
                        i := i+1;
                END LOOP;

                RAISE NOTICE 'O indexador vale 10 agora!!!';

                RETURN i;
END;
$$ LANGUAGE plpgsql;
```

33.4.2.15.4 - FOR (variação inteira)

Cria um loop que interage em um intervalo de valores inteiros. A variável "nome" é definida automaticamente como sendo do tipo integer, e **somente dentro do loop**.

As duas expressões que são os limites inferior e superior são avaliadas somente uma vez, ao entrar no laço. Por padrão a razão da interação é +1, quando o parâmetro "REVERSE" é especificado passa a ser -1.

Sintaxe:

```
[<<rótulo>>]
FOR nome IN [ REVERSE ] expressão .. expressão LOOP
        comandos
END LOOP;
```

Exemplos:

```
CREATE OR REPLACE FUNCTION teste_loop() RETURNS int AS $$
        DECLARE
                i int:=0;

        BEGIN

                FOR i IN 0..9 LOOP
                        RAISE NOTICE 'Ainda não... :(';
                        i := i+1;
                END LOOP;

                RAISE NOTICE 'O indexador vale 10 agora!!!';

                RETURN i;
        END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION teste_loop() RETURNS int AS $$
        DECLARE
                i int:=0;

        BEGIN

                FOR i IN REVERSE 20..11 LOOP
                        RAISE NOTICE 'Ainda não... :(';
                        i := i-1;
                END LOOP;

                RAISE NOTICE 'O indexador vale 10 agora!!!';

                RETURN i;
        END;
$$ LANGUAGE plpgsql;
```

Obs.: Se o limite inferior for maior que o limite superior (ou menor, no caso de `REVERSE`), o loop não será executado sem gerar erro.

33.4.2.15.5 - Loop por Queries

É possível interagir através do resultado de uma query e manipular dados utilizando um laço `FOR` diferente.

Cada linha resultante de comando (`SELECT`) é atribuída, sucessivavemnte, à variável registro ou linha, e o corpo do loop é executado uma vez para cada linha.

Se o loop for finalizado por uma `EXIT`, o último valor de linha atribuído ainda é acessível após o laço.

Sintaxe:

```
[<<rótulo>>]
FOR registro_ou_linha IN comando LOOP
        comandos
END LOOP;

Exemplo:

CREATE OR REPLACE FUNCTION loop_for() RETURNS int AS $$
DECLARE

var1 colaboradores%ROWTYPE;
i int :=0;

BEGIN
        FOR var1 IN SELECT * FROM colaboradores WHERE uf='SP' LOOP
                i :=i+1;
        END LOOP;

        -- Retorna o número de linhas resultantes da query:
        RETURN i;

END;
$$ LANGUAGE plpgsql;
```

33.4.2.16 - FOR-IN-EXECUTE

É uma outra forma de interagir sobre linhas, semelhante à anterior, porém o código fonte do SELECT é definido como uma expressão cadeia de caracteres, que é avaliada e replanejada a cada iteração do loop. Isso permite ao programador escolher entre a velocidade de consulta pré-planejada e a flexibilidade da consulta dinâmica, da mesma forma que no comando EXECUTE puro.

Sintaxe:

```
[<<rótulo>>]
FOR registro_ou_linha IN EXECUTE string_de_comando LOOP
        comandos
END LOOP;

Exemplo:

CREATE OR REPLACE FUNCTION for_in_exec(var_uf char(2)) RETURNS int AS $$
DECLARE
        var1 colaboradores%ROWTYPE;
i int :=0;

BEGIN
        FOR var1 IN EXECUTE 'SELECT * FROM colaboradores WHERE uf='||''''||var_uf||'''' LOOP
                i :=i+1;
        END LOOP;

        -- Retorna o número de linhas resultantes da query:
        RETURN i;
END;
$$ LANGUAGE plpgsql;
```

Obs.: Atualmente o analisador PL/pgSQL distingue os dois tipos de loops FOR (inteiro e resultado de uma query), verificando se encontra ".." fora de parênteses entre IN e LOOP. Caso não encontrar, o loop é entendido como sendo de linhas. Se ".." for escrito de maneira equivocada, pode causar um erro notificando que “a variável do laço, para laço sobre linhas, deve ser uma variável registro ou linha”, ao invés de um simples erro de sintaxe.

33.4.2.17 - Tratamento de Erros

Por padrão, qualquer erro em uma função causa sua interrupção. Pode-se capturar e se recuperar de erros utilizando a cláusula `EXCEPTION`.

Se não ocorrer erro algum todos comandos antes de `EXCEPTION` serão executados.

Sintaxe:

```
[ <<rótulo>> ]
[ DECLARE
     declarações ]
BEGIN
     comandos
EXCEPTION
     WHEN condição [ OR condição ... ] THEN
          comandos_do_tratador
     [ WHEN condição [ OR condição ... ] THEN
       comandos_do_tratador
 ... ]
END;
```

Exemplo:

```
CREATE OR REPLACE FUNCTION trata_except(num1 int,num2 int) RETURNS int AS $$
        DECLARE
                resultado int;
        BEGIN
                resultado:=num1/num2;
                RETURN resultado;

        EXCEPTION
            WHEN division_by_zero THEN
                 RAISE EXCEPTION 'Não dividirás por Zero!!!';
     END;
$$ LANGUAGE plpgsql;
```

33.4.2.18 - Mensagens e Erros

O comando `RAISE` gera mensagens que informam ou causam erros.

Sintaxe:

```
RAISE nível 'formato' [, variável [, ...]];
```

Sendo que "nível" pode ser `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, e `EXCEPTION`. O nível `EXCEPTION` provoca um erro (que normalmente interrompe a transação atual); os outros níveis geram apenas mensagens com diferentes prioridades.

Na string de formatação, o caractere "%" representa o próximo argumento (opcional). Para se obter uma mensagem
com o caractere "%" literalmente deve-se fazer: %%.

Exemplos:

```
RAISE NOTICE 'Total da semana: %', total; -- Exibe a frase com o conteúdo da variável
"total"
RAISE NOTICE '% %% de lucro!',50; -- Exibe o valor e também o caractere "%"
```

33.4.2.19 - Cursores em Funções PL/pgSQL

Cursores facilitam muito em consultas cujo retorno so muitas linhas, pois ao invés de fazer a consulta de uma vez, pode-se ler algumas linhas por vez.

Declaração de Cursores

Cursores são um tipo de variável, portanto devem ser estar dentro de `DECLARE`.

Exemplo:

```
...
DECLARE
	cursor_vazio refcursor;
	cursor_determinado FOR  SELECT * colaboradores;
	cursor_com_arg (argumento int) IS SELECT * FROM colaboradores WHERE cargo IS NOT
NULL;
...
```

Foram criados dois cursores, ambos são do mesmo tipo (refcursor),no entanto, "cursor_vazio" não foi determinado em sua declaração e pode ser usado para qualquer query e "cursor_determinado" já tem uma
query determinada para executar.

33.4.2.19.1 - Abertura de Cursores

Para se fazer uso de um cursor em uma função é preciso abrí-lo.

Cursores Vazios:

```
OPEN cursor_vazio FOR SELECT * FROM colaboradores;
OPEN FOR EXECUTE;
```

Cursores Vazios Para Executar Strings:

```
OPEN cursor_vazio FOR EXECUTE 'SELECT * FROM colaborades WHERE uf='||$1||' ORDER BY nome';
```

Cursores Determinados:

```
OPEN cursor_determinado;
```

Cursores Com Argumentos:

```
OPEN cursor_com_arg(77);
```

33.4.2.19.2 - Utilização e Fechamento de Cursores

Como já foi visto, o comando "FETCH" serve para recuperar informações de cursores.
Com o comando "FETCH" pode recuperar a(s) linha determinada(s) jogando as informações em variáveis que podem ser do tipo ROWTYPE, RECORD ou variáveis de algum desses tipos separadas por vírgulas.

Exemplo:

```
FETCH cursor_x INTO var1;
FETCH cursor_y INTO var1,var2,var3,var4;
```

A variável `FOUND` pode ser utilizada pra se saber se houve alguma linha retornada.
Após a utilização, um cursor tem que ser fechado com o comando "CLOSE" para liberar recursos. Porém o mesmo pode ser aberto novamente.
ser aberto novamente.

Exemplo:

```
CLOSE cursor_x;
```

33.4.2.19.3 - Funções PL/pgSQL que Retornam Cursores

Funções PL/pgSQL podem retornar cursores, que é útil para retornar muitas linhas ou colunas, principalmente conjuntos de resultados muito grandes.

A função deve abrir o cursor e retornar o nome do cursor para quem fez a chamada (ou simplesmente abrir o cursor utilizando o nome do portal especificado por, ou de outra forma conhecido por quem chamou). Quem chamar poderá ler as linhas usando o cursor.

Quem chama pode fechar o cursor, ou o mesmo será fechado automaticamente quando a transação for finalizada.

O nome do portal para o cursor pode ser definido pelo programador ou gerado automaticamente.

Para definir o nome do portal deve simplesmente atribuir uma string à variável do tipo "refcursor" antes de abrí-la.

A string da variável será utilizada pelo "OPEN" como o nome do portal subjacente.

Mas se a variável do tipo "refcursor" é nula, o comando "OPEN" automaticamente gera um nome que não dê conflito com nenhum outro portal que já exista e atribui esse nome à variável do tipo "refcursor".

Obs.: Uma variável cursor ativada é inicializada com o valor da string relativa ao seu nome, então, o nome do portal é o mesmo da variável cursor, a não ser que o programador mude esse nome fazendo uma atribuição antes de abrir o cursor. Mas uma variável cursor desativada tem inicialmente seu valor nulo por padrão, então, recebe um nome único gerado de forma automática, se esse não for mudado.

Exemplos:

```
CREATE TEMP TABLE tb_teste(campo text);
INSERT INTO tb_teste VALUES ('123');

CREATE OR REPLACE FUNCTION retorna_cursor(refcursor) RETURNS refcursor AS $$
       BEGIN
           OPEN $1 FOR SELECT campo FROM tb_teste;
           RETURN $1;
       END;
       $$ LANGUAGE plpgsql;


BEGIN; -- Abertura de uma transação;

       SELECT retorna_cursor('teste_cursor');

       FETCH ALL IN teste_cursor;

COMMIT;

/* O próximo exemplo gera o nome do cursor automaticamente */

CREATE OR REPLACE FUNCTION retorna_cursor() RETURNS refcursor AS $$
       DECLARE
            ref refcursor;
       BEGIN
            OPEN ref FOR SELECT campo FROM tb_teste;
            RETURN ref;
       END;
       $$ LANGUAGE plpgsql;


BEGIN;
       SELECT retorna_cursor();

retorna_cursor
--------------------
 <unnamed portal 1>
(1 row)
```

```
FETCH ALL IN "<unnamed portal 1>";

COMMIT;

/* Retornando vários cursores: */

CREATE OR REPLACE FUNCTION retorna_cursor(refcursor, refcursor)
RETURNS SETOF refcursor AS $$

BEGIN
       OPEN $1 FOR SELECT * FROM tabela_1;
       RETURN NEXT $1;
       OPEN $2 FOR SELECT * FROM tabela_2;
       RETURN NEXT $2;
       RETURN;
END;
$$ LANGUAGE plpgsql;

-- é necessário estar em uma transação para poder usar cursor
BEGIN;
SELECT * FROM retorna_cursor('a', 'b');
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

34 - Gatilhos – Triggers

Triggers são objetos que determinam ações a serem executadas antes ou depois de um evento em uma tabela. Essas ações podem ser disparadas por comando SQL (statement) ou por linha (row).

Para se criar um gatilho é necessário que já exista uma função (strored procedure), a qual será usada como ação disparada por um evento . E essa função deverá ter o retorno do tipo `TRIGGER`.

Sintaxe:

```
CREATE TRIGGER nome { BEFORE | AFTER } { evento [ OR ... ] }
       ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]
       EXECUTE PROCEDURE nome_função ( argumentos );
```

Detalhes:

- nome: Nome do trigger a ser criado;
- BEFORE: A função será executada antes do evento;
- AFTER: A função será executada depois do evento;
- evento: INSERT, UPDATE ou DELETE;
- tabela: Nome da tabela que terá um gatilho vinculado;
- ROW: A função será disparada para cada linha;
- STATEMENT: A função será disparada apenas uma vez;
- nome_função: Nome da função vinculada ao gatilho.

Quando uma função PL/pgSQL é chamada como um trigger, diversas variáveis especiais são criadas automaticamente no bloco de nível superior.

Elas são:

Nome	Tipo	Descrição
NEW	RECORD	Contém a nova linha do banco de dados para operações do tipo INSERT/UPDATE nos gatilhos no nível de linha. O valor dessa variável é nulo (NULL) nos gatilhos no nível de instrução (statement).
OLD	RECORD	Contém a antiga linha do banco de dados (antes da modificação), para operações do tipo UPDATE/DELETE nos gatilhos no nível de linha. O valor dessa variável é nulo (NULL) nos gatilhos no nível de instrução (statement).
TG_NAME	name	Contém o nome do gatilho disparado.
TG_WHEN	text	String que contém BEFORE ou AFTER, dependendo da definição do gatilho.
TG_LEVEL	text	String que contém ROW ou STATEMENT, conforme a definição do gatilho.
TG_OP	text	String que contém INSERT, UPDATE, ou DELETE, informando para qual operação o gatilho foi disparado.
TG_RELID	oid	O ID de objeto da tabela que disparou o gatilho.
TG_RELNAME	name	O nome da tabela que disparou o gatilho.
TG_NARGS	integer	Número de argumentos fornecidos ao procedimento de gatilho na instrução “CREATE TRIGGER”.
TG_ARGV[]	text	Os argumentos da instrução CREATE TRIGGER. O contador do índice começa por 0. Índices inválidos (menor que 0 ou maior ou igual a tg_nargs) resultam em um valor nulo.

Exemplo:

```
        /* Criação da tabela original */
        CREATE TEMP TABLE tb_teste(
              campo1 int
        );

        /* Criação da tabela de logs */
        CREATE TEMP TABLE tb_teste_log(
             codigo serial primary key,
           data date,
           usuario varchar(15),
           modificacao char(6)
        );

        /* Criação da função que será vinculada ao trigger */
        CREATE FUNCTION func_log() RETURNS trigger AS $$
               BEGIN
                    INSERT INTO tb_teste_log(data, usuario, modificacao) VALUES (now(), user,
TG_OP);
                    RETURN NEW;
               END;
               $$ LANGUAGE plpgsql;

        /* Criação do trigger */
        CREATE TRIGGER tg_teste_log AFTER INSERT OR UPDATE OR DELETE ON tb_teste
        FOR EACH ROW EXECUTE PROCEDURE func_log();

        /* Inserções, Remoções e Modificações na tabela tb_teste */
        INSERT INTO tb_teste VALUES (1),(2),(3),(4);
        DELETE FROM tb_teste WHERE campo1 = 2 OR campo1 = 4;
        UPDATE tb_teste SET campo1 = 7 WHERE campo1 = 3;

        /* Verificando a tabela de logs */
        SELECT * FROM tb_teste_log;
```

35 - Personalizando Operadores

O PostgreSQL permite aos usuários criarem operadores personalizados, ao invés de se utilizar funções com o mesmo propósito.

No entanto, assim como triggers, cada operador deve ser assimilado a uma função.

Podem ser criado operadores dos tipos unário esquerdo, unário direito e binário.

Assim como acontece com funções também é possível fazer sobrecarga de operadores, cujo comportamento irá variar conforme o(s) tipo(s) de dados em que for usado.

Sintaxe geral da criação de um operador:

```
CREATE OPERATOR simbolo_operador (
       PROCEDURE = funcao_atrelada,
       RIGHTARG = tipo_argumento_direito,
       LEFTARG = tipo_argumento_esquerdo
```

Exemplo:

```
/* Criação da função para o operador binário */
CREATE OR REPLACE FUNCTION fc_operador_potencia (int,int) RETURNS double precision AS $$
       SELECT power($1,$2);
$$ LANGUAGE SQL;


/* Operador Binário */
CREATE OPERATOR ** (
LEFTARG = int,
RIGHTARG = int,
PROCEDURE = fc_operador_potencia
);

/* Criação da função para os operadores unários */
CREATE OR REPLACE FUNCTION fc_operador_potencia (int) RETURNS double precision AS $$
       SELECT power($1,2);
$$ LANGUAGE SQL;

/* Operadores Unários */


/* Esquerdo */
CREATE OPERATOR ** (
LEFTARG = int,
PROCEDURE = fc_operador_potencia
);

/* Direito */
CREATE OPERATOR ** (
RIGHTARG = int,
PROCEDURE = fc_operador_potencia
);

/* Utilização do operador */

SELECT 2**9;  -- 2 elevado a 9 (operador binário)
SELECT 3**;  -- 3 ao quadrado (operador unário esquerdo)
SELECT **3;  -- 3 ao quadrado (operador unário direito)
```

No exemplo acima podemos ver a sobrecarga não só da função, como também do próprio operador.

O operador quando usado de forma unária simplesmente eleva ao quadrado e na forma binária, o argumento esquerdo é elevado ao direito (potência).

36 - Regras - Rules

RULES é um sistema de regras (próprio do PostgreSQL, não é padrão SQL), o qual permite definir uma ação alternativa a ser realizada nos eventos em tabelas ou visões. Grosso modo, uma regra faz com que sejam executados comandos adicionais quando é executado um determinado comando em uma determinada tabela.

Na verdade, uma regra é um mecanismo que transforma comandos. Tal transformação ocorre antes de se executar o comando.

Regras não disparam independentemente para cada linha física como pode ser definido em um gatilho.

Os eventos que podem disparar uma RULE são: SELECT, INSERT, UPDATE e DELETE.

Sintaxe:

```
CREATE [ OR REPLACE ] RULE nome_rule AS ON evento TO tabela [ WHERE condição ]
       DO [ ALSO | INSTEAD ] { NOTHING | comando | ( comando ; comando ... ) };
```

O que vem após o DO, como mostra a sintaxe acima, é a ação executada pela regra disparada por um evento que pode ser:

- INSTEAD: Indica que o(s) comando(s) deve(m) ser executado(s) ao invés do comando original;
- ALSO: Indica que o(s) comando(s) deve(m) ser executado(s) em adição ao comando original. Caso ALSO ou INSTEAD não forem especificados, ALSO é o padrão.;
- NOTHING: Não faz coisa alguma, ou seja, anula o comando dado ou não faz coisa alguma além do comando;
- comando: O(s) comando(s) que faz a ação da regra. Os comandos válidos são SELECT, INSERT, UPDATE, DELETE, ou NOTIFY.

Exemplo:

```
/* Criação da tabela de funcionários que já pertenceram à empresa */
CREATE TABLE ex_colaboradores(
       mat_id character(5) NOT NULL,
       nome character varying(15) NOT NULL,
       snome character varying(30) NOT NULL,
       dt_demis date,
       CONSTRAINT ex_colaboradores_pkey PRIMARY KEY (mat_id)
);

/* Criação da RULE */
CREATE OR REPLACE RULE rl_ex_colaboradores AS ON DELETE TO colaboradores
       DO INSERT INTO ex_colaboradores VALUES (OLD.mat_id,OLD.nome,OLD.snome,now());

/* Demissão de um funcionário */
DELETE FROM ONLY colaboradores WHERE mat_id = '00016';

/* Constatação do preenchimento da tabela */
SELECT * FROM ex_colaboradores ;
```

Há casos em que em dados eventos, para se proteger os dados, nada deve ser feito:

```
/* Criação da tabela a ser protegida */
CREATE TEMP TABLE protegida( campo1 int );

/* Inserção de valores */
INSERT INTO protegida VALUES (1),(2),(3),(4),(5),(6),(7);

/* Criando regras para proteger contra modificações ou remoções, respectivamente */
CREATE OR REPLACE RULE rl_escudo_protegida_upd AS ON UPDATE TO protegida DO INSTEAD NOTHING;
CREATE OR REPLACE RULE rl_escudo_protegida_del AS ON DELETE TO protegida DO INSTEAD NOTHING;

/* Tentativas (frustradas) de violar os dados inseridos na tabela */
UPDATE protegida SET campo1 = 1000 WHERE campo1 = 1;
DELETE FROM protegida WHERE campo1 = 2;
```

36.1 - Atualizando Views com Rules

É sabido que uma view nada mais é do que uma consulta personalizada de uma tabela, sendo um objeto apenas para visualização de dados, não sendo possível inserir e nem apagar dados dela.

No entanto, aplicando uma regra isso pode ser possível desviando o evento da view para a tabela de origem:

```
/* Criação da view */
CREATE VIEW vw_colaboradores_sp AS SELECT * FROM colaboradores WHERE uf='SP' AND cidade = 'São Paulo';

/* Criação da rule */
CREATE RULE rl_vw_colaboradores_sp AS ON INSERT TO vw_colaboradores_sp DO INSTEAD
INSERT INTO colaboradores (mat_id,nome,snome,dt_admis,chefe_direto,uf,cidade) VALUES
                        (NEW.mat_id,NEW.nome,NEW.snome,now(),NEW.chefe_direto,'SP','São Paulo');

/* Inserção de um registro na view */
INSERT INTO vw_colaboradores_sp (mat_id,nome,snome,chefe_direto) VALUES ('00051','Escendino','Matos','00001');

/* Verificando o resultado */
SELECT * from vw_colaboradores_sp ;
```

36.2 - Removendo Rules

Para apagar uma `RULE` é necessário especificar também o nome da relação à qual está atrelada:

Sintaxe:

```
DROP RULE [ IF EXISTS ] nome_rule ON relação [ CASCADE | RESTRICT ]
```

Exemplo:

```
DROP RULE rl_vw_colaboradores_sp ON vw_colaboradores_sp;
```

36.3 - Rules X Trigggers

Rules

- Não é possível usar uma linguagem procedural;
- Substitui, Adiciona ou desabilita comandos;
- Ao contrário de triggers, podem alterar a consulta executada;
- É independente de funções definidas pelo usuário.

Triggers

- É um mecanismo mais complexo, porém muito mais poderoso;
- Determinam funções a serem executadas antes ou depois de um tipo de evento;
- Antes de sua criação é preciso criar uma função cujo retorno é do tipo trigger e não pode receber parâmetros;
- Pode executar um comando que acione outro trigger;

37 - Herança de Tabelas

Assim como em linguagens orientadas a objetos, de maneira similar, há o conceito de herança de tabelas no PostgreSQL.

A Tabela-mãe possui campos comuns a todas as outras que dela se originarão, e as tabelas-filhas, cada uma terá colunas próprias além das herdadas.

O vínculo de uma tabela-mãe com uma tabela-filha se efetiva com a palavra chave “INHERITS”.

Exemplo:

```
/* Tabela-mãe */
CREATE TEMP TABLE pessoa(
       nome varchar(15),
       endereco varchar(30),
       telefone char(10)
);

/* Tabelas-filhas */
CREATE TEMP TABLE pessoa_juridica(
       razao_social varchar(40),
       cnpj char(14)
) INHERITS (pessoa);

CREATE TEMP TABLE pessoa_fisica(
       sobrenome varchar(40),
       cpf char(11)
) INHERITS (pessoa);
```

Exibição da estrutura de cada tabela:

```
\d pessoa
```

Coluna	Tipo
nome	character varying(15)
endereco	character varying(30)
telefone	character(10)

```
\d pessoa_juridica
```

Coluna	Tipo
nome	character varying(15)
endereco	character varying(30)
telefone	character(10)
razao_social	character varying(40)
cnpj	character(14)

```
\d pessoa_fisica
```

Coluna	Tipo
nome	character varying(15)
endereco	character varying(30)
telefone	character(10)
sobrenome	character varying(40)
cpf	character(11)

37.1 - Inserindo Dados em uma Tabela-Filha

Ao se inserir dados em uma tabela-filha, automaticamente eles serão também vistos na tabela-mãe, por padrão, nos campos que ela possuir em comum com as outras:

```
 INSERT INTO pessoa_fisica VALUES ('Georg','Rua da Resistência,
35','1122223333','Ohm','12345678901');

/* Verificando os dados na tabela-filha */
SELECT * FROM pessoa_fisica;

/* Verificando os dados na tabela-mãe */
SELECT * FROM pessoa;
```

Como pôde ser constatado, a princípio, tudo que as filhas tiverem a mãe também terá. Se por acaso em uma coluna de valores únicos comum entre mãe e filhas, o mesmo valor for inserido tanto na mãe quanto em uma filha, se não for usado parâmetro “ONLY” antes do nome da tabela-mãe exibirá o mesmo valor mais de uma vez.

```
/* Visualizando apenas os resultados da tabela-mãe */
SELECT * FROM ONLY pessoa;
```

Além do SELECT, o parâmetro “ONLY” pode ser usado também com DELETE e UPDATE.

38 - Array de Colunas

Também conhecido como vetor ou matriz é um recurso muito útil em linguagens de programação e que também está disponível como recurso
em colunas de tabelas do PostgreSQL.

Para se definir um campo como vetor, adiciona-se colchetes ([]) logo depois do tipo de dado.

Exemplo:

```
CREATE TABLE tb_array(
	campo_array_a int[],
	campo_array_b varchar(10)[][][] -- Uma matriz de 3 dimensões
);
```

Até o momento, se especificar um tamanho fixo para o array ([n]), não fará diferença, pois a implementação atual não força a obdecer o tamanho definido.

38.1 - Inserindo Dados em Arrays

Os valores a serem inseridos em um vetor devem ser envoltos por chaves ({}) e separados por vírgulas.

Em matrizes multidimensionais devem ter expressões de matriz com dimensões correspondentes.

Exemplo:

```
INSERT INTO tb_array VALUES (
	'{1,2,3,4,5}',
	'{{"valorX1","valorX2","valorX3"},{"valorY1","valorY2","valorY3"},
		{"valorZ1","valorZ2","valorZ3"}}'
);
```

38.2 - Selecionando Dados de Arrays

Para selecionarmos os dados de um vetor ou um array definimos sua posição com apenas um par de colchetes:

```
/* Selecionar a segunda posição */
SELECT campo_array_a[2] FROM tb_array;
```

É possível definir um faixa de valores de um array para consulta, usando como delimitador dois pontos (:):

```
/* Selecionar da terceira à quinta posição */
SELECT campo_array_a[3:5] FROM tb_array;
```

No caso de matrizes (vetor de mais de uma dimensão), são necessários 2 pares de colchetes, sendo que o primeiro informa em qual dimensão se encontra e o segundo em que posição:

```
/* Dimensão 3, Posição 1 */
SELECT campo_array_b[3][1] FROM tb_array;
```

39 - Particionamento de Tabelas

Particionar tabelas consiste em dividir uma tabela que recebe um grande volume de dados. Essa divisão é feita entre outras tabelas que na verdade são tabelas-filhas cujos dados são recebidos por redirecionamentos conforme uma regra (geralmente faixas de valores). Cada tabela-filha é chamada de partição, nesse caso. A tabela-mãe é vazia geralmente, sua existência serve apenas para representar os dados por inteiro.

Importante: O conceito de particionamento aqui agora apresentado **não tem nada a ver** com o particionamento de discos.

Benefícios:

- Performance melhorada para consultas;
- Quando consultas ou atualizações acessam uma grande porcentagem de uma única partição, o desempenho pode ser melhorado pela vantagem de varredura sequencial daquela partição ao invés de usar um índice e acessos aleatórios que faz a leitura espalhada por toda a tabela;
- Grandes cargas e remoções podem ser efetudadas por adições ou remoções de partições. `ALTER TABLE` é muito mais rápido do que em operações de grande carga. Evita também sobrecarga de `VACUUM` causada por um grande volume de `DELETE`.

39.1 - Formas de Particionamento do PostgreSQL

- Range Partitioning

É a forma mais utilizada. A tabela é particionada dentro de faixas de dados definidas por uma coluna-chave ou um conjunto de colunas.

- List Partitioning

A tabela é particionada explicitamente listando que valores chaves aparecem em cada partição.

39.2 - Implementação do Particionamento

1. Crie a tabela-mãe, de onde as partições irão se originar. Esta tabela não deve ter dados. Não defina qualquer restrição do tipo "CHECK", a não ser que pretenda aplicá-la igualmente a todas as partições:

```
CREATE TABLE tb_mae( valor int);
```

2. Crie as tableas-filhas (as partições) que herdarão as propriedades da tabela-mãe:

```
CREATE TABLE tb_filha_1() INHERITS (tb_mae);
CREATE TABLE tb_filha_2() INHERITS (tb_mae);
CREATE TABLE tb_filha_3() INHERITS (tb_mae );
```

3. Adicione restrições às partições para definir os valores chaves permitidos em cada partição:

```
ALTER TABLE tb_filha_1 ADD CONSTRAINT chk_valor CHECK (valor BETWEEN 0 AND 9999);
ALTER TABLE tb_filha_2 ADD CONSTRAINT chk_valor CHECK (valor BETWEEN 10000 AND 19999);
ALTER TABLE tb_filha_3 ADD CONSTRAINT chk_valor CHECK (valor BETWEEN 20000 AND 29999);
```

4. Para cada partição criar um índice na coluna principal:

```
CREATE INDEX idx_tb_filha_1_valor ON tb_filha_1 (valor);
CREATE INDEX idx_tb_filha_2_valor ON tb_filha_2 (valor);
CREATE INDEX idx_tb_filha_3_valor ON tb_filha_3 (valor);
```

5. Verifique se o parâmetro "constraint_exclusion" está desabilitado (off) no arquivo postgresql.conf. Caso estiver, as consultas não serão otimizadas como desejado. Com esse parâmetro desabilitado, as consultas vasculharão cada uma das partições.

6. Crie uma RULE ou um TRIGGER para fazer o redirecionamento da tabela-mãe para a tabela-filha conforme o valor citado:

```
/* I) Com Trigger */

-- Criação da Função para o Trigger

CREATE OR REPLACE FUNCTION fc_particionamento() RETURNS TRIGGER AS $$
       BEGIN
               IF (NEW.valor BETWEEN 0 AND 9999) THEN
                       INSERT INTO tb_filha_1 VALUES (NEW.*);
               ELSIF(NEW.valor BETWEEN 10000 AND 19999) THEN
                       INSERT INTO tb_filha_2 VALUES (NEW.*);
               ELSIF(NEW.valor BETWEEN 20000 AND 29999) THEN
                       INSERT INTO tb_filha_3 VALUES (NEW.*);
               ELSE
                       RAISE EXCEPTION 'Valor Inválido!!!';
               END IF;

               RETURN NULL;

               END;
$$ LANGUAGE PLPGSQL;

-- Criação do Triger

CREATE TRIGGER tg_particionamento BEFORE INSERT ON tb_mae
FOR EACH ROW EXECUTE PROCEDURE fc_particionamento();

/*=========================================================*/

/* II) Inserção e verificação de valores */

-- Inserção
INSERT INTO tb_mae VALUES (55),(100),(20700),(14777);

-- Verificando na tabela principal:
SELECT * FROM tb_mae;

-- Verificando em cada partição se os valores correspondem à faixa
SELECT * FROM tb_filha_1;
SELECT * FROM tb_filha_2;
SELECT * FROM tb_filha_3;

/*=========================================================*/

/* III) Com Rules */

-- Eliminação do Trigger e sua Função

DROP FUNCTION fc_particionamento() CASCADE;

-- Criação da Rule para a Partição 1

CREATE OR REPLACE RULE rl_particionamento_1 AS ON INSERT TO tb_mae
WHERE valor    BETWEEN 0 AND 9999 DO INSTEAD
INSERT INTO tb_filha_1 VALUES (NEW.*);

-- Criação da Rule para a Partição 2

CREATE OR REPLACE RULE rl_particionamento_2 AS ON INSERT TO tb_mae
WHERE valor BETWEEN 10000 AND 19999 DO INSTEAD
INSERT INTO tb_filha_2 VALUES (NEW.*);
```

```
        -- Criação da Rule para a Partição 3

        CREATE OR REPLACE RULE rl_particionamento_3 AS ON INSERT TO tb_mae WHERE valor   BETWEEN
20000 AND 29999 DO INSTEAD
        INSERT INTO tb_filha_3 VALUES (NEW.*);
```

No exemplo apresentado tanto faz usar TRIGGER ou RULEs, o resultado será o mesmo (que pode ser conferido conforme o passo II).

40 - Administração de Usuários

40.1 - Roles

O PostgreSQL gerencia permissões de acesso usando o conceito de "roles", cujo uso pode é para usuários e grupos, dependendo de como um "role" é definido.

"Roles" podem possuir objetos do banco de dados e determinar privilégios nesses objetos para outros "roles", de modo a controlar quem deverá ter acesso a esses objetos.

Além disso, é possível conceder um membro de um "role" para outro, permitindo assim usufruir dos privilégios que foram concedidos a esse outro "role".

O conceito de "role" engloba o conceito de usuários e grupos. Em versões do PostgreSQL antes da 8.1, usuários e grupos eram distintos tipos de entidades, mas agora há apenas "roles". Qualquer "role" pode serr um usuário, um grupo ou ambos.

"Roles" do banco de dados são conceitualmente completamente separados dos usuários do sistema operacional. Na prática, pode ser conveniente manter uma correspondẽncia, mas isso não é requerido.

Sintaxe:

```
CREATE ROLE nome_role [ [ WITH ] opções [ ... ] ]
```

Onde opções podem ser:

- **SUPERUSER | NOSUPERUSER** : Determinam se o novo "role" é um super usuário, que pode sobrepor todas restrições de acesso. O status de super usuário é perigoso e deve ser usado somente quando é realmente necessário. Você mesmo deve ser um super usuário para criar um novo super usuário. Se não for especificado NOSUPERUSER é o padrão;
- **CREATEDB | NOCREATEDB** : Determinam se um "role" pode criar um banco de dados. Se CREATEDB é especificado, o "role" poderá criar novos bancos de dados. Especificando NOCREATED", que é padrão, negará ao "role" poder criar banco de dados;
- **CREATEROLE | NOCREATEROLE** : Determinam se um "role" pode criar novos "roles". Um "role" com o privilégio CREATEROLE pode também alterar ou apagar outros "roles".
- Se nada for especificado NOCREATEROLE é o padrão;
- **CREATEUSER | NOCREATEUSER** : Essas cláusulas são obsoletas, mas ainda são aceitas, São equivalentes a CREATEROLE e NOCREATEROLE, respectivamente;
- **INHERIT | NOINHERIT** : Determinam um "role" herdando os privilégios de "roles" que é membro. Um "role" com o atributo INHERIT pode automaticamente usar qualquer privilégio do banco de dados que foram concedidos diretamente ou indiretamente a um membro. Sem INHERIT, a adesão em outro "role" apenas garante definir um "role" para outro "role". Se nada for especificado INHERIT é o padrão";
- **LOGIN | NOLOGIN** : Determinam se um "role" está permitido a entrar no sistema, isso é, se o "role" poderá ter autorização a se conectar. Um"role" tento o atributo LOGIN pode ser pensado como um usuário. "Roles" sem esse atributo são úteis para gerenciar privilégios de bancos de dados, mas não são como usuários. Se não for especificado NOLOGIN é o padrão", exceto quando CREATE ROLE é executado usando o parâmetro CREATE USER;
- **CONNECTION LIMIT connlimit** : Se o "role" puder entrar no sistema, isso especifica como muitas conexões concorrentes o "role" pode fazer. O padrão é "-1", que siginifica: sem limites;
- **[ENCRYPTED | UNENCRYPTED] PASSWORD 'password'** : Configura a senha do "role", sendo apenas para "roles" que tenham o atributo LOGIN, mas pode-se definir para um que não tenha. Se não planeja usar autenticação com senha, pode-se omitir essa opção. Se nenhuma senha for especificada, será nula e a autenticação sempre falhará para esse usuário. Uma senha nula pode ser opcionalmente ser escrita explicitamente como PASSWORD NULL. ENCRYPTED ou UNENCRYPTED controlam se a senha é armazenada encriptada no sistema de catálogos. Se nenhuma for especificada, o comportamento padrão é determinado pelo parâmetro de configuração "password encryptation", no arquivo postgresql.conf. Se a senha apresentada já estiver no formato encriptado MD5, então ela será armazenada como é, independentemente se ENCRYPTED ou UNENCRYPTED for especificados (desde que o sistema não possa descriptografar a string especificada senha

criptografada). Isso permite recarga de senhas criptografadas durante "dump" ou "restore". Clientes mais antigos podem não ter suporte para autenticação com o mecanismo MD5 que é preciso para funcionar com senhas que são armazenadas encriptadas;

- **VALID UNTIL 'timestamp'** : Configura a data e hora, que após isso a senha não será mais aceita. Se essa cláusula for omitida a senha será válida para sempre;
- **IN ROLE rolename [, ...]** : Lista um ou mais "roles" existentes para quais o novo "role" será imediatamente adicionado como membro. (Note que não há opção para adicionar o novo "role" como um administrador, use o comando `GRANT` separadamente para isso;
- **IN GROUP rolename [, ...]** : Para versões a partir da 8.1 é totalmete obsoleta, cujo uso se equivale a "`IN ROLE`";
- **ROLE rolename [, ...]** : Lista um ou mais "roles" existentes que são automaticamente adicionado como membros do novo "role". (Faz o papel de um novo grupo);
- **ADMIN rolename [, ...]** : É como `ROLE`, mas os nomes de "roles" adicionados ao novo "role" `WITH ADMIN`, dando-lhes o direito de conceder a adesão neste "role" para outros;
- **USER rolename [, ...]** : Seu uso é obsoleto, equivale a `ROLE`;
- **SYSID uid**: É uma cláusula ignorada, porém aceita para compatibilidade com versões anteriores.

Exemplos:

```
/* Criação de um usuário*/
CREATE ROLE usuario1 LOGIN;

/* Criação de usuario comum com senha definida em "123" */
CREATE ROLE usuario2 LOGIN ENCRYPTED PASSWORD '123';

/* Criação de super usuário com senha definida em "123" */
CREATE ROLE dbmaster LOGIN SUPERUSER ENCRYPTED PASSWORD '123';

/* Criação do role contâiner (grupo) financeiro */
CREATE ROLE financeiro;

/* Criação do role marcia dentro do grupo financeiro */
CREATE ROLE marcia LOGIN ENCRYPTED PASSWORD '123' IN ROLE financeiro;

/* Criação do role clara dentro do grupo financeiro */
CREATE ROLE clara LOGIN ENCRYPTED PASSWORD '123'  IN ROLE financeiro;
```

40.2 - Apagando Roles

Como em outros tipos de objeto, o comando `DROP` é usado para remoção. Porém, se um “role” possui algum objeto isso não é possível sem antes repassar os objetos de sua propriedade para outro “role” com o comando `REASSIGN OWNED BY role_original TO role_novo` ou mesmo apagando todos objetos de sua propriedade com o comando `DROP OWNED BY role_...`

Exemplo:

```
/* Criação de um objeto do tipo tabela */
CREATE TEMP TABLE tb_1(valor int);

/* Atribuição de propriedade da tabela criada para usuario1 */
ALTER TABLE tb_1 OWNER TO usuario1 ;

/* Tentativa (frustrada) de remoção do role usuario1 */
DROP ROLE usuario1;

/* Passando as propriedades de usuario 1 para usuario2@db1 */
REASSIGN OWNED BY usuario1 TO usuario2 ;

/* Tentativa (com sucesso) de remoção do role usuario1 */
DROP ROLE usuario1;

/* Removendo todos os objetos de propriedade de usuario2 sem remover o role */
DROP OWNED BY usuario2;
```

40.3 - Grupos

Também conhecidos como “roles contâiners” são como grupos de usuários. São “roles” que contém outros “roles”.

Exemplos:

```
/* Roles contâiner */
CREATE ROLE comercial;
CREATE ROLE vendas;

/* Criação de contâiner com 2 roles assimilados */
CREATE ROLE diretores WITH ROLE clara,marcia;

/* Inserindo um role de login em um role contâinter */
GRANT vendas TO marcia;

/* Removendo um role de login de um role contâinter */
REVOKE financeiro FROM clara;

/* Usando o comando "GRANT" e o parâmetro "WITH ADMIN OPTION"
permite conceder o direito de administração do role a outro role */
GRANT vendas TO clara WITH ADMIN OPTION;

/* Inversamente podemos revogar tal direito */
REVOKE ADMIN OPTION FOR vendas FROM clara;

/* Exibindo todos os roles */
\du

/* ...individualmente */
\du clara
```

40.4 - Concedendo ou Revogando Acesso a Objetos

Na criação de um objeto, quem cria (o “role”), passa a ser o dono e tem poderes sem restrições. Para que outros roles tenham acesso é preciso conceder esse privilégio, que pode ser dado a “roles” contâiners, em que seus membros poderão usufruir dos mesmos direitos. Os comandos `GRANT` e `REVOKE`, respectivamente, concedem ou tiram privilégios, fazem parte da subdivisão DCL (Data Control Language – Linguagem de Controle de Dados) que tratam dessa parte da linguagem SQL. Os tipos de privilégios que podem ser concedidos são:

- **SELECT**: Permite selecionar qualquer coluna de uma especificada tabela, visão ou sequẽncia. Também permite o uso de `COPY TO`. Esse privilégio é também necessário para referenciar valores existentes em colunas em `UPDATE` ou `DELETE`. Para sequências, este privilégio também permite o uso da função `currval`;
- **INSERT**: Permite inserir uma nova linha em uma tabela especificada. Permite também `COPY FROM`;
- **UPDATE**: Permite atualizar qualquer coluna de uma tabela especificada. (Na prática, qualquer comando de atualização não trivial requirirá privilégio `SELECT` bem como, uma vez que as colunas da tabela para determinar que linhas atualizar, e/ou calcular novos valores para colunas). `SELECT ... FOR UPDATE` e `SELECT ... FOR SHARE` também requerem esse privilégio, em adição ao privilégio de `SELECT`. Para sequências, este privilégio permite o uso das funções `nextval` e `setval`;
- **DELETE**: Permite deltar uma linha da tabela especificada. (Na prática, qualquer `DELETE` não trivial requirirá o privilégio `SELECT` bem como, uma vez que as colunas da tabela tem de referência para determinar quais as linhas que deseja excluir);
- **REFERENCES**: Para criar uma restrição de chave estrangeira, é necessário ter este privilégio em ambas as tabelas referenciadas;
- **TRIGGER**: Permite a criação de um gatilho na tabela especificada;
- **CREATE**: Para bancos de dados, permite novos esquemas serem criados dentro do banco de dados. Para esquemas, permite novos objetos serem criados dentro do esquema. Para renomear

um objeto existente, você deve possuir o objeto e ter este privilégio para o esquema que o contém. Para "tablespaces", permite tabelas, índices e arquivos temporários serem criados dentro do "tablespace", e permite bancos de dados serem criados terem o "tablespace" como padrão. (Note que revogando este privilégio não poderá alterara localização dos objetos existentes);

- **CONNECT**: Permite ao usuário conectar ao banco de dados especificado. Este privilégio é checado no início da conexão (em adição checa qualquer restrições impostas pelo arquivo pg_hba.conf);
- **TEMPORARY | TEMP**: Permite tabelas temporárias serem criadas enquanto se usa o banco de dados especificado;
- **EXECUTE**: Permite o uso da função especificada e o uso de quaisquer operadores que são implementados no topo da função. Este é o único tipo de privilégio que é aplicável a funções. (Sua sintaxe funciona para funções de agregação);
- **USAGE**: Para linguagens procedurais, permite o uso de uma linguagem especificada para criação de funções nela. Este é o único privilégio que é aplicável a linguagens procedurais. Para esquemas, permite acesso a objetos contidos no esquema especificado (supondo que os requirimentos próprios de objetos também estejam preenchidos). Essencialmente isso permite a quem é concedido "procurar" objetos dentro do esquema. Sem essa permissão, ainda é possível ver os nomes dos objetos, por exemplo, consultando as tabelas de sistema. Também, após revogar esta permissão, "backends" existentes devem ter declarações que já tenham realizado essa pesquisa, então não é uma maneira completamente segura para prevenir o acesso ao objeto. Para sequências, este privilégio permite o uso das funções `currval` e `nextval`;
- **ALL PRIVILEGES**: Concede todos os privilégios imediatamente. A palavra-chave `PRIVILEGES` é opcional no PostgreSQL, embora seja requirida pelo SQL padrão. Os privilégios requiridos por outros comandos são listados na página de referência do mesmo.

Exemplos:

```
/* Role contâiner */
CREATE ROLE masters SUPERUSER;

/* Usuário simples */
CREATE ROLE newusr NOSUPERUSER LOGIN ENCRYPTED PASSWORD '123';

/* Tabelas para teste */
CREATE TEMP TABLE tb_acesso1(valor int);
CREATE TEMP TABLE tb_acesso2(valor int);
CREATE TEMP TABLE tb_acesso3(valor int);
CREATE TEMP TABLE tb_acesso4(valor int);
CREATE TEMP TABLE tb_acesso5(valor int);

/* Conceder direito de INSERT em 2 tabelas */
GRANT INSERT ON tb_acesso1,tb_acesso2 TO masters;

/* Conceder direito de SELECT para qualquer um */
GRANT SELECT ON tb_acesso1 TO PUBLIC;

GRANT masters TO newusr;

/* Revogando todos os privilegios de todos os usuários em 3 tabelas */
REVOKE ALL ON tb_acesso2,tb_acesso3,tb_acesso4 FROM PUBLIC;

/* Revogando o direito de inserir na tabela tb_acesso5 para os roles newusr e masters */
REVOKE INSERT ON tb_acesso5 FROM newusr,masters;
```

<table>
<tr><td rowspan="39">GRANT</td><td colspan="2">SELECT</td><td rowspan="39">ON</td><td rowspan="15">[TABLE] tablename [, ...]</td><td rowspan="39">TO</td><td rowspan="15">[GROUP]
rolename</td><td rowspan="39">[WITH
GRANT
OPTION]</td></tr>
<tr><td colspan="2">UPDATE</td></tr>
<tr><td colspan="2">INSERT</td></tr>
<tr><td colspan="2">DELETE</td></tr>
<tr><td colspan="2">TRUNCATE</td></tr>
<tr><td colspan="2">REFERENCES</td></tr>
<tr><td colspan="2">TRIGGER</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td>SELECT</td><td rowspan="2">(column [, ...])
[,...]</td></tr>
<tr><td>INSERT</td></tr>
<tr><td>UPDATE</td><td rowspan="2">ALL [PRIVILEGES]
(column [, ...]) }</td></tr>
<tr><td>REFERENCES</td></tr>
<tr><td colspan="2"></td><td></td><td rowspan="11">[GROUP]
rolename</td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="5">SEQUENCE sequencename [, ...]</td></tr>
<tr><td colspan="2">SELECT</td></tr>
<tr><td colspan="2">UPDATE</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="5">DATABASE dbname
[, ...]</td></tr>
<tr><td colspan="2">CONNECT</td></tr>
<tr><td colspan="2">TEMPORARY | TEMP</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="2">FOREIGN DATA WRAPPER fdwname
[, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td><td rowspan="15">[GROUP]
rolename</td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="2">FOREIGN SERVER servername
[, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">EXECUTE</td><td rowspan="2">FUNCTION funcname ([[argmode] [argname] argtype
[, ...]]) [, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="2">LANGUAGE langname [, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="5">SCHEMA schemaname [, ...]</td></tr>
<tr><td colspan="2">USAGE</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="2">TABLESPACE tablespacename [, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td><td></td></tr>
<tr><td colspan="2">Role [, ...]</td><td>TO</td><td colspan="4">Rolename [, ...]</td></tr>
</table>

<table>
<tr><td rowspan="39">REVOKE [GRANT OPTION FOR]</td><td colspan="2">SELECT</td><td rowspan="39">ON</td><td rowspan="15">[TABLE] tablename [, ...]</td><td rowspan="39">FROM</td><td rowspan="15">[GROUP]
rolename</td><td rowspan="39">[CASCADE
|
RESTRICT]</td></tr>
<tr><td colspan="2">UPDATE</td></tr>
<tr><td colspan="2">INSERT</td></tr>
<tr><td colspan="2">DELETE</td></tr>
<tr><td colspan="2">TRUNCATE</td></tr>
<tr><td colspan="2">REFERENCES</td></tr>
<tr><td colspan="2">TRIGGER</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td>SELECT</td><td rowspan="2">(column [, ...])
[,...]</td></tr>
<tr><td>INSERT</td></tr>
<tr><td>UPDATE</td><td rowspan="2">ALL [PRIVILEGES]
(column [, ...]) }</td></tr>
<tr><td>REFERENCES</td></tr>
<tr><td colspan="2"></td><td></td><td rowspan="10">[GROUP]
rolename</td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="10">SEQUENCE sequencename [, ...]</td></tr>
<tr><td colspan="2">SELECT</td></tr>
<tr><td colspan="2">UPDATE</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="10">DATABASE dbname
[, ...]</td></tr>
<tr><td colspan="2">CONNECT</td></tr>
<tr><td colspan="2">TEMPORARY | TEMP</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="3">FOREIGN DATA WRAPPER fdwname
[, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="3">FOREIGN SERVER servername
[, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">EXECUTE</td><td rowspan="3">FUNCTION funcname ([[argmode] [argname] argtype [, ...]]) [, ...]</td><td rowspan="15">[GROUP]
rolename</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">USAGE</td><td rowspan="3">LANGUAGE langname [, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="4">SCHEMA schemaname [, ...]</td></tr>
<tr><td colspan="2">USAGE</td></tr>
<tr><td colspan="2">[,...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td><td></td></tr>
<tr><td colspan="2">CREATE</td><td rowspan="3">TABLESPACE tablespacename [, ...]</td></tr>
<tr><td colspan="2">ALL [PRIVILEGES]</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td>REVOKE [ADMIN OPTION FOR]</td><td colspan="2">role [, ...]</td><td>FROM</td><td colspan="3">rolename [, ...]</td></tr>
</table>

40.5 - Verificando os Privilégios de Objetos

Há duas maneiras simples e equivalentes para se consultar os privilégios que “roles” têm sobre um determinado objeto:

```
\dp objeto
```
(pode ser interpretado como “display privileges”; exibir privilégios)

ou

```
\z objeto
```
(seu efeito é exatamente igual ao anterior)

Exemplo:

```
GRANT ALL ON colaboradores TO aluno ;
GRANT SELECT ON colaboradores TO PUBLIC ;

\dp colaboradores
```

Produziu a seguinte saída:

```
\dp colaboradores

                                  Access privileges
 Schema |     Name      | Type  |     Access privileges             | Column access
privileges
--------+---------------+-------+---------------------------+-------------------------
 public | colaboradores | table | postgres=arwdDxt/postgres |
                                      : aluno=arwdDxt/postgres
                                       : =r/postgres
```

No exemplo “postgres”, que por padrão é role de administrador tem todos os privilégios, assim como “aluno”, cujos priviégios foram concedidos por “postgres” e PUBLIC tem acesso de somente leitura, ou seja, só pode fazer consultas (SELECT).

Abaixo estão maiores informações e siglas vistas no exemplo:

rolename=xxxx -- privileges granted to a role
=xxxx -- privileges granted to PUBLIC
r -- SELECT ("read": leitura)
w -- UPDATE ("write": escrita)
a -- INSERT ("append": acres)
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
arwdDxt -- ALL PRIVILEGES (Todos os privilégios para tabelas, varia para outros objetos)
* -- opção de concessão de privilégio anterior
/yyyy -- role que concedeu esse privilégio

nome_de_role=xxxx -- Privilégios concedidos a um usuário ou grupo
=xxxx -- Privilégios concedidos para PUBLIC

40.6 - Concedendo ou Revogando Privilégios em Colunas

Antes da versão 8.4 só era possível fazer o controle de acesso de forma mais granular a uma tabela criando uma view especificando as colunas permitidas.

Agora pode dar ou tirar privilégios da seguinte forma:

```
GRANT SELECT (coluna1,coluna2,...) ON tabela TO role;
```

Para o exemplo prático a criação de role:

```
CREATE ROLE limitado LOGIN NOSUPERUSER;
```

Só terá acesso às colunas “nome” e “snome”, sendo de somente leitura da tabela colaboradores:

```
GRANT SELECT (nome,snome) ON colaboradores TO limitado;
```

Visualização dos privilégios:

```
\z colaboradores
```

```
                                    Access privileges
 Schema |     Name      | Type  |     Access privileges             | Column access
privileges
--------+---------------+-------+---------------------------+--------------------------
 public | colaboradores | table | postgres=arwdDxt/postgres | nome:
                                : aluno=arwdDxt/postgres      :   limitado=r/postgres
                                  : =r/postgres               : snome:
                                                              :   limitado=r/postgres
```

De forma análoga será revogado (retirado) o acesso à coluna “snome”:

```
REVOKE SELECT (snome) ON colaboradores FROM limitado;
```

Visualização dos privilégios novamente:

```
\z colaboradores
```

```
                                    Access privileges
 Schema |     Name      | Type  |     Access privileges             | Column access privileges
--------+---------------+-------+---------------------------+-------------------------
 public | colaboradores | table | postgres=arwdDxt/postgres | nome:
                                : aluno=arwdDxt/postgres               :
limitado=r/postgres
                                : =r/postgres
```

41 - Dblink – Acessando um Outro Banco de Dados

DBlink é um modulo que suporta conexões a outros bancos de dados PostgreSQL dentro de uma sessão de banco de dados.

Sinataxe:

Para consultas:

```
SELECT * FROM dblink ('dbname=banco_de_dados hostaddr=endereço_de_host user=usuário password=senha port=nº_da_porta',consulta_no_banco) AS tabela(campo1 tipo,campo2 tipo, ...);
```

Para execuções:

```
SELECT dblink_exec ('dbname=banco_de_dados hostaddr=endereço_de_host user=usuário password=senha port=nº_da_porta',comando_sql);
```

Instalação:

Como o aprendizado deste curso é baseado na distribuição Debian, usaremos suas ferramentas. E primeiramente é preciso saber qual é o nome do pacote que contém o DBlink:

```
apt-cache search dblink
```

De posse do valor retornado:

```
apt-get install postgresql-contrib
```

Agora é necessário saber onde estão os arquivos referentes ao DBlink:

```
dpkg -L postgresql-contrib | grep dblink.sql
```

```
/usr/share/postgresql/8.4/contrib/dblink.sql
/usr/share/postgresql/8.43/contrib/uninstall_dblink.sql
```

Sendo que, ao ler o nome dos arquivos, fica óbvio constatar que o primeiro instala e o segundo desinstala.

A instalação do módulo:

```
psql -d template1 -U aluno -f /usr/share/postgresql/8.4/contrib/dblink.sql
```

Exemplo:

```
/* Criação dos bancos de dados */
CREATE DATABASE banco1;
CREATE DATABASE banco2;

/* Conectando ao banco1 */
\c banco1

/* Criação de uma tabela de exemplo e inserção de dados */
CREATE TABLE tb1(campo int);
INSERT INTO tb1 VALUES (1),(2),(3),(4),(5),(6),(7);

/* Conectando ao banco2 */
\c banco2
```

```
        /* Acessando dados da tabela tb1 que está em banco1 */

        SELECT * FROM dblink ('dbname=banco1 hostaddr=127.0.0.1 user=aluno password=123
port=5432','SELECT * FROM tb1') AS tb2(campo int);

        /* Inserindo dados na tabela tb1 que está em banco1 */

        SELECT dblink_exec ('dbname=banco1 hostaddr=127.0.0.1 user=aluno password=123
port=5432','INSERT INTO tb1 VALUES (8),(9)');
```

42 - VACUUM – Alocando Espaços sem Desperdício

O comando VACUUM recupera as áreas de armazenamento ocupadas por registros mortos. Em uma operação normal do PostgreSQL, registros que são deletados ou transformados em obsoletos por um UPDATE não são fisicamente removidos de suas tabelas, eles permanecem presentes até o comando VACUUM ser feito. Portanto, é necessário executar o VACCUM periodicamente, especialmente em tabelas frequentemente atualizadas.

O comando VACCUM ANALIZE além do comportamento padrão de VACUUM faz uma análise para cada tabela selecionada. Essa é uma forma de combinação acessível para scripts de manutenção de rotina. Veja adiante sobre o comando ANALIZE para mais detalhes sobre como ele processa.

VACUUM (sem o parâmetro FULL) apenas recupera o espaço e o torna disponível para reutilização. Essa forma de comando pode operar em paralelo com leituras e escritas normais da tabela, devido ao fato de não haver uma trava ("lock") exclusiva.

Já VACUUM FULL faz processamento mais extensivo, incluindo mover registros através de blocos para tentar compactar a tabela para o mínimo número blocos de disco. Essa forma é muito lenta e requer uma trava exclusiva em cada tabela enquanto é processada.

Sintaxe:

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ tabela [ (coluna [, ...] ) ] ]
```

Parãmetros:

FULL	Recupera mais espaço, mas leva mais tempo e requer exclusividade de trava ("lock") de tabela.
FREEZE	Provoca o congelamento dos registros. É um parâmetro obsoleto e será removido num futuro lançamento.
VERBOSE	Imprime na tela a atividade de relatório de "VACUUM" para cada tabela. Pode ser usado para ajudar a determinar configurações para "max_fsm_pages", "max_fsm_relations" e "default_statistics_targets" que se encontram no arquivo "postgresql.conf".
ANALYZE	Atualiza estatísticas usadas pelo planejador para determinar a o modo mais eficiente para executar uma consulta.

Exemplo:

```
VACUUM ANALYZE VERBOSE colaboradores ;
```

43 - ANALYZE - Coleta de Estatísticas Sobre uma Base de Dados

O comando ANALYSE coleta estatísticas sobre o conteúdo de tabelas no banco de dadoes e guarda os resultados na tabela de sistema "pg_statistic". Subsequencianmente, o planejador de consultas usa essas estatísticas para ajudar a determinar o plano de execução mais eficiente para "queries".

Sem parâmetros, ANALYSE examina todas tabelas no banco de dados corrente. Com um parâmetro, examina apenas a tabela determinada.

Com o parâmetro VERBOSE, habilita a exibição de mensagens de progressso.

Sintaxe:

```
ANALYZE [ VERBOSE ] [ tabela [ ( coluna [, ...] ) ] ]
```

Exemplo:

```
ANALYZE VERBOSE colaboradores;
```

Obs.: Na configuração padrão do PostgreSQL, o daemon de "auto-vacuum" cuida da análise automática de tabelas quando são carregadas pela primeira vez com dados e como elas mudam toda operação regular. Quando "autovacuum" é desabilitado, é uma boa prática executar ANALYZE periodicamente, ou apenas após fazer grandes mudanças no conteúdo da tabela. Estatísticas ajudam o planejador a escolher o mais apropriado plano de consultas e assim melhorar sua velocidade de processamento. Uma estratégia comum é rodar o ANALYZE uma vez ao dia durante um período de pouco uso.

44 - EXPLAIN

Esse comando exibe o plano de execução que o planejador PostgreSQL gera para cada instrução. O plano de execução exibe como a(s) tabela(s) referenciada(s) por uma instrução será examinada por um exame sequencial plano, exame de índices, etc. Se múltiplas tabelas forem referenciadas, quais algoritmos de junção serão usados para unir linhas necessárias de cada tabela envolvida.

Sintaxe:

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] instrução
```

Parâmetros:

ANALIZE	Executa o comando e mostra o tempo real de execução
VERBOSE	Mostra a representação interna completa da árvore de planos, ao invés de apenas um resumo. Usualmente essa opção é útil apenas para propósitos de "debug" especializados.
instrução	Qualquer instrução SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE ou DECLARE, cujo plano de execução deeja visualizar.

Exemplo:

```
EXPLAIN ANALYZE SELECT * FROM colaboradores;
```

45 - Backup & Restore

Como tudo que tem dados, bancos de dados PostgreSQL devem ser "backupeados" regularmente. Fundalmentalmente, há três diferentes tipos de técnicas para se fazer backup do PostgreSQL:

- SQL dump: on line, com pg_dump ou pg_dumpall;
- Em nível de sistema de arquivos: off line, com tar, cp, gzip, bzip2, etc;
- Arquivamento contínuo: online, utilizando Point In Time Recovery.

Backups on line não param o serviço do banco, mas reduz a performance. Já os backups off line necessitam que o serviço do banco esteja parado.
Cada um tem seus pontos positivos e negativos, que serão discutidos.
É altamente recomendável efetuar backups em períodos que a utilização seja a menor possível.

45.1 - Backup SQL

45.1.1 - pg_dump

A idéia por trás deste método de "dump" é gerar um arquivo de texto com comandos SQL, que quando restaurado recriará o banco de dados no mesmo estado com era quando o backup foi efetudado. O PostgreSQL disponibiliza um utilitário, para ser usado no shell, o bpg_dump. O pg_dump é uma aplicação cliente, que permite fazer backups remotamente. E seu uso básico é:

```
pg_dump nome_do_banco > arquivo_de_saida
```

Por padrão, o pg_dump faz backups em formato texto. Há a possibilidade de se fazer backups em formato binário e comprimido da seguinte forma:

```
pg_dump -Fc -f arquivo.dump nome_do_banco
```

45.1.1.1 - Restaurando do pg_dump e o Utilitário pg_restore

Os arquivos de texto criados pelo pg_dump podem ser interpretados pelo utilitário psql. A forma geral de comando para restaurar é:

```
psql nome_do_banco < arquivo_de_entrada
```

Onde "arquivo_de_entrada" é o arquivo de saída usado pelo comando pg_dump.

Obs.: Ele não recria o banco de dados com o comando `CREATE DATABASE ...`, caso o banco não exista mais e for preciso restaurar através do arquivo gerado pelo pg_dump, antes pode recriar o banco com o utilitário shell, cuja função equivale ao comando SQL `CREATE DATABASE`:

```
createdb nome_do_banco
```

No entanto o método acima é válido para “dumps” feitos em modo texto. Para restaurar arquivos binários compactados pelo pg_dump utiliza-se o programa pg_restore:

```
pg_restore -Fc arquivo.dump -d nome_do_banco
```

45.1.2 - pg_dumpall

O utilitário pg_dump faz cópia de segurança de um único banco de dados por vez, e não guarda informações sobre roles ou tablespaces, pois são objetos que não dependem de um banco. Para se fazer backups de todo conteúdo de um cluster de um cluster, o utilitário pg_dumpall é fornecido. O utilitário pg_dumpall efetua backups de cada base de dados que um cluster contém e também preserva dados mais abrangentes como roles e definições de tablespaces. Não faz backup de BLOBs. Seu uso básico é:

```
pg_dumpall > arquivo
```

O backup resultante pode ser restaurado pelo utilitário psql da seguinte forma:

```
psql -f arquivo postgres
```

(Atualmente, pode ser especificado algum nome de base de dados existente para iniciar o cluster dela, mas se está recarregando as informações em um cluster vazio, então o nome "postgres" deve ser usado).

45.2 - Backups para Bancos de Dados Grandes

Há a possibilidade de um arquivo de backup ser maior do que o máximo tamanho permitido para o sistema de arquivos (file system) utilizado. Levando-se em conta que o utilitário pg_dump pode imprimir sua saída para a tela, é possível usar ferramentas padrão Unix para contornar esse possível problema. Aplicando o uso do pipe "|" e utilitários com compressão mais poderosa do que o formato binário comprimido do pg_dump.

Exemplo:

- Comprimindo

```
pg_dump curso | bzip2 -9 > banco.bz2
```

No comando acima, o pg_dump enviaria sua saída para a tela, mas o pipe redireciona para o bzip2 (utilitário de compressão), que gera o arquivo "banco.bz2".

- Restaurando

```
bzip2 -d -c banco.bz2 | psql curso
```

O arquivo é descompactado e mandaria sua saída para a tela, mas novamente o pipe redireciona, desta vez para o psql que especifica para qual base de dados.

Quando só a compressão não resolve podemos dividir o arquivo resultante em vários pedaços com o comando split:

```
pg_dump nome_do_banco | split -b 10m - arquivo
```

No comando acima, o pipe repassa o que seria exibido em tela para o comando split que dividirá o backup em arquivos de no máximo 10MB

A restauração é feita da seguinte maneira:

```
cat arquivo* | psql nome_do_banco
```

45.3 - Backup em Nível de Sistema de Arquivos

Uma estratégia alternativa de backup é copiar diretamente os arquivos que o PostgresSQL usa para armazenar os dados da base.

Para isso deve-se saber onde estão tais arquivos. No caso da instalação via apt-get do postgresql desta versão está em "/var/lib/postgresql/8.4/main". Localização obtida no arquivo postgresql.conf, na configuração de nome "data_directory".

Exemplo usando o utilitário tar com compressão bzip2:

```
tar -cvjf cluster.tar.bz2 /var/lib/postgresql/8.4/main/ --exclude=pg_xlog/*
```

45.4 - Arquivamento Contínuo

45.4.1 - WAL (Write-Ahead-Log: Registro Prévio de Escrita)

Write-Ahead Loggin (WAL) é o método padrão para garantir integridade de dados.

O conceito central do WAL é que as mudanças nos arquivos de dados (onde tabelas e índices estão) devem ser gravadas somente depois que tenham sido escritas em log, ou seja, as alterações serão efetivadas fisicamente (nos arquivos de dados).

Os arquivos de log de transação são conhecidos como “xlog” e estão armazenados em em subdiretório do diretório de dados do postgresql (“data_directory” no arquivo de configuração postgresql.conf ou a variável de ambiente “PGDATA”). Então o caminho para os logs de transação em sistemas Linux/Unix é “PGDATA/pg_xlog” ou em sistemas Windows: “%PGDATA%\pg_xlog”, isso se, as variáveis de ambientes foram devidamente configuradas.

Os nomes dos arquivos de log são números hexadecimais, sendo que os oito primeiros dígitos são referentes à sua cronologia. Exemplo: **00000003**0000000000000012.

Em caso de “crash” (falha) de banco de dados é possível recuperar as transações registradas, sendo que para aplicar usa-se “rolling forward” e para desfazer “rolling back”.

O WAL faz com que não seja preciso sincronizar a área de cache em memória da base de dados com os arquivos físicos a cada efetivação de uma transação.

No arquivo “postgresql.conf”, os parâmetros de configuração do WAL são:

Parâmetro	Padrão	Descrição
• **Definições**		
fsync (boolean)	on	Ativado (on), o servidor PostgreSQL tentará checar se as atualizações estão fisicamente escritas em disco, mediante chamadas de sistema fsync() ou vários métodos equivalentes (veja wal_sync_method). Isso assegura que a base de dados pode ser recuperada para um estado consistente após um crash de hardware ou de sistema operacional. No entanto, o uso de fsync resulta em perda de performance: quando uma transação é efetivada, o PostgreSQL deve esperar pelo sistema operacional para liberar o WAL para o disco. Quando o fsync está desabilitado (off), permite ao sistema operacional fazer melhores "buffering", ordenação e temporização de escrita. Resulta em uma insignificante melhora de performance, mas se houver um crash de sistema, os resultados das poucas últimas transações efetivadas serão perdidas parcialmente ou completamente.
synchronous_commit (boolean)	on	Define se a efetivação da transação irá esperar por registros WAL serem escritos no disco antes do comando retornar uma indicação de sucesso para o cliente. Se for desativado (off), poderá haver um atraso durante o momento de sucesso que é reportado ao cliente e quando a transação é realmente garantida para ser segura contra um crash de servidor.
wal_sync_method (string)	fsync	Método usado para forçar atualizações do WAL para o disco. Se fsync estiver desativado esta configuração é irrelevante, já que as atualizações não serão todas forçadas a sair. Os possíveis valores são: • open_datasync (grava arquivos WAL com a opção open() O_DSYNC); • fdatasync (chama fdatasync() a cada efetivação); • fsync_writethrough (chama fsync() a cada efetivação, forçando write-through de qualquer cache de disco); • fsync (chama fsync() a cada efetivação);

		• open_sync (grava arquivos WAL com a opção open() O_SYNC). Obs.: Nem todas as opções estão disponíveis para todas as plataformas.
full_page_writes (boolean)	on	Ativado este parâmetro faz com que o servidor PostgreSQL escreva o conteúdo inteiro de cada página de disco para o WAL durante a primeira modificação de página depois de um checkpoint.
wal_buffers	64kB	O total de memória usada na memória compartilhada (shared memory) para dados do WAL. Aumentar esse valor pode significar maior desempenho para bases de dados que têm transações grandes, no entanto, faz com que o PostgreSQL requeira mais memória compartilhada do que as configurações padrão do sistema operacional oferecem.
wal_writer_delay (integer)	200ms	Define o tempo entre ciclos de atividade para o escritor do WAL.
commit_delay (integer)	0	Tempo entre a escrita de uma efetivação para o buffer do WAL e liberação do buffer para o disco em microsegundos. Configurando com um valor diferente de zero pode permitir múltiplas transações para serem efetivadas com apenas uma chamada de sistema fsync(), se a carga de sistema é alta o suficiente de modo que transações adicionais fiquem prontas para efetivar dentro do intervalo especificado.
commit_siblings (integer)	5	Número mímino de transações simultâneas abertas antes da requisição de tempo de commit_delay. Um valor maior torna possível pelo menos uma ou outra transação fique pronta para efetivar durante o intervalo de tempo.
• **Checkpoints**		
checkpoint_segments (integer)	3	Número máximo de segmentos de arquivos de log entre checkpoints do WAL (cada segmento tem 16 MB). Aumentando esse parâmetro pode aumentar também o tempo necessário para um "crash recovery".
checkpoint_timeout (integer)	5min	Intervalo entre checkpoints automáticos do WAL (por padrao em segundos). Varia de 30s a 1h.
checkpoint_completion_target (floating point)	0.5	Especifica a duração do alvo de checkpoints, como uma fração de intervalo de verificação.
checkpoint_warning (integer)	30s	Escreve uma mensagem no log do servidor se verificações causadas pelo preenchimento de arquivos de checkpoint acontecer mais próximo do que este número de segundos. Zero desabilita a mensagem.
• **Arquivamento**		
archive_mode (boolean)	off	Quando habilitado, segmentos WAL completados são copiados para locais determinados (ver archive_command).
archive_command (string)		Comando utilizado para arquivar um segmento do WAL. Na string podem ser usados os seguintes artifícios: %p = O caminho do arquivo WAL a ser arquivado %f = Nome do arquivo WAL É importante que o comando retorne zero em caso de sucesso. Exemplos: archive_command = 'cp "%p" /caminho/do/diretorio/de/arquivamento/"%f"' archive_command = 'copy "%p" "C:\\caminho\\do\\diretorio\\de\\arquivamento\%f"' # Windows
archive_timeout (integer)	0	Tempo máximo em segundos entre arquivamentos do WAL, ou seja, determina o intervalo de reciclagem do arquivo de log, forçando o arquivamento. Zero desabilita. Caso tiver um valor muito baixo comprometerá a performance do banco de dados.

45.4.1.1 - Benefícios do WAL

- Redução significativa do número de escritas em disco, pois quando uma transação é efetivada só é preciso descarregar o arquivo de registro ao invés de todos os arquivos de dados modificados pela transação;
- Consistência das páginas de dados, pois sem o WAL não é possível garantir consistência em caso de queda de energia. E qualquer queda durante a escrita poderia resultar em:
    1. Linhas de índice apontando para linhas que não existem;
    2. Perda de linhas de índice nas operações de quebra de página (split);
    3. Tabelas ou índices corrompidos devidos a dados parcialmente gravados.
- Alta disponibilidade com a cópia dos arquivos de log para outra máquina: “Log shipping”.

45.4.1.2 - PITR - Point In Time Recovery

Com o PITR é possível restaurar o estado do cluster de um tempo passado.

Durante todo o tempo, o PostgreSQL mantém o WAL no subdiretório px_log do diretório do cluster, o qual contém cada mudança para os arquivos da base de dados. Esse log existe principalmente contra falhas de sistema, pois se isso acontecer, o banco pode ser restaurado consistentemente refazendo as entradas efetuadas desde o último checkpoint. Mas, a existência de logs torna possível outra estratégia para se fazer backup de bancos de dados, pode-se combinar backup em nível de sistema de arquivos com backups dos arquivos do WAL. Se houver necessidade de recuperação, restauram-se o backup, para o momento atual.

Obs.: Índices que não sejam B-Tree não estão presentes no WAL, se o banco em questão tiver índices diferentes desse tipo deve ser reindexado, utilizando o comando `REINDEX`, após o término do PITR.

45.4.1.2.1 - Configuração do Arquivamento

Primeiramente deve ser criado ou determinado o diretório de backup para onde irão os arquivos de log. No exemplo será usado `/srv/wal_backup`:

```
mkdir -p /srv/wal_backup              # Criação do diretório
chgrp -R postgres /srv/wal_backup     # Definindo o grupo de usuários do diretório
chmod -R 775 /srv/wal_backup          # Permissão a membros do grupo para escrita
```

No arquivo postgresql.conf fazer as seguintes modificações:

- archive_mode = on
  Habilita o arquivamento;
- achive_command = 'cp %p /srv/wal_backup/%f'

Após as modificações serem feitas é preciso reiniciar o serviço do PostgreSQL para reconhecer essas novas configurações:

```
/etc/init.d/postgresql restart
```

Obs.: Se o diretório px_xlog for perdido, os dados dos arquivos que não foram arquivados serão perdidos.

45.4.1.2.2 - Fazendo um Backup da Base

O procedimento para fazer o backup base é relativamente simples:

1. Verifique se o WAL archiving está abilitado;

2. Conecte a um banco de dados como super usuário e dê o comando:

```
SELECT pg_start_backup('rotulo');
```

Onde "rotulo" é qualquer string escolhida para identificar unicamente essa operação de backup (uma boa prática é usar o caminho inteiro que se pretende destinar o arquivo de backup).

A função pg_start_backup cria um arquivo de rótulo, cujo nome é "backup_label", no diretório do cluster como informações sobre o backup ou então cria um arquivo de extensão ".backup" no subdiretório "pg_xlog".

Não importa em qual banco de dados está conectado para dar esse comando.

A função pg_start_backup pode levar muito tempo para finalizar, devido ao fato de ele realizar um checkpoint, e as requisições de I/O (input/output = entrada/saída) para um checkpoint serão distribuídas ao longo de um período de tempo significativo, por padrão metade do intervalo de inter-checkpoint (ver configuração "checkpoint_completion_target"). Normalmente isso é o desejado porque minimaliza o impacto no processamento de queries.

Se desejar inicializar o backup assim que possível, execute o comando CHECKPOINT (que realiza um checkpoint o mais rápido possível) e em seguida, imediatamente execute pg_start_backup. A função pg_start_backup terá pouco para fazer, pois não vai demorar.

3. Realize o backup em nível de sistema de arquivo no cluster usando qualquer ferramenta conveniente como tar ou cpio. Não é necessário e nem desejável parar o banco de dados enquanto se faz isso.
   Por exemplo, dê o comando no shell do sistema operacional:

```
tar -cvjf cluster.tar.bz2 /var/lib/postgresql/8.4/main/ --exclude=pg_xlog/*
```

Obs.: O comando acima criará o backup no diretório corrente, para saber em qual diretório está dê o comando "pwd" no shell.

4. Após o backup em nível de sistema de arquivos conecte a um banco de dados dê o comando:

```
SELECT pg_stop_backup();
```

Isso terminará o modo de backup e realizará uma mudança automática para o próximo segmento WAL e também gerará um histórico de backup na área de arquivamento.

O intervalo entre pg_start_backup e pg_stop_backup pode ser demorado.

45.4.1.2.3 - Recuperação PITR

O pior aconteceu!!! É preciso recuperar o backup!

Procedimento:

1. Pare o serviço, se ele estiver rodando;

   ```
   /etc/init.d/postgresql stop
   ```

   ou

   ```
   pg_ctl stop
   ```

2. Se houver espaço para fazer, copie todo o diretório de cluster e qualquer tablespace para uma localidade temporária, pois pode precisar depois. Caso não houver espaço, no mínimo copiar o conteúdo do diretório pg_xlog, pois pode ter logs que não foram arquivados antes do sistema cair;

   ```
   cp -a $PGDATA/pg_xlog/* /srv/wal_backup
   ```

3. Apague todos arquivos e diretórios que estiverem dentro do diretório do cluster e sob a raiz de qualquer tablespace que estiver em uso;

   ```
   rm -fr $PG_DATA
   ```

4. Restaure os arquivos de seu backup. É importante ter assegurar as propriedades corretas de permissão para o usuário do sistema do banco de dados (geralmente o usuário postgres) e não com as permissões de root. Caso usar tablespaces, deve verificar se os links simbólicos em $PGDATA/tblspc/ foram corretamente restaurados;

   ```
   tar -xvjf cluster.tar.bz2
   ```

5. Apague qualquer arquivo presente em $PGDATA/pg_xlog, pois se originam da restauração do backup e provavelmente são obsoletos. Se não tiver o conteúdo de de pg_xlog, ele deve ser recriado, tendo cuidado ao assegurar que foi restabelecido como link simbólico, se foi criado dessa forma antes.

   ```
   mkdir -p $PGDATA/pg_xlog/archive_status
   ```

6. Caso tenha armazenado os arquivos de segmentos WAL (ver passo 2), devem ser copiados em $PGDATA/pg_xlog.
   É melhor copiá-los e não movê-los, pois assim ainda terá os arquivos não modificados se um problema ocorrer, para evitar começar de novo.

   ```
   cp -a  /srv/wal_backup/* $PGDATA/pg_xlog
   ```

7. Criar o arquivo recovery.conf no diretório de cluster. Na instalação padrão há um modelo em "/usr/share/postgresql/8.4/recovery.conf.sample".

   ```
   cp /usr/share/postgresql/8.4/recovery.conf.sample $PGDATA/recovery.conf
   ```

   Obs.: É aconselhável fazer uma modificação temporária no arquivo pg_hba.conf para prevenir que usuários comuns se conectem até tiver certeza que a restauração funcionou.

45.4.1.2.4 - Configuração de Restauração (recovery.conf)

Estas configurações só podem ser feitas no arquivo recovery.conf, e aplicáveis somente durante o processo de recovery (restauração).

- restore_command (string)

  É o comando para executar a restauração de WAL arquivados e única **configuração obrigatória**.

  ```
  restore_command = 'cp /srv/wal_backup/%f "%p"'
  ```

- recovery_end_command (string)

Define um comando que será executado assim que a restauralção acabou. Seu propósito é prover um mecanismo para fazer limpeza de arquivos após uma replicação ou recuperação.

- recovery_target_time (timestamp)

  Determina até que data e hora a recuperação deve ser feita, o padrão é recuperar até o fim dos logs WAL.

- recovery_target_xid (string)

  Indica até que ID de transação a recuperação será feita

- recovery_target_inclusive (boolean)

  Determina se vai parar logo após o alvo de restauração (true) ou pouco antes dele (false).

- recovery_target_timeline (string)

Determina a recuperação de uma determinada cronologia. Por padrão recupera ao longo da cronologia que foi a cronologia atual quando foi feito o backup da base.

45.4.2 - Cronologias – Timelines

A capacidade de restaurar uma base de dados a um ponto anterior no tempo cria algumas complexidades que são semelhantes a histórias de ficção científica sobre viagem no tempo e universos paralelos.

Num caso real de banco de dados, talvez você apagou uma tabela crítica às 17:15 na terça-feira, mas não percebeu o erro até quarta-feira ao meio-dia. Tranqüilamente, você pega seu backup e o restaura para 17:14 de terça-feira. Está rodando! Nessa história de universo de banco de dados, você nunca apagou toda a tabela. Mas suponha que depois você percebeu que após tudo isso não foi uma grande idéia, e gostaria de voltar para algum tempo na manhã de quarta-feira na história original. Você não poderá fazer isso se, enquanto seu banco de dados estiver rodando, ele sobrescreveu algum dos segmentos seriais do WAL que levou até o momento que você deseja agora retornar. Então você realmente quer distinguir as séries de gravações do WAL geradas após terminar o PITR daqueles que foram gerados na história original do banco de dados.

Para lidar com esses problemas o PostgreSQL tem a noção de timelines (cronologias). Sempre que um arquivo de restauração é completado, uma nova timeline é criada para identificar as séries das gravações de WAL geradas após essa restauração. O número timeline ID é parte dos nomes de arquivos de segmentos WAL, e então uma nova timeline não vai sobrescrever dados gerados por timelines anteriores.

É de fato possível armazenar muitas cronologias diferentes. Embora possa parecer um recurso inútil, muitas vezes salva vidas. Considere a situação onde você não tenha certeza que ponto no tempo recuperar, e então tem que fazer muitos PITRs por tentativa e erro até encontrar o melhor lugar para ramificar a partir do histórico antigo. Sem cronologias esse processo logo geraria uma bagunça incontrolável. Com cronologias, você pode recuperar para qualquer estado anterior, incluindo estados de

ramificações de cronologias que você mais tarde abandonou.

Cada vez que uma nova cronologia é criada, o PostgreSQL cria um arquivo de "histórico da linha do tempo" que exibe que cronologia ramifica de onde e quando. Esses arquivos de históricos são necessários para permitir ao sistema pegar os arquivos certos de segmento WAL quando se recupera de um arquivo que contém várias cronologias. Portanto, elas são armazenadas na área de arquivo WAL assim como arquivos de segmentos do WAL.

Os históricos são apenas pequenos arquivos de texto, então não custa muito e é adequado mantê-los indefinidamente (ao contrário dos arquivos de segmento que são muito grandes). Se quiser, você pode adicionar comentários ao arquivo de histórico para fazer suas próprias anotações sobre como e porque esta cronologia foi criada. Tais comentários serão especialmente importantes quando você tem um emaranhado de cronologias diferentes como resultado de experimentação. O comportamento padrão de uma restauração é recuperar ao longo da mesma cronologia que era quando o backup da base de dados foi tomado.

Se desejar recuperar dentro da mesma cronologia filha (isto é, você quer retornar para algum estado que foi-se gerado após uma tentativa de restauração), você precisa especificar o ID da cronologia alvo no arquivo recovery.conf (mais especificamente, no parâmetro “recovery_target_timeline”). Você não poderá recuperar dentro de cronologias que ramificaram mais cedo do que o backup da base.

45.4.2.1 - Recuperação PITR (continuação)

8. Inicie o servidor. O servidor entrará no modo de recuperação e fará a leitura dos arquivos WAL armazenados que precisa. Se a recuperação for terminada por causa de um erro externo, o servidor pode simplesmente ser reiniciado e vai continuar a recuperação. Após a conclusão do processo de recuperação, o servidor renomeará o arquivo recovery.conf para recovery.done (para prevenir entrar no modo de recuperação acidentalmente em caso de falha posterior) e então iniciar as operações normais de banco de dados.

9. Examine o conteúdo do banco de dados para garantir que você tenha recuperado para onde você quer estar. Se não, volte ao passo 1. Se tudo estiver bem, permita conexão dos usuários novamente restaurando o arquivo pg_hba.conf ao normal. A parte fundamental de tudo isso é configurar o arquivo de comando de recuperação que descreve como você quer restaurar e até quando o recovery deve rodar.

46 - Configurações de Memória e Ajustes de Desempenho

Muitas pessoas erroneamente pensam que o PostgreSQL é um SGBD muito lento. No entanto, o que acontece é que por questões de compatibilidade com configurações de hardware modestas, os ajustes padrões são também modestos. Para configurações de hardware mais potentes podem ser feitas modificações que aumentam o desempenho consideravelmente. Essas modificações, no postgresql.conf são feitas seguindo o modelo parâmetro = valor, sendo que para valores numéricos não se usa separador de milhar e para valores não numéricos (texto/string) o valor deve ficar entre aspas simples: parâmetro='valor'.

Antes de qualquer modificação, vamos criar uma tabela de exemplo e populá-la significativamente:

Criação da tabela:

```
CREATE TABLE tb_tuning_teste(
       codigo serial primary key,
       valor1 int,
       valor2 int,
       valor3 int,
       texto varchar(50)
);
```

No shell digite:

```
for i in $(seq 1 10000);
do psql -d curso -c \
"INSERT INTO tb_tuning_teste (valor1,valor2,valor3,texto) VALUES \
($i*2,$i*3,$i*4,now()||' '||'$i'),($i*2,$i*3,$i*4,now()||' '||'$i'),\
($i*2,$i*3,$i*4,now()||' '||'$i'),($i*2,$i*3,$i*4,now()||' '||'$i'),\
($i*2,$i*3,$i*4,now()||' '||'$i'),($i*2,$i*3,$i*4,now()||' '||'$i'),\
($i*2,$i*3,$i*4,now()||' '||'$i'),($i*2,$i*3,$i*4,now()||' '||'$i');"
done
```

O comando acima gerará 100.000 registros na tabela criada. Essa operação demorará alguns minutos:

No psql digite:

```
EXPLAIN ANALYZE SELECT  * from tb_tuning_teste WHERE (codigo%2=0) ORDER BY valor3 DESC;
```

Repita cinco vezes e veja os resultados. Serão exibidos como segue:

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=2502.71..2503.96 rows=500 width=50) (actual time=191.541..248.919 rows=50000 loops=1)
   Sort Key: valor3
   Sort Method:  external merge  Disk: 2952kB
   ->  Seq Scan on tb_tuning_teste  (cost=0.00..2480.30 rows=500 width=50) (actual
time=0.023..77.225 rows=50000 loops=1)
         Filter: ((codigo % 2) = 0)
 Total runtime: 281.811 ms
(6 rows)
```

46.1 - Shared Memory

Também conhecida como **memória compartilhada**, é uma quantidade de memória disponibilizada pelo sistema operacional para fins de comunicação entre processos (IPC: Inter-Process Comunication).

A configuração da memória compartilhada é feita através de parâmetros do kernel:

Parâmetro	Descrição	Sugestão de Valor
SHMMAX	Quantidade máxima (em bytes) de um segmento de memória compartilhada	250kB + (8.2kB * shared_buffers) + (14.2kB * max_connections)
SHMALL	Total de memória compartilhada disponível (em páginas de 4KB)	O valor equivalente ao de SHMMAX = SHMMAX / 4096

46.1.1 - Visualizando os Valores de Memória Compartilhada

Respectivamente, para cada um dos parâmetros do kernel citados, no shell dê os comandos:

```
cat /proc/sys/kernel/shmmax
cat /proc/sys/kernel/shmall
```

46.1.2 - Modificando os Valores de Memória Compartilhada

Há três maneiras para fazer a modificação, ambas como usuário root do sistema:

1. A forma não efetiva, ao reiniciar a máquina a configuração voltará ao que era antes:

```
echo 322838528 > /proc/sys/kernel/shmmax
echo 78818 > /proc/sys/kernel/shmall
```

2. Forma definitiva por edição, abrindo o arquivo /etc/sysctl.conf e então altere ou adicione as seguintes linhas referentes ao parâmetros kernel.shmmax e kernel.shmall:

```
kernel.shmmax =  322838528
kernel.shmall =  78818
```

Salve, saia e dê o comando no prompt:

```
sysctl -p
```

3. A forma definitiva por comando, sem precisar editar arquivo:

```
sysctl -w kernel.shmmax=322838528
sysctl -w kernel.shmall=78818
```

O utilitário sysctl é usado para modificar parâmetros do kernel em tempo de execução.

46.2 - Configurações de Consumo de Recursos no postgresql.conf

46.2.1 - Gerenciamento de Trava – Lock Management

46.2.1.1 - deadlock_timeout (integer) → Padrão: 1s

Vamos assumir de uma forma otimista que dead locks não são comuns em aplicações de produção e apenas espera por uma trava enquanto antes de começar a buscar por um dead lock. Aumentando este valor reduz a quantidade de tempo gasto em checagens de dead lock sem necessidade.

O valor padrão de um segundo (1s) é provavelmente o menor valor que vai querer na prática. Em um servidor muito carregado você pode querer aumentar esse valor. Idealmente o ajuste deve exceder o tempo típico de transação, de modo a melhorar as chances que uma trava será liberada antes da decisão do sistema para checar por dead locks.

Quando o parâmetro log_lock_waits está ajustado, determina também o tempo de espera antes que uma mensagem de log é enviada sobre a espera de lock (trava). Se você está investigando atrasos de locking você deve querer ajustar um valor menor para log_check_waits do que um deadlock_timeout normal.

46.2.1.2 - max_locks_per_transaction (integer) → Padrão: 64

Controla o número médio de travas alocadas para cada transação; transações individuais podem travar mais objetos enquanto como as travas de todas transações se ajustam na tabela de trava (lock table). Este não é o número de linhas que devem ser travadas, esse é ilimitado.

O valor padrão (64), tem historicamente tem se comprovado suficiente, mas você deve precisar aumentar se tem clientes que acionam tabelas diferentes em uma única transação. Ao aumentar esse parâmetro pode fazer com que o PostgreSQL requira mais memória compartilhada do que sua configuração padrão permite.

46.2.2 - Memória

46.2.2.1 - shared_buffers (integer) → Padrão: 24MB

Um dos mais importantes parâmetros que influencia diretamente no desempenho do sistema de banco de dados. Determina a quantidade de memória (buffer) da memória compartilhada que o PostgreSQL usa. Esse parâmetro é expresso escrevendo o valor diretamente com o sufixo de medida (KB,MB,..) ou um número puro que representa a quantidade de blocos de 8 KB.

Exemplos equivalentes:

```
shared_buffers = 24MB          # Valor em MB
```

ou

```
shared_buffers = 3072                  # 3072 páginas de 8KB → (Qtd em MB x 1024) / 8
```

E eis então a questão:

"*Com qual valor ajusto shared_buffers?"*

A resposta mais adequada é:

"*A quantidade adequada para este parâmetro é o maior número possível que não prejudique outras atividades no servidor."*

Para reajustar esse parâmetro é preciso também reajustar os valores de shmmax e shmall do

kernel.

Primeiro se define a quantidade desejada para o parâmetro shared_buffers, e então a quantidade para a memória compartilhada (shmmax) será determinada pela seguinte fórmula:

SHMMAX = 250kB + (8.2kB * shared_buffers) + (14.2kB * max_connections)

Exemplo:

Supondo que a máquina que o servidor Postgres esteja tenha apenas 1 GB (1024 MB), e 300 MB seja uma quantidade segura de memória para destinar ao parâmetro shared buffers e o parâmetro max_connections seja igual a 10:

shared_buffers = (300x1024) / 8 → shared_buffers = 38400

então:

SHMMAX = 250kB + (8.2kB *38400)+(14.2kB * 10) = 315272 kB

Valor em bytes: SHMMAX = 315272 x 1024 = 322838528

Precisamos agora do valor para SHMALL:

SHMALL = SHMMAX / 4096 = 322838528 / 4096 = 78818

Ok. Agora temos os valores para shmmax e shmall. É preciso fazer com que o kernel reconheça esses valores:

```
sysctl -w kernel.shmmax=322838528
sysctl -w kernel.shmall=78818
```

E por fim, após todos os cáculos, no arquivo postgresql.conf, vamos definir finalmente, a quantidade de 300MB para o parâmetro shared buffers:

shared_buffers = 38400

ou

shared_buffers = 300MB

46.2.2.2 - temp_buffers (integer) → Padrão: 8MB

Configura a quantidade máxima de buffers temporários usados em cada sessão de banco de dados. Esses são buffers de sessão local **apenas usados para tabelas temporárias**. A configuração pode ser alterada com sessões individuais, mas apenas até o primeiro uso de tabelas temporárias dentro de uma sessão; tentativas subsequentes para mudar o valor não terá efeito nessa sessão.

Uma sessão sessão alocará buffers temporários conforme o necessário até o limite dado de temp_buffers. O custo de configurar um valor muito alto em sessões que realmente não precisam de muitos buffers temporários é apenas um descritor de buffer, ou cerca de 64 bytes por incremento em temp_buffers. Porém, se um buffer é realmente usado um adicional de 8192 bytes (8KB) será consumido por ele.

Exemplo:

temp_buffers = 8MB

46.2.2.3 - max_prepared_transactions (integer) → Padrão: 0

Configura o máximo número de transações que podem estar em um estado “preparadas” simultaneamente (PREPARE TRANSACTION do manual do PostgreSQL). Ajustando este parâmetro para zero desabilita a característica de transação preparada (prepared transaction).

Se não planeja usar esse recurso, seu valor deve ser zero para prevenir criação acidental de transações preparadas. Caso estiver usando transações preparadas, provavelmente gostaria que max_prepared_transactions seja no mínimo o mesmo valor de max_connections, de modo que cada sessão pode ter uma transação preparada pendente.

Aumentanto max_prepared_transactions custa aproximadamente 600 bytes de de memória compartilhada por slot de transação, mais lock space (veja max_locks_per_transaction).

46.2.2.4 - work_mem → Padrão 1MB

Determina a quantidade de buffers de memória compartilhada utilizada pelo Postgres. Tem efeito em operações de ordenação para manipulação e/ou consulta de índices. Especifica a quantidade de memória usada por operações internas de classificação e tabelas hash antes de alternar para arquivos temporários em disco. Para cada consulta complexa, muitas consultas e operações de hash devem rodar paralelamente; cada uma será permitida para usar tanta memória quanto este valor especifica antes dele começar a inserir os dados em arquivos temporários. Além disso, várias sessões em execução poderiam fazer tais operações simultaneamente. Então o total de memória usado pode ser muitas vezes o valor de work_mem; é necessário para manter esse fato em mente quando escolher o valor. operações de classificação são usadas por ORDER BY, DISTINCT e JOINs. Tabelas hash são usadas em JOINs de hash, agregação baseada em hash e hashes baseados em processamento de subconsultas IN.

Vamos alterar para 30MB o valor:

```
work_mem = 30MB
```

46.2.2.5 - maintenance_work_mem (integer) → Padrão: 16MB

Especifica a quantidade máxima de memória para ser usada em operações de manutenção, como VACUUM, CREATE INDEX, e ALTER TABLE ADD FOREIGN KEY. Desde que apenas uma dessas operações possa ser executada em um momento por uma sessão, e uma instalação normalmente não tenha muitas delas rodando atualmente, é seguro configurar este valor significantemente maior do que work_mem. Maiores configurações devem melhorar a performance para VACUUM e para restaurar backups. Note que enquanto o autovacuum roda, até autovacuum_max_workers essa memória deve der alocada, então tenha cuidado para não configurar o valor padrão tão alto.

Exemplo:

```
maintenance_work_mem = 100MB
```

46.2.2.6 - max_stack_depth (integer) → Padrão: 2MB

Determina a profundidade máxima segura de pilhas em execução. A configuração ideal para este parâmetro é o atual limite de tamanho de pilha executada pelo kernel menos uma margem segura de 1MB. A margem de segurança é necessária porque a profundidade de pilha não é checada em todas rotinas no servidor, mas apenas nas principais rotinas potencialmente recursivas como avaliação de expressão. A configuração padrão é um valor seguro contra falhas, mas muito baixo para permitir execução de funções complexas. Apenas super usuários podem mudar essa configuração. Definir max_stack_depth com um valor mais alto do que o atual limite do kernel significa que uma função recursiva que escapar pode quebrar um processo individual de backend. Em plataformas onde o Postgres pode determinar o limite do kernel, não vai deixar definir essa variável para um valor não seguro. Porém, nem todas as plataformas fornecem a

informação, então é recomendado ter cautela ao configurar esse valor.

Exemplo:

max_stack_depth = 5MB

46.2.3 - Uso de Recursos do Kernel

46.2.3.1 - max_files_per_process (integer) → Padrão: 1000

Configura o número máximo permitido de arquivos abertos simultaneamente para cada subprocesso do servidor. Se o kernel impor um limite seguro por processo, não precisa se preocupar com esta configuração. Mas em algumas plataformas (a maioria dos sistemas BSD), o kernel permitirá processos individuais a abrir muito mais arquivos do que o sistema pode realmente suportar quando um grande número de processos de todos os que tentam abrir muitos arquivos. Se aparecer uma mensagem de falha: "Too many open files", tente reduzir esta configuração.

Exemplo:

max_files_per_process = 1000

46.2.3.2 - shared_preload_libraries (string) → Padrão: [em branco]

Esta variável define uma ou mais bibliotecas compartilhadas que devem ser carregadas na inicialização do servidor. Se deve carregar mais de uma biblioteca, separe seus nomes com vírgulas. Por exemplo: '$libdir/mylib' abrirá mylib.so (em alguns sistemas; mylib.sl) para ser carregada do diretório de instalação padrão de bibliotecas.

Bibliotecas de linguagens procedurais podem ser carregadas desse jeito, tipicamente usando a sintaxe '$libdir/plXXX' onde XXX é pgsql, perl, tcl ou python.

Carregando a biblioteca compartilhada, seu tempo de inicialização é evitado quando usada pela primeira vez. No entanto, o tempo para inicializar cada novo processo de servidor deve aumentar ligeiramente, mesmo se esses processos nunca usem essa biblioteca. Então, alterar esse parâmetro é recomendado apenas para bibliotecas que serão usadas na maior parte das sessões.

Se uma biblioteca especificada não for encontrada, o servidor falhará ao inicializar.

Todas bibliotecas suportadas pelo PostgreSQL tem um “bloco mágico” (magic bloc) que é checado para garantir compatibilidade. Por essa razão, bibliotecas não-PostgreSQL não poderão ser carregadas desse jeito.

Exemplo:

shared_preload_libraries =

46.2.4 - Custo Baseado no Tempo de VACUUM

Durante a execução de comandos VACCUM e ANALYZE, o sistema mantém um contador interno que mantém o controle de custo estimado de várias operações I/O (input/output → entrada/saída) que são realizadas. Quando o custo acumulado alcança um limite (especificado por vacuum_cost_delay). Então ele vai zerar o contador e continua a execução.

A intenção dessa característica é permitir administradores reduzir o impacto de I/O desses comandos em atividades de banco de dados simultâneas. Há muitas situações em que não é muito importante que comandos de manutenção como VACUUM e ANALYZE terminem rapidamente; porém, é muito importante que esses comandos normalmente não interfiram significantemente na habilidade que o sistema realiza operações de banco de dados. Custo baseado no tempo de VACUUM (Cost-based vacuum delay) fornece uma maneira para administradores alcançarem isso.

Esta característica é desabilitada por padrão para comandos VACUUM enviados. Para habilitar defina o parâmetro vaccum_cost_delay para um valor diferente de zero.

46.2.4.1 - vacuum_cost_delay (integer) → Padrão: 0ms

Duração do tempo em milisegundos, que o processo aguardará quando o limite de custo (cust limit) for excedido. Valores positivos habilitam esse parâmetro. Em muitos sistemas, a resolução efetiva de hibernação é de 10ms; configurando vacuum_cost_delay para um valor que não é um múltiplo de 10 deve ter os mesmos resultados como os configurados no próximo número maior que 10.

Quando o custo baseado em vácuo (cost-based vacuuming) é usado de forma apropriada para vacuum_cost_delay são normalmente valores muito pequenos, talvez 10 ou 20 milisegundos. Ajustando o recurso de consumo de vácuo é melhor feito ajustando outros parâmetros de custo de vácuo.

Exemplo:

```
vacuum_cost_delay = 0ms
```

46.2.4.2 - vacuum_cost_page_hit (integer) → Padrão: 1

O custo estimado para vácuo de um buffer encontrado no cache de shared buffer. Ele representa o custo para travar as requisições de buffer, pesquisa a tabela de hash compartilhada e vasculha o conteúdo da página.

Exemplo:

```
vacuum_cost_page_hit = 1
```

46.2.4.3 - vacuum_cost_page_miss (integer) → Padrão: 10

O custo estimado para vácuo de um buffer que tem que ser lido do disco. Isso representa o esforço para travar as requisições de buffer, pesquisa a tabela de hash compartilhada, lê o bloco desejado do disco e vasculha seu conteúdo.

Exemplo:

```
vacuum_cost_page_miss = 10
```

46.2.4.4 - vacuum_cost_page_dirty (integer) → Padrão: 20

O custo estimado cobrado quando o vácuo modifica um bloco que foi previamente limpo. Ele representa o I/O requirido para liberar o bloco limpo no disco novamente.

Exemplo:

```
vacuum_cost_page_dirty = 20
```

46.2.4.5 - vacuum_cost_limit (integer) → Padrão: 200

O custo acumulado que causará a hibernação do processo de vácuo.
Há determinadas operações que mantêm bloqueios críticos e deve então completar o mais rápido possível. Custos baseados no tempo de vácuo não ocorrem durante durante tais operações. Porém, é possível que o custo acumule muito mais do que o limite especificado. Para evitar longos atrasos nesses casos, o atual (delay) é caulculado por vacuum_cost_delay * accumulated_balance / vaccum_cost_limit com um máximo de vaccum_cost_delay * 4.

Exemplo:

```
vacuum_cost_limit = 200
```

46.2.5 - Background Writer

Há um processo separado chamado background writer, cuja função é enviar escritas de shared buffers sujos (dirty). A intenção é que o processo do servidor manipule consultas de usuários para que raramente ou nunca tenha que esperar para que uma escrita ocorra, porque o background writer o fará. No entanto, há um aumento significativo de carga de I/O, porque uma página repetidamente suja deve ser escrita apenas uma vez por intervalo de checkpoint, mas o background writer deve gravá-lo muitas vezes no mesmo intervalo. Os parâmetros discutidos em seguida podem ser usados para melhorar o desempenho do comportamento para necessidades locais.

46.2.5.1 - bgwriter_delay (integer) → Padrão: 200ms

Especifica o tempo entre rodadas de atividade para o background writer. Em cada round (rodada) o writer escreve para um determinado número de buffers sujos (controlável pelos parâmetros seguintes). Em seguida ele espera pelo tempo do valor de bgwriter e repete. Em muitos sistemas, a resolução efetiva de tempo de hibernação é 10ms; configurando bgwriter_delay para um valor que não é um múltiplo de 10 deve ter os mesmos resultados com configurá-lo para o próximo múltiplo de 10.

Exemplo:

```
bgwriter_delay = 200ms
```

46.2.5.2 - bgwriter_lru_maxpages (integer) → Padrão: 100

Em cada rodada, não mais do que isso muitos buffers serão escritos pelo background writer. Configurando isso para zero desabilita background writing (exceto para atividades de checkpoint).

Exemplo:

```
bgwriter_lru_maxpages = 100
```

46.2.5.3 - bgwriter_lru_multiplier (floating point) → Padrão: 2.0

O número de buffers sujos escritos em cada rodada é baseado no número de novos buffers que tem sido necessários por processos do servidor durante rodadas recentes. A média recente necessária é multiplicada pelo bgwriter_lru_multiplier para chegar a um número estimado de número de buffers que é preciso durante a próxima rodada. Buffers sujos são escritos até que haja muitos limpos, bufferes disponíveis reutilizáveis. (porém, não mais do que buffers bgwriter_lru_maxpages serão escritos por rodada.) Assim, uma configuração de 1.0 representa uma política “just in time” de escrita exatamente o número de buffers previsto para ser necessário. Valores maiores fornecem amortecimento contra picos de demanda, enquanto valores menores intencionalmente deixa escritas para serem feitas por processos do

servidor.

Valores menores de bgwriter_lru_maxpages e bgwriter_lru_multiplier reduz a carga extra de I/O causada pelo background writer, torna mais provável que que os processos do servidor terá que enviar escritas para si mesmos, atrasando consultas interativas.

Exemplo:

```
bgwriter_lru_multiplier = 2.0
```

46.2.6 - Comportamento Assíncrono

46.2.6.1 - effective_io_concurrency (integer) → Padrão: 1

Configura o número de operações simultâneas de I/O de disco que o PostgreSQL espera serem executadas. Aumentando este valor eleva o número de operações de I/O que qualquer sessão individual do PostgreSQL tenta iniciar em paralelo. A faixa de valores permita é de 1 a 1000, ou zero para desativar solicitações assíncronas de I/O.

Um bom ponto de partida para essa configuração é o número de drives separados, inclusive para RAID. No entanto, se o banco de dados está frequentemente ocupado com múltiplas consultas feitas em sessões concorrentes, valores mais baixos devem ser suficiente para manter o disco de array ocupado. Um valor mais alto do que o necessário para manter os discos ocupados resultarão apenas em overhead extra de CPU.

Exemplo:

```
effective_io_concurrency = 1
```

46.2.7 - Constantes de Custo do Planejador

As variáveis de custo descritas nessa seção são mensuradas em uma escala arbitrária. Apenas seus valores relativos importam, portanto dimensionando todos eles para cima ou para baixo pelo mesmo fator resultará em nenhuma mudança nas escolhas do planejador. Tradicionalmente, essas variáveis tem sido referenciadas para buscas de páginas sequenciais como unidade de custo, isto é, seq_page_cost é convencionalmente ajustada para 1.0 e as outras variáveis de custo são definidas com referência a essa. Mas você pode usar uma escala diferente se preferir, como os tempos de execução em milisegundos em uma máquina específica.

Infelizmente. não há nenhum método bem definido para determinar valores ideais para variáveis de custo. Eles são melhores tratados como médias sobre todas as consultas que uma instalação particular fará. Isso significa que mudando-os com base em algumas poucas experiências é arriscado

46.2.7.1 - seq_page_cost (floating point) → Padrão: 1.0

Ajusta a estimativa do planejador de custo de uma página do disco buscar qual é a parte de uma série de busca sequencial.

Exemplo:

```
seq_page_cost = 1.0
```

46.2.7.2 - random_page_cost (floating point) → Padrão: 4.0

Configura a estimativa do planejador de custo de uma página de disco não-sequencialmente-

buscada. Reduzindo este valor relativo a seq_page_cost fará com que o sistema prefira buscas indexadas, aumentando fará as buscas indexadas parecerem mais custosas. Você pode aumentar ou diminuir ambos os valores juntos para mudar a prioridade de custos de I/O de discos para custos de CPU, que serão descritos pelos parâmetros seguintes.

Dica: Apesar de o sistema permitir definir ramdom_page_cost para menos do que seq_page_cost, ele não é fisicamente sensato fazê-lo. Porém, ajustando-os igualmente faz sentido se o banco de dados estiver totalmente no cache da memória RAM, porque neste casonão haverá penalidade para manejar páginas fora de sequência. Além disso, em um banco de dados pesadamente em cache você deverá abaixar ambos os valores relativos para os parâmetros de CPU, desde que o custo para buscar uma página já em RAM é muito menor do que ele normalmente seria.

Exemplo:

```
random_page_cost = 4.0
```

46.2.7.3 - cpu_tuple_cost (floating point) → Padrão: 0.01

Define a estimativa do planejador de custo de processamento de cada linha, durante uma consulta.

Exemplo:

```
cpu_tuple_cost = 0.01
```

46.2.7.4 - cpu_index_tuple_cost (floating point) → Padrão: 0.005

Define a estimativa do planejador de custo de processamento de cada entrada de índice durante a varredura do índice.

Exemplo:

```
cpu_index_tuple_cost = 0.005
```

46.2.7.5 - cpu_operator_cost (floating point) → Padrão: 0.0025

Define a estimativa do planejador de custo de processamento de cada operador ou função executados durante uma consulta.

Exemplo:

```
cpu_operator_cost = 0.0025
```

46.2.7.6 - effective_cache_size (integer) → Padrão: 128MB

This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes.

Define a suposição do planejador sobre o tamanho efetivo de cache de disco que está disponível para uma única consulta. É um dos elementos de estimativa de custo de uso de um índice; um valor mais alto provalvelmente fará com que varreduras de índices sejam utilizadas, um valor mais baixo provavelmente fará mais buscas sequenciais. Ao configurar este parâmetro deve considerar os shared buffers do PostgreSQL e a porção de memória cache do disco do kernel que será usado para arquivos de dados. Além disso, considerar o número de consultas simultâneas em tabelas diferentes, uma vez que terá de compartilhar o espaço disponível.

Este parâmetro não tem efeito no tamanho da memória compartilhada alocada pelo Postgres, nem faz reserva de cache de disco do kernel, só é usado para fins de estimativa.
Um bom valor sugerido para este parâmetro é 2/3 da memória RAM.

Exemplo:

```
effective_cache_size = 682MB
```

47 - Instalação a Partir do Código Fonte → Compilação

Instalar o Postgres a partir do código fonte permite uma flexibilidade maior para um DBA, podendo incluir recursos. Vejamos algumas das vantagens em fazer a instalação a partir do código fonte:

- Escolher uma versão específica ao invés de apenas a que está no repositório;
- Fazer a compilação de acordo com a arquitetura do processador;
- Dentre várias coisas, pode-se definir em que porta as conexões poderão escutar, diretórios, etc;
- O processo de compilação analisa seu hardware (principalmente processador e memória), fazendo assim com que a instalação aproveite melhor os recursos do servidor;

47.1 - Instalando

1. Instalar programas e bibliotecas necessários para compilação do source do Postgres:

   Como usuário root:

   ```
   apt-get install build-essential libreadline5-dev zlib1g-dev gettext
   ```

2. No seu browser entre no endereço: http://www.postgresql.org/ftp/source/

Ao abrir o site, serão exibidas algumas pastas que representam uma versão, clique sobre a versão desejada e então aparecerá outra tela com os arquivos disponíveis para baixar. Esses arquivos estão nos formatos tar.gz e tar.bz2 e além desses há um arquivo .md5 para cada um, para fazer o checksum.
No nosso exemplo baixaremos apenas o arquivo .bz2, o qual devido à sua compressão ser maior é um arquivo de menor tamanho e consequentemente mais rápido de baixar.
Escolha um mirror (clique), para baixar o arquivo .bz2. De preferência baixe o arquivo no diretório “/tmp”.

3. Com o arquivo baixado precisamos descompactá-lo:

   ```
   tar xvjf postgresql-XXXX.tar.bz2
   ```

   Onde “XXXX” é a versão do software baixado.

4. Vamos criar um usuário de sistema que será o usuário do cluster, seu grupo e também definir o local do cluster, que também será o diretório home do usuário:

   Grupo: dbgroup
   Usuário: dbmaster
   Diretório do Cluster: /db_data
   Diretório de instalação do PostgreSQL: /usr/local/postgresql

   Execute os dois comandos como root:

   ```
   groupadd dbgroup
   useradd -g dbgroup -d /db_data -s /bin/bash -c "Postgres User" dbmaster -m
   ```

Sendo que: -g → grupo do usuário, -d → diretório home do usuário, -c → comentário e -m → força a criação do diretório home.

```
passwd dbmaster # define a senha do usuário dbmaster
```

5. Dê a propriedade do código para o usuário dbmaster e para seu grupo:

   ```
   chown -R dbmaster:dbgroup postgresql-XXXX/
   ```

6. Entre no diretório do source:

```
cd postgresql-XXXX/
```

7. Antes de começarmos a compilar para que o código seja compilado de uma maneira a oferecer um maior desempenho vamos definir algumas variáveis de ambiente:

   Para dividir a compilação em 2 trabalhos (jobs), para CPUs de 2 núcleos:

```
export MAKEOPTS="-j2"
```

   Memória cache para compilação de 500M:

```
export CCACHE_SIZE="500M"
```

   Específicas para sistemas de 32 ou 64 bits:

   Para sistemas de 32 Bits:

```
export CHOST="i686-pc-linux-gnu"
export CFLAGS="-march=native -O3 -m32 -pipe"
export CXXFLAGS="${CFLAGS}"
```

   Para sistemas de 64 Bits:

```
export CHOST="x86_64-pc-linux-gnu"
export CFLAGS="-march=native -O3 -m64 -pipe"
export CXXFLAGS="${CFLAGS}"
```

Obs.: No caso das variáveis específicas (32 ou 64 bits) são de acordo com a versão do seu sistema instalado, ressaltando que sistemas de 64 bits têm desempenho muito superior aos de 32. O suporte a um sistema de 64 bits exige um processador de 64 bits.

Explicação das flags usadas nas variáveis:

-march=cpu-type

Gera instruções para o tipo de processador da máquina.

-native

Produz o código otimizado para a máquina local, auto detectando o modelo de CPU.

-m32
-m64

Gera código, respectivamente, para ambientes de 32 bits ou de 64 bits.

-pipe

Acelera o processo de compilação, mas não há ganhos no tempo de execução.

-O[n] (letra "O" maiúscula)

Otimização. Faz com que compilação leve mais tempo e muito mais memória. Com esta opção o compilador tenta reduzir o tamanho do código e o tempo de execução.
Há três níveis de otimização: "-O1", "-O2" e "-O3", sendo que "-O3" é o nível mais alto.

8. Agora vamos à compilação em si. Dê os comandos:

```
su dbmaster -c "./configure --prefix=/usr/local/postgresql --bindir=/usr/local/bin \
--sbindir=/usr/local/sbin"
```

```
make && make install
```

Como nossas otimizações exigem muito do processador, é hora de tomar um café!

9. Para iniciarmos o postgresql precisamos do script de inicialização:

```
cp /tmp/postgresql-XXXX/contrib/start-scripts/linux /etc/init.d/postgresql
chmod 755 /etc/init.d/postgresql
update-rc.d postgresql start 70 S . stop 21 0 6 .
```

Obs.: Na pasta contrib do código fonte do PostgreSQL há também funções que por padrão não estão no PostgreSQL, como a função dblink.

É preciso editar o arquivo /etc/init.d/postgresql e mude os seguintes valores:

prefix: /usr/local/pgsql → prefix=/usr/local/postgresql
PGDATA: /usr/local/pgsql/data → /db_data/base
PGUSER: postgres → dbmaster
DAEMON="$prefix/bin/postmaster" → "/usr/local/bin/postmaster"
PGCTL="$prefix/bin/pg_ctl" → "/usr/local/bin/pg_ctl"

10. Crie o cluster:

```
su dbmaster -c "/usr/local/bin/initdb -E utf8 -D /db_data/base"
```

11. Como root inicialize o processo do postgres com o script:

```
/etc/init.d/postgresql start
```

12. Opcionalmente pode ser adicionada a seguinte linha, relativa à variável de diretório do cluster:

export PGDATA="/db_data/base"

As variáveis só passarão a valer pro usuário na próxima vez que ele logar, a não ser que ele dê o comando:

```
source ~/.bashrc
```

48 - Instalação do PostgreSQL no Windows

O uso do PostgreSQL na plataforma Windows não é encorajado, principalmente pela performance muito inferior a sistemas operacionais Unix like (Linux, Solaris, BSDs).

Mas para quem não tem nenhum desses sistemas operacionais instalados em seu computador, abaixo estão os procedimentos para a atual versão (8.4.2):

1. Baixe o instalador do seguinte endereço: www.postgresql.org/download/windows

2. Vá até o diretório onde baixou o instalador e dê um duplo clique sobre ele e siga os passos:

- Setup - PostgreSQL → Next
- Installation Directory → C:\Arquivos de programas\PostgreSQL\8.4 → Next
- Data Directory → C:\Arquivos de programas\PostgreSQL\8.4\data → Next
- Password -> Digite e redigite uma senha para o super usuário do Postgres -> Next
- Port → 5432
- Advanced Options → Locale=[Default locale], [ ] Install pl/pgsql in template1 database? → Next
- Read to Install → Next
- Completing the PostgreSQL Setup Wizard → [ ] Launch Stack Builder at exit? → Finish
- Iniciar -> Configurações -> Painel de Controle -> Sistema
- Guia "Avançado" -> Variáveis de Ambiente
- Variáveis do sistema -> Path -> Editar -> Adicione no final da linha: ";C:\Arquivos de programas\PostgreSQL\8.4\bin" (sem as aspas) -> OK -> OK -> OK
- Iniciar -> Executar... -> Arir: cmd -> OK
- psql -U postgres

Extras

Código do Banco de Dados Usado como Exemplo (Copie, cole e salve)

```
	--CRIAÇÃO DO BANCO DE DADOS "curso"
	--CREATE DATABASE curso ENCODING='utf8' LC_COLLATE='pt_BR.UTF-8' LC_CTYPE='pt_BR.UTF-8'
TEMPLATE=template0;
	CREATE DATABASE curso;

	\c curso

	--TABELA COLABORADORES

	CREATE TABLE colaboradores
	(
	  mat_id character(5) NOT NULL,
	  nome character varying(15) NOT NULL,
	  snome character varying(30) NOT NULL,
	  cargo character varying(15),
	  setor character varying(15),
	  uf character(2) DEFAULT 'SP'::bpchar,
	  cidade character varying(15),
	  salario real,
	  dt_admis date,
	  chefe_direto character(5),
	  CONSTRAINT colaboradores_pkey PRIMARY KEY (mat_id),
	  CONSTRAINT chefe_direto_fk  FOREIGN KEY (chefe_direto) REFERENCES colaboradores(mat_id)
	)
	WITH (OIDS=FALSE);
	ALTER TABLE colaboradores OWNER TO postgres;
	COMMENT ON TABLE colaboradores IS 'Funcionários da empresa';

	--TABELA PRODUTOS

	CREATE TABLE produtos
	(
	  prod_id character varying(5) NOT NULL,
	  nome character varying(15) NOT NULL,
	  preco real,
	  qtd integer,
	  descr text,
	  CONSTRAINT produtos_pkey PRIMARY KEY (prod_id)
	)
	WITH (OIDS=FALSE);
	ALTER TABLE produtos OWNER TO postgres;

	--TABELA PEDIDO

	CREATE TABLE pedido
	(
	  ped_id character varying(5) NOT NULL,
	  prod character varying(5),
	  vendedor character varying(5),
	  CONSTRAINT pedido_pkey PRIMARY KEY (ped_id),
	  CONSTRAINT prod_fk FOREIGN KEY (prod)
	      REFERENCES produtos (prod_id) MATCH SIMPLE
	      ON UPDATE NO ACTION ON DELETE NO ACTION,
	  CONSTRAINT vend_fk FOREIGN KEY (vendedor)
	      REFERENCES colaboradores (mat_id) MATCH SIMPLE
	      ON UPDATE NO ACTION ON DELETE NO ACTION
	)
	WITH (OIDS=FALSE);
	ALTER TABLE pedido OWNER TO postgres;

	--TABELA FUNCIONARIOS PREMIADOS

	CREATE TABLE funcionarios_premiados(
	       premio_id varchar(5) primary key,
	       mat_id varchar(15),
	       CONSTRAINT pk_funcionario FOREIGN KEY (mat_id) REFERENCES colaboradores (mat_id)
	);
```

```
	--INSERTS
	--Inserção de dados na tabela colaboradores

	INSERT INTO colaboradores VALUES
	('00001','Chiquinho','da Silva','Diretor','Marketing',DEFAULT,'São Paulo',8500.51,'1999-10-
20','00001'),
	('00002','Almerinda','Santos','Recepcionista','Recepção',DEFAULT,'São Paulo',850.63,'1997-
04-04','00003'),
	('00003','Estrovézio','da Silva','Diretor','Publicidade',DEFAULT,'São Paulo',8500.51,'1997-
05-30','00001'),
	('00004','Wolfrâmio','Leite','Químico','Laboratório 1',DEFAULT,'São Paulo',5400.32,'1999-03-
13','00003'),
	('00005','Estrôncio','Junqueira','Faxineiro','Limpeza',DEFAULT,'São Paulo',600.00,'2000-10-
25','00003'),
	('00006','Azevino','Xavier','Porteiro','Portaria','MG','Belo Horizonte',1020.34,'1997-03-
15','00007'),
	('00007','Sandra','Lima','Diretor','Logística','MG','Belo Horizonte',3200.51,'2003-04-
04','00001'),
	('00008','Vagner','da Silva','Office Boy','Expedição','SP','Jundiaí',550.48,'2002-06-
23','00012'),
	('00009','Elisa','Macedo','Vendedor','Marketing','MG','Belo Horizonte',2300.00,'2005-09-
19','00007'),
	('00010','Teobaldo','Moura','Web Designer','Publicidade','RJ','Niterói',3210.00,'2000-08-
27','00031'),
	('00011','Radônio','Xavier','Químico','Laboratório 13','RJ','Niterói',5400.32,'2003-11-
11','00031'),
	('00012','Cláudia','Carvalho','Contador','Financeiro',DEFAULT,'Jundiaí',2500.51,'2002-12-
12','00001'),
	('00013','Natália','Costa','Diretor','TI',DEFAULT,'São Paulo',4559.72,'2002-02-02','00001'),
	('00014','Agevésio','Ramalho','Contador','Financeiro',DEFAULT,'São Paulo',3002.00,'2005-05-
04','00003'),
	('00015','Gertrudes','Trevizan','Auxiliar','Expedição',DEFAULT,'São Paulo',850.00,'2000-06-
06','00003'),
	('00016','Estrôncio','da Silva','Engenheiro','Laboratório 3','SP','São Paulo',7100.00,'1997-
07-07','00003'),
	('00017','Agatemor','Santos','Programador','TI',DEFAULT,'São Paulo',2500.00,'1998-11-
03','00003'),
	('00018','Alzevina','dos Santos','Estagiário','Marketing',DEFAULT,'São Paulo',822.20,'2008-
01-09','00003'),
	('00019','Deoclécio','Diniz','Motorista','Logística',DEFAULT,'São Paulo',1500.00,'2005-05-
04','00003'),
	('00020','Angélica','Santos','Estagiário','Publicidade','AM','Manaus',1000.00,'2009-04-
04','00021'),
	('00021','Sandra','Vilares','Gerente','Financeiro','AM','Manaus',5000.00,'2007-05-
04','00001'),
	('00022','Joana','Ferraz','Programador','TI','AM','Manaus',2500.00,'2009-01-04','00021'),
	('00023','Elmeraldo','da Silva','Motorista','Logística','SP','São Paulo',1000.00,'2009-02-
04','00003'),
	('00024','Beatriz','Xavier','Químico','Laboratório 13','RJ','Niterói',3500.00,'2008-11-
20','00031'),
	('00025','Fernanda','Toledo','Vendedor','Marketing','MG','Belo Horizonte',1400.00,'2006-12-
13','00007'),
	('00026','Tânia','Costa','An. de Sistemas','TI','AM','Manaus',4500.00,'2008-12-10','00021'),
	('00027','Zeoclécio','Teodoro','Motorista','Logística','AM','Manaus',1200.00,'2008-01-
31','00021'),
	('00028','Mariana','Xavier','Vendedor','Logística','SP','Jundiaí',1200.00,'2009-07-
22','00003'),
	('00029','Lucrécia','Ramalho','Programador','TI',DEFAULT,'Jundiaí',2200.00,'2007-07-
07','00003'),
	('00030','Martina','Santos','An. de Sistemas','TI','SP','Jundiaí',4200.00,'2007-09-
30','00003'),
	('00031','Alice','Santos','Engenheiro','Laboratório 13','RJ','Niterói',7200.00,'2008-01-
12','00031'),
	('00032','Paula','Franco','Programador','TI','SP','Jundiaí',2100.00,'2007-09-30','00003');

	INSERT INTO colaboradores (mat_id,nome,snome) VALUES
	('00033','Zé','Ninguém'),
	('00034','%///teste','teste'),
	('00035','_///teste','teste');

	--Inserção de dados na tabela funcionarios premiados
```

```
        INSERT INTO funcionarios_premiados VALUES
        ('00001','00007'),
        ('00002','00013'),
        ('00003','00004'),
        ('00004','00021'),
        ('00005','00010');

        INSERT INTO funcionarios_premiados (premio_id) VALUES ('00006'),('00007'),('00008'),
('00009'),('00010'),('00011'),('00012');
```

Dicas

- **Em que endereço está escutando?**

```
SHOW listen_addresses ;

 listen_addresses
------------------------
 localhost,192.168.10.7
(1 row)
```

- **Criando banco de dados latin1 em um cluster utf-8**

```
CREATE DATABASE sistema_outros ENCODING='latin1'LC_COLLATE='pt_BR.ISO-8859-1'
LC_CTYPE='pt_BR.ISO-8859-1' TEMPLATE=template0;
```

- **Número de conexões abertas no servidor**

```
SELECT count(procpid) FROM pg_stat_activity;
```

- **Número de conexões abertas em um determinado banco**

```
SELECT count(procpid) FROM pg_stat_activity WHERE datname='nome_do_banco';
```

- **Número de conexões abertas em cada banco do servidor**

```
SELECT datname,numbackends FROM pg_stat_database;
```

- **Quais ROLES estão conectados ao servidor, sendo que oid = 10 é o usuario que inicializa o cluster**

```
SELECT oid,rolname FROM pg_authid;
```

Bibilografia

Documentação oficial do PostgreSQL
http://www.postgresql.org/

Documentação oficial do GNU Compiler Collection (GCC)
http://gcc.gnu.org/