PostgreSQL - Administração

Juliano Atanazio

2018-04-10

PostgreSQL - Administração

Sobre esta Obra...

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

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

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

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

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

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

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

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

Juliano Atanazio
juliano777@gmail.com

Dedicatórias

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

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

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

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

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

À minha família e amigos.

Ao Criador.

Sumário

A Apostila

- Sobre a Apostila
- Créditos de Softwares Utilizados

Sobre a Apostila

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

Comando a Ser Digitado

Cliente de modo texto do PostgreSQL:

```
$ psql
```

Consulta envolvendo condição de filtragem:

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

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

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

Texto Impresso em Tela

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

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

Arquivos ou Trechos de Arquivos

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

São demonstrações de como um arquivo deve ser alterado, como deve ser ou

partes do próprio.

Se tiver três pontos seguidos separados por espaços significa que há outros conteúdos que ou não importam no momento ou que há uma continuação (a qual é anunciada).

Aviso

Aviso:

O mal uso deste comando pode trazer consequências indesejáveis.

Precauções a serem tomadas para evitar problemas.

Observação

Obs.:

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

Observações e dicas sobre o assunto tratado.

Créditos de Softwares Utilizados

Software utilizados para elaboração desta obra.

Editor de Texto

LibreOffice Writer

www.libreoffice.org

Editor de Gráficos Vetoriais

Inkscape

www.inkscape.org

Editor de Imagens

GIMP

www.gimp.org

Sistema Operacional

Linux

www.linux.com

Distribuições

Ubuntu

www.ubuntu.com

Linux Mint

www.linuxmint.com

Apresentação

- Objetivo
- Cenário do Curso
- Database Administrator: Seu Papel e Suas Responsabilidades

Objetivo

Este curso tem como grande objetivo trazer o devido conhecimento para realizar tarefas administrativas do PostgreSQL.

Cujas tarefas muitas vezes exigirão que atuemos em um nível mais baixo, mexendo diretamente com o sistema operacional onde será instalado.

Modos de instalação, configurações, autenticação, gerenciamento de usuários e estratégias de backup serão os temas mais abordados.

Capacitar o aluno a realizar tarefas administrativas do PostgreSQL.

Cenário do Curso

Os testes de laboratório serão feitos em Linux, CentOS.

A prática deste curso terá como base o banco de dados de exemplo `pagila` [1].

[1] http://pgfoundry.org/projects/dbsamples/

Database Administrator: Seu Papel e Suas Responsabilidades

Administrador de Banco de Dados, também conhecido como DBA (DataBase Administrator), é o profissional responsável pela instalação, configuração, manutenção e outras atividades relativas a bancos de dados.

Seu trabalho é contínuo, sendo que muitas vezes é necessário interagir com outros profissionais de TI, como programadores, administradores de sistema operacional e de hardware, gestores de projeto e etc.

Na concepção de uma nova base de dados ou um servidor o DBA deve ser envolvido, participando ativamente do projeto de modo a dimensionar os recursos para obter os melhores resultados.

Principais Responsabilidades

- Planejamento;
- Instalação;
- Configuração;
- Atualização;
- Administração;
- Monitoramento;
- Manutenção;
- Segurança.

Habilidades

- Comunicação, saber se expressar bem (verbal e escrita);
- Conhecimento de teoria de bancos de dados;
- Conhecimento de projeto de banco de dados;
- Conhecimento sobre o conceito de SGBDs, e. g. PostgreSQL, Oracle Database, IBM DB2, Microsoft SQL Server, MySQL, etc;
- Conhecimento sobre Structured Query Language (SQL), ou em adição a SQL, SQL/PSM (SQL/Persistent Stored Modules);
- Conhecimento geral de arquitetura de computação distribuída, e. g. Client/Server, Internet/Intranet, Enterprise;
- Conhecimento sobre o sistema operacional que roda o SGBD, e. g. Linux, FreeBSD, Unix, Solaris, Windows, AIX, etc;
- Conhecimento sobre tecnologias de armazenamento, gerenciamento de memória, arrays de disco, NAS/SAN, redes;
- Conhecimentos de rotinas de manutenção, recuperação e backup de bancos de dados.

Tarefas

- Instalação e atualização do servidor e ferramentas de aplicação;
- Alocar sistema de armazenamento e planejar futuros requerimentos de armazenamento para o sistema de banco de dados;
- Modificar a estrutura da base de dados, quando necessário, conforme informações fornecidas por desenvolvedores da aplicação que a acessa;
- Cadastramento de usuários e manter a segurança;
- Controlar e monitorar o acesso de usuários;
- Assegurar a conformidade com o acordo de licença do fornecedor do banco de dados;
- Monitorar e otimizar a performance;
- Planejamento para backup e restauração (recovery);
- Gerar relatórios via consultas conforme a necessidade;
- Contactar suportes conforme a demanda.

PostgreSQL
–
Administração

1 Arquitetura do PostgreSQL

- Como uma Conexão é Estabelecida
- Fundamentos da Arquitetura
- Plataformas Suportadas
- O Conceito de Cluster de Banco de Dados PostgreSQL
- Layout de Arquivos Físicos do Banco de Dados

1.1 Como uma Conexão é Estabelecida

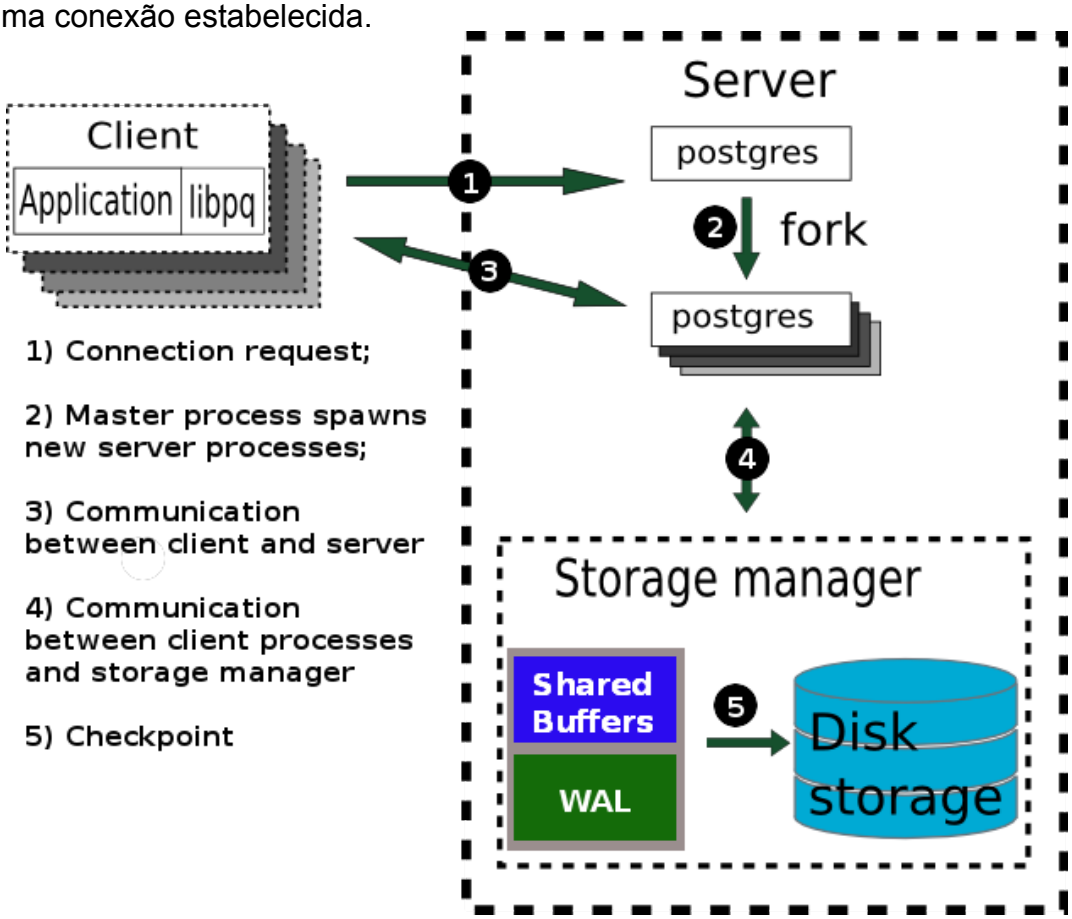
O PostgreSQL é implementado usando um simples processo por usuário no modelo cliente/servidor. Nesse modelo há um processo cliente conectado a exatamente um processo servidor.

Como não sabemos previamente quantas conexões serão feitas, temos que usar o processo mestre que gera um novo processo servidor toda vez que uma conexão é requerida. Esse processo mestre é chamado `postgres` e escuta em uma porta TCP/IP (por padrão a 5432) para conexões de entrada.

Sempre que uma requisição de conexão é detectada o processo `postgres` gera um novo processo; um *fork*. As tarefas do servidor comunicam entre si utilizando semáforos e memória compartilhada para garantir integridade de dados a todos acessos de dados simultâneos.

O processo cliente pode ser qualquer programa que entenda o protocolo PostgreSQL. Muitos clientes são baseados na biblioteca em Linguagem C, a libpq, mas há muitas implementações independentes, como o driver Java JDBC.

Uma vez que uma conexão é estabelecida o processo cliente pode enviar uma consulta para o *backend* (servidor). A consulta é transmitida usando texto plano, não há análise (parsing) no frontend (cliente). O servidor analisa a consulta, cria o plano de execução, executa o plano e retorna as traz as linhas para o cliente transmitindo-as em sobre uma conexão estabelecida.



1.2 Fundamentos da Arquitetura

Uma sessão do PostgreSQL consiste nos seguintes processos (programas):

- **backend**: Um processo servidor que gerencia os arquivos físicos do banco de dados. Aceita conexões ao banco, vindas de aplicações clientes e executa ações em prol desses clientes. Antigamente esse aplicativo servidor se chamava `postmaster`, hoje é simplesmente chamado de `postgres`.

- **frontend**: A aplicação cliente do usuário que executa operações na base de dados. Aplicações clientes podem ser de natureza diversificada: um cliente pode ser uma ferramenta em modo texto, um aplicativo gráfico, um servidor web (como com PHP, Django, Ruby On Rails, JBoss, etc), ou uma ferramenta especializada em manutenção de banco de dados. Alguns aplicativos clientes são distribuídos com o próprio PostgreSQL.

Normalmente em aplicações do tipo cliente/servidor, o cliente e o servidor podem estar em diferentes máquinas. Nesse caso, a comunicação será por uma conexão TCP/IP.

O servidor PostgreSQL pode gerenciar múltiplas conexões simultâneas desses clientes. Para que isso seja possível, a cada nova conexão, é feito um *fork* do processo servidor original. A partir desse ponto, o cliente e um novo processo servidor comunicam sem intervenção do processo `postgres` original. Dessa forma, o processo servidor mestre estará sempre rodando, esperando por conexões clientes, enquanto que um cliente e seu respectivo processo vem e vão. Tudo isso é transparente para o usuário.

1.2.1 libpq

A `libpq` é a API escrita em Linguagem C para o PostgreSQL. `libpq` é uma biblioteca que tem um conjunto de funções que permite a aplicações clientes passar consultas (*queries*) para o *backend* servidor do PostgreSQL e receber o resultado dessas consultas.

A `libpq` é também o motor que está por baixo de várias outras interfaces para o PostgreSQL, incluindo aquelas escritas em C++, Python, Perl, Tcl e ECPG. Há também vários exemplos completos de aplicações no diretório `src/test/examples` na distribuição do código-fonte.

Programas clientes que usam a libpq devem incluir o arquivo de cabeçalho (*header*) `libpq-fe.h` e deve fazer a ligação com a biblioteca `libpq`.

1.3 Plataformas Suportadas

Uma plataforma (que é, uma combinação de arquitetura de CPU e sistema operacional) é considerada suportada pela comunidade de desenvolvimento do PostgreSQL se seu código contém as disposições para funcionar nessa plataforma e tem sido recentemente verificado para compilar e passar em testes de regressão.

Atualmente, a maior parte dos testes de compatibilidade de plataforma é feito por máquinas de teste na PostgreSQL Build Farm [1].

Quem estiver interessado em utilizar o PostgreSQL em uma plataforma que não está representada na *build farm*, mas o código funciona ou pode fazer funcionar, está fortemente encorajado para configurar uma máquina membro da *build farm* de modo que a continuidade da compatibilidade seja assegurada.

Em geral, o PostgreSQL, espera-se que ele funcione nestas arquiteturas de CPU: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K, e PA-RISC.

Há suporte de código para M32R, NS32K, e VAX, mas essas arquiteturas não sabe-se se foram testadas recentemente. É frequentemente possível compilar em uma CPU não suportada, no momento do `./configure` utilizando a opção `--disable-spinlocks`, mas a performance será horrível.

O PostgreSQL é compatível com os seguintes sistemas operacionais:

- Linux (todas distribuições recentes);
- FreeBSD;
- NetBSD;
- OpenBSD;
- Solaris;
- AIX;
- HP/UX;
- Windows (Win2000 SP4 ou superior);
- Mac OS X;
- IRIX;
- Tru64 Unix;
- UnixWare.

Outros sistemas operacionais Unix-*like* devem funcionar, mas não foram testados até então. Na maioria dos casos, todas arquiteturas de CPU suportadas por um dado sistema operacional funcionará.

Na documentação oficial do PostgreSQL há informações específicas para instalação e configuração de sistemas operacionais específicos.

[1] https://buildfarm.postgresql.org/

1.4 O Conceito de Cluster de Banco de Dados PostgreSQL

No PostgreSQL, o conceito de *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.

É possível ter mais de um *cluster* em uma mesma máquina, para cada um terá um processo independente, que por sua vez também devem escutar em portas TCP/IP diferentes entre si.

1.5 Layout de Arquivos Físicos do Banco de Dados

Esta seção descreve o formato de armazenamento no nível de arquivos e diretórios.

Todos os dados necessários para um *cluster* é armazenado no diretório de dados do cluster, mais conhecido como `PGDATA`, uma variável de ambiente.

A localização do `PGDATA` varia conforme o sistema operacional e a forma de instalação.

Podem haver vários *clusters* gerenciados como diferentes instâncias de servidor na mesma máquina.

O diretório do `PGDATA` contém vários subdiretórios e arquivos de controle.

Por padrão, arquivos de configuração do `cluster` são armazenados na raiz do diretório `$PGDATA`: `postgresql.conf`, `postgresql.auto.conf`, `pg_ident.conf` e `pg_hba.conf`.

1.5.1 Conteúdo do PGDATA

Diretórios (o que contém)

- **`base`**

  Subdiretórios referentes a bases de dados, sendo que cada um corresponde a um banco de dados no *cluster*. Em cada diretório de um banco, há principalmente arquivos de dados, cujo tamanho máximo e divisão em páginas, por padrão são respectivamente 1 GB e 8 kB. Se for ultrapassado o limite máximo de tamanho de arquivo para um objeto (e. g. tabela, índice), outro arquivo é criado para esse objeto.

- **`global`**

  Tabelas de sistema do *cluster*, como a *pg_database*.

- **`pg_commit_ts`**

  Dados de timestamp de commit de transação.

- **`pg_dynshmem`**

  Arquivos usados pelo subsistema de memória compartilhada dinâmica.

- **`pg_logical`**

  Dados de status para decodificação lógica.

- **pg_multixact**

  Dados de estado de multi-transação (usados para travas compartilhadas de linhas).

- **pg_notify**

  Estados de dados `LISTEN` / `NOTIFY`.

- **pg_replslot**

  Dados de *slots* de replicação.

- **pg_serial**

  Informações sobre transações serializáveis efetivadas.

- **pg_snapshots**

  Snapshots exportados.

- **pg_stat**

  Arquivos permanentes para o subsistema de estatísticas.

- **pg_stat_tmp**

  Arquivos temporários para o subsistema de estatísticas. É uma boa prática colocá-lo como ponto de montagem em um pequeno *ramdisk*.

- **pg_subtrans**
  Estados de dados de subtransações.

- **pg_tblspc**
  Links simbólicos para *tablespaces*.

- **pg_twophase**

  Arquivos de estado para *prepared transactions*.

- **pg_wal**

  Segmentos do WAL, que são os logs de transação em formato binário.

- **pg_xact**

  Dados de status de commit de transação.

Arquivos

- **PG_VERSION**

  Número principal da versão do PostgreSQL.

- **current_logfiles**

  Seu contéudo exibe o destino de log (log_destination) e seu respectivo arquivo de log.

- **postgresql.conf**

  Arquivo de configuração principal do PostgreSQL.

- **postgresql.auto.conf**

  Arquivo de configuração secundário do PostgreSQL.
  São configurações que têm prioridade sobre o arquivo principal, geradas pelo comando `ALTER SYSTEM`.

- **postmaster.opts**

  Mantém gravadas as opções de linha de comando em que o servidor foi iniciado pela última vez.

- **postmaster.pid**

  Arquivo de bloqueio (lock) que grava o ID do processo do postmaster atual (PID), o caminho do diretório de dados do cluster, o timestamp de início do postmaster, o número da porta, o caminho do diretório de soquete do domínio Unix (vazio no Windows), primeiro endereço de escuta válido (endereço IP ou * ou vazio, se não ouvindo TCP) e ID de segmento de memória compartilhada (este arquivo não está presente após o desligamento do servidor).

2 Instalação do PostgreSQL

- Sobre Instalação do PostgreSQL
- Instalação via Pacotes
- Instalação via Código-Fonte
- SSH sem Senha

2.1 Sobre Instalação do PostgreSQL

A instalação do PostgreSQL tem formas diferentes de ser feita, seja ela qual em qual sistema operacional estiver sendo feita.

Em sistemas operacionais da família Unix, temos duas opções: via pacotes binários ou via instalação compilando o código fonte.

2.1.1 Instalação via Pacotes

É a forma mais simples de instalação.

Uma instalação via pacotes se dá através de um sistema de gerenciamento de pacotes, tais como o `aptitude` (Debian/Ubuntu), `yum` (RedHat/CentOS) ou `pkgng` (FreeBSD).

O sistema gerenciador de pacotes baixa o(s) pacote(s) do(s) binário(s) pré compilados e resolve qualquer dependência que existir.

Pacotes são binários compilados de uma forma geral para poder rodar em qualquer máquina e não têm otimizações específicas para um determinado hardware.

2.1.1.1 Repositórios PGDG

Para distribuições Linux baseadas em Debian ou RedHat, há um repositório oficial do PGDG (*PostgreSQL Global Development Group* – Grupo Global de Desenvolvimento do PostgreSQL).

Utilizar um repositório PGDG nos permite instalar a última versão do PostgreSQL pelo gerenciador de pacotes, o que é algo vantajoso, pois nos repositórios oficiais das distribuições, a versão disponível para instalação está sempre defasada.

Debian: https://apt.postgresql.org/
RedHat: https://yum.postgresql.org/

2.1.2 Instalação Via Compilação de Código Fonte

É o tipo de instalação mais complicada e demorada a se fazer.

No entanto, a mesma tem suas vantagens.

No processo de compilação podem ser especificadas opções de acordo com o desejo do usuário, além claro, de poder alterar o código fonte.

O PostgreSQL instalado por compilação pode ter um desempenho superior a um pacote binário se tiver otimizações para a máquina na qual será instalado.

Existem diversas formas de fazer esse tipo de instalação.

2.1.3 Independência do Usuário de Sistema

Uma peculiaridade incômoda quando se faz uma instalação no Linux via pacotes é a falta de certas opções para o usuário de sistema do PostgreSQL, que no Linux normalmente é o usuário `postgres`.

Tal usuário não tem em sua variável de ambiente `$PATH` o caminho para aplicativos como o `pg_ctl` (gerenciamento de clusters) ou o `initdb` (criação de novos *clusters*).

Operações de gerenciamento como parar (`stop`), iniciar (`start`), reinicializar (`restart`) ou recarregar (`reload`) configurações serem muito mais complicados de serem feitas.

Ao tornar independente o usuário de sistema do serviço do PostgreSQL tudo será mais fácil e dispensará o usuário `root` para tarefas triviais de um DBA.

2.2 Instalação via Pacotes

Linux atualmente é o sistema operacional mais utilizado para servidores PostgreSQL e é também o preferido pelos desenvolvedores.

Arquitetura de Diretórios

Instalação	/usr/pgsql-${PGVERSION}
Configuração	/var/lib/pgsql/${PGVERSION}/data
Dados	/var/lib/pgsql/${PGVERSION}/data
Binários	/usr/pgsql-{PGVERSION}/bin/

2.2.1.1 Instalação via Repositório PGDG

Tenha em mente qual versão majoritária do PostgreSQL será instalada.

Entre no site http://yum.postgresql.org/repopackages.php e instale o arquivo .rpm conforme a versão do PostgreSQL e a versão de sua distro.

Configure o repositório para não instalar versões mais antigas, adicionando a seguinte linha conforme a distro:

```
exclude=postgresql*
```

Distro	Arquivo	Sessões
RedHat	/etc/yum/pluginconf.d/rhnplugin.conf	main
CentOS	/etc/yum.repos.d/CentOS-Base.repo	base, updates

Ou simplesmente utilize o script:

```
# if [ -f /etc/yum.repos.d/CentOS-Base.repo ]; then
    # CentOS
    sed -i 's/\(\[base\]\|\[updates\]\)/\1\nexclude=postgresql*/g' \
/etc/yum.repos.d/CentOS-Base.repo

else
    # RedHat
    sed -i 's/\(\[main\]\)/\1\nexclude=postgresql*/g' \
/etc/yum/pluginconf.d/rhnplugin.conf

fi
```

Atribuir à variável de ambiente PGVERSION, a versão a ser instalada:

```
# read -p 'Digite a versão majoritária do PostgreSQL a ser instalada: ' \
PGVERSION

Digite a versão majoritária do PostgreSQL a ser instalada:
```

Instalação do arquivo rpm de repositório PGDG:

```
# rpm -Uvh <link_do_rpm_do_repositorio>
```

Exemplo:

```
# rpm -Uvh \
https://download.postgresql.org/pub/repos/yum/10/\
redhat/rhel-7-x86_64/pgdg-centos10-10-2.noarch.rpm
```

Instalação do pacote:

```
# yum install -y postgresql${PGVERSION}-server && yum clean all
```

Habilitando o serviço na inicialização:

```
# systemctl enable postgresql-${PGVERSION}
```

Criando o cluster:

```
# /usr/pgsql-${PGVERSION}/bin/postgresql-${PGVERSION}-setup initdb
```

Iniciando o serviço:

```
# systemctl start postgresql-${PGVERSION}
```

2.2.1.2 CentOS: Indenpendência do Usuário de Sistema

Copiando os arquivos de usuário do diretório de template de usuário:

```
# rsync -v /etc/skel/.* ~postgres/
```

Variável de ambiente do diretório de dados:

```
# PGDATA="/var/lib/pgsql/${PGVERSION}/data"
```

Variável de ambiente do diretório de configurações:

```
# PGCONF="/etc/pgsql/${PGVERSION}/main"
```

Parar o serviço do PostgreSQL:

```
# systemctl stop postgresql-${PGVERSION}
```

Criar diretório de configurações:

```
# mkdir -p ${PGCONF}
```

Mover arquivos de configuração do diretório de dados para o diretório de configurações:

```
# mv ${PGDATA}/*.conf ${PGCONF}/
```

Criar links simbólicos de arquivos de configuração no diretório de dados:

```
# ls ${PGCONF}/* | xargs -i ln -sf {} ${PGDATA}/
```

Variáveis de ambiente para o usuário de sistema do serviço:

```
# cat << EOF > ~postgres/.pgvars
export PGVERSION="${PGVERSION}"
export PGDATA="/var/lib/pgsql/\${PGVERSION}/data"
export PGCONF="/etc/pgsql/\${PGVERSION}/main"
export PGDATABASE='postgres'
export PGUSER='postgres'
export PATH="\${PATH}:/usr/pgsql-${PGVERSION}/bin/"
export OOMScoreAdjust=-1000
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
EOF
```

Faz com que o arquivo de variáveis seja lido pelo arquivo de perfil:

```
# echo 'source .pgvars' >> ~postgres/.bash_profile
```

Ajustar permissão de diretórios:

```
# chmod -R 0700 ${PGCONF} /var/lib/pgsql ~postgres
```

Ajustar permissão de arquivos:

```
# find /var/lib/pgsql ${PGCONF} -type f -exec chmod 0600 {} \;
```

Usuário e grupo postgres como donos dos diretórios:

```
# chown -R postgres: ${PGCONF} /var/lib/pgsql ~postgres
```

Mudar para usuário de sistema do serviço:

```
# su - postgres
```

Iniciando o serviço pelo usuário de sistema:

```
$ pg_ctl start
```

Testando:

```
$ psql -c 'SELECT true;'
 bool
------
 t
```

2.3 Instalação via Código-Fonte

Criar grupo de sistema postgres:

```
# groupadd -r postgres &> /dev/null
```

Criar usuário de sistema postgres:

```
# useradd -s /bin/bash -k /etc/skel -d /var/lib/pgsql \
-g postgres -m -r postgres  &> /dev/null
```

Qual versão completa (X.Y) do PostgreSQL deve ser instalada?:

```
# read -p \
'Digite o número de versão completo (X.Y) do PostgreSQL a ser baixado: ' \
PGVERSIONXY
```

Baixando o código-fonte silenciosamente e em background:

```
# (wget --quiet -c \
https://ftp.postgresql.org/pub/source/v${PGVERSIONXY}/\
postgresql-${PGVERSIONXY}.tar.bz2 -P /tmp/) &
```

Versão majoritária (X.Y) do PostgreSQL:

```
# PGVERSION=`echo ${PGVERSIONXY} | cut -f1 -d.`
```

Local de Instalação:

```
# PG_INSTALL_DIR="/usr/local/pgsql/${PGVERSION}"
```

Diretório de binários:

```
# PGBIN="${PG_INSTALL_DIR}/bin"
```

Diretório de bibliotecas:

```
# PG_LD_LIBRARY_PATH="${PG_INSTALL_DIR}/lib"
```

Diretório de manuais:

```
# PG_MANPATH="${PG_INSTALL_DIR}/man"
```

Diretório de arquivos de configuração:

```
# PGCONF="/etc/pgsql/${PGVERSION}"
```

Diretório de arquivos de log:

```
# PGLOG="/var/log/pgsql/${PGVERSION}"
```

Diretório de dados (variável temporária):

```
# PGDATA="/var/lib/pgsql/${PGVERSION}/data"
```

Diretório de estatísticas temporárias:

```
# PG_STATS_TEMP="/var/lib/pgsql/${PGVERSION}/pg_stat_tmp"
```

Diretório de logs de transação (WAL):

```
# PG_WAL="/var/lib/pgsql/${PGVERSION}/pg_wal"
```

Diretório de bibliotecas PYTHON:

```
# PYTHONDIR="/var/lib/pgsql/${PGVERSION}/python"
```

Opções do initdb (com pg_wal fora de PGDATA):

```
# INITDB_OPTS="-k \
-D ${PGDATA} -E utf8 -U postgres \
--locale=pt_BR.utf8 \
--lc-collate=pt_BR.utf8 \
--lc-monetary=pt_BR.utf8 \
--lc-messages=en_US.utf8 \
-T portuguese \
-X ${PG_WAL}"
```

Tamanho padrão do ponto de montagem em RAM para estatísticas temporárias:

```
# PG_STATS_TEMP_SIZE='32M'
```

Criação de diretórios:

```
# mkdir -p ${PG_INSTALL_DIR}/src/ ${PGCONF} ${PGLOG} \
${PG_WAL} ${PGDATA} ${PG_STATS_TEMP} ${PYTHONDIR}
```

Linha em /etc/fstab para o diretório de estatísticas temporárias:

```
# echo -e "\ntmpfs ${PG_STATS_TEMP} tmpfs \
size=${PG_STATS_TEMP_SIZE},uid=postgres,gid=postgres 0 0" >> /etc/fstab
```

Monta tudo definido em /etc/fstab:

```
# mount -a
```

PL/Python

PL/Python é a implementação de Python no PostgreSQL como linguagem procedural. Para podermos usufruir de todo poder dessa linguagem é necessário instalar Python e seus *headers* para a compilação, além de declarar qual é a localização do binário principal que será o interpretador.

Localizando o pacote com a versão mais recente:

```
# PYTHON_PKG=`yum search python3 | egrep '^python3.\.x86_64' | awk '{print $1}' \
| sort | tail -1 | cut -f1 -d.`
```

Variável para o pacote principal com seu respectivo pacote de headers:

```
# PYTHON_PKG="${PYTHON_PKG} ${PYTHON_PKG}-devel"
```

Pacotes a serem instalados (exceto Python):

```
# PKG="bison gcc flex gettext make readline-devel openssl-devel libxml2-devel \
openldap-devel perl-devel perl-ExtUtils-MakeMaker \
perl-ExtUtils-Embed"
```

Instalação de pacotes necessários para a compilação e limpando os pacotes baixados:

```
# yum -y install ${PKG} ${PYTHON_PKG} && yum clean all
```

Variável de ambiente para o binário de Python:

```
# export PYTHON="/usr/bin/`ls /usr/bin/python3* | \
sed 's:/usr/bin/::g' | egrep '\.[0-9]$'`"
```

Mover o código-fonte baixado para o sub-diretório src no diretório de instalação:

```
# mv /tmp/postgresql-${PGVERSIONXY}.tar.bz2 ${PG_INSTALL_DIR}/src/
```

Descompactar o código-fonte:

```
# tar xf ${PG_INSTALL_DIR}/src/postgresql-${PGVERSIONXY}.tar.bz2 \
-C ${PG_INSTALL_DIR}/src/
```

Protege o processo principal do OOM Killer:

```
# CPPFLAGS="-DLINUX_OOM_SCORE_ADJ=0"
```

Opções do make com número de jobs conforme a quantidade cores de CPU (cores + 1):

```
# MAKEOPTS="-j`expr \`cat /proc/cpuinfo | egrep ^processor | wc -l\` + 1`"
```

Tipo de hardware:

```
# CHOST="x86_64-unknown-linux-gnu"
```

Flags de otimização para o make:

```
# CFLAGS="-march=native -O2 -pipe" && CXXFLAGS="$CFLAGS"
```

Opções do configure:

```
# CONFIGURE_OPTS="
--prefix=${PG_INSTALL_DIR} \
--with-perl \
--with-python \
--with-libxml \
--with-openssl \
--with-ldap \
--mandir=/usr/local/pgsql/${PGVERSION}/man \
--docdir=/usr/local/pgsql/${PGVERSION}/doc"
```

Ir ao diretório onde estão os fontes:

```
# cd ${PG_INSTALL_DIR}/src/postgresql-${PGVERSIONXY}
```

Processo de configure, compilação (com manuais e contrib) e instalação:

```
# ./configure ${CONFIGURE_OPTS} && make world && make install-world
```

Criar o arquivo .pgvars com no diretório do usuário home postgres:

```
# cat << EOF > ~postgres/.pgvars

# Environment Variables

export PGVERSION='${PGVERSION}'
export LD_LIBRARY_PATH="/usr/local/pgsql/\${PGVERSION}/lib:\${LD_LIBRARY_PATH}"
export MANPATH="/usr/local/pgsql/\${PGVERSION}/man:\${MANPATH}"
export PATH="/usr/local/pgsql/\${PGVERSION}/bin:\${PATH}"
export PGDATA="/var/lib/pgsql/\${PGVERSION}/data"
export PGCONF="/etc/pgsql/\${PGVERSION}"
export OOMScoreAdjust=-1000
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
export PYTHONPATH="\${PYTHONPATH}:/var/lib/pgsql/\${PGVERSION}/python"
EOF
```

Faz com que o arquivo .pgvars seja lido a cada login:

```
# echo 'source ~/.pgvars' >> ~postgres/.bashrc
```

Definição do arquivo de serviço para SystemD:

```
# cat << EOF > /lib/systemd/system/postgresql-${PGVERSION}.service
[Unit]
Description=PostgreSQL ${PGVERSION} database server
After=syslog.target
After=network.target
[Service]
Type=forking
User=postgres
Group=postgres
Environment=PGDATA=${PGDATA}
Environment=PYTHONPATH=${PYTHONDIR}
OOMScoreAdjust=-1000
ExecStart=${PGBIN}/pg_ctl start -D ${PGDATA} -s -w -t 300
ExecStop=${PGBIN}/pg_ctl stop -D ${PGDATA} -s -m fast
ExecReload=${PGBIN}/pg_ctl reload -D ${PGDATA} -s
OOMScoreAdjust=-1000
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment=PG_OOM_ADJUST_VALUE=0
TimeoutSec=300
[Install]
WantedBy=multi-user.target
EOF
```

Habilita o serviço na inicialização:

```
# systemctl enable postgresql-${PGVERSION}.service
```

Dar propriedade a usuário e grupo postgres aos diretórios:

```
# chown -R postgres: ${PGCONF} ${PGLOG} ${PG_WAL} ${PGDATA} \
${PG_STATS_TEMP} ~postgres
```

Criação de cluster:

```
# su - postgres -c "initdb ${INITDB_OPTS}"
```

Mover arquivos de configuração para o diretório de configuração:

```
# su - postgres -c "mv ${PGDATA}/*.conf ${PGCONF}/"
```

Criar link para cada configuração no diretório de dados:

```
# su - postgres -c "ls ${PGCONF}/* | xargs -i ln -sf {} ${PGDATA}/"
```

Modificações no postgresql.conf:

```
# sed "s:\(^#listen_addresses.*\):\1\nlisten_addresses = '*':g" \
-i ${PGCONF}/postgresql.conf

# sed "s:\(^#log_destination.*\):\1\nlog_destination = 'stderr':g" \
-i ${PGCONF}/postgresql.conf

# sed "s:\(^#logging_collector.*\):\1\nlogging_collector = on:g" \
-i ${PGCONF}/postgresql.conf

# sed "s:\(^#\)\(log_filename.*\):\1\2\n\2:g" \
-i ${PGCONF}/postgresql.conf

# sed "s:\(^#log_directory.*\):\1\nlog_directory = '${PGLOG}':g" \
-i ${PGCONF}/postgresql.conf

# sed "s:\(^#stats_temp_directory.*\):\1\nstats_temp_directory = \
'${PG_STATS_TEMP}':g" -i ${PGCONF}/postgresql.conf
```

Ouvir em todas interfaces de rede, configurações de log e diretório de estatísticas temporárias.

Inicia o serviço do PostgreSQL:

```
# systemctl start postgresql-${PGVERSION}
```

Ajustes para o psql:

```
# su - postgres -c "cat << EOF > ~/.psqlrc
\set HISTCONTROL ignoredups
\set COMP_KEYWORD_CASE upper
\x auto
EOF
"
```

Desinstalar pacotes que não são mais necessários:

```
# yum erase -y ${PKG}
```

Testando:

```
$ su - postgres -c "psql -c 'SELECT true;'"

 bool
------
 t
```

2.4 SSH sem Senha

Após instalar um servidor PostgreSQL é interessante que a administração do servidor pertinente ao serviço de banco de dados seja feita somente pelo usuário postgres, que é o usuário de sistema.

Evitar utilizar o usuário root e fazer com que o administrador se conecte ao servidor de banco de dados através de uma chave pública permitida no servidor.

Armazenar o endereço na variável de ambiente:

```
$ read -p 'Digite o endereço do Servidor PostgreSQL: ' PGSERVER

Digite o IP do Servidor PostgreSQL:
```

Caso não exista a chave na máquina local, a mesma será criada:

```
$ if [ ! -e ~/.ssh/id_rsa ]; then
    ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa;
fi
```

Adicionando a chave pública do usuário e máquina local para o usuário root do servidor:

```
$ ssh-copy-id root@${PGSERVER}
```

Criando chaves para o usuário postgres:

```
$ ssh root@${PGSERVER} \
"su - postgres -c \"ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa\""
```

Adicionando a chave pública para o usuário postgres do servidor:

```
$ cat ~/.ssh/id_rsa.pub | \
ssh root@${PGSERVER} "cat - >> ~postgres/.ssh/authorized_keys"
```

Teste de acesso como usuário postgres:

```
$ ssh postgres@${PGSERVER}
```

3 Gerenciamento de Clusters PostgreSQL

- Variáveis de Ambiente do PostgreSQL
- O Arquivo ~/.pgpass
- Arquivo de Serviço de Conexão
- initdb
- O Aplicativo postgres
- pg_ctl
- Arquivos Físicos e OID

3.1 Variáveis de Ambiente do PostgreSQL

As seguintes variáveis de ambiente podem ser usadas para definir valores padrões de conexão que são utilizadas pelas funções da libpq `PQconnectdb`, `PQsetdbLogin` e `PQsetdb` se não houver nenhum outro valor especificado pelo código de chamada.

São úteis para evitar embutir informações de conexão de banco de dados em aplicações de clientes simples.

- **PGHOST**
  Tem seu comportamento igual ao parâmetro de conexão **`host`**.

- **PGHOSTADDR**
  Mesmo comportamento do parâmetro de conexão **`host`**. Pode ser configurado invés de ou em adição a `PGHOST` para evitar *overhead* de busca DNS.

- **PGPORT**
  Seu efeito é o mesmo do parâmetro de conexão **`port`**. Especifica a porta de conexão ao servidor.

- **PGDATABASE**
  Especifica que base de dados será usada para a conexão.

- **PGUSER**
  Define o usuário de conexão.

- **PGPASSWORD**
  Variável de ambiente usada para armazenar como valor a senha da conexão. Por questões de segurança seu uso é desencorajado. É altamente aconselhável usar o arquivo `~/.pgpass` ao invés dessa variável de ambiente.

- **PGPASSFILE**
  Seu valor aponta para qual arquivo será utilizado como arquivo de senhas. Por padrão é o arquivo contido dentro do diretório do usuário `.pgpass` ou `~/.pgpass`.

- **PGSERVICE**
  Nome da conexão de serviço.

- **PGSERVICEFILE**
  Similarmente à variável `PGPASSFILE`, a variável `PGSERVICEFILE` aponta qual é o arquivo utilizado para conexão de serviço. Por padrão é `~/.pg_service.conf`.

- **PGREALM**
  Define o domínio Kerberos a ser usado com o PostgreSQL.

- **PGOPTIONS**
  Parâmetros de conexão.

- **PGAPPNAME**
  Define o parâmetro de conexão **`application_name`**.

- **PGSSLMODE**
  Parâmetro de conexão **`sslmode`**.

- **PGREQUIRESSL**
  Parâmetro de conexão **`requiressl`**.

- **PGSSLCOMPRESSION**
  Parâmetro de conexão **`sslcompression`**.

- **PGSSLCERT**
  Parâmetro de conexão **`sslcert`**.

- **PGSSLKEY**
  Parâmetro de conexão **`sslkey`**.

- **PGSSLROOTCERT**
  Parâmetro de conexão **`sslrootcert`**.

- **PGSSLCRL**
  Parâmetro de conexão **`sslcrl`**.

- **PGREQUIREPEER**
  Parâmetro de conexão **`requirepeer`**.

- **PGKRBSRVNAME**
  Parâmetro de conexão **`krbsrvname`**.

- **PGGSSLIB**
  Parâmetro de conexão **`gsslib`**.

- **PGCONNECT_TIMEOUT**
  Parâmetro de conexão **`connect_timeout`**.

- **PGCLIENTENCODING**
  Parâmetro de conexão **`client_encoding`**.
  As seguintes variáveis de ambiente podem ser usadas para especificar o comportamento padrão de cada sessão PostgreSQL:

- **PGDATESTYLE**
  Configura o estilo de representação de data e hora. (Equivalente a **`SET`** `datestyle` **`TO`** ....)

- **PGTZ**
  Configura o fuso horário (*time zone*). (Equivalente a **`SET`** `timezone` **`TO`** ....)

- **PGGEQO**
  Configura o modo padrão para o otimizador genético de consultas. (Equivalente a `SET geqo TO` ....)

Obs.:

Consulte o comando SQL SET para informações de valores corretos para estas variáveis de ambiente. As variáveis de ambiente seguintes determinam o comportamento interna da *libpq*; elas sobrescrevem os valores padrões com que foi compilado.

- **PGSYSCONFDIR**

Configura em que diretório está armazenado o arquivo *pg_service.conf* e em uma futura versão possivelmente outros arquivos de configuração para todo o sistema.

- **PGLOCALEDIR**

Configura em que diretório estão armazenados os arquivos de localidade (*locale files*) para internacionalização de mensagens.

Obs.:

Ver também os comandos `ALTER ROLE` e `ALTER DATABASE` para maneiras de configurar o comportamento padrão em uma base por usuário ou por banco de dados.

3.2 O Arquivo ~/.pgpass

O arquivo `.pgpass` em um diretório de usuário ou o arquivo referenciado pela variável de ambiente `PGPASSFILE` pode conter senhas para serem usadas se uma conexão requer uma senha (e nenhuma senha foi especificada de nenhuma outra maneira).

No Windows o arquivo é chamado `\%APPDATA%\postgresql\pgpass.conf`, onde `%APPDATA%` refere ao subdiretório `Application Data` no diretório do usuário.

O arquivo deve ter linhas no seguinte formato:

```
hostname:port:database:username:password
```

O caractere sustenido (#) é utilizado para se fazer comentários. Nos primeiros quatro campos pode ser utilizado o caractere asterisco ("*"), o que significa casar com qualquer coisa. O campo de senha (`password`) da primeira linha que combinar com a conexão atual será usado. Portanto, insira entradas mais específicas primeiro ao utlizar curingas.

Se uma entrada precisa conter "**:**" ou "", escape-os com "\". Um hostname como "`localhost`" combina ambos TCP (nome de máquina "`localhost`") e soquete de domínio Unix (*Unix domain socket*) da máquina local.

Em um servidor *standby*, o nome do banco de dados de replicação trata conexões de replicação *streaming* feitas parar o servidor *master*.

Em sistemas Unix, as permissões do arquivo `.pgpass` não deve permitir qualquer acesso para todos ou grupo, cuja permissão octal deve ser `0600`. Se as permissões estiverem menos restritas do que isso, o arquivo será ignorado. No Windows, assume-se que o arquivo está armazenado em um diretório que é seguro, então nenhuma checagem de permissão especial é feita.

Obs.:

O campo para base de dados (`database`) é utilidade limitada porque usuários têm a mesma senha para todas as bases no mesmo *cluster*. Por isso, no campo `database` utilizamos o caractere *.

Para testes crie um segundo cluster:

```
$ initdb -U postgres -E utf8 -D /tmp/cluster2
```

Mude a porta padrão no postgresql.conf e inicialize o cluster:

```
$ sed -i 's/\#port = 5432/port = 5433/g' \
/tmp/cluster2/postgresql.conf && pg_ctl -D /tmp/cluster2 start
```

Atribuindo uma senha para o papel postgres:

```
$ psql -p 5433 -c "ALTER ROLE postgres ENCRYPTED PASSWORD '123';"
```

Criação do usuário zezinho com senha definida:

```
$ psql -p 5433 -c "CREATE ROLE zezinho LOGIN ENCRYPTED PASSWORD '123';"
```

Deixando o pg_hba do cluster com apenas uma linha de modo que sempre peça senha:

```
$ echo 'local all all md5' > /tmp/cluster2/pg_hba.conf
```

Reload no cluster para aplicar o novo pg_hba.conf:

```
$ pg_ctl -D /tmp/cluster2 reload
```

Tentativa de conexão que pedirá senha:

```
$ psql -p 5433
```

Criação do arquivo ~/.pgpass:

```
$ cat << EOF > ~/.pgpass
# hostname:port:database:username:password
localhost:5433:*:postgres:123
localhost:5433:*:zezinho:1234
EOF
```

Somente o usuário dono poderá ter acesso ao arquivo:

```
$ chmod 0600 ~/.pgpass
```

Nesta nova tentativa não pedirá senha devido ao arquivo pgpass:

```
$ psql -p 5433
```

Tentativa de conexão com o usuário zezinho:

```
$ psql -U zezinho -p 5433 postgres

psql: FATAL:  password authentication failed for user "zezinho"
password retrieved from file "/home/usuario/.pgpass"
```

O arquivo de senhas foi criado propositalmente com a senha errada para o papel `zezinho`. Por isso houve um erro relacionado a senha incorreta.

3.3 Arquivo de Serviço de Conexão

O arquivo de serviço de conexão permite parâmetros de conexão da libpq para serem associados com um único nome de serviço.

Esse nome de serviço pode então ser especificado por uma conexão libpq, e as configurações associadas serão utilizadas.

O nome do serviço pode também ser especificado utilizando a variável de ambiente PGSERVICE.

O arquivo de serviço de conexão pode ser por usuário (~/.pg_service.conf) ou em outro local especificado pela variável de ambiente PGSERVICEFILE ou pode ser para todo o sistema em /etc/pg_service.conf ou no diretório da variável de ambiente PGSYSCONFDIR. Se definições de serviço com mesmo nome existirem tanto no arquivo global (de todo o sistema) e no local (do usuário), o arquivo do usuário terá prioridade.

O arquivo usa o formato INI onde o nome da seção é o nome do serviço e os parâmetros são os de conexão.

Crie o arquivo ~/.pg_service.conf com o seguinte conteúdo:

```
$ cat << EOF > ~/.pg_service.conf
# comentário
# bla bla bla bla
[mydb]
host=localhost
port=5432
user=postgres
dbname=template1
application_name=teste_pg_service
EOF
```

Ajustes no arquivo de autenticação:

```
$ cat << EOF > ${PGDATA}/pg_hba.conf
local all all trust
host all all 127.0.0.1/32 trust
host all all ::1/128 trust
EOF
```

Reload no cluster para aplicar o novo pg_hba.conf:

```
$ pg_ctl reload
```

Testando:

```
$ psql 'service=mydb' -c 'SHOW application_name;'

 application_name
------------------
 teste_pg_service
```

3.4 Aplicatiovs de Gerenciamento

3.4.1 initdb

O `initdb` é um aplicativo cuja função é criar um novo *cluster* de banco de dados PostgreSQL.
Sintaxe:

```
initdb [opção(s)...] [--pgdata | -D] diretório
```

3.4.2 postgres: O Aplicativo servidor

É o servidor de banco de dados PostgreSQL, o aplicativo que gera os processos servidores.

Em versões mais antigas era chamado de `postmaster`, hoje é um alias obsoleto de `postgres`.

3.4.3 pg_ctl: start, stop, restart, reload, status

O pg_ctl é um utilitário para gerenciamento (inicialização, parada, reinicialização, reload, etc...) de clusters PostgreSQL ou exibe o status de um servidor.

Embora o servidor (o aplicativo postgres) possa ser inicializado manualmente, o pg_ctl encapsula tarefas como redirecionamento de saída de log separando adequadamente do terminal e do grupo de processo. Para inicializar um cluster o utilitário `pg_ctl` precisa que seja fornecido o diretório do cluster com o argumento `-D` e em seguida a localização do diretório do cluster ou se esse diretório estiver configurado como valor da variável de ambiente PGDATA não é necessário.

Para parar e reinicializar um cluster via `pg_ctl` a forma geral de sintaxe é a seguinte:

```
pg_ctl stop [-D $PGDATA] [-m SHUTDOWN-MODE]
pg_ctl restart [-D $PGDATA] [-m SHUTDOWN-MODE]
```

Há três modos (*SHUTDOWN-MODE*) de se parar um *cluster* como valor do argumento `-m` ou `--mode`.

Os modos são **s**`mart`, **f**`ast` ou **i**`mmediate`, ou a primeira letra de cada um dos três.

Para ilustrar melhor, uma variável de ambiente (`PGPID`) é criada para armazenar os PIDs de processos do postgres:

```
$ PGPID=`pidof postgres`
```

3.4.3.1 Modos para start e restart

- **smart** (padrão) - `SIGTERM -15`:

Espera todos clientes ativos se desconectem e qualquer backup online finalizar.

Se o servidor estiver em hot standby, recovery e replicação streaming será terminado uma vez que todos clientes tenham desconectado.

Equivalentes do usando o comando `kill`:

```
kill ${PGPID}
kill -15 ${PGPID}
kill -s TERM ${PGPID}
```

- **fast** - SIGINT -2:

Não espera pela desconexão dos clientes e termina *backup on line* em progresso.
É dado ROLLBACK em todas transações ativas, todos clientes são forçadamente desconectados e então o servidor é parado.

Equivalentes do usando o comando `kill`:

```
kill -2 ${PGPID}
kill -s INT ${PGPID}
```

- **immediate**: SIGQUIT -3

Aborta todos processos servidores imediatamente sem fazer uma parada limpa.
Isso faz com que rode uma recuperação de desastre (*crash-recovery*) na próxima vez que o servidor subir através dos logs de transação.

Equivalentes do usando o comando `kill`:

```
kill -3 ${PGPID}
kill -s QUIT ${PGPID}
```

3.4.3.2 reload

Simplesmente envia um sinal SIGHUP ao processo `postgres`, fazendo com que os arquivos de configuração sejam lidos novamente.

Isso permite que se mude opções de arquivos de configuração que não requerem um *restart* para serem aplicadas as mudanças feitas.

Equivalentes do usando o comando `kill`:

```
kill -1 ${PGPID}
kill -s HUP ${PGPID}
```

Aviso:

Se possível, não use SIGKILL para matar o processo principal do `postgres`!

Se o fizer, impedirá que o `postgres` libere recursos do sistema (e. g.: memória compartilhada e semáforos) antes de terminá-los.

Isso pode causar problemas para inicializar uma nova instância de processo do `postgres`.

Criação de um cluster de teste:

```
$ initdb -D /tmp/data -U postgres
```

Modificando o postgresql.conf para que o cluster escute na porta 5433:

```
$ sed -i 's/#port = 5432/port = 5433/g' /tmp/data/postgresql.conf
```

Inicializando o cluster especificando o diretório do cluster:

```
$ pg_ctl -D /tmp/data/ start
```

Testando...:

```
$ psql -U postgres -p 5433 -c 'SHOW port;'

 port
------
 5433
```

Parando o cluster de teste:

```
$ pg_ctl -D /tmp/data/ stop
```

Mudando a variável de ambiente PGDATA para o diretório do cluster de teste:

```
$ PGDATA='/tmp/data'
```

Inicializando o novo cluster sem precisar especificar o diretório:

```
$ pg_ctl start
```

Testando...:

```
$ psql -U postgres -p 5433 -c 'SHOW port;'

 port
------
 5433

$ pg_ctl stop
```

3.5 Arquivos Físicos e OID

Arquivos físicos da base são os arquivos de dados. São os arquivos que conforme o OID (*Object ID*) representam um determinado objeto do banco e esse OID é o próprio nome do arquivo.

Cada base do *cluster* também tem um OID, mas sua representação é como diretório e não como arquivo, porém o nome do diretório é também o OID da base.

3.5.1 Filenodes e OID

Filenode é o nome do arquivo físico que representa uma tabela, índice, sequência e etc.

OID é a identificação de um objeto, um número.

A princípio filenode e OID são iguais, mas se o arquivo físico que representa o objeto mudar o OID permanece o mesmo e o filenode terá outro número para o objeto, por exemplo, quando se faz um `VACUUM FULL` em uma tabela, ou também um `TRUNCATE`.

Como usuário postgres entrar no psql:

```
$ psql
```

ID de uma base de dados:

```
> SELECT datid
    FROM pg_stat_database
    WHERE datname = 'postgres';

 datid
-------
 12332
```

```
> SELECT oid
    FROM pg_database
    WHERE datname = 'postgres';

  oid
-------
 12332
```

O ID do banco é o nome da pasta que conterá seus objetos.

O Catálogo pg_class

É uma tabela de sistema que guarda informações sobre alguns tipos (classes) de relações (*relations*) que podem ser:

c = tipo composto
i = índice
m = view materializada
p = tabela particionada
r = tabela comum
S = sequência
t = tabela TOAST
v = visão

Criação de tabela de teste:

```
> CREATE TABLE tb_tmp(id serial PRIMARY KEY);
```

Verificando a estrutura da tabela:

```
> \d tb_tmp

                          Table "public.tb_tmp"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 id     | integer |           | not null | nextval('tb_tmp_id_seq'::regclass)
Indexes:
    "tb_tmp_pkey" PRIMARY KEY, btree (id)
```

Nota-se que por causa da declaração do campo id como serial é um inteiro com um valor default atrelado a uma sequence; tb_tmp_id_seq.
Essa tabela também tem um índice; tb_tmp_pkey.

Consultando no catálogo pg_class conforme a tabela de teste criada:

```
> SELECT oid, relfilenode, relname, relkind
    FROM pg_class WHERE relname ~ 'tb_tmp' ORDER BY oid;

  oid  | relfilenode |    relname    | relkind
-------+-------------+---------------+---------
 17727 |       17727 | tb_tmp_id_seq | S
 17729 |       17729 | tb_tmp        | r
 17733 |       17733 | tb_tmp_pkey   | i
```

ID, arquivo de dados, nome e tipo de relação.
Observa-se que até então oid e relfilenode são iguais.

De posse do datid, acessando seu respectivo diretório:

```
$ ls $PGDATA/base/<id da base de dados>
```

ou utilizando Shell Script para o mesmo fim:

```
$ ls $PGDATA/base/`psql -Aqtc \
"SELECT datid FROM pg_stat_database WHERE datname = 'postgres';"`
```

...

Procure os arquivos cujos nomes são os relfilenodes da tabela, da sequence e do índice.

VACUUM FULL na tabela de teste:

```
> VACUUM FULL tb_tmp;
```

Consultando no catálogo pg_class conforme a tabela de teste criada:

```
> SELECT oid, relfilenode, relname, relkind
    FROM pg_class WHERE relname ~ 'tb_tmp' ORDER BY oid;
```

```
  oid  | relfilenode |    relname    | relkind
-------+-------------+---------------+---------
 17727 |       17727 | tb_tmp_id_seq | S
 17729 |       17747 | tb_tmp        | r
 17733 |       17750 | tb_tmp_pkey   | i
```

Observa-se agora que para a tabela e para o índice os relfilenodes mudaram.

Consultando no catálogo pg_class conforme a tabela de teste criada:

```
> TRUNCATE tb_tmp RESTART IDENTITY;
```

Consultando no catálogo pg_class conforme a tabela de teste criada:

```
> SELECT oid, relfilenode, relname, relkind
    FROM pg_class WHERE relname ~ 'tb_tmp' ORDER BY oid;
```

```
  oid  | relfilenode |    relname    | relkind
-------+-------------+---------------+---------
 17727 |       17753 | tb_tmp_id_seq | S
 17729 |       17751 | tb_tmp        | r
 17733 |       17752 | tb_tmp_pkey   | i
```

Após o TRUNCATE mudou também o filenode da sequence, por causa da cláusula RESTART IDENTITY.

Para determinar qual é o objeto do arquivo físico:

```
> SELECT
    relname "Nome do objeto",
    CASE relkind
        WHEN 'r' THEN 'tabela comum'
        WHEN 'i' THEN 'índice'
        WHEN 'S' THEN 'sequência'
        WHEN 'v' THEN 'visão'
        WHEN 'c' THEN 'tipo composto'
        WHEN 't' THEN 'tabela TOAST'
        ELSE '---'
    END "Tipo de objeto"
    FROM pg_class WHERE relfilenode = 17750;

 Nome do objeto | Tipo de objeto
----------------+----------------
 tb_tmp_pkey    | índice
```

4 Configuração

- postgresql.conf
- Outras Formas de Configurar Parâmetros
- A view pg_settings
- ALTER SYSTEM

4.1 postgresql.conf

O `postgresql.conf` é o arquivo principal de configurações do PostgreSQL.

É um arquivo cujo formato é `parâmetro = valor`, cada linha.

Comentários são feitos pelo caractere sustenido (`#`).

Algumas configurações especificam um valor de tempo ou de memória.

Cada um desses tem uma unidade implícita, que é ou kilobytes, blocos (normalmente de 8kb), milissegundos, segundos ou minutos.

Unidades padrões podem ser encontradas em `pg_settings.unit`: `s`, `min`, `kB`, `8kB`, `ms`.

O multiplicador de memória é 1024 e não 1000.

Pra facilitar, uma unidade diferente pode também ser especificada explicitamente.

Unidades de memória válidas são `kB` (*kilobytes*), `MB` (*megabytes*) e `GB` (*gigabytes*);

Unidades de tempo válidas são: `ms` (milissegundos), `s` (segundos), `min` (minutos), `h` (horas) e `d` (dias).

4.1.1 Tipos de Valores em postgresql.conf

- **`bool`** - Valores "booleanos": verdadeiro (`on`, `true`, `1`) ou falso (`off`, `false`, `0`);
- **`enum`** - Enumeração: é aceito um valor de uma lista permitida do parâmetro. Os valores permitidos podem ser encontrados em `pg_settings.enumvals` e seus valores são *case-insensitive*;
- **`string`** - Texto: seu valor deve ser envolvido por apóstrofos (' ');
- **`integer`** - Inteiro: valor numérico inteiro;
- **`real`** - Real: valor númerico real, ou seja, são usadas casas decimais, cujo caractere utilizado como separadaor de casas decimais é o ponto.

4.1.2 A Directiva include

Além dos parâmetros de configuração, o `postgresql.conf` pode conter directivas de inclusão (`include`), que especifica outro arquivo para ler e processar como se seu conteúdo estivesse inserido naquele ponto.

Essa funcionalidade permite que um arquivo de configuração seja dividido fisicamente em partes separadas.

Sintaxe:

```
include 'nome_do_arquivo'
```

Se o nome do arquivo não estiver em um caminho absoluto, será tomado como referência o diretório onde está o arquivo referenciador.

É permitido fazer inclusões aninhadas.

Há também a directiva `include_if_exists`, que atua da mesma forma que a directiva `include`, exceto pelo comportamento quando o arquivo referenciado não existe ou não pode ser lido.

A directiva `include` considera isso uma condição de erro, enquanto `include_if_exists` apenas adiciona uma mensagem de log e continua a processar o arquivo referenciador.

O arquivo de configuração é relido sempre que o processo principal de servidor

recebe um sinal `SIGHUP` (que é mais facilmente emitido pelo comando no shell `pg_ctl reload`).

O processo servidor principal também propaga esse sinal para todos os processos atuais que estejam rodando de forma que suas sessões existentes vão já trabalhar com o novo valor. Alternativamente, pode-se mandar o sinal para um único processo diretamente.

Alguns parâmetros podem apenas ser configurados na inicialização do servidor; quaisquer mudanças a suas entradas no arquivo de configuração serão ignoradas até que o servidor seja reiniciado.

Parâmetros de configuração inválidos no arquivo de configuração são igualmente ignorados (mas logados) durante o processamento de `SIGHUP`.

4.1.2.1 Contexto de Parâmetros do postgresql.conf

Contexto	**Como as Configurações Podem ser Mudadas**
`internal`	Só por compilação ou opções do initdb.
`postmaster`	Só por start ou restart.
`sighup`	Um reload no serviço é suficiente.
`backend`	Pode ser alterada usando a variável de ambiente `PGOPTIONS`.
`superuser-backend`	Igual a backend, mas somente para superusuário.
`user`	Pode ser alterada na sessão via comando SET.
`superuser`	Igual a user, mas somente para superusuário.

4.2 Outras Formas de Configurar Parâmetros

Uma segunda forma de definir esses parâmetros de configuração é fornecê-los como opção de linha de comando do utilitário `postgres`.

Passando parâmetros de configuração ao daemon:

```
$ postgres -c log_connections=yes -c log_destination='syslog'
```

Aviso:

Ao utilizar opções de linha de comando sobrescreve as configurações no `postgresql.conf`.

O que significa não poder alterá-las.

Ocasionalmente isso é útil para uma sessão particular apenas.

A variável de ambiente `PGOPTIONS` pode ser utilizada para esse propósito no lado do cliente:

```
$ export PGOPTIONS='-c geqo=off' psql
```

Isso funciona para qualquer aplicação cliente baseada na libpq, não apenas o psql.

Vale lembrar que só serão aceitos parâmetros ajustáveis em sessões. Além disso, é possível atribuir um conjunto de configurações de parâmetros para um usuário ou base de dados.

Sempre que uma sessão é iniciada, as configurações padrões para o usuário e base de dados são carregadas.

Os comandos `ALTER ROLE` e `ALTER DATABASE`, respectivamente, são usados para isso.

Configurações por base de dados sobrescrevem tudo vindo de linha de comando ou do arquivo de configuração, e por sua vez são substituídas pelas configurações por usuário, ambas são substituídas pelas configurações por sessão.

4.2.1 Examinando Configurações de Parâmetros

O comando `SHOW` permite inspecionar o valor atual de todos parâmetros.

A tabela virtual `pg_settings` também permite exibir e atualizar parâmetros em tempo de execução; `pg_settings` é equivalente a `SHOW` e `SET`, mas pode ser mais conveniente em usá-la fazendo junções com outras tabelas, ou qualquer outra consulta que for útil utilizá-la. Nela também contém mais informações sobre cada parâmetro do que o que está disponível em `SHOW`.

4.2.1.1 SHOW

Sua única função é exibir o valor de uma determinada configuração do PostgreSQL.

Exibir o valor da porta de conexão:

```
> SHOW port;

 port
------
5432
```

4.2.1.2 SET

O comando `SET` altera qualquer parâmetro de configuração que possa ser mudado dentro de uma sessão, além de sobrescrever qualquer outro meio de configuração.

Definição de parâmetro dentro de uma sessão:

```
> SET application_name = 'minha_aplicacao';
```

4.3 A view pg_settings

Nessa view não se pode inserir ou deletar linhas, mas alterações são aceitas.

Um `UPDATE` aplicado a um registro dela é equivalente a executar o comando `SET` em um dado parâmetro.

A alteração apenas afeta o valor usado pela sessão atual.

Se um `UPDATE` for emitido dentro de uma transação que depois for abortada, os efeitos do comando `UPDATE` desaparecerão quando a transação for revertida (`ROLLBACK`).

Uma vez que a transação que envolve o comando for efetivada, os efeitos persistirão até o fim da sessão, a não ser que sejam sobrescritos por outro `UPDATE` ou `SET`.

Nome	Tipo	Descrição
name	text	Nome do parâmetro de configuração.
setting		Atual valor do parâmetro.
unit		Unidade implícita do parâmetro.
category		Grupo lógico do parâmetro.
short_desc		Descrição curta do parâmetro.
extra_desc		Descrição do parâmetro mais detalhada.
context		Contexto requerido para ajustar o valor do parâmetro.
vartype		Tipo de valor do parâmetro (bool, enum, integer, real, or string).
source		Origem do valor atual.
min_val		Valor mínimo permitido para ajustar o parâmetro (nulo para valores não numéricos).
max_val		Valor máximo permitido para ajustar o parâmetro (nulo para valores não numéricos).
enumvals	text[]	Valores permitidos de um parâmetro enumerado (nulo para valores não enumerados).
boot_val	text	Valor do parâmetro assumido na inicialização do servidor se não foi de outra forma definido.
reset_val		Ao se usar o comando RESET parametro, o parâmetro declarado assume este valor.
sourcefile		Arquivo de configuração em que o valor atual foi definido (nulo para valores definidos de outras origens além de arquivos de configuração ou quando inspecionado por um não superuser). Muito útil quando se usa a directiva include em arquivos de configuração.
sourceline	integer	Número da linha no arquivo de configuração que a configuração atual está situada (nulo para valores ajustados por origens que não sejam arquivos de configuração, ou quando examinado por um não super user).

Tabela 1: Siginificado de cada campo da view pg_settings

Através do comando SHOW, exibir o valor de shared_buffers:

```
> SHOW shared_buffers;

 shared_buffers
----------------
 24MB
```

O comando `SHOW` exibiu o valor de `shared_buffers` de forma “amigável”, uma forma mais humanamente legível.

Pela view pg_settings verificar valor, contexto, tipo de valor e unidade:

```
> SELECT setting, context, vartype, unit FROM pg_settings
    WHERE name = 'shared_buffers';

 setting |  context   | vartype | unit
---------+------------+---------+------
 3072    | postmaster | integer | 8kB
```

O valor de shared_buffers exibido aqui é de 3072 páginas de 8kB.

O tipo de valor é inteiro e é um parâmetro que seu valor só terá efeito após reinicializar o serviço do PostgreSQL.

Convertendo o valor para MB:

```
> SELECT
    ((setting::int * 8) / 1024) "shared_buffers (MB)"
    FROM pg_settings WHERE name = 'shared_buffers';

 shared_buffers (MB)
---------------------
                  24
```

3072 * 8 nos dá o resultado em kB, então para sabermos o equivalente em MB dividimos por 1024.

Visualizando o parâmetro application_name:

```
> SHOW application_name;

 application_name
------------------
 psql
```

Alterando o valor do parâmetro application_name:

```
> SET application_name = 'curso_adm_postgres';
```

Valor, contexto, tipo de valor e unidade de application_name:

```
> SELECT setting, context, vartype, unit FROM pg_settings
    WHERE name = 'application_name';

      setting       | context | vartype | unit
--------------------+---------+---------+------
 curso_adm_postgres | user    | string  |
```

Uma modificação feita com `SET` podemos visualizar com uma consulta em `pg_settings` em vez de usarmos `SHOW`.

`application_name` é um parâmetro que pode ser mudado por sessão, o tipo de valor é uma string e não há unidade para esse parâmetro.

A `view pg_settings` nos dá maiores detalhes o que para muitas situações é muito útil.

4.4 ALTER SYSTEM

Um dos interessantes recursos vindos com a versão 9.4 foi a possibilidade de podermos alterar qualquer configuração do PostgreSQL via comando SQL, dispensando a necessidade de editar o arquivo *postgresql.conf*.

Um novo arquivo auxiliar foi introduzido nessa versão; o *postgresql.auto.conf*.

Funciona da seguinte forma; o servidor dará preferência a esse arquivo. As configurações que nele estiverem declaradas terão prioridade com relação às mesmas do *postgresql.conf*.

O comando `ALTER SYSTEM` faz as alterações de configurações escritas no *postgresql.auto.conf*.

Verificando em qual porta o PostgreSQL está escutando:

```
$ netstat -nltp | fgrep postgres

(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp    0      0 127.0.0.1:5432    0.0.0.0:*    LISTEN      3947/postgres
```

Para fins de testes acrescentar uma linha manualmente ao arquivo postgresql.auto.conf:

```
$ echo 'port = 5433' >> ${PGDATA}/postgresql.auto.conf
```

Exibir o conteúdo do arquivo auxiliar postgresql.auto.conf:

```
$ cat ${PGDATA}/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
port = 5433
```

Reiniciando o serviço do PostgreSQL:

```
$ pg_ctl restart
```

Verificando em qual porta o PostgreSQL está escutando:

```
$ netstat -nltp | fgrep postgres

(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)

tcp    0    0    127.0.0.1:5433    0.0.0.0:*    LISTEN    3964/postgres
```

No próprio servidor se conectar pela porta 5433:

```
$
```

Verificando via SQL em qual porta o PostgreSQL está escutando:

```
> SHOW port;

 port
------
 5433
```

Alterando a porta de escuta do serviço para 5432:

```
> ALTER SYSTEM SET port = 5432;
```

Reiniciando o serviço:

```
$ pg_ctl restart
```

Verificando em qual porta o PostgreSQL está escutando:

```
$ netstat -nltp | fgrep postgres

(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)

tcp    0    0    127.0.0.1:5432    0.0.0.0:*    LISTEN    4018/postgres
```

Exibindo o conteúdo de postgresql.auto.conf:

```
$ cat ${PGDATA}/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
port = '5432'
```

Alteração do parâmetro log_min_duration_statement:

```
> ALTER SYSTEM SET log_min_duration_statement = '5s';
```

Dentro do psql executar o comando de shell para visualizar o conteúdo de postgresql.auto.conf:

```
> \! cat ${PGDATA}/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
port = '5432'
log_min_duration_statement = '5s'
```

Isso não faz a mudança no postgresql conf na mesma hora. Em vez disso ele escreve a configuração em um arquivo chamado postgresql.auto.conf

Isso não muda o *postgresql.conf*. Ao invés disso a configuração é escrita para o arquivo *postgresql.auto.conf*. Esse arquivo **sempre** será lido por último, o que faz com que qualquer parâmetro de configuração nele declarado sobrescreva o que está no *postgresql.conf*.

Mudar port para o padrão:

```
> ALTER SYSTEM SET port TO DEFAULT;
```

Dentro do psql executar o comando de shell para visualizar o conteúdo de postgresql.auto.conf:

```
> \! cat ${PGDATA}/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
log_min_duration_statement = '5s'
```

Mudar log_min_duration_statement para o padrão:

```
> ALTER SYSTEM SET log_min_duration_statement TO DEFAULT;
```

Dentro do psql executar o comando de shell para visualizar o conteúdo de postgresql.auto.conf:

```
> \! cat ${PGDATA}/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
```

Uma vantagem de fazer mudanças utilizando `ALTER SYSTEM` é a questão da segurança no sentido de que se um parâmetro for configurado com um valor inválido, a mudança não será efetivada. O que evita de em caso de um restart, o servidor não subir devido a uma má configuração.

Tentativa de alterar o parâmetro wal_level com um valor não permitido:

```
> ALTER SYSTEM SET wal_level = 'cold_standby';

ERROR:  invalid value for parameter "wal_level": "cold_standby"
HINT:  Available values: minimal, archive, hot_standby, logical.
```

5 Tablespaces

- Conceito
- Como Criar um Tablespace

5.1 Conceito

Tablespace é a localização no sistema de arquivos, onde objetos do banco de dados são criados.

Tal recurso permite que administradores de banco de dados criem e/ou alterem objetos em tablespaces diferentes do padrão.

O tablespace padrão é no diretório `PGDATA`.

Uma vez criado, o *tablespace* pode ser referido pelo seu nome.

Tablespaces são muito úteis para gerenciamento de armazenamento e performance de discos, por exemplo, uma determinada tabela que é muito mais acessada em um sistema do que as demais, talvez seja o caso de alocá-la em um tablespace, cujo diretório seja um ponto de montagem (Unix *like*) em um outro disco, de forma a evitar concorrência de I/O. Ou seja, é um recurso extremamente útil na obtenção de maior desempenho.

5.2 Como Criar um Tablespace

O(s) diretório(s) a se(rem) utilizado(s) como tablespace(s) deve(m) ter permissão de escrita para o usuário postgres. No nosso exemplo, vamos criar dentro do diretório do usuário postgres os diretórios que serão tablespaces.

Voltando ao diretório do usuário:

```
$ cd ${HOME}
```

Criação de diretórios para tablespaces:

```
$ mkdir -p ${PGVERSION}/tablespaces/{ts0,ts1,ts2}
```

Criação de tablespace:

```
$ psql -c "CREATE TABLESPACE ts_alpha \
LOCATION '/var/lib/postgresql/${PGVERSION}/tablespaces/ts0';"
```

Criação de tablespace:

```
$ psql -c "CREATE TABLESPACE ts_beta \
LOCATION '/var/lib/postgresql/${PGVERSION}/tablespaces/ts1';"
```

Criação de tablespace:

```
$ psql -c "CREATE TABLESPACE ts_gama LOCATION \
 '/var/lib/postgresql/${PGVERSION}/tablespaces/ts2';"
```

Listando tablespaces

```
$ psql -c '\db'

                       List of tablespaces
    Name    |  Owner   |                Location
------------+----------+----------------------------------------
 pg_default | postgres |
 pg_global  | postgres |
 ts_alpha   | postgres | /var/lib/postgresql/9.3/tablespaces/ts0
 ts_beta    | postgres | /var/lib/postgresql/9.3/tablespaces/ts1
 ts_gama    | postgres | /var/lib/postgresql/9.3/tablespaces/ts2
```

Aviso:

`LOCATION` deve ser um caminho absoluto.

Utilitário psql:

```
$ psql
```

Criação de uma nova tabela com tablespace definido:

```
> CREATE TABLE tb_teste_tablespace(
  campo1 INT2,
  campo2 TEXT,
  CONSTRAINT pk_campo1_teste_tablespace PRIMARY KEY (campo1))
  TABLESPACE ts_alpha;

NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"pk_campo1_teste_tablespace" for table "tb_teste_tablespace"
CREATE TABLE
```

Verificando em qual tablespace está o índice da tabela recém criada:

```
> SELECT tablespace
    FROM pg_indexes WHERE indexname = 'pk_campo1_teste_tablespace';

 tablespace
------------
```

Apagando a tabela:

```
> DROP TABLE tb_teste_tablespace;
```

Recriando a tabela com tablespace para índices definido:

```
> CREATE TABLE tb_teste_tablespace(
    campo1 INT2,
    campo2 TEXT,
    CONSTRAINT pk_campo1_teste_tablespace PRIMARY KEY (campo1)
    USING INDEX TABLESPACE ts_beta)
    TABLESPACE ts_alpha;
```

Verificando em qual *tablespace* está o índice da tabela recém-criada:

```
> SELECT tablespace FROM pg_indexes WHERE indexname = 'pk_campo1_teste_tablespace';

 tablespace
------------
 ts_beta
```

Verficando em qual tablespace está a tabela criada:

```
> SELECT tablespace FROM pg_tables WHERE tablename = 'tb_teste_tablespace';

 tablespace
------------
 ts_alpha
```

Alterando o tablespace da tabela:

```
> ALTER TABLE tb_teste_tablespace SET TABLESPACE ts_gama;
```

Nova verificação de tablespace da tabela:

```
> SELECT tablespace FROM pg_tables WHERE tablename = 'tb_teste_tablespace';

 tablespace
------------
 ts_gama
```

Obs.:

No diretório `$PGDATA/pg_tblspc` há um link para cada *tablespace* criado.

---

6 Roles

- Sobre Papéis (Roles)
- Atributos
- Parâmetros de Configuração por Papel

6.1 Sobre Papéis (Roles)

A palavra “papel” é no sentido de função, qual papel desempenhará no sistema gerenciador de banco de dados.

O comando `CREATE ROLE` adiciona um novo papel ao *cluster* do PostgreSQL.

Um papel é uma entidade que pode possuir objetos de banco de dados e ter privilégios de banco de dados; um papel pode ser considerado um usuário, um grupo, ou ambos dependendo de como é usado.

É necessário ter o privilégio `CREATEROLE` ou ser superusuário para usar este comando.

Obs.:

Papéis são definidos no nível de *cluster*, e então são válidos em todas as bases de dados no *cluster*.

Obs.:

Cada objeto criado por um papel, faz com que esse objeto seja propriedade dele. Ou seja, esse papel tem poder total sobre esse objeto.

Obs.:

O prompt do psql muda conforme o tipo de papel: normal ou superusuário.

Para papéis comuns: `nome_da_base=>`

Para superusuários: `nome_da_base=#`

- **Usuários**

  É o tipo de papel (*role*), cuja função é operar o sistema através de login.

- **Grupos**

  É o tipo de papel que agrupa outros papéis.

6.2 Atributos

Um papel de banco de dados tem um certo número de atributos que definem seus privilégios e interage com sistema de autenticação de clientes.

- **SUPERUSER / NOSUPERUSER**

Determinam o se o novo papel é um “superusuário”, que sobrescreve todas restrições de acesso internamente no banco de dados. O status de superusuário é perigoso e deve ser usado apendas quando realmente for necessário.

Para criar um superusuário é preciso ser um. Se não for especificado, `NOSUPERUSER` é o padrão.

- **CREATEDB / NOCREATEDB**

Permite ou não ao papel criar bases de dados.

Se `CREATEDB` for especificado, o papel poderá criar novas bases de dados.

Especificando `NOCREATEDB` negará ao papel a habilidade de criar bases de dados.

O valor padrão é `NOCREATEDB`.

- **CREATEROLE / NOCREATEROLE**

Permite ou não a criação de outros papéis.

Um papel com o privilégio `CREATEROLE` pode também alterar e apagar outros papéis.

O valor padrão é `NOCREATEROLE`.

- **CREATEUSER / NOCREATEUSER**

Essas cláusulas, apesar de ainda aceitas, são obsoletas.

Note que não são equivalentes a `CREATEROLE / NOCREATEROLE` como talvez pudesse se supor.

- **INHERIT / NOINHERIT**

Permite ou não se um papel herda os privilégios de papéis que forem membros de.

Um papel com o atributo `INHERIT` pode automaticamente usar seja qual for o privilégio de banco de dados ter sido concedido para todos os papéis que ele for membro diretamente.

Sem `INHERIT`, associação em outro papel apenas concede a habilidade `SET ROLE` para que outro papel; os privilégios de outro papel estejam disponíveis apenas depois de terem feito então.

Seu valor padrão é `INHERIT`.

- **LOGIN / NOLOGIN**

Determina se um papel pode ou não logar, privilégio no qual um papel pode obter uma autorização inicial de uma sessão durante a conexão de um cliente.

Um papel tendo o atributo `LOGIN` pode ser pensado como um usuário. Papéis sem esse atributo são úteis para gerenciamento de privilégios de bancos de dados, mas não são usuários propriamente ditos, mas sim algo como grupos. Por padrão é `NOLOGIN`, exceto quando `CREATE ROLE` for invocado pela sua forma alternativa `CREATE USER`.

- **REPLICATION / NOREPLICATION**

Permite ou não a um papel iniciar uma replicação via *streaming* ou colocar o sistema em modo de backup.

Um papel que tem o atributo `REPLICATION` é um papel altamente privilegiado, e deve ser apenas usado para replicação.

`NOREPLICATION` é o valor padrão.

- **CONNECTION LIMIT**

Se um papel tiver o privilégio `LOGIN` (poder se logar), esse atributo especifica quantas conexões simultâneas o papel pode fazer.

O valor padrão é `-1`, que significa sem limite.

- **PASSWORD**

Só se aplica a papéis que tem o atributo `LOGIN`, sua função é definir uma senha para o papel. Mesmo assim é possível usar em papéis que não tenham o atribulo `LOGIN`.

Se não pretende usar autenticação por senha, essa opção pode ser omitida. Se nenhuma senha for especificada, a mesma será nula e a autenticação sempre falhará para o papel.

Uma senha nula pode ser explicitamente definida como `PASSWORD NULL`.

- **ENCRYPTED / UNENCRYPTED**

Essas palavras-chave controlam se a senha é armazenada encriptada no catálogo de sistema.

Se nenhum for especificado, o comportamento padrão será determinado pela configuração `password_encryption`, no arquivo `postgresql.conf`.

Se a *string* de senha já estiver no formato encriptado MD5, então ela será armazenada como está, independente se for especificado `ENCRYPTED` ou `UNENCRYPTED`.

É possível que clientes mais antigos não tenham suporte a autenticação MD5.

- **VALID UNTIL**

  Configura a data e hora que após o papel não será mais válido.
  Por omissão é infinito.

- **IN ROLE**

Lista um ou mais papéis existentes que o novo papel será imediatamente adicionado como um novo membro.
Não há opção para adicionar o novo papel como um administrador, então deve-se usar o comando `GRANT` separadamente para fazer isso.

- **IN GROUP**

  É uma forma obsoleta de se escrever `IN ROLE`.

- **ROLE**

Esta cláusula lista um ou mais papéis existentes que serão automaticamente adicionados como membros do novo papel.
O que na prática, tem o mesmo efeito como um grupo.

- **ADMIN**

A cláusula `ADMIN` é como `ROLE`, mas os nomes dos papéis são adicionados para o novo papel com a opção `WITH ADMIN`, dando a eles o direito para conceder filiação nesse papel para outros.

- **USER**

  É uma forma obsoleta de escrever a cláusula `ROLE`.

- **SYSID**

  Esta cláusula é ignorada, mas é aceita para compatibilidade retroativa.

Como usuário postgres utilizar o psql:

```
$ psql
```

Criação do papel financeiro:

```
> CREATE ROLE financeiro;
```

Criação do papel comercial:

```
> CREATE ROLE comercial;
```

Criação do papel marcia com LOGIN:

```
> CREATE ROLE marcia LOGIN;
```

Criação do papel jose com LOGIN:

```
> CREATE ROLE jose LOGIN;
```

Criação do papel silvana com LOGIN:

```
> CREATE ROLE silvana LOGIN;
```

Criação do papel chiquinho com LOGIN e já como membro dos papéis comercial e financeiro:

```
> CREATE ROLE chiquinho LOGIN IN ROLE comercial, financeiro;
```

Criação do papel g_teste_priv:

```
> CREATE ROLE g_teste_priv;
```

Comando para exibir os papéis:

```
> \du
```

```
                                   List of roles
   Role name   |                   Attributes                   |       Member of
---------------+------------------------------------------------+------------------------
 chiquinho     |                                                | {financeiro,comercial}
 comercial     | Cannot login                                   | {}
 g_teste_priv  | Cannot login                                   | {}
 financeiro    | Cannot login                                   | {}
 jose          |                                                | {}
 marcia        |                                                | {}
 postgres      | Superuser, Create role, Create DB, Replication | {}
 silvana       |                                                | {}
```

Atribuindo senha para o papel chiquinho:

```
> ALTER ROLE chiquinho PASSWORD '123';
```

Atribuindo senha para o papel jose:

```
> ALTER ROLE jose PASSWORD '123';
```

Atribuindo senha para o papel marcia:

```
> ALTER ROLE marcia PASSWORD '123';
```

Atribuindo senha para o papel silvana:

```
> ALTER ROLE silvana PASSWORD '123';
```

Verificando o usuário atual:

```
> SELECT current_user;

 current_user
--------------
 postgres
```

Mudando o usuário atual:

```
> SET role marcia;
```

Verificando o usuário atual:

```
> SELECT current_user;

 current_user
--------------
 marcia
```

6.3 Parâmetros de Configuração por Papel

Em alguns casos é interessante ter configurações para papéis (grupos e / ou usuários), como para fins de auditoria, por exemplo.

Criação do papel de teste:

```
> CREATE ROLE user_foo LOGIN;
```

Estrutura da view de papéis:

```
> \d pg_roles
```

```
                        View "pg_catalog.pg_roles"
     Column     |           Type           | Collation | Nullable | Default
----------------+--------------------------+-----------+----------+---------
 rolname        | name                     |           |          |
 rolsuper       | boolean                  |           |          |
 rolinherit     | boolean                  |           |          |
 rolcreaterole  | boolean                  |           |          |
 rolcreatedb    | boolean                  |           |          |
 rolcanlogin    | boolean                  |           |          |
 rolreplication | boolean                  |           |          |
 rolconnlimit   | integer                  |           |          |
 rolpassword    | text                     |           |          |
 rolvaliduntil  | timestamp with time zone |           |          |
 rolbypassrls   | boolean                  |           |          |
 rolconfig      | text[]                   |           |          |
 oid            | oid                      |           |          |
```

Alterando parâmetros do usuário:

```
> ALTER ROLE user_foo SET work_mem = '77MB';

> ALTER ROLE user_foo SET log_min_duration_statement = 0;
```

Verificando os parâmetros ajustados na view:

```
> SELECT rolconfig FROM pg_roles WHERE rolname = 'user_foo';
```

```
                 rolconfig
-------------------------------------------
 {work_mem=77MB,log_min_duration_statement=0}
```

Conectando ao banco modelo template1 como user_foo:

```
> \c template1 user_foo
```

Com o próprio papel, verificando os parâmetros:

```
> SHOW log_min_duration_statement;

 log_min_duration_statement
----------------------------
 0

> SHOW work_mem;

 work_mem
----------
 77MB
```

ou

```
> SELECT name, pg_size_pretty(setting::int8 * 1024)
    FROM pg_settings
    WHERE name IN ('log_min_duration_statement', 'work_mem');

            name            | pg_size_pretty
----------------------------+----------------
 log_min_duration_statement | 0 bytes
 work_mem                   | 77 MB
```

E se o próprio usuário com esse papel quiser alterar algum parâmetro?:

```
> ALTER ROLE user_foo SET log_min_duration_statement  = -1;

ERROR:  permission denied to set parameter "log_min_duration_statement"
```

7 Privilégios

- Sobre Privilégios
- REASSIGN OWNED
- Permissões em Schemas

7.1 Sobre Privilégios

Quando um objeto é criado, é assimilado um dono a ele, que normalmente o papel que executou o comando de criação.

Para a maioria dos tipos de objetos, o estado inicial é que apenas o dono (ou um superusuário) possa fazer qualquer coisa com o objeto.

Para permitir outros papéis a utilizar esse objeto, privilégios têm que ser concedidos.

Os privilégios que são aplicáveis a um objeto em particular variam dependendo do tipo de objeto (tabela, função, etc.).

Somente o dono do objeto pode modificá-lo ou destruí-lo (DDL).

Um objeto pode ser assimilado a um novo proprietário com o comando `ALTER` do tipo apropriado de objeto, e. g. `ALTER TABLE`.

Superusuários podem sempre fazer isso; um papel comum só poderá fazer se for o atual proprietário do objeto (ou membro de um grupo proprietário) e um membro do novo papel proprietário.

7.1.1 Tipos de privilégios

- **SELECT (r)**

Permite `SELECT` em qualquer coluna, ou colunas especificadas por lista, de uma tabela, view ou sequência.

Também permite o uso de `COPY TO`. Esse privilégio é também necessário para referenciar valores existentes de colunas em `UPDATE` ou `DELETE`.

Para sequências, este privilégio também permite o uso da função currval. Para grandes objetos, permite que o mesmo seja lido.

- **INSERT (a)**

Permite INSERT de um novo registro em uma tabela.

Se colunas específicas forem listadas, apenas nessas será assimilado o privilégio, enquanto que as outras colunas receberão valores padrões. Também permite `COPY FROM`.

- **UPDATE (w)**

Permite `UPDATE` de qualquer coluna, ou em colunas especificadas por lista, de uma tabela especifica.

Na prática, qualquer `UPDATE` fora do comum vai requerer também o privilégio `SELECT`, uma vez que as colunas de uma tabela devem ser referenciadas para determinar quais linhas devem ser atualizadas, e / ou para calcular novos valores para as colunas.

`SELECT ... FOR UPDATE` e `SELECT ... FOR SHARE` também requerem esse privilégio em, pelo menos, uma coluna, em adição ao privilégio `SELECT`.

Para sequências, este privilégio permite o uso das funções `nextval` e `setval`. Para grandes objetos (BLOBs) este privilégio permite gravar ou truncar o objeto.

- **DELETE (d)**

  Permite `DELETE` de registros em uma tabela especificada.

  Na prática, qualquer `DELETE` fora do comum vai requerer também o privilégio `SELECT`, uma vez que as colunas de uma tabela devem ser referenciadas para determinar que linhas serão apagadas.

- **TRUNCATE (D)**

  Permite o comando `TRUNCATE` em uma tabela.

- **REFERENCES (x)**

  Para criar uma restrição (*constraint*) chave estrangeira (*foreign key*), é necessário ter este privilégio em ambas as colunas referenciadas.

  O privilégio deve ser concedido para todas as colunas de uma tabela ou apenas em específicas.

- **TRIGGER (t)**

  Permite a criação de um gatilho (*trigger*) em uma tabela especificada.

- **CREATE (C)**

  Para bases de dados, permite novos esquemas serem criados internamente.

  Para esquemas, permite que novos objetos sejam criados dentro dele.

  Para renomear um objeto que já existe, deve ser o proprietário do objeto e ter este privilégio no esquema que o contém.

  Para *tablespaces*, permite tabelas, índices e arquivos temporários serem criados dentro do *tablespace* e permite bases de dados serem criadas que tenham o *tablespace* como padrão.

  Revogando este privilégio não vai alterar a localização de objetos existentes.

- **CONNECT (c)**

  Permite ao usuário se conectar à base especificada.

  Este privilégio é checado ao iniciar a conexão em adição à checagem de quaisquer restrições impostas pelo `pg_hba.conf`.

- **TEMPORARY/TEMP (T)**

  Permite a criação de objetos temporários (tabelas, *views* ou sequências) na base de dados especificada.

- **EXECUTE (X)**

  Permite o uso de uma função especificada e o uso de quaisquer operadores que são implementados no topo da função.

  Isto é o único tipo de privilégio que é aplicável a funções.

- **USAGE (U)**

Para linguagens procedurais, permite o uso de uma linguagem especificada para criação de funções.

Para esquemas, permite acesso a objetos contidos no mesmo (levar em conta também os privilégios dos próprios objetos). Essencialmente isso permite que sejam feitas buscas em objetos dentro do esquema.

Sem essa permissão, ainda é possível ver os nomes de objetos, e. g. consultando tabelas de sistema.

Também, após revogar essa permissão, *backends* existentes podem ter statements que tiveram feito previamente essa busca, então isso não é completamente seguro para prevenir acessos a objetos.

Para sequências, este privilégio permite o uso das funções `currval` e `nextval`.

Para tipos e domínios, este privilégio permite o uso do tipo ou domínio na criação de tabelas, funções e outros objetos de esquema. Observe que ele não controla o uso geral de tipo, tal como valores de tipo aparecendo em consultas. Apenas previne que objetos criados dependam desse tipo.

O principal propósito do privilégio é controlar que usuários criam dependências em um tipo, o que poderia impedir o proprietário de alterar o tipo mais tarde.

Para *foreign-data wrappers*, este privilégio habilita a criar novos servidores que utilizem dados externos encapsulados (*foreign-data wrapper*).

Para servidores, este privilégio permite criar, alterar e descartar usuário de seu próprio mapeamento de usuários associados com esse servidor.

Também habilita a consultar as opções do servidor e mapeamentos de usuários associados.

- **ALL PRIVILEGES (arwdDxt)**

Concede todos os privilégios disponíveis imediatamente.

A palavra-chave PRIVILEGES é opcional no PostgreSQL, no entanto é requerida pelo padrão SQL.

Os privilégios requeridos por outros comandos são listados na página de referência do respectivo comando.

- **GRANT**

Concede privilégios de acesso a objetos para papéis.

- **REVOKE**

Revoga (tira) privilégios de acesso a objetos de papéis.

Criação do banco de dados db_teste_priv (como usuário postgres):

```
> CREATE DATABASE db_teste_priv;
```

Acessando o banco criado:

```
> \c db_teste_priv
```

Criando uma tabela com 15 registros inseridos a partir da função generate_series:

```
> SELECT
    generate_series(1, 15) AS campo1,
    (generate_series(1, 15) * 3) AS campo2
    INTO tb_teste_priv;
```

Verificando privilégios da tabela tb_teste_priv:

```
> \z tb_teste_priv

                              Access privileges
 Schema |     Name      | Type  | Access privileges | Column access privileges
--------+---------------+-------+-------------------+--------------------------
 public | tb_teste_priv | table |                   |
```

Acessando o banco db_teste_priv com o papel marcia:

```
> \c db_teste_priv marcia

FATAL:  Peer authentication failed for user "marcia"
Previous connection kept
```

Não foi possível a conexão devido a um impedimento feito pelo arquivo pg_hba.conf, que é responsável pela autenticação baseada em hosts.

Mudanças no arquivo de autenticação:

```
$ cat << EOF > ${PGDATA}/pg_hba.conf
local    all              postgres              trust
local    db_teste_priv    +g_teste_priv         md5
EOF
```

O sinal de + significa considerar como grupo e não como usuário.

Reload no serviço e conexão em seguida:

```
$ pg_ctl reload && psql
```

Concedendo o direito de membro do papel g_teste_priv a outros papéis:

```
> GRANT g_teste_priv TO chiquinho, jose, marcia, silvana;
```

Nova tentativa de acesso do papel marcia ao banco db_teste_priv:

```
> \c db_teste_priv marcia

Password for user marcia:

You are now connected to database "db_teste_priv" as user "marcia".
db_teste_priv=>
```

Tentativa de consulta aos campos da tabela tb_teste_priv:

```
> SELECT campo1, campo2 FROM tb_teste_priv LIMIT 5;

ERROR:  permission denied for relation tb_teste_priv
```

Em outro terminal com o superusuário postgres, conexão ao banco de testes:

```
> \c db_teste_priv
```

Concedendo o direito de ler (SELECT) na tabela ao papel g_teste_priv:

```
> GRANT SELECT ON tb_teste_priv TO g_teste_priv;
```

Verificando os privilégios de acesso à tabela:

```
> \z tb_teste_priv

                                 Access privileges
 Schema |     Name      | Type  |     Access privileges     | Column access privileges
--------+---------------+-------+---------------------------+--------------------------
 public | tb_teste_priv | table | postgres=arwdDxt/postgres+|
        |               |       | g_teste_priv=r/postgres   |
```

Legenda:

- rolename=xxxx: Privilégios concedidos a um papel;
- =xxxx: Privilégios concedidos a PUBLIC;
- r: SELECT ("read");
- w: UPDATE ("write");
- a: INSERT ("append");
- d: DELETE;
- D: TRUNCATE;
- x: REFERENCES;
- t: TRIGGER;
- X: EXECUTE;
- U: USAGE;
- C: CREATE;
- c: CONNECT;
- T: TEMPORARY;
- arwdDxt: ALL PRIVILEGES (para tabelas, varia para outros objetos);
- *: grant option para o privilégio precedido;
- /yyyy: papel que concedeu esse privilégio.

Voltando ao terminal do papel marcia, nova tentativa de consulta à tabela:

```
> SELECT campo1, campo2 FROM tb_teste_priv LIMIT 5;

 campo1 | campo2
--------+--------
      1 |      3
      2 |      6
      3 |      9
      4 |     12
      5 |     15
```

No terminal do superusuário, revogar (tirar) o direito de leitura na tabela para o papel g_teste_priv:

```
> REVOKE SELECT ON tb_teste_priv FROM g_teste_priv;
```

silvana membro de financeiro:

```
> GRANT financeiro TO silvana;
```

jose e marcia são membros de comercial:

```
> GRANT comercial TO jose, marcia;
```

Concedendo o direito de leitura (SELECT) na coluna campo1 da tabela tb_teste_priv para comercial (e seus membros):

```
> GRANT SELECT (campo1) ON tb_teste_priv TO comercial;
```

Concedendo o direito de leitura (SELECT) na coluna campo2 da tabela tb_teste_priv para financeiro (e seus membros):

```
> GRANT SELECT (campo2) ON tb_teste_priv TO financeiro;
```

Verificando os privilégios de acesso na tabela tb_teste_priv:

```
> \z tb_teste_priv
```

ou

```
> \dp tb_teste_priv
```

```
                                Access privileges
 Schema |     Name      | Type  |     Access privileges     | Column access privileges
--------+---------------+-------+---------------------------+--------------------------
 public | tb_teste_priv | table | postgres=arwdDxt/postgres | campo1:                 +
        |               |       |                           |   comercial=r/postgres  +
        |               |       |                           | campo2:                 +
        |               |       |                           |   financeiro=r/postgres
```

Papel marcia tentando ler todas as colunas da tabela:

```
> SELECT campo1, campo2 FROM tb_teste_priv LIMIT 5;
```

```
ERROR:  permission denied for relation tb_teste_priv
```

Papel marcia fazendo consulta referenciando apenas o campo que pode ler:

```
> SELECT campo1 FROM tb_teste_priv LIMIT 5;

 campo1
--------
      1
      2
      3
      4
      5
```

No exemplo anterior pode-se ver como funciona o sistema de privilégios granulares em tabelas.

Revogando todos privilégios públicos:

```
> REVOKE ALL PRIVILEGES ON DATABASE db_teste_priv FROM PUBLIC;
```

Permissões de acesso à base db_teste_priv:

```
> SELECT datacl AS "Permissões de Acesso" FROM pg_database
    WHERE datname = 'db_teste_priv';

  Permissões de Acesso
------------------------
 {postgres=CTc/postgres}
```

Aviso:

Usuários logados antes de uma revogação de acesso a um banco de dados não são afetados enquanto não logam novamente.

Pelo terminal do superusuário uma nova conexão ao banco db_teste_priv com o papel marcia:

```
> \c db_teste_priv marcia

Password for user marcia:

FATAL:  permission denied for database "db_teste_priv"
DETAIL:  User does not have CONNECT privilege.
Previous connection kept
```

Em um terminal de superusuário conceder os privilégios de acesso ao banco:

```
> GRANT CONNECT, CREATE, TEMPORARY ON DATABASE db_teste_priv TO g_teste_priv;
```

Consultando os privilégios de acesso:

```
> SELECT datacl AS "Permissões de Acesso"
    FROM pg_database WHERE datname = 'db_teste_priv';

               Permissões de Acesso
-------------------------------------------------
 {postgres=CTc/postgres,g_teste_priv=CTc/postgres}
```

Mudando o papel dono da tabela para chiquinho:

```
> ALTER TABLE tb_teste_priv OWNER TO chiquinho;
```

Criação de uma nova tabela:

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

Mudando o dono da tabela:

```
> ALTER TABLE tb_xyz OWNER TO chiquinho;
```

Criando uma nova sequência:

```
> CREATE SEQUENCE sq_xyz;
```

Mudando o dono da sequência:

```
> ALTER SEQUENCE sq_xyz OWNER TO chiquinho;
```

Verificando relações (tabelas, views e sequências) do banco:

```
> \d

              List of relations
 Schema |     Name      |   Type   |   Owner
--------+---------------+----------+-----------
 public | sq_xyz        | sequence | chiquinho
 public | tb_teste_priv | table    | chiquinho
 public | tb_xyz        | table    | chiquinho
```

7.2 REASSIGN OWNED

Muda o dono desses objetos.

Sintaxe:

```
REASSIGN OWNED BY antigo_dono [, ...] TO novo_dono
```

Reassimilando a propriedade dos objetos que eram de chiquinho a g_teste_priv;

```
> REASSIGN OWNED BY chiquinho TO g_teste_priv;
```

Eis que então, agora todos os membros de g_teste_priv têm total acesso aos objetos:

```
> \d
```

```
                List of relations
 Schema |     Name      |   Type   |     Owner
--------+---------------+----------+---------------
 public | sq_xyz        | sequence | g_teste_priv
 public | tb_teste_priv | table    | g_teste_priv
 public | tb_xyz        | table    | g_teste_priv
```

Definição de senha para o usuário postgres:

```
> ALTER ROLE postgres ENCRYPTED PASSWORD '123';
```

No shell do sistema operacional, mudança no pg_hba.conf para a deixá-lo mais seguro:

```
$ cat << EOF > ${PGDATA}/pg_hba.conf
local    all               postgres             md5
local    db_teste_priv     +g_teste_priv        md5
EOF
```

Reload no serviço e conexão ao banco em seguida:

```
$ pg_ctl reload && $ psql db_teste_priv
```

Reassimilação de propriedade para postgres:

```
> REASSIGN OWNED BY g_teste_priv TO postgres;
```

7.3 Permissões em Schemas

E se precisar dar permissões a vários objetos num mesmo esquema?
Tem que ser um por um?

Criação de um novo esquema:

```
> CREATE SCHEMA sc_teste_priv;
```

Criação de tabela dentro do esquema:

```
> CREATE TABLE sc_teste_priv.tb_teste1(campo int);
```

Criação de tabela dentro do esquema:

```
> CREATE TABLE sc_teste_priv.tb_teste2(campo int);
```

Criação de tabela dentro do esquema:

```
> CREATE TABLE sc_teste_priv.tb_teste3(campo int);
```

Listando as tabelas criadas:

```
> \dt sc_teste_priv.*

              List of relations
    Schema     |   Name    | Type  |  Owner
---------------+-----------+-------+----------
 sc_teste_priv | tb_teste1 | table | postgres
 sc_teste_priv | tb_teste2 | table | postgres
 sc_teste_priv | tb_teste3 | table | postgres
```

Conexão ao banco com o papel silvana:

```
> \c db_teste_priv silvana
```

Tentativa de consulta do papel silvana na tabela dentro do esquema criado:

```
> SELECT * FROM sc_teste_priv.tb_teste1 LIMIT 5;

ERROR:  permission denied for schema sc_teste_priv
LINE 1: SELECT * FROM sc_teste_priv.tb_teste1;

                      ^
```

Com o superusuário dar o privilégio de uso no esquema para o papel silvana:

```
> GRANT USAGE ON SCHEMA sc_teste_priv TO silvana;
```

Nova tentativa de consulta pelo papel silvana:

```
> SELECT * FROM sc_teste_priv.tb_teste1;

ERROR:  permission denied for relation tb_teste1
```

Verificando se o papel silvana tem o direito de fazer consultas na tabela:

```
> SELECT has_table_privilege('silvana', 'sc_teste_priv.tb_teste1', 'SELECT');

 has_table_privilege
---------------------
 f
```

Concedendo a permissão de consulta em todas as tabelas do esquema:

```
> GRANT SELECT ON ALL TABLES IN SCHEMA sc_teste_priv TO silvana;
```

Nova tentativa de consulta do papel silvana:

```
> SELECT * FROM sc_teste_priv.tb_teste1;

campo
--
```

Consulta em sc_teste_priv.tb_teste2

```
> SELECT * FROM sc_teste_priv.tb_teste2;

campo
--
```

Consulta em sc_teste_priv.tb_teste3:

```
> SELECT * FROM sc_teste_priv.tb_teste3;

campo
--
```

Em outro terminal com o papel marcia logar no banco:

```
> \c db_teste_priv marcia
```

Tentativa de consulta:

```
> SELECT * FROM sc_teste_priv.tb_teste1;

ERROR:  permission denied for schema sc_teste_priv
LINE 1: SELECT * FROM sc_teste_priv.tb_teste1;
```

Invertendo a ordem de como foi feito com o papel silvana, para marcia primeiro será dado o privilégio nas tabelas e depois no esquema.

Concedendo o direito de consulta a todas as tabelas dentro do esquema para marcia:

```
> GRANT SELECT ON ALL TABLES IN SCHEMA sc_teste_priv TO marcia;
```

Verificar se marcia tem o privilégio de leitura na tabela tb_teste1 dentro do esquema:

```
> SELECT has_table_privilege('marcia','sc_teste_priv.tb_teste1','SELECT');

  has_table_privilege
----------------------
 t
```

Será que marcia conseguirá ler a tabela? Tentativa de consulta:

```
> SELECT * FROM sc_teste_priv.tb_teste1;

ERROR:  permission denied for schema sc_teste_priv
LINE 1: SELECT * FROM sc_teste_priv.tb_teste1;
                      ^
```

Com o superusuário dar o privilégio de acesso ao esquema:

```
> GRANT USAGE ON SCHEMA sc_teste_priv TO marcia;
```

No terminal de marcia nova tentativa de consulta:

```
> SELECT * FROM sc_teste_priv.tb_teste1;

campo
--
```

Consulta em tabela do esquema:

```
> SELECT * FROM sc_teste_priv.tb_teste2;

campo
--
```

Consulta bem-sucedida:

```
> SELECT * FROM sc_teste_priv.tb_teste3;

campo
--
```

O que acontece ao darmos privilégios de acesso a objetos a um papel e depois removemos esse papel?

Concedendo direito de uso no esquema para jose:

```
> GRANT USAGE ON SCHEMA sc_teste_priv TO jose;
```

Concedendo direito de leitura a todas as tabelas dentro do esquema para jose:

```
> GRANT SELECT ON ALL TABLES IN SCHEMA sc_teste_priv TO jose;
```

Tentativa de remoção do papel jose:

```
> DROP ROLE jose;

ERROR:  role "jose" cannot be dropped because some objects depend on it
DETAIL:  privileges for table sc_teste_priv.tb_teste3
privileges for table sc_teste_priv.tb_teste2
privileges for table sc_teste_priv.tb_teste1
privileges for schema sc_teste_priv
```

Consultando os privilégios de uma das tabelas do esquema:

```
> \z sc_teste_priv.tb_teste1

                                 Access privileges
    Schema     |   Name    | Type  |     Access privileges     | Column access privileges
---------------+-----------+-------+---------------------------+--------------------------
 sc_teste_priv | tb_teste1 | table | postgres=arwdDxt/postgres+|
               |           |       | silvana=r/postgres       +|
               |           |       | marcia=r/postgres        +|
               |           |       | jose=r/postgres           |
```

Revogando todos privilégios que jose tinha nas tabelas dentro do esquema:

```
> REVOKE ALL ON ALL TABLES IN SCHEMA sc_teste_priv FROM jose;
```

Nova consulta de priviégios em uma tabela do esquema:

```
> \z sc_teste_priv.tb_teste1

                                 Access privileges
    Schema     |   Name    | Type  |     Access privileges     | Column access privileges
---------------+-----------+-------+---------------------------+--------------------------
 sc_teste_priv | tb_teste1 | table | postgres=arwdDxt/postgres+|
               |           |       | silvana=r/postgres       +|
               |           |       | marcia=r/postgres         |
```

Nova tentativa de apagar jose:

```
> DROP ROLE jose;

ERROR:  role "jose" cannot be dropped because some objects depend on it
DETAIL:  privileges for schema sc_teste_priv
```

Revogando todos privilégios de jose no esquema:

```
> REVOKE ALL ON SCHEMA sc_teste_priv FROM jose;
```

Remoção do papel jose:

```
> DROP ROLE jose;
```

Se um papel tem privilégios, para apagá-lo é preciso primeiro revogar os privilégios que tem nos objetos.

8 RLS: Row Level Security - Segurança em Nível de Linha

- Conceito

8.1 Conceito

É um recurso disponibilizado a partir da versão 9.5, que veio a complementar o gerenciamento de usuários do Postgres.

Tal recurso cria políticas de segurança em nível de linhas conforme expressões definidas.

É preciso habilitar o RLS na tabela desejada com o comando `ALTER TABLE <tabela> ENABLE ROW LEVEL SECURITY`.

Para criar uma nova política usa-se o comando `CREATE POLICY`, cuja sintaxe é:

```
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

`name`	Nome da política
`table_name`	Nome da tabela
`FOR`	Para qual comando DML (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) a política será usada. A palavra-chave "`ALL`" representa todos comandos.
`role_name`	Papel (grupo / usuário) em que a política será aplicada.
`USING (using_expression)`	A cláusula `USING` é utilizada para linhas pré existentes na tabela e using_expression é a expressão utilizada para validar isso.
`WITH CHECK (check_expression)`	Para linhas que serão criadas via `INSERT` ou `UPDATE`, a checagem é com a cláusula `WITH CHECK` que tem também sua expressão que é avaliada.

Políticas podem ser aplicadas para comandos específicos ou para papéis específicos. Por omissão, novas políticas criadas serão aplicadas para todos comandos e papéis, a não ser que sejam especificados.

Se múltiplas políticas forem aplicadas para um determinado comando, elas serão combinadas usando a lógica `OR`.

Para comandos que tem ambas as políticas `USING` e `WITH CHECK` (`ALL` e `UPDATE`), se a política `WITH CHECK` não for definida, então a política `USING` será usada tanto para as linhas que estão visíveis (uso normal de `USING`) e para as linhas que serão permitidas para serem adicionadas (`WITH CHECK`).

Se a tabela tiver o recurso de RLS habilitado e não tiver nenhuma política definida, por padrão a política de negação será assumida. Ou seja, nenhuma linha será visível ou alterável.

Políticas não se aplicam a superusuários.

Se a tabela que a política se aplica for apagada, a política deixará de existir também.

Para a parte prática a seguir serão utilizados dois terminais (Terminal I e Terminal II), ambos inicialmente conectados como usuário postgres e no banco de dados de teste criado:

Terminal I

Criação do banco de dados de teste:

```
> CREATE DATABASE db_rls;
```

Terminal I e Terminal II

Conexão ao banco:

```
> \c db_rls
```

Terminal I

Criação de papéis com login (usuários):

```
> CREATE ROLE admin LOGIN; -- Administrador

> CREATE ROLE alice LOGIN; -- Usuário comum

> CREATE ROLE joana LOGIN; -- Usuário comum
```

Criação da tabela para testes:

```
> CREATE TABLE tb_usuario(
    username text PRIMARY KEY, -- Usuário
    pw TEXT, -- Senha
    nome_real text NOT NULL, -- Nome
    shell text NOT NULL -- Shell
);
```

Inserção de valores:

```
> INSERT INTO tb_usuario VALUES
    ('admin', '###', 'Administrador', '/bin/tcsh'),
    ('alice', '###', 'Alice', '/bin/bash'),
    ('joana', '###', 'Joana', '/bin/zsh');
```

Nota-se que como username foram inseridos os mesmos nomes dos papéis criados.

Permitir ao usuário admin ler, inserir, atualizar e pagar:

```
> GRANT SELECT, INSERT, UPDATE, DELETE ON tb_usuario TO admin;
```

Usuários comuns só terão acesso a colunas públicas:

```
> GRANT SELECT (username, nome_real, shell) ON tb_usuario TO PUBLIC;
```

Permitir a usuários comuns a mudar certas colunas:

```
> GRANT UPDATE (pw, username, nome_real, shell) ON tb_usuario TO PUBLIC;
```

Verificando a tabela:

```
> TABLE tb_usuario;
```

```
 username | pw  |   nome_real   |   shell
----------+-----+---------------+-----------
 admin    | ### | Administrador | /bin/tcsh
 alice    | ### | Alice         | /bin/bash
 joana    | ### | Joana         | /bin/zsh
```

Habilitando RLS na tabela:

```
> ALTER TABLE tb_usuario ENABLE ROW LEVEL SECURITY;
```

Terminal II

Conexão ao banco:

```
> \c db_rls
```

Mudando para o papel admin:

```
> SET role admin;
```

Checando nome de usuário:

```
> SELECT current_user;

 current_user
--------------
 admin
```

Verificando a toda a tabela:

```
> TABLE tb_usuario;

 username | pw | nome_real | shell
----------+----+-----------+-------
```

Terminal I

Em outro terminal como superusuário (role postgres):

```
> CREATE POLICY po_admin_all_priv_usuario
    ON tb_usuario
    TO admin
    USING (true)
    WITH CHECK (true);
```

Terminal II

Com o papel admin:

```
> TABLE tb_usuario;

 username | pw  |   nome_real   |   shell
----------+-----+---------------+-----------
 admin    | ### | Administrador | /bin/tcsh
 alice    | ### | Alice         | /bin/bash
 joana    | ### | Joana         | /bin/zsh
```

Utilizar o papel alice:

```
> SET role alice;
```

Verificar a tabela:

```
> SELECT username, nome_real, shell FROM tb_usuario;

 username | nome_real | shell
----------+-----------+-------
```

Terminal I

Será que podemos criar política negando um super usuário?:

```
> CREATE POLICY po_deny_su
    ON tb_usuario
    TO postgres
    USING (false)
    WITH CHECK (false);
```

Testando:

```
> TABLE tb_usuario;

 username | pw  |   nome_real   |   shell
----------+-----+---------------+-----------
 admin    | ### | Administrador | /bin/tcsh
 alice    | ### | Alice         | /bin/bash
 joana    | ### | Joana         | /bin/zsh
```

Pode-se constatar que para um superusuário não se aplicam políticas.

Veriricando quais políticas existem na base de dados atual:

```
> TABLE pg_policies;

 schemaname | tablename  |        policyname         |  roles     | cmd | qual  | with_check
------------+------------+---------------------------+------------+-----+-------+------------
 public     | tb_usuario | po_admin_all_priv_usuario | {admin}    | ALL | true  | true
 public     | tb_usuario | po_deny_su                | {postgres} | ALL | false | false
```

Apagando uma política:

```
> DROP POLICY po_deny_su ON tb_usuario;
```

Para apagar uma política é necessário além do nome da mesma, especificar em qual tabela ela se aplica.

Política para visualização de dados para todos usuários:

```
> CREATE POLICY po_all_view_usuario ON tb_usuario FOR SELECT USING (true);
```

Usuários normais podem atualizar seus registros, mas limita que shells um usuário normal pode configurar:

```
> CREATE POLICY po_users_mod_usuario ON tb_usuario
    FOR UPDATE
    USING (current_user = username)
    WITH CHECK (
        current_user = username AND
        shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh'));
```

Terminal II

Papel alice:

```
> SET role alice;
```

Visualizando todos os registros com as colunas públicas:

```
> SELECT username, nome_real, shell FROM tb_usuario;
```

```
 username |   nome_real   |   shell
----------+---------------+-----------
 admin    | Administrador | /bin/tcsh
 alice    | Alice         | /bin/bash
 joana    | Joana         | /bin/zsh
```

Tentativa de alteração de próprio nome e shell:

```
> UPDATE tb_usuario SET (nome_real, shell) = ('Alice Santos', '/bin/fish');
```

```
ERROR:  new row violates row-level security policy for table "tb_usuario"
```

Foi feita uma tentativa de tentar alterar para um shell que não constava na lista de permitidos.

Atualizando o próprio nome e shell devidamente:

```
> UPDATE tb_usuario SET (nome_real, shell) = ('Alice Santos', '/bin/zsh');
```

Terminal I

Criação de tabela:

```
> CREATE TABLE tb_anotacao(
    id serial PRIMARY KEY,
    username text DEFAULT current_user
        REFERENCES tb_usuario (username),
    dt timestamptz DEFAULT now(),
    title varchar(30),
    description text);
```

A idéia aqui é que o admin possa ver todas anotações e os outros usuários só possam ver as suas próprias.

Habilitando RLS na tabela:

```
> ALTER TABLE tb_anotacao ENABLE ROW LEVEL SECURITY;
```

Permitir ao usuário admin ler, inserir, atualizar e pagar:

```
> GRANT SELECT, INSERT, UPDATE, DELETE ON tb_anotacao TO admin;
```

Permitir a usuários comuns a mudar certas colunas:

```
> GRANT UPDATE (dt, title, description) ON tb_anotacao TO PUBLIC;
```

Permissões públicas para buscar, inserir (algumas colunas) e apagar:

```
> GRANT SELECT, INSERT (dt, title, description), DELETE ON tb_anotacao TO PUBLIC;
```

Conceder uso da sequência que está atrelada à tabela:

```
> GRANT USAGE ON tb_anotacao_id_seq TO PUBLIC;
```

Acesso total ao usuário admin:

```
> CREATE POLICY po_admin_all_priv_anotacao
    ON tb_anotacao
    FOR ALL
    TO admin
    USING (true)
    WITH CHECK (true);
```

Somente o próprio usuário pode ler / alterar seus registros:

```
> CREATE POLICY po_users_rw_anotacao
    ON tb_anotacao
    FOR ALL
    TO PUBLIC
    USING (current_user = username)
    WITH CHECK (current_user = username);
```

Terminal II

Usuário joana:

```
> SET role joana;
```

Inserindo registros:

```
> INSERT INTO tb_anotacao (dt, title, description) VALUES
    (now(), 'Teste', 'Primeira anotação da Joana'),
    ('2016-10-07', 'Segundo Teste', 'Segunda anotação da Joana'),
    (now() - '2 days'::interval, 'Título', 'difdopifikerm fefejkfejkej');
```

Usuário alice:	Verificando a tabela:

`> SET role alice;`	`> TABLE tb_anotacao;`
	
	id | username | dt | title | description
	`----+----------+----+-------+-------------`
	Nenhum resultado, pois o usuário não tem acesso.

Inserindo registros:

```
> INSERT INTO tb_anotacao (dt, title, description) VALUES
    (now(), 'Teste 1', 'Primeira anotação da Alice'),
    (now() - '2 weeks'::interval, 'Título', 'difdopifikerm fefejkfejkej');
```

Usuário admin:

```
> SET role admin;
```

Verificando a tabela:

```
> TABLE tb_anotacao;
```

```
 id | username |              dt                |     title     |        description
----+----------+--------------------------------+---------------+---------------------------
  1 | joana    | 2016-10-07 11:54:12.912176-03  | Teste         | Primeira anotação da Joana
  2 | joana    | 2016-10-07 00:00:00-03         | Segundo Teste | Segunda anotação da Joana
  3 | joana    | 2016-10-05 11:54:12.912176-03  | Título        | difdopifikerm fefejkfejkej
  4 | alice    | 2016-10-07 11:56:06.083534-03  | Teste 1       | Primeira anotação da Alice
  5 | alice    | 2016-09-23 11:56:06.083534-03  | Título        | difdopifikerm fefejkfejkej
```

Usuário joana:

```
> SET role joana;
```

Verificando a tabela:

```
> TABLE tb_anotacao;
```

```
 id | username |              dt                |     title     |        description
----+----------+--------------------------------+---------------+---------------------------
  1 | joana    | 2016-10-07 11:54:12.912176-03  | Teste         | Primeira anotação da Joana
  2 | joana    | 2016-10-07 00:00:00-03         | Segundo Teste | Segunda anotação da Joana
  3 | joana    | 2016-10-05 11:54:12.912176-03  | Título        | difdopifikerm fefejkfejkej
```

Usuário Alice:

```
> SET role alice;
```

Verificando a tabela:

```
> TABLE tb_anotacao;
```

```
 id | username |              dt                |  title  |        description
----+----------+--------------------------------+---------+---------------------------
  4 | alice    | 2016-10-07 11:56:06.083534-03  | Teste 1 | Primeira anotação da Alice
  5 | alice    | 2016-09-23 11:56:06.083534-03  | Título  | difdopifikerm fefejkfejkej
```

9 Autenticação

- pg_hba.conf - Host-Based Authentication (Autenticação Baseada em Máquina)
- Autenticação no OpenLDAP
- Função e View pg_hba_file_rules

9.1 pg_hba.conf - Host-Based Authentication (Autenticação Baseada em Máquina)

A autenticação de clientes é controlada por um arquivo de configuração, que tradicionalmente é chamado `pg_hba.conf` e armazenado no diretório de dados do *cluster* de banco de dados (`$PGDATA`).

Um arquivo `pg_hba.conf` padrão é instalado quando o diretório de dados é inicializado pelo initdb.

É possível guardá-lo em outro lugar, o que é determinado pelo parâmetro de configuração do PostgreSQL `hba_file`.

O formato geral do `pg_hba.conf` é um conjunto de registros, um por linha. Linhas em branco são ignoradas, assim como qualquer texto depois do caractere de comentário "`#`". Os registros não podem ser continuados através das linhas.

Um registro é composto de um certo número de campos que são separados por espaços e/ou tabs. Esses campos podem conter espaço em branco se for envolvido por aspas.

Ao envolver palavras-chaves com aspas nos campos `database`, `user` ou `address` (e. g. `all` ou `replication`) faz com que a palavra perca seu caráter especial.

Forma geral de um registro do `pg_hba.conf`:

- **Conexão Local**

```
local      DATABASE  USER  METHOD  [OPTIONS]
```

- **Outros Tipos de Conexão**

```
TYPE       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

Aviso:

Se houverem registros conflitantes no `pg_hba.conf`, prevalescerá o que for especificado primeiro.

9.1.1 Campos do pg_hba.conf

TYPE

É o tipo de conexão utilizada, que pode ser:

- **`local`** - Tenta utilizar soquetes de domínio Unix. Sem um registro desse tipo, conexões de soquete de domínio Unix não são permitidas;

- **`host`** - Tenta utilizar conexões usando o protocolo TCP/IP, que podem ser tanto com ou sem SSL.
  Conexões TCP/IP remotas não serão possíveis a não ser que o servidor seja inicializado com o valor apropriado para o parâmetro de configuração `listen_addresses`, uma vez que o comportamento padrão é ouvir apenas conexões TCP/IP vindas do endereço local de *loopback*; *localhost*.

- **`hostssl`** - Tenta utilizar conexões usando o protocolo TCP/IP, mas apenas conexões com encriptação SSL.
  Para fazer uso dessa opção o servidor deve ter sido compilado com suporte a SSL. Além disso, SSL deve ser habilitada ao inicializar do servidor com o parâmetro de configuração `ssl`.

- **`hostnossl`** - Tem o efeito oposto de `hostssl`; apenas conexões sem SSL, via TCP/IP são aceitas.

DATABASE

Especifica que nome de base(s) de dados combina(m) com o registro.
É possível especificar mais de uma base de dados no registro, separando os nomes por vírgulas.
Também é possível especificar um arquivo separado que contenha uma lista de nomes de bases de dados.
O nome desse arquivo deve ser precedido com o caractere "`@`".
Há as palavras-chaves que podem ser especificadas, que são:

- **`all`** - Todas bases de dados;

- **`sameuser`** - A base de dados tem que ter o mesmo nome do usuário requisitante;

- **`samerole/samegroup`** - Força com que o usuário requisitante seja membro de um papel cujo nome seja idêntico ao da base que quer se conectar. `samegroup` é obsoleto, mas ainda é aceito e tem o mesmo efeito;

- **`replication`** - Para fins de replicação via *streaming*.

USER

Especifica que usuário(s) do banco de dados combina(m) com o registro. Pode-se especificar mais de um nome de usuário, separando-os por vírgulas.

Um arquivo separado que contenha nomes de usuários pode ser especificado, o nome desse arquivo deve ser precedido por "@".

O valor all especifica que combina com todos usuários. Caso contrário, este é o nome de um papel de banco de dados específico, ou um nome de grupo precedido por "+".

ADDRESS

Especifica endereços de máquinas clientes que combinem com esse registro.

Este campo se aplica apenas quando o tipo (TYPE) for host, hostssl, ou hostnossl.

Este campo pode conter um hostname, endereço IP, endereço de rede ou uma palavra-chave.

Palavras-chave

- **`all`**: Qualquer endereço IP;
- **`samehost`**: Qualquer IP do host;
- **`samenet`**: Qualquer IP da rede que estiver diretamente conectado.

METHOD

Método de autenticação utilizado.

- **`trust`**: Permite conexões incondicionalmente. Este método permite qualquer um a se conectar ao servidor PostgreSQL com qualquer papel que desejar sem precisar fornecer senha ou qualquer outro tipo de autenticação.
- **`reject`**: Rejeita a conexões incondicionalmente. É útil para filtrar certos hosts de um grupo, por exemplo, uma linha de rejeição pode impedir uma máquina de se conectar, enquanto que linhas posteriores podem permitir que as outras máquinas se conectem normalmente.
- **`md5`**: Requer que o cliente forneça uma senha encriptada em MD5 para autenticação.
- **`password`**: Requer que o cliente forneça uma senha não encriptada. Uma vez que essa senha é enviada em texto claro na rede, não deve ser utilizado em redes não confiáveis.
- **`gss`**: Usa GSSAPI para autenticação. Está disponível apenas para conexões TCP/IP.
- **`sspi`**: Usa SSPI para autenticação. Disponível apenas no Windows.

- **krb5**: Usa autenticação Kerberos V5. Está disponível apenas para conexões TCP/IP.

- **ident**: Obtém o nome de usuário do sistema operacional do cliente consultando o servidor ident no cliente e checando se combina com um usuário do banco de dados. Autenticação ident só pode ser utilizada em conexões TCP/IP. Quando for especificada uma conexão local, a autenticação peer será utilizada em seu lugar.

- **peer**: Idêntico ao método ident, mas disponível apenas para conexões locais.

- **ldap**: Autentica usando um servidor LDAP.

- **radius**: Autentica usando um servidor RADIUS.

- **cert**: Autentica usando certificados SSL do cliente.

- **pam**: Autentica usando o serviço Pluggable Authentication Modules (PAM) fornecido pelo sistema operacional.
-
- **scram-sha-256**: Requer que o cliente forneça uma senha encriptada em SCRAM-SHA-256 para autenticação.

- **bsd**: Autenticação BSD, serviço fornecido pelo sistema operacional.

OPTIONS

Após o campo de método de autenticação, podem ter campos na forma `nome = valor` que especificam opções para o método de autenticação.

Tornar-se usuário postgres:

```
# su - postgres
```

Acessar o diretório onde está o pg_hba.conf:

```
$ cd ${PGCONF}
```

Fazendo um backup do arquivo original:

```
$ cp pg_hba.conf pg_hba.conf.bkp
```

Substituindo o conteúdo original, reload no serviço, limpando a tela e exibindo o atual conteúdo:

```
$ cat << EOF > pg_hba.conf && pg_ctl reload && clear && cat pg_hba.conf
local all all trust
EOF
```

Atribuindo uma senha para o usuário de banco postgres:

```
$ psql -c "ALTER ROLE postgres ENCRYPTED PASSWORD '123';"
```

Novo conteúdo para o pg_hba.conf com o método trust primeiro:

```
$ cat << EOF > pg_hba.conf && pg_ctl reload && clear && cat pg_hba.conf
local all all trust
local all all md5
EOF
```

Teste:

```
$ psql -c "SELECT 'OK' AS teste;"

 teste
-------
  OK
```

Novo conteúdo para o pg_hba.conf com o método md5 primeiro:

```
$ cat << EOF > pg_hba.conf && pg_ctl reload && clear && cat pg_hba.conf
local all all md5
local all all trust
EOF
```

Teste:

```
$ psql -c "SELECT 'OK' AS teste;"

Password:

 teste
-------
  OK
```

Novo conteúdo para o pg_hba.conf com o método trust primeiro:

```
$ cat << EOF > pg_hba.conf && pg_ctl reload && clear && cat pg_hba.conf
local all all trust
local all all reject
EOF
```

Teste:

```
$ psql -c "SELECT 'OK' AS teste;"

 teste
-------
 OK
```

Novo conteúdo para o pg_hba.conf com o método reject primeiro:

```
$ cat << EOF > pg_hba.conf && pg_ctl reload && clear && cat pg_hba.conf
local all all reject
local all all trust
EOF
```

Teste:

```
$ psql -c "SELECT 'OK' AS teste;"

psql: FATAL:  pg_hba.conf rejects connection for host "[local]", user "postgres", database "postgres", SSL off
```

Voltar o pg_hba original:

```
$ cat pg_hba.conf.bkp > pg_hba.conf && pg_ctl reload
```

Teste:

```
$ psql -c "SELECT 'OK' AS teste;"

 teste
-------
 OK
```

9.2 Autenticação no OpenLDAP

OpenLDAP é um software livre que implementa o protocolo LDAP como serviço de diretório.

Tem como maior aplicabilidade a centralização de autenticação para vários sistemas.

9.2.1 Autenticação LDAP no PostgreSQL

Esse método de autenticação opera similarmente ao password exceto que ele utiliza o LDAP como método de verificação de senha.

LDAP é utilizado apenas para validar o par usuário/senha.

No entanto, o usuário (papel com login) deve pré existir no servidor de banco de dados antes de LDAP ser utilizado como método de autenticação.

As seguintes opções de configuração são suportadas para LDAP:

- **`ldapserver`**: Nomes ou endereços IP de servidores LDAP para conectar. Múltiplos servidores podem ser especificados separados por espaços.

- **`ldapport`**: Número da porta do servidor LDAP para se conectar. Se for omitido, será utilizado o padrão da biblioteca LDAP.

- **`ldaptls`**: Configurando seu valor para 1 faz a conexão entre os servidores PostgreSQL e LDAP utilizar encriptação TLS.
  Observe que isso apenas encripta o tráfego para o servidor LDAP, a conexão para o cliente ainda não será encriptada a não ser que se use SSL.

- **`ldapprefix`**: A string que precede o nome de usuário quando forma o DN (nome distinto = distinguished name) para serem ajuntados, quando se faz uma autenticação simples bind.

- **`ldapsuffix`**: String para ser adicionada ao nome do usuário quando forma o DN para serem vinculados, quando se faz uma autenticação simples bind.

- **`ldapbasedn`**: DN raiz (root DN) para iniciar uma busca por um usuário quando se faz autenticação search+bind.

- **`ldapbinddn`**: DN do usuário para juntar ao diretório quando se faz autenticação search+bind.

- **`ldapbindpasswd`**: Senha para usuário para juntar ao diretório para fazer a busca quando se faz autenticação search+bind.

- **`ldapsearchattribute`**: Atributo para combinar com o usuário quando se faz autenticação search+bind.

Exemplo:

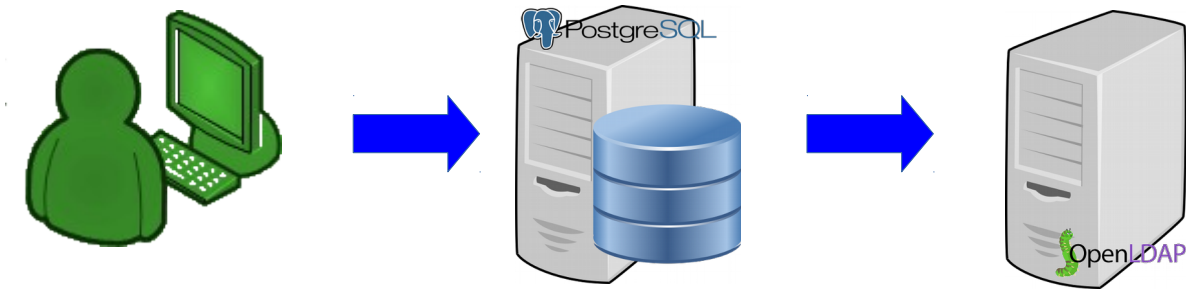
```
ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=net"
```

Nesta prática será utilizado um servidor externo OpenLDAP, o qual será configurado com LDAP BaseDN: `"dc=curso,dc=dominio"`.

No teste, haverão 3 (três) máquinas: cliente, servidor PostgreSQL e servidor OpenLDAP.

O cliente solicitará autenticação no servidor PostgreSQL e esse mesmo repassará a autenticação para o servidor OpenLDAP para permitir acesso à base PostgreSQL.

O servidor OpenLDAP será um CentOS.



Parte I – Instalação e Configuração do Servidor OpenLDAP

Instalação do OpenLDAP:

```
$ yum install -y openldap-{servers,clients} && yum clean all
```

Cópia do carquivo de exemplo DB_CONFIG:

```
$ cp /usr/share/openldap-servers/DB_CONFIG.example /var/lib/ldap/DB_CONFIG
```

Usuário e grupo ldap como proprietários do arquivo:

```
$ chown ldap:ldap /var/lib/ldap/DB_CONFIG
```

Habilitando o serviço OpenLDAP:

```
$ systemctl enable slapd
```

Iniciando o serviço OpenLDAP:

```
$ systemctl start slapd
```

Adicionando configurações de schema do OpenLDAP:

```
$ ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/cosine.ldif

$ ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/nis.ldif

$ ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/inetorgperson.ldif
```

Variáveis de ambiente de senha para o usuário administrador do OpenLDAP (Manager) e o usuário de teste do banco (tux):

```
$ ADMIN_PW=`slappasswd`

$ TUX_PW=`slappasswd`
```

Criação de arquivo ldif para modificar a base padrão do OpenLDAP:

```
$ cat << EOF > /tmp/0.ldif
dn: olcDatabase={2}hdb,cn=config
replace: olcSuffix
olcSuffix: dc=curso,dc=dominio
-
replace: olcRootDN
olcRootDN: cn=Manager,dc=curso,dc=dominio
-
add: olcRootPW
olcRootPW: ${ADMIN_PW}
EOF
```

Modificando a base padrão conforme o arquivo ldif:

```
$ ldapmodify -Y EXTERNAL -H ldapi:/// -f /tmp/0.ldif
```

Criação de arquivo ldif para o usuário de teste do banco:

```
$ cat << EOF > /tmp/1.ldif
dn: dc=curso,dc=dominio
dc: curso
objectClass: dcObject
objectClass: organizationalUnit
ou: Curso PostgreSQL - Administração

# dbuser ou
dn: ou=dbuser,dc=curso,dc=dominio
ou: dbuser
objectClass: organizationalUnit
objectClass: top

# tux uid
dn: uid=tux,ou=dbuser,dc=curso,dc=dominio
uid: tux
cn: tux
objectClass: account
objectClass: posixAccount
objectClass: top
objectClass: shadowAccount
userPassword: ${TUX_PW}
shadowLastChange: 15140
shadowMin: 0
shadowMax: 99999
shadowWarning: 7
loginShell: /bin/false
homeDirectory: /dev/null
uidNumber: 10000
gidNumber: 10000
EOF
```

Adicionando entradas OpenLDAP pelo arquivo ldif:

```
$ ldapadd -c -xD 'cn=Manager,dc=curso,dc=dominio' -W -H ldapi:/// -f /tmp/1.ldif
```

Teste de autenticação do usuário tux na base LDAP (se o resultado for zero o procedimento ocorreu com sucesso):

```
$ ldapsearch -xD 'uid=tux,ou=dbuser,dc=curso,dc=dominio' -W -b \
'dc=curso,dc=dominio' $> /dev/null ; echo $?
```

Parte II – Configuração do Servidor PostgreSQL

Variáveis de ambiente para o IP do servidor OpenLDAP:

```
$ read -p 'Digite o IP do Servidor OpenLDAP: ' SRV_OPENLDAP
```

Variável de ambiente para o endereço de rede com máscara:

```
$ read -p 'Digite IP/Máscara (XXX.XXX.XXX.XXX/XX) da rede: ' NET_ADDR
```

Arquivo pg_hba e reload no serviço do PostgreSQL:

```
$ cat << EOF > ${PGDATA}/pg_hba.conf && pg_ctl reload
local   all    postgres                          trust
host    all    all         127.0.0.1/32          md5
host    all    tux         ${NET_ADDR}    ldap \
ldapserver=${SRV_OPENLDAP} ldapbasedn="dc=curso,dc=dominio" \
ldapsearchattribute=uid
local   all    tux                               md5
EOF
```

Criação do papel tux:

```
$ psql -c 'CREATE ROLE tux LOGIN;'
```

Criação de senha para o papel tux:

```
$ psql -c "ALTER ROLE tux ENCRYPTED PASSWORD '789';"
```

Parte III – Testes de Conexão

Variável de ambiente para o servidor PostgreSQL:

```
$ read -p 'Digite o IP do Servidor PostgreSQL: ' SRV_PGSQL
```

Teste de conexão ao PostgreSQL com autenticação LDAP (máquina cliente):

```
$ psql -h ${SRV_PGSQL} -U tux -d template1 -c 'SELECT true;'
```

Teste de conexão local (servidor PostgreSQL):

```
$ psql -U tux postgres -c 'SELECT true;'
```

Nota-se que pela rede, conforme determinado no arquivo pg_hba.conf foi utilizada a autenticação LDAP e localmente no próprio PostgreSQL.

9.3 Função e View pg_hba_file_rules

Para facilitar a administração dos dados no arquivo de autenticação (`pg_hba.conf`) há uma *view* de sistema (catálogo) chamada `pg_hba_file_rules` cujos dados vêm da função de mesmo nome que por sua vez tem sua origem no arquivo de autenticação.

Estrutura e detalhes da view pg_hba_file_rules:

```
> \d+ pg_hba_file_rules

                        View "pg_catalog.pg_hba_file_rules"
   Column    |  Type   | Collation | Nullable | Default | Storage  | Description
-------------+---------+-----------+----------+---------+----------+-------------
 line_number | integer |           |          |         | plain    |
 type        | text    |           |          |         | extended |
 database    | text[]  |           |          |         | extended |
 user_name   | text[]  |           |          |         | extended |
 address     | text    |           |          |         | extended |
 netmask     | text    |           |          |         | extended |
 auth_method | text    |           |          |         | extended |
 options     | text[]  |           |          |         | extended |
 error       | text    |           |          |         | extended |
View definition:
 SELECT a.line_number,
    a.type,
    a.database,
    a.user_name,
    a.address,
    a.netmask,
    a.auth_method,
    a.options,
    a.error
   FROM pg_hba_file_rules() a(line_number, type, database, user_name, address, netmask, auth_method,
options, error);
```

Como pode se notar, na definição é feita uma chamada na função de mesmo nome da view.

Estrutura e detalhes da função pg_hba_file_rules:

```
> \df+ pg_hba_file_rules

Schema              | pg_catalog
Name                | pg_hba_file_rules
Result data type    | SETOF record
Argument data types | OUT line_number integer, OUT type text, OUT database text[], OUT user_name
text[], OUT address text, OUT netmask text, OUT auth_method text, OUT options text[], OUT error text
Type                | func
Volatility          | volatile
Parallel            | safe
Owner               | postgres
Security            | invoker
Access privileges   | postgres=X/postgres
Language            | internal
Source code         | pg_hba_file_rules
Description         | show pg_hba.conf rules
```

10 Write Ahead Log

- WAL: Write Ahead Log, Integridade de Dados

10.1 WAL: Write Ahead Log, Integridade de Dados

Write-Ahead Log (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 em disco).

Os arquivos de log de transação são conhecidos como “xlog” e estão armazenados no subdiretório `pg_wal` (do PostgreSQL 10 em diante, antes era `pg_xlog`).

Então o caminho para os logs de transação em sistemas Linux/Unix é nomalmente "`$PGDATA/pg_wal`" ou em sistemas Windows: "`\%PGDATA%\pg_wal`", isso se as variáveis de ambiente foram devidamente configuradas. Porém isso pode ser determinado de outra forma com o aplicativo `initdb` com a opção `-X`.

Os nomes dos arquivos de log são números hexadecimais, sendo que os oito primeiros dígitos são referentes à sua cronologia.

Exemplo: `000000030000000000000012`.

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.

Os arquivos do WAL são gravados de forma serial e não aleatória, enquanto que os arquivos físicos do banco são gravados de forma randômica. A gravação serial é significantemente mais rápida.

Quando uma transação é efetivada, os dados não vão direto para os arquivos físicos do banco, mas sim simultaneamente para o WAL e para memória, então depois de algum tempo é feito o despejo dessas “páginas sujas” nos arquivos físicos. Esse despejo é chamado de *checkpoint*. Se por algum motivo não houver como gravar nos arquivos do WAL a transação é abortada.

Por padrão cada segmento do WAL tem o tamanho fixo de 16 MB, a cada um que for preenchido é criado um novo.

Também por padrão os segmentos do WAL são divididos em páginas de 8 kB.

Resumindo:

Transação efetivada → gravação simultânea em memória e no WAL → checkpoint

Com esse procedimento, não é preciso despejar as páginas sujas em disco a cada efetivação de transação, porque como sabe-se num possível *crash* será possível recuperar a base utilizando os *xlogs*: quaisquer mudanças que não foram aplicadas para as páginas de dados podem ser refeitas dos registros dos logs de transação. Esse tipo de recuperação é também conhecida como *REDO*.

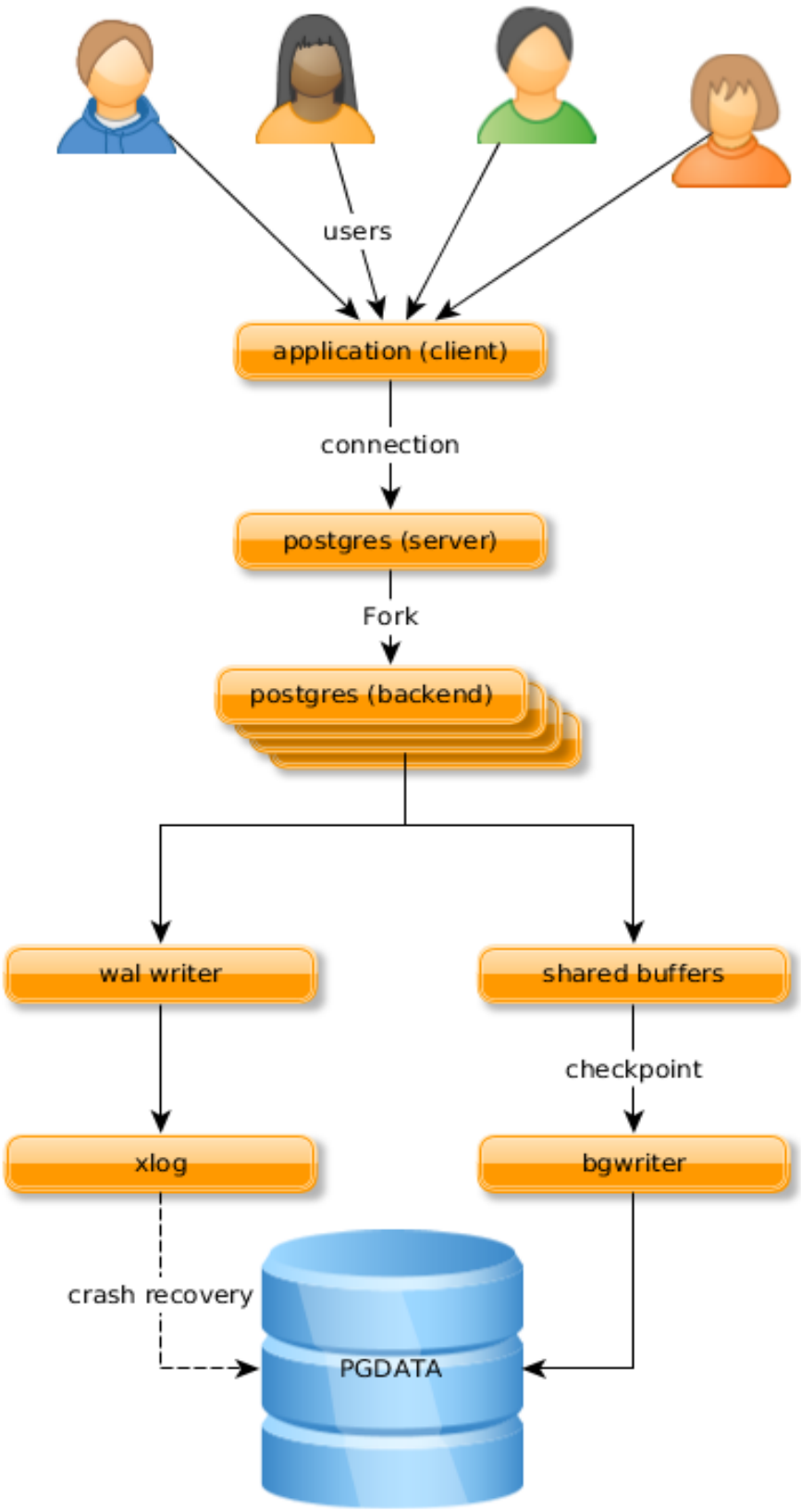


Figura 1: Diagrama de escrita de dados

Parar o serviço (como usuário postgres):

```
$ pg_ctl stop
```

Tira a permissão de escrita nos xlogs existentes:

```
$ ls -lhd ${PGDATA}/pg_wal/* | egrep -v '^d' | \
awk '{print $(NF)}' | xargs chmod -w
```

Inicialização do serviço:

```
$ pg_ctl start
```

Tentativa de gravação criando uma base de dados:

```
$ psql -c 'CREATE DATABASE db_teste;'

Expanded display is used automatically.
2016-10-10 16:40:39 BRT [2511-1] PANIC:  could not open file
"pg_xlog/000000010000000000000001": Permission denied

. . .
```

A mensagem de erro de permissão negada se deve ao fato de que foi feita uma tentativa de gravação nos logs de transação e não foi possível.

Dessa forma então o serviço do PostgreSQL foi parado.

Dando permissão de leitura e escrita nos xlogs:

```
$ ls -lhd ${PGDATA}/pg_wal/* | egrep -v '^d' | \
awk '{print $(NF)}' | xargs chmod +w
```

Inicializando o serviço:

```
$ pg_ctl start
```

Tirando a permissão de escrita nos xlogs após o serviço subir:

```
$ ls -lhd ${PGDATA}/pg_wal/* | egrep -v '^d' | \
awk '{print $(NF)}' | xargs chmod -w
```

Consulta simples (somente leitura):

```
$ psql -c "SELECT 'OK' AS teste;"

 teste
-------
 OK
```

Tentativa de criação de tabela (escrita):

```
$ psql -c 'CREATE TABLE tb_teste_wal();'
```

Erro de permissões em arquivos de log de transação e o serviço foi parado novamente.

Voltando as permissões para leitura e escrita:

```
$ ls -lhd ${PGDATA}/pg_wal/* | egrep -v '^d' | \
awk '{print $(NF)}' | xargs chmod +w
```

Subindo o serviço novamente:

```
$ pg_ctl start
```

Novo teste de leitura:

```
$ psql -c "SELECT 'OK' AS teste;"

 teste
-------
 OK
```

11 Checkpoint

- Sobre Checkpoint

11.1 Sobre Checkpoint

Um *checkpoint* é um ponto no log da sequência de transação que os arquivos de dados são atualizados para refletir as informações no log (de transação).

Todas as *dirty pages* (páginas sujas), cujos dados foram gravados na memória (os mesmos dados gravados nos WAL), antes do checkpoint serão despejadas em disco.

Na hora do *checkpoint*, todas páginas sujas são despejadas em disco e um registro especial de *checkpoint* é gravado para o arquivo de log (os registros de mudança foram previamente despejados para os arquivos de WAL).

Em caso de *crash* de sistema, o procedimento de recuperação de *crash* verifica o último registro de *checkpoint* para determinar o ponto no log (também chamado registro *redo*) de onde deve partir a operação *REDO*.

Quaisquer alterações feitas nos arquivos de dados antes desse ponto são garantidos já estarem em disco.

Assim, após um *checkpoint*, os segmentos de log anteriores ao que contém o registro REDO não são mais necessários e podem ser reciclados ou removidos. (Quando o arquivamento de WAL está pronto, os segmentos de log devem ser arquivos antes de serem reciclados ou removidos.

Um *checkpoint* é iniciado após encher os logs do WAL (quantidade declarada em checkpoint_segments) ou o tempo configurado em `checkpoint_timeout`, o que vier primeiro.

Se nenhum arquivo do WAL foi alterado desde o último *checkpoint*, novos checkpoints serão ignorados mesmo que tenha passado o tempo de `checkpoint_timeout`.

É possível também forçar um *checkpoint* utilizando o comando SQL `CHECKPOINT`.

Reduzindo `checkpoint_timeout` e / ou `max_wal_size` faz com que os *checkpoints* ocorram com mais frequência.

Se *checkpoints* acontecem mais perto do que `checkpoint_warning`, uma mensagem será escrita no log do servidor recomendando aumentar o valor de `max_wal_size`.

A aparição ocasional dessa mensagem não é motivo para alarme, mas se aparecer frequentemente, então os parâmetros de controle de *checkpoint* devem ser aumentados.

Operações em massa como uma grande transferência via `COPY` causam vários avisos (*warning*) se o parâmetro `max_wal_size` não tiver seu valor alto o suficiente.

Nesta parte prática veremos os *WAL* sendo reciclados.

Alterar parâmetros no postgresql.conf:

```
$ vim ${PGDATA}/postgresql.conf

wal_level = logical
```

Verificando o contexto da configuração:

```
$ psql -c "SELECT context FROM pg_settings
    WHERE name = 'wal_level';"

 context
---------
 postmaster
```

Devido ao contexto de um dos parâmetros alterados ser postmaster será preciso reiniciar o serviço. Restart do serviço:

```
$ pg_ctl restart
```

Em terminais diferentes (chamaremos de X e Y) fazer as seguintes operações:

Terminal X

Monitoramento de arquivos de log do WAL:

```
$ while :; do
  clear
  # Utilizando regex para arquivos
  # com 24 caracteres hexadecimais
  ls -1 ${PGDATA}/pg_wal | \
  awk '/^[0-9,A-F]+$/ {if (length($1) == 24) {print $1} }'
  sleep 3;
done
```

Esse monitoramento rústico se baseia em um laço (*loop) while* infinito que lista todos os *WAL* do diretório, cuja nomenclatura se baseia em 24 (vinte e quatro) caracteres que são números hexadecimais.

O laço é repetido a cada 3 (três) segundos, exibindo os arquivos, o que nos possibilita verificar novos *WAL* sendo gerados.

Terminal Y

Criação de tabela com muitos registros (também como usuário postgres):

```
$ psql -c "SELECT generate_series(1,70000000) AS campo1 \
INTO tb_teste_checkpoint;"
```

Variável que armazena a quantidade de arquivos de log de transação:

```
$ QTD_WAL=`ls -1 ${PGDATA}/pg_wal | \
awk '/^[0-9,A-F]+$/ {if (length($1) == 24) {print $1} }' | wc -l`
```

Exibindo o valor da variável:

```
$ echo ${QTD_WAL}
```

```
64
```

Exibindo o valor do parâmetro max_wal_size:

```
$ psql -Atc 'SHOW max_wal_size;'
```

```
1GB
```

Multiplicando a quantidade de logs de transação por 16:

```
$ echo "${QTD_WAL} * 16" | bc
```

```
1024
```

O resultado da multiplicação de 64 logs de transação x 16 MB resultou em 1024 MB, que é igual a 1 GB. Isso bate com o valor de max_wal_size.

12 Backup e Restauração

- Sobre Backup e Restauração
- Backup Lógico (SQL Dump)
- Backup Físico Off Line: Snapshot
- Backup Física On Line
- pg_basebackup

12.1 Sobre Backup e Restauração

Como tudo que tem dados valiosos, bancos de dados PostgreSQL necessitam que sejam feitos backups regularmente.

Embora os procedimentos sejam essencialmente simples, é importante ter um entendimento claro das técnicas subjacentes e suas premissas.

Há fundamentalmente três diferentes abordagens para se fazer backups de dados PostgreSQL:

- Backup Lógico (SQL *Dump*);
- Backup em Nível de Sistema de Arquivos (*File System Level Backup*);
- Arquivamento Contínuo (*Continuous Archiving*).

Cada um com seus pontos positivos e negativos; que serão discutidos a seguir.

12.2 Backup Lógico (SQL Dump)

A idéia por trás deste método é gerar um arquivo de texto com comandos SQL que, ao ser utilizado para uma restauração para o servidor, recriará o banco de dados ou o cluster no mesmo estado como estava na hora do dump.

12.2.1 pg_dump

É um utilitário para fazer backup de uma base de dados do PostgreSQL.

Faz backups consistentes mesmo se a base de dados estiver sendo usada concorrentemente, como também não bloqueia outros usuários a acessarem a base (leitura ou escrita).

Os *dumps* podem ser feitos em formato de texto plano; que é restaurado pelo utilitário `psql`, ou em formato binário; restaurado pelo utilitário `pg_restore`:

- Texto plano: psql;
- Binário: pg_restore;

A partir da versão 9.3 do PostgreSQL o `pg_dump` conta com uma opção muito interessante:

`-j njobs` ou `--jobs=njobs`

Tal opção permite o *dump* rodar em paralelo dividindo em N trabalhos (*jobs*) simultaneamente.

Essa opção reduz o tempo em que o *dump* é feito, mas também aumenta a carga no servidor de banco de dados.

Só pode ser usada com o formato de saída de diretório (`-Fd`), porque é o único formato onde múltiplos processos podem escrever seus dados ao mesmo tempo.

O `pg_dump` abre N *jobs* + 1 conexões, o que depende da configuração de `max_connections` ser alta o suficiente para acomodar todas conexões.

12.2.2 pg_restore

O `pg_restore` é um utilitário para restaurar uma base PostgreSQL de um arquivo criado pelo `pg_dump` em um dos formatos não texto plano. Ele emitirá comandos necessários para reconstruir a base ao estado como era no momento que o arquivo de dump foi salvo.

Os arquivos de dump são projetos para serem portáveis entre arquiteturas.

Dentre seus parâmetros de configuração é interessante comentar sobre:

`-j number-of-jobs, --jobs=number-of-jobs`

Faz com que a restaurado seja dividida em N *jobs* (trabalhos), sendo que cada *job* é um processo ou uma *thread*, dependendo do sistema operacional e usa uma conexão separada para o servidor. É aceito apenas no formato *custom*.

Dividindo a tarefa em vários *jobs* reduz o tempo, porém exige mais do servidor.

Com o usuário postgres, criar o banco de dados vazio:

```
$ createdb pagila
```

Download do banco compactado em formato .zip no diretório do usuário postgres:

```
$ wget -c \
http://pgfoundry.org/frs/download.php/1719/pagila-0.10.1.zip \
-O ~/pagila.zip
```

Descompactando o arquivo baixado:

```
$ unzip ~/pagila.zip -d /tmp/
```

Gerando a estrutura do banco pelo arquivo SQL:

```
$ psql -f /tmp/pagila-0.10.1/pagila-schema.sql pagila
```

Inserindo os dados:

```
$ psql -f /tmp/pagila-0.10.1/pagila-data.sql pagila
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Criação de diretórios:

```
$ mkdir -pm 700 /tmp/pg_dumps/{texto,custom,tar,compact}
```

Dump formato diretório em 7 jobs:

```
$ pg_dump -j7 -Fd pagila -f /tmp/pg_dumps/diretorio
```

Apagando o banco de dados pagila e criando um novo vazio:

```
$ dropdb pagila && createdb pagila
```

Fazendo o restore do formato diretório:

```
$ pg_restore -d pagila -Fd /tmp/pg_dumps/diretorio
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Dump formato tar:

```
$ pg_dump -Ft pagila > /tmp/pg_dumps/tar/pagila.tar
```

Apagando o banco de dados pagila e criando um novo vazio:

```
$ dropdb pagila && createdb pagila
```

Fazendo o restore do formato tar:

```
$ pg_restore -d pagila -Ft /tmp/pg_dumps/tar/pagila.tar
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Dump formato custom:

```
$ pg_dump -Fc pagila > /tmp/pg_dumps/custom/pagila.dump
```

Apagando o banco de dados pagila e criando um novo vazio:

```
$ dropdb pagila && createdb pagila
```

Fazendo o restore do formato custom:

```
$ pg_restore -j5 -d pagila -Fc /tmp/pg_dumps/custom/pagila.dump
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Dump formato texto com instruções de criação da base de dados:

```
$ pg_dump -C pagila > /tmp/pg_dumps/texto/pagila.sql
```

Apagando o banco de dados pagila:

```
$ dropdb pagila
```

Restaurando o banco de dados pagila pelo arquivo texto:

```
$ psql -f /tmp/pg_dumps/texto/pagila.sql
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Dump com criação da base e compactado em formato gzip:

```
$ pg_dump -C pagila | gzip -9 > /tmp/pg_dumps/compact/pagila.gz
```

Apagando o banco de dados pagila:

```
$ dropdb pagila
```

Restaurando o banco a partir de um arquivo gzip:

```
$ gunzip -c /tmp/pg_dumps/compact/pagila.gz | psql
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Dump com criação da base e compactado em formato bzip2:

```
$ pg_dump -C pagila | bzip2 -9 > /tmp/pg_dumps/compact/pagila.bz2
```

Apagando o banco de dados pagila:

```
$ dropdb pagila
```

Restaurando o banco a partir de um arquivo bzip2:

```
$ bunzip2 -c /tmp/pg_dumps/compact/pagila.bz2 | psql
```

Listando as tabelas do banco de dados pagila:

```
$ psql -c "SELECT relname tabela FROM pg_stat_user_tables \
ORDER by tabela LIMIT 5;" pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

No final das contas, vemos que podemos reduzir o tempo do dump ou reduzir o tamanho do arquivo ou diretório resultante conforme o tipo de dump utilizado:

Tamanho	Arquivo/Diretório
477K	pagila.bz2
611K	pagila.gz
688K	pagila.dump
744K	diretorio
2,8M	pagila.sql
2,9M	pagila.tar

Tabela 2: Resultados em tamanho dos dumps

12.2.3 pg_dumpall

O pg_dumpall é um utilitário que extrai um cluster de banco de dados para um script.

Seu *modus operandi* é similar ao pg_dump, no entanto, o pg_dumpall somente tem o formato de texto plano e faz dump de todo o cluster e não somente de uma base de dados.

O pg_dumpall também é muito utilizado para fazer migrações, seja da mesma versão do PostgreSQL ou entre versões diferentes.

Criação de outro cluster:

```
$ initdb -U postgres -D /tmp/cluster2
```

Modificando a porta de escuta de serviço no postgresql.conf para 5433:

```
$ sed -i 's/\#port = 5432/port = 5433/g' /tmp/cluster2/postgresql.conf
```

Inicialização do cluster:

```
$ pg_ctl -D /tmp/cluster2/ start
```

Listando as bases de dados do novo cluster:

```
$ psql -p 5433 -U postgres -c 'SELECT datname FROM pg_database;'

  datname
-----------
 template1
 template0
 postgres
```

Dump direto de um cluster para outro (velho para novo):

```
$ pg_dumpall | psql -p 5433 -U postgres
```

Listar tabelas do banco de dados pagila no novo cluster:

```
$ psql -p 5433 -U postgres -c \
'SELECT relname tabela FROM pg_stat_user_tables ORDER by tabela LIMIT 5' pagila

  tabela
----------
 actor
 address
 category
 city
 country
```

Redirecionando a saída do dump para um arquivo:

```
$ pg_dumpall > /tmp/cluster.dump.sql
```

Redirecionando a saída do dump para o compactador bzip2 que cria um arquivo compactado:

```
$ pg_dumpall | bzip2 -9 -c > /tmp/cluster.dump.sql.bz2
```

Listando os dois arquivos criados e seus respectivos tamanhos:

```
$ ls -lh /tmp/cluster.* | awk '{print $(NF) " => " $5}'

/tmp/cluster.dump.sql => 2.8M
/tmp/cluster.dump.sql.bz2 => 477K
```

Parando e depois excluindo o novo cluster:

```
$ pg_ctl -D /tmp/cluster2 -m immediate stop && rm -fr /tmp/cluster2
```

Criação de outro cluster:

```
$ initdb -U postgres -D /tmp/cluster2
```

Modificando a porta de escuta de serviço no postgresql.conf para 5433:

```
$ sed -i 's/\#port = 5432/port = 5433/g' /tmp/cluster2/postgresql.conf
```

Inicialização do cluster:

```
$ pg_ctl -D /tmp/cluster2/ start
```

Cronometrando a importação de um arquivo de dump de texto puro:

```
$ time psql -p 5433 -f /tmp/cluster.dump.sql

. . .

real    0m2.464s
user    0m0.016spsql -f /tmp/pagila-0.10.1/pagila-data.sql pagila
sys     0m0.304s
```

Parando e depois excluindo o novo cluster:

```
$ pg_ctl -D /tmp/cluster2 -m immediate stop && \
rm -fr /tmp/cluster2
```

Criação de outro cluster:

```
$ initdb -U postgres -D /tmp/cluster2
```

Modificando a porta de escuta de serviço no postgresql.conf para 5433:

```
$ sed -i 's/\#port = 5432/port = 5433/g' \
/tmp/cluster2/postgresql.conf
```

Inicialização do cluster:

```
$ pg_ctl -D /tmp/cluster2/ start
```

Cronometrando a importação de um arquivo de dump compactado:

```
$ time bunzip2 -dc /tmp/cluster.dump.sql.bz2 | psql -p 5433

. . .

real    0m2.620s
user    0m0.072s
sys     0m0.424s
```

Ao utilizar compactação, economizará espaço, mas se for preciso fazer uma restauração a partir de um dump compactado levará mais tempo.

Parando o cluster:

```
$ pg_ctl -D /tmp/cluster2/ stop
```

12.3 Backup Físico Off Line: Snapshot

Para esta estratégia de backup é necessário que o cluster esteja inativo.
É feita uma cópia do diretório de dados (`$PGDATA`).

Parando o cluster de trabalho:

```
$ pg_ctl stop
```

Acessa o diretório pai de PGDATA:

```
$ cd `dirname "${PGDATA}"`
```

A partir do diretório pai cria um tar.gz em /tmp/ do diretório do cluster (data):

```
$ tar cvzf /tmp/cluster.tar.gz `basename ${PGDATA}`
```

Descompactar o arquivo em /tmp:

```
$ tar xvf /tmp/cluster.tar.gz -C /tmp/
```

A partir da descompactação criou-se o diretório `data` em `/tmp`.

Será preciso editar seu postgresql.conf e desativar alguns parâmetros de configuração:

```
$ vim /tmp/data/postgresql.conf
```

Desativar os seguintes parâmetros comentando-os:

- data_directory;
- hba_file;
- ident_file;
- external_pid_file.

Esses pârametros têm vínculos com o diretório de dados do cluster principal, por isso precisam ser comentados.

Inicializando o cluster de teste:

```
$ pg_ctl -D /tmp/data start
```

Listando os bancos do cluster:

```
$ psql -c 'SELECT datname FROM pg_database;'

  datname
-----------
 template1
 template0
 postgres
 pagila
```

Parando o cluster de teste:

```
$ pg_ctl -D /tmp/data/ stop
```

Inicializando o cluster de trabalho:

```
$ pg_ctl start
```

12.4 Backup Físico On Line

O Backup físico *on line* os arquivos de dados da base são copiados com o serviço do banco rodando.

Antes de mais nada é preciso verificar como estão alguns parâmetros.

Verificando o contexto do parâmetro:

```
$ psql -c "SELECT name, setting, context FROM pg_settings \
WHERE name IN ('wal_level', 'archive_mode', 'archive_command');"
```

```
      name       |  setting   |  context
-----------------+------------+------------
 archive_command | (disabled) | sighup
 archive_mode    | off        | postmaster
 wal_level       | replica    | postmaster
```

Criação de diretórios de backup no mesmo nível de PGDATA:

```
$ mkdir -pm 700 `dirname "${PGDATA}"`/bkp_{data,wal}
```

No mesmo nível hierárquico de diretório de PGDATA foram criados os diretórios `bkp_data` e `bkp_wal`.

Edição do arquivo de configuração:

```
$ vim ${PGDATA}/postgresql.conf

wal_level = replica
archive_mode = on
archive_command = 'rsync -a %p `dirname "${PGDATA}"`/bkp_wal/%f'
...
```

Reinicialização do cluster:

```
$ pg_ctl restart
```

Variável de ambiente para identificar o momento atual:

```
$ NOW=`date +%Y%m%d-%H%M`
```

Verificando o valor da variável:

```
$ echo ${NOW}

20180312-1342
```

Função pg_start_backup que avisa o servidor que será feito um backup:

```
$ CHKPNT_START=`psql -Atqc "SELECT pg_start_backup('${NOW}');"`
```

Localização inicial do log de transações do backup:

```
$ echo ${CHKPNT_START}
```

```
0/5000028
```

Verificando o conteúdo do arquivo de etiqueta de backup:

```
$ cat ${PGDATA}/backup_label
```

```
START WAL LOCATION: 0/5000028 (file 000000010000000000000005)
CHECKPOINT LOCATION: 0/5000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2018-03-12 13:43:17 -03
LABEL: 20180312-1342
```

Converte a localização de backup em nome do arquivo de segmento do WAL:

```
$ psql -c "SELECT pg_walfile_name('${CHKPNT_START}');"
```

```
     pg_walfile_name
--------------------------
 000000010000000000000005
```

Ir ao diretório pai de PGDATA:

```
$ cd `dirname "${PGDATA}"`
```

Cria um backup compactado do diretório PGDATA sem o caminho inteiro:

```
$ tar czf bkp_data/cluster.tar.gz `basename ${PGDATA}`
```

Avisa o servidor que o backup foi concluído:

```
$ CHKPNT_FINAL=`psql -Atqc "SELECT pg_stop_backup();"`
```

Localização final do log de transações do backup:

```
$ echo ${CHKPNT_FINAL}
```

```
0/5000130
```

Converte a localização de backup em nome do arquivo de segmento do WAL:

```
$ psql -c "SELECT pg_walfile_name('${CHKPNT_FINAL}');"

     pg_walfile_name
--------------------------
 000000010000000000000005
```

Parando o cluster de trabalho:

```
$ pg_ctl stop
```

Apagando o diretório de dados:

```
$ rm -fr ${PGDATA}
```

Restaurando o backup:

```
$ tar -xvf `dirname "${PGDATA}"`/bkp_data/cluster.tar.gz \
-C `dirname "${PGDATA}"`/
```

Apagando o diretorio de dados:

```
$ ls ${PGCONF}/* | xargs -i ln -sf {} ${PGDATA}/
```

Inicializando o cluster de teste:

```
$ pg_ctl start
```

Listando os bancos do cluster:

```
$ psql -c 'SELECT datname FROM pg_database;'

  datname
-----------
 template1
 template0
 postgres
 pagila
```

12.5 pg_basebackup

É um utilitário para fazer backups *on* line de *clusters* PostgreSQL. Os *backups* são feitos sem afetar outros clientes na base e podem ser usados para PITR (*Point In Time Recovery*) e / ou como ponto de partida para *log shipping* ou servidores *standby* de replicação via *streaming*.

Diretório pai de PGDATA:

```
$ cd `dirname "${PGDATA}"`
```

Backup do cluster no diretório data.new:

```
$ pg_basebackup -P -D data.new
```

```
pg_basebackup: could not connect to server: FATAL:  no pg_hba.conf entry for replication connection from host
"[local]", user "postgres", SSL off
pg_basebackup: removing data directory "data.new"
```

O utilitário pg_basebackup cria o diretório declarado.

É necessário ter uma entrada no `pg_hba.conf` para replicação, cuja entrada é inserida na posição de nome de banco com a palavra reservada *replication*.

Inserindo a nova linha em pg_hba.conf:

```
$ echo 'local    replication    postgres    trust' >> ${PGDATA}/pg_hba.conf
```

Recarregando as novas configurações:

```
$ pg_ctl reload
```

Backup on line do cluster em /tmp/data:

```
$ pg_basebackup -P -D data.new
```

```
pg_basebackup: could not connect to server: FATAL:  number of requested
standby connections exceeds max_wal_senders (currently 0)
```

Por padrão, o valor do parâmetro `max_wal_senders` é 0 (zero). Para poder utilizar o `pg_basebackup` é preciso que esse valor seja pelo menos 1.

Edição do postgresql.conf:

```
$ vim ${PGDATA}/postgresql.conf

max_wal_senders = 2
```

Verificando o contexto do parâmetro:

```
$ psql -c "SELECT context FROM pg_settings WHERE name = 'max_wal_senders';"

  context
------------
 postmaster
```

Contexto postmaster: é preciso reiniciar o serviço!

Reiniciando o serviço:

```
$ pg_ctl restart
```

Fazendo o backup on line cujo diretório de cluster é /tmp/data:

```
$ pg_basebackup -P -D data.new
```

Parando o cluster de trabalho:

```
$ pg_ctl stop
```

Renomeando o diretório data para data.old:

```
$ mv data data.old
```

Renomeando o diretório data.new para data (PGDATA):

```
$ mv data.new data
```

Criação de links no diretório de dados:

```
$ ls ${PGCONF}/* | xargs -i ln -sf {} ${PGDATA}/
```

Inicializando o cluster de teste:

```
$ pg_ctl start
```

Listando os bancos do cluster:

```
$ psql -c 'SELECT datname FROM pg_database;'

  datname
-----------
 template1
 template0
 postgres
 pagila
```

Remoção do diretório de dados antigo:

```
$ rm -fr data.old/
```

Backup em /tmp/data:

```
$ pg_basebackup -P -D /tmp/data
```

Parando o cluster de trabalho:

```
$ pg_ctl stop
```

Para gerarmos os arquivos de configuração vamos criar um cluster temporário:

```
$ initdb -U postgres -E utf8 -D /tmp/pgtmp
```

Copiando arquivos de configuração:

```
$ cp -f /tmp/pgtmp/*.conf /tmp/data/
```

Inicializando o cluster de teste:

```
$ pg_ctl -D /tmp/data start
```

Listando os bancos do cluster:

```
$ psql -c 'SELECT datname FROM pg_database;'

  datname
-----------
 template1
 template0
 postgres
 pagila
```

Parando o cluster de teste:

```
$ pg_ctl -D /tmp/data stop
```

Inicializando o cluster de trabalho:

```
$ pg_ctl start
```

O novo backup será feito pelo utilitário *pg_basebackup* criando um novo diretório referenciando data e hora atual.

Variável de ambiente para identificar o momento atual:

```
$ NOW=`date +%Y%m%d-%H%M`
```

Fazendo o backup em tar comprimido por gzip:

```
$ pg_basebackup -P -Ft -z -Z 9 \
-D `dirname "${PGDATA}"`/bkp_data/${NOW}/
```

Dentro do diretório criado há um arquivo base.tar.gz.

Criação de cluster de teste:

```
$ mkdir -pm 700 /tmp/data2
```

Descompactação os backups dos arquivos físicos e do WAL:

```
$ tar xf bkp_data/${NOW}/base.tar.gz -C /tmp/data2/

$ tar xf bkp_data/${NOW}/pg_wal.tar.gz -C /tmp/data2/pg_wal/
```

Copiando arquivos de configuração:

```
$ cp /tmp/pgtmp/*.conf /tmp/data2/
```

Mudando a configuração de porta para 5433:

```
$ sed -i 's/\#port = 5432/port = 5433/g' /tmp/data2/postgresql.conf
```

Ajuste de permissões:

```
$ chmod -R 700 /tmp/data2 && find /tmp/data2 -type f -exec chmod 0600 {} \;
```

Inicializando o cluster de teste:

```
$ pg_ctl -D /tmp/data2/ start
```

Listando os bancos do cluster:

```
$ psql -c 'SELECT datname FROM pg_database;' -p 5433

  datname
-----------
 template1
 template0
 postgres
 pagila
```

Parando o cluster de teste:

```
$ pg_ctl -D /tmp/data2/ stop
```

13 PITR

- Arquivamento Contínuo
- PITR: Point In Time Recovery, a Máquina do Tempo!

13.1 Arquivamento Contínuo

Logs de transação (xlogs) possibilitam usar uma terceira estratégia para fazer backup de bases de dados: podemos combinar backup em nível de sistema de arquivos com o backup de arquivos do WAL.

Se precisar fazer uma recuperação, restaura-se o backup do sistema de arquivos e então reaplica-se os arquivos do WAL que foram guardados em backup para trazer o sistema ao estado atual. Essa abordagem é mais complexa para administrar do que cada uma das outras abordagens, mas tem alguns benefícios significantes:

- Dispensa backup de sistema de arquivos consistente como ponto de partida. Qualquer inconsistência interna no backup será corrigida pela reaplicação do log de transação (isso não é significantemente diferente do que acontece durante recuperação de falha). Não é necessário um sistema de arquivos com a capacidade de fazer *snapshots*, apenas o `tar` ou uma simples ferramenta de arquivamento similar.

- Por podermos combinar uma longa sequência indefinidamente de arquivos WAL para *replay*, o backup contínuo pode ser alcançado simplesmente pela continuação de arquivamento de arquivos do WAL. Isso é particularmente valioso para grandes bases de dados, onde não é conveniente fazer *backups* inteiros frequentemente.

- Não precisa fazer o *replay* de entradas do WAL de todo caminho até o fim. Poderia parar o *replay* a qualquer ponto e ter um *snapshot* consistente da base de dados como era antes. Assim, essa técnica suporta recuperação em um ponto no tempo (*point-in-time recovery*): é possível restaurar a base de dados para seu estado em qualquer tempo desde que seu backup foi feito.

- Se continuamente alimentarmos as entradas de arquivos de WAL para outra máquina que tem sido carregada com o mesmo arquivo de backup base, tem-se um sistema de *warm standby*: a qualquer ponto pode-se levantar a segunda máquina e vai ter uma cópia quase atual da base de dados.

Aviso:

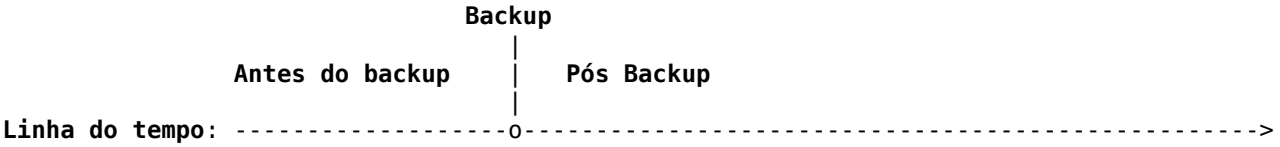
`pg_dump` e `pg_dumpall` não produzem *backups* em nível de sistema de arquivos e não podem ser usados como parte de uma solução de arquivamento contínuo. Tais *dumps* são lógicos e não contêm informação suficiente para ser usada pelo *replay* do WAL.

13.2 PITR: Point In Time Recovery, a Máquina do Tempo!

Em português significa “Recuperar em um Ponto no Tempo”.

É um recurso extremamente interessante para não somente poder recuperar uma base de dados, mas também ter a facilidade de recuperar a base em um determinado momento.

Esse determinado momento é a partir de quando fazemos um backup da base. Tudo o que for feito depois desse backup pode ser recuperado como se fosse uma máquina do tempo. Ou seja, se o ponto que quer recuperar é depois do backup, é possível fazê-lo!



Como demonstrado no diagrama acima, conforme a linha do tempo, só a partir do momento que se faz um backup é que se pode fazer o PITR.

Ou seja, antes do backup não é possível fazer o *Point in Time Recovery*.

13.2.1 pg_archivecleanup vs pypg_wal_archive_clean

Ambos têm a mesma função, que é limpar o diretório de backups de xlogs, eliminando os arquivos de log de transação mais antigos.

- pg_archivecleanup [1]: Feito em Linguagem C, é o aplicativo padrão para esse fim.
- pypg_wal_archive_clean [2]: Script feito em Python, cuja vantagem está em listar os arquivos que podem ser apagados, além de oferecer a opção de em vez de um diretório passar o caminho para um xlog arquivado. Se assim for especificado, ou seja, o caminho para um xlog arquivado, todos os arquivos mais antigos do que ele serão removidos.

[1] http://www.postgresql.org/docs/current/static/pgarchivecleanup.html

[2] https://github.com/juliano777/pypg_tools/blob/master/py/pypg_wal_archive_clean.py

13.2.2 recovery.conf: Configuração de Recuperação

Suas configurações só aplicam durante a recuperação.
Não podem ser alteradas uma vez que a operação de restauração se iniciou.
Suas configurações, assim como no `postgresql.conf` são especificadas no formato `nome = 'valor'`; um parâmetro especificado por linha.
O caractere sustenido (`#`) é utilizado para comentários.
Para embutir um apóstrofo em um valor de parâmetro, escreva dois apóstrofos (`''`).
Um exemplo está disponível no diretório `share` da instalação (`share/recovery.conf.sample`).

Variáveis Especiais

- **`%f`**: Nome do arquivo de segmento para buscar do arquivamento, por exemplo: `000000020000000000000001A`;

- **`%p`**: Caminho completo de destino, por exemplo:

  `/var/lib/pgsql/10/data/pg_wal/000000020000000000000001A`

- **`%r`**: Nome do arquivo que contém o último ponto de reinício válido (e. g. `000000020000000000000001D`), que é o primeiro arquivo (mais antigo) que deve ser mantido para permitir que uma restauração seja inicializável, tal informação pode ser usada para truncar o arquivo para apenas o mínimo requerido para suportar reinicialização da restauração atual. `%r` é tipicamente usada apenas para configurações *warm standby*;

- **`%%`**: Escreve o caractere %.

Como exemplo será visto logo a seguir utilizando o banco de dados pagila [1] para fazermos os testes de laboratório.

[1] http://pgfoundry.org/projects/dbsamples/

De usuário root para usuário postgres:

```
# su - postgres
```

Como usuário postgres criar a base de dados (vazia) pagila:

```
$ createdb pagila
```

Baixar a base de dados pagila (formato zip):

```
$ wget -c \
http://pgfoundry.org/frs/download.php/1719/pagila-0.10.1.zip \
-O ~/pagila.zip
```

Descompactar o arquivo em /tmp:

```
$ unzip ~/pagila.zip -d /tmp/
```

Primeiro, criamos a estrutura da base que está no script SQL pagila-schema.sql:

```
$ psql -f /tmp/pagila-0.10.1/pagila-schema.sql pagila
```

Segundo, importamos os dados que estão no script SQL pagila-data.sql:

```
$ psql -f /tmp/pagila-0.10.1/pagila-data.sql pagila
```

Criação de diretórios de backup de dados (arquivos físicos do banco) e logs de transação (xlogs) além do diretório de recover, o qual conterá somente os xlogs necessários para a recuperação:

```
$ mkdir -pm 700 `dirname "${PGDATA}"`/{bkp_{data,wal},recover}
```

O comando anterior criou os diretórios na hierarquia de diretórios no mesmo nível de `$PGDATA` e com permissão octal 0700.

Diretórios criados:

- `bkp_data`: Backup de arquivos físicos do banco;
- `bkp_wal`: Backup de arquivos de log de transação;
- `recover`: Diretório de recuperação, só conterá os xlogs necessários.

É preciso fazer alterações no arquivo de configurações principal do PostgreSQL:

```
$ vim ${PGDATA}/postgresql.conf
```

Modificar os seguintes parâmetros conforme segue abaixo:

```
wal_level = logical
archive_mode = on
archive_command = 'rsync -a %p `dirname "${PGDATA}"`/bkp_wal/%f'
```

Moficamos o nível de informação mínimo necessário para se fazer um processo de PITR, habilitamos o arquivamento e por fim especificamos o comando de arquivamento de logs de transação utilizar.

Após fazer as alterações no postgresql.conf, salvar e sair, reiniciar o serviço:

```
$ pg_ctl restart
```

OK! Nosso servidor está devidamente configurado e pronto para que se dê início a um processo de PITR.

Variável de ambiente para identificar o momento atual:

```
$ NOW=`date +%Y%m%d-%H%M`
```

Verificando o valor da variável:

```
$ echo ${NOW}

20180313-1152
```

Função pg_start_backup que avisa o servidor que será feito um backup:

```
$ CHKPNT_START=`psql -Atqc "SELECT pg_start_backup('${NOW}');"`
```

Localização inicial do log de transações do backup:

```
$ echo ${CHKPNT_START}

0/3000028
```

Qual é o arquivo xlog de localização inicial de log de transação?:

```
$ psql -c "SELECT pg_walfile_name('${CHKPNT_START}');"

     pg_walfile_name
--------------------------
 000000010000000000000003
```

O servidor está em processo de backup?:

```
$ psql -qc "SELECT pg_is_in_backup();"

 pg_is_in_backup
-----------------
 t
```

Que horas o backup iniciou?:

```
$ psql -qc "SELECT pg_backup_start_time();"

  pg_backup_start_time
------------------------
 2018-03-13 11:53:22-03
```

Com o início do backup, através da função pg_start_backup, foi criado um arqquivo, o backup_label, que contém algumas informações interessantes:

```
$ cat ${PGDATA}/backup_label

START WAL LOCATION: 0/3000028 (file 000000010000000000000003)
CHECKPOINT LOCATION: 0/3000098
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2018-03-13 11:53:22 -03
LABEL: 20180313-1152
```

Com o comando rsync fazemos o backup de $PGDATA:

```
$ rsync --delete-before -Pav $PGDATA/ `dirname "${PGDATA}"`/bkp_data/tmp/
```

Após o rsync avisar ao servidor que o processo de backup parou:

```
$ psql -qc 'SELECT pg_stop_backup();'

 NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/3000168
```

Inserir dados a mais na tabela para fins de teste:

```
$ psql -qc "INSERT INTO actor (first_name,last_name) VALUES
       ('Roberto','Vivar'),
       ('Ramón','Bolaños'),
       ('Florinda','de las Nieves'),
       ('Carlos','Valdez'),
       ('María Antonieta','Meza'),
       ('Edgar','Villagrán'); " pagila
```

Hora antes do delete no formato que o recovery.conf pede:

```
$ STATE_0=`date "+%Y-%m-%d %H:%M:%S %Z"`
```

YYYY-mm-dd HH:MM:SS Z

- YYYY: Ano com 4 (quatro) dígitos;
- mm: Mês com 2 (dois) dígitos;
- dd: Dia
- HH: Hora com 2 (dois) dígitos;
- MM: Minuto com 2 (dois) dígitos;
- SS: Segundos com 2 (dois) dígitos;
- Z: Fuso horário (timezone).

Um pequeno tempo de espera, antes do DELETE:

```
$ sleep 7
```

O DELETE simulando um comando desastroso para o banco, dados que deverão ser recuperados via PITR e parando o serviço logo em seguida:

```
$ psql -c "DELETE FROM actor WHERE actor_id > 200;" pagila && pg_ctl stop
```

Via rsync, copiando somente os arquivos de log de transação para o diretório de backup de logs de transação (bkp_wal):

```
$ ls ${PGDATA}/pg_wal/ | \
egrep --color=never '^[[:xdigit:]]{24}$' | \
xargs -i rsync -av ${PGDATA}/pg_wal/{} \
`dirname "${PGDATA}"`/bkp_wal/
```

Limpeza de arquivos de arquivos de log de transação:

```
$ pypg_wal_archive_clean --remove `dirname "${PGDATA}"`/bkp_wal/
```

Removendo o diretório de dados:

```
$ rm -fr ${PGDATA}/*
```

Restaurando o diretório de dados:

```
$ rsync --exclude pg_wal/* -Pav --delete-before \
`dirname "${PGDATA}"`/bkp_data/tmp/ ${PGDATA}/
```

Recriando o diretório archive_status:

```
$ mkdir ${PGDATA}/pg_wal/archive_status
```

Copiando os logs de transação para o diretório de recover:

```
$ rsync --delete-before -Pav `dirname "${PGDATA}"`/bkp_wal/* \
`dirname "${PGDATA}"`/recover/
```

Criando o arquivo recovery.conf:

```
$ cat << EOF > ${PGDATA}/recovery.conf
restore_command = 'cp  `dirname "${PGDATA}"`/recover/%f %p'
recovery_target_time = '${STATE_0}'
archive_cleanup_command = 'pypg_wal_archive_clean --remove  `dirname
"${PGDATA}"`/bkp_wal/'
EOF
```

Inicializando o serviço do PostgreSQL:

```
$ pg_ctl start
```

Testando:

```
$ psql -c \
"SELECT first_name||' '||last_name actor FROM actor \
WHERE actor_id > 200;" pagila

         actor
-----------------------
 Roberto Vivar
 Ramón Bolaños
 Florinda de las Nieves
 Carlos Valdez
 María Antonieta Meza
 Edgar Villagrán
```

14 Exportação de Resultados de Consultas

- Sobre Exportação de Resultados de Consultas

14.1 Sobre Exportação de Resultados de Consultas

Pode ser conveniente ter o resultado de uma consulta armazenado em um arquivo.
O utilitário psql, nativo do PostgreSQL fornece opções para que isso seja concretizado nos formatos CSV e HTML.

14.1.1 Formato CSV

Para converter para o formato de valores separado por vírgulas é preciso que a saída seja desalinhada (opção -A) e utilizar como separador de campos (opção -F) com ponto e vírtula (;).

Todos registros da tabela actor em formato CSV:

```
$ psql -A -F ';' -c 'TABLE actor;' pagila > /tmp/pagila_actor.csv
```

14.1.2 Formato HTML

A opção -H do psql faz a conversão da saída para o formato HTML.

Criação de cabeçalho do arquivo HTML:

```
$ cat << EOF > /tmp/pagila_actor.html
<!DOCTYPE HTML>
<html lang="pt-br">
<head>
<meta charset="UTF-8">
<title>Banco pagila - Tabela actor</title>
</head>
<body>
EOF
```

Os registros:

```
$ psql -H -c 'TABLE actor;' pagila >> /tmp/pagila_actor.html
```

Rodapé do arquivo HTML:

```
$ echo -e "</body>\n</html>" >> /tmp/pagila_actor.html
```

15 Manutenção

- VACUUM
- autovacuum

15.1 Sobre Manutenção

Como qualquer outro SGBD, o PostgreSQL requer que certas tarefas sejam feitas regularmente para se ter uma performance otimizada.

As tarefas que serão discutidas neste capítulo são necessárias, mas por natureza são repetitivas e podem facilmente ser automatizadas utilizando ferramentas padrão como o agendador de tarefas do sistema operacional.

É de responsabilidade do administrador configurar os *scripts* apropriados (no agendador de tarefas) e checar se os mesmos são executados com sucesso.

15.2 VACUUM

15.2.1 Rotina de Vacumização

Bases de dados PostgreSQL precisam de um tipo de manutenção periódica chamada de vacumização (*vacuuming*).

Para muitas instalações é suficiente que esse processo seja deixado para o *daemon* de autovacuum, cuja configuração é ajustada no `postgresql.conf` (categoria `# AUTOVACUUM PARAMETERS` ou `Autovaccum` na view de catálogo `pg_settings`).

Alguns administradores vão querer completar ou mesmo substituir atividades do *daemon* por comandos `VACUUM` gerenciados manualmente, que tipicamente são executados por *scripts* que rodam por tarefas agendadas no sistema operacional.

Para configurar adequadamente vacumizações feitas manualmente, é essencial entender questões discutidas adiante. O que também é válido para administradores que optarem por depender do *daemon* de auto-vacumização.

15.2.2 Princípios Básicos da Vacumização

O comando `VACUUM` do PostgreSQL tem que processar cada tabela em uma base regular por várias razões:

- Recuperar ou reusar espaço em disco ocupado por linhas atualizadas ou removidas (tuplas mortas);
- Atualizar dados estatísticos usados pelo planejador de consultas do PostgreSQL;
- Atualizar o mapa de visibilidade, que aumenta a velocidade de buscas apenas por índices (*index-only scans*);
- Proteger contra perdas de dados muito antigos devido a ID de transações envolventes (*transaction ID wraparound*).

Há duas variantes principais de vacumização: `VACUUM` padrão e `VACUUM FULL`.

`VACUUM FULL` pode recuperar mais espaço em disco, mas funciona muito mais lentamente.

A forma padrão de `VACUUM` pode rodar em paralelo com operações de banco de dados de produção.

Comandos como `SELECT`, `INSERT`, `UPDATE`, e `DELETE` continuarão a funcionar normalmente, embora não seja possível modificar a estrutura de uma tabela (como `ALTER TABLE`) enquanto durar o processo de vacumização.

`VACUUM FULL` exige trava exclusiva na tabela que ele estiver trabalhando, e portanto não pode ser feito em paralelo com outro uso de tabela.

Normalmente, administradores devem se esforçar para usar `VACUUM` padrão e evitar `VACUUM FULL`.

`VACUUM` cria um grande tráfego I/O, que pode causar um desempenho ruim para outras sessões ativas.

Existem parâmetros de configuração que podem ser ajustados para reduzir o impacto na performance da vacumização rodando em *background*.

15.2.3 Recuperando Espaço em Disco

No PostgreSQL, um `UPDATE` ou `DELETE` de uma linha não remove imediatamente a versão antiga dessa linha.

Essa abordagem é necessária para obter benefícios de controle de concorrência multiversão (MVCC): a versão da linha não deve ser apagada enquanto ainda estiver potencialmente visível para outras transações.

Mas eventualmente, uma versão de linha desatualizada ou removida não é mais necessária para qualquer transação.

Essas linhas que não têm mais utilidade após serem apagadas são conhecidas também por como tuplas mortas.

O espaço que ela ocupa deve ser recuperado para reúso para novas linhas, para evitar crescimento de requerimentos de disco sem limites. Isso é feito ao se rodar o `VACUUM`.

A forma padrão de `VACUUM` remove as tuplas mortas em tabelas e índices e marca o espaço disponível para futuro reúso. No entanto, isso não fará com que retorne o espaço liberado para o sistema operacional, exceto em um caso especial onde uma ou mais páginas no fim da tabela se torne inteiramente livre e se consiga facilmente uma trava (*lock*) exclusiva de tabela.

Em contraste, `VACUUM FULL` compacta ativamente tabelas escrevendo completamente a nova versão do arquivo da tabela sem espaços mortos. Isso minimiza o tamanho da tabela, mas pode levar muito tempo e também requer espaço extra em disco para a nova cópia da tabela até que se finalize a operação.

Obs.:

Se você tem uma tabela cujo conteúdo inteiro é apagado em uma base periódica, considere fazer isso como **`TRUNCATE`** em vez de `DELETE` seguido de `VACUUM`.

O comando `TRUNCATE` remove o conteúdo inteiro de uma tabela imediatamente, sem requerer em seguida um `VACUUM` ou `VACUUM FULL` para recuperar o espaço em disco não utilizado. A desvantagem é que semânticas MVCC estritas são violadas.

15.2.4 Atualizando o Planejador de Estatísticas

O planejador de consultas do PostgreSQL depende de informações estatísticas sobre conteúdos de tabelas para gerar bons planos de consultas.

Essas estatísticas são coletadas pelo comando `ANALYZE`, que pode ser invocado por si só ou como uma etapa opcional em `VACUUM`.

É importante ter estatísticas razoáveis, caso contrário escolhas pobres de planos podem degradar o desempenho do banco de dados.

O *daemon autovacuum*, se habilitado, automaticamente envia comandos `ANALYZE` sempre que o conteúdo de uma tabela tiver mudado suficientemente.

Mas, administradores devem preferir depender de operações `ANALYZE` manualmente agendadas, especialmente se é sabido que a atividade de atualização em uma tabela não afetará as estatísticas de colunas “interessantes”.

O *daemon* agenda `ANALYZE` estritamente como uma função de número de linhas inseridas ou atualizadas; ele não tem conhecimento do que se vai levar a alterações estatísticas significativas. Assim como acontece em vacumização para recuperação de espaço, atualizações frequentes de estatísticas são mais úteis para tabelas atualizadas massivamente do que para aquelas raramente atualizadas.

15.2.5 Atualizando o Mapa de Visibilidade

O `VACUUM` mantém um mapa de visibilidade pra cada tabela para manter o controle de que páginas contém apenas tuplas que se sabe que devem ser visíveis para todas transações ativas (e todas transações futuras, até a página ser novamente modificada). Isso tem dois propósitos.

Primeiro, a vacumização por si só pode pular tais páginas na próxima vez que rodar, desde que não tenha nada para limpar.

Segundo, permite ao PostgreSQL responder algumas consultas usando apenas índices, sem referenciar a tabela subjacente.

Desde que índices PostgreSQL não contenham informação de visibilidade de tuplas, uma pesquisa normal de índice busca a pilha de tupla para cada entrada de índice correspondente, para verificar se deve ser visto pela transação atual. Uma busca *index-only*, por outro lado, verifica o mapa de visibilidade primeiro. Se é sabido que todas as tuplas na página são visíveis, a busca em pilha pode ser ignorada. Isso é mais perceptível em grandes conjuntos de dados, onde o mapa de visibilidade pode impedir os acessos ao disco.

O mapa de visibilidade é muito menor do que a pilha, de modo que pode ser facilmente armazenado em cache, mesmo quando a pilha é muito grande.

15.2.6 Prevenindo Falhas de ID de Transações Envolventes (Transaction ID Wraparound)

A semântica de transação MVCC do PostgreSQL depende de ser capaz de comparar números de ID de transação (XID): uma versão de linha com uma inserção XID maior do que a atual XID de transação estar “no futuro” e não deve ser visível para a transação atual.

Mas uma vez que IDs de transação tem tamanho limitado (32 bits) um *cluster* que roda por um longo tempo (mais do que 4 bilhões de transações) sofreriam com a ID de transação envolvente: o contador de XID envolve em torno de zero e repentinamente transações que estavam no passado parecem estar no futuro; o que significa que sua saída pode se tornar invisível. Resumindo, perda de dados catastrófica. Na realidade os dados ainda estão lá, mas o que não é muito confortante se não conseguir acessá-los.

Para prevenir isso, é necessário vacumizar cada tabela em cada banco de dados pelo menos uma vez a cada dois bilhões de transações.

A razão pela qual a vacumização periódica resolve o problema é que o PostgreSQL reserva uma XID especial como *FrozenXID* (XID congelada).

Essa XID não segue a comparação de XID normal e é sempre considerada mais antiga do que toda XID normal.

XIDs normais são comparadas usando aritmética modulo-2^{32}. Isso significa que para toda XID normal, há dois bilhões de XIDs que são mais “antigas” e dois bilhões que são “mais novas”; outra maneira de dizer que o espaço de uma XID normal é circular sem

ponto final.

Portanto, uma vez que uma versão de linha tenha sido criada com uma XID normal particular, a versão de linha parecerá estar “no passado” para as próximas dois bilhões de transações, não importa que XID normal está se falando sobre.

Se a versão da linha ainda existe após mais do que dois bilhões de transações, ela parecerá de repente a estar no “futuro”.

Para evitar isso, as versões antigas de linhas devem ser transferidas a XID *FrozenXID* algum tempo antes que eles atinjam o marca de dois bilhões de transações de idade. Uma vez que essas XIDs especiais são atribuídas, parecem estar “no passado” para todas as operações normais, independente de questões correlatas, e então tais versões de linha serão válidas até serem apagadas, não importa quanto tempo é isso.

Essa reassimilação de antigas XIDs é controlado pelo `VACUUM`.

15.2.7 O Comando VACUUM

Coleta de lixo (*garbage collect*) e opcionalmente analisa uma base de dados.

Sem parâmetros, o `VACUUM` processa cada tabela no banco de dados atual que o usuário que executa tiver permissão.

Com um parâmetro, o `VACUUM` interpreta como nome de tabela e processa apenas ela.

Quando a lista de opções é envolvida por parênteses, as opções podem ser escritas em qualquer ordem.

Sem parênteses (obsoleto), asopções devem ser especificadas na ordem exata.

Com parênteses foi adicionada na versão 9.0.

- **`FULL`**: Faz a vacumização “completa”, que pode recuperar mais espaço, mas leva mais tempo e exige trava exclusiva de tabela. Precisa de espaço em disco extra, uma vez que ele faz uma nova cópia da tabela e não libera a cópia antiga até que a operação esteja completa. Normalmente isso deve apenas ser usado quando uma quantidade significante de espaço precisa ser recuperada internamente em uma tabela;

- **`FREEZE`**: Faz a vacumização com “congelamento” agressivo de tuplas. Especificando `FREEZE` é equivalente a realizar a vacumização com o parâmetro `vacuum_freeze_min_age` configurado para zero;

- **`VERBOSE`**: Modo verboso;

- **`ANALYZE`**: Atualiza estatísticas usadas pelo planejador para determinar o caminho mais eficiente para executar uma consulta.

- **`table_name`**: O nome (opcionalmente qualificado de esquema) de uma tabela específica para vacumização.

- **`column_name`**: O nome de uma coluna específica para analisar. Por padrão todas as colunas. Se uma lista de colunas for especificada, `ANALYZE` está implícito.

Cliente psql já se conectando à base de dados pagila:

```
$ psql pagila
```

Habilitar o cronômetro do psql:

```
> \timing
```

Verificando o oid e relfilenode da tabela actor:

```
> SELECT oid, relfilenode FROM pg_class WHERE relname = 'actor';

  oid  | relfilenode
-------+-------------
 16387 |       16387
```

Repare que ambos inicialmente são iguais.

Vacumização da tabela actor:

```
> VACUUM actor;
```

Verificando o oid e relfilenode da tabela actor:

```
> SELECT oid, relfilenode FROM pg_class WHERE relname = 'actor';

  oid  | relfilenode
-------+-------------
 16387 |       16387
```

Continuam iguais.

Vacumização completa da tabela actor:

```
> VACUUM FULL actor;
```

Verificando o oid e relfilenode da tabela actor:

```
> SELECT oid, relfilenode FROM pg_class WHERE relname = 'actor';

  oid  | relfilenode
-------+-------------
 16387 |       16869
```

Houve mudança do `relfilenode`.

Vacumização completa, exibindo detalhes e com análise da base de dados corrente:

```
> VACUUM (FULL, VERBOSE, ANALYZE);
```

Vacumização completa, exibindo detalhes e com análise da tabela actor:

```
> VACUUM (FULL, VERBOSE, ANALYZE) actor;
```

Apagando um registro da tabela actor:

```
> DELETE FROM actor WHERE actor_id = 201;
```

Vacumização completa, exibindo detalhes e com análise da tabela actor:

```
> VACUUM (FULL, VERBOSE, ANALYZE) actor;
```

Contando quantos registros há na tabela actor:

```
> SELECT count(*) FROM actor;

 count
-------
   205
```

205 registros

De acordo com a tabela de sistema pg_class, quantas tuplas existem na tabela actor:

```
> SELECT reltuples FROM pg_class WHERE relname = 'actor';

 reltuples
-----------
       205
```

205 tuplas

Inserindo um novo registro na tabela actor:

```
> INSERT INTO actor (first_name, last_name) VALUES ('Roberto', 'Vivar');
```

Contando quantos registros há na tabela actor:

```
> SELECT count(*) FROM actor;

 count
-------
   206
```

206 registros

De acordo com a tabela de sistema pg_class, quantas tuplas existem na tabela actor:

```
> SELECT reltuples FROM pg_class WHERE relname = 'actor';

 reltuples
-----------
       205
```

205 tuplas: Ops! Números divergentes!

Executando vacumização na tabela:

```
> VACUUM actor;
```

De acordo com o catálogo pg_class, quantas tuplas existem na tabela actor:

```
> SELECT reltuples FROM pg_class WHERE relname = 'actor';

 reltuples
-----------
       206
```

206 tuplas: Após a vacumização mostra realmente quantas tuplas existem.

Selecionando o último registro inserido:

```
> SELECT first_name FROM actor WHERE last_name = 'Vivar';

 first_name
------------
 Roberto
```

Removendo:

```
> DELETE FROM actor WHERE last_name = 'Vivar';
```

De acordo com a tabela de sistema pg_class, quantas tuplas existem na tabela actor:

```
> SELECT reltuples FROM pg_class WHERE relname = 'actor';

 reltuples
-----------
       206
```

206 tuplas

Vacumização:

```
> VACUUM actor;
```

De acordo com a tabela de sistema pg_class, quantas tuplas existem na tabela actor:

```
> SELECT reltuples FROM pg_class WHERE relname = 'actor';

 reltuples
-----------
       205
```

205 tuplas.

15.3 autovacuum

O autovacuum é uma característica opcional do PostgreSQL altamente recomendada, cujo propósito é automatizar a execução dos comandos VACUUM e ANALYZE.

Quando habilitado, o autovacuum verifica a existência de tabelas que tenham uma grande quantidade de linhas inseridas, atualizadas ou apagadas.

Essas verificações usam a facilidade de coleção de estatísticas; portanto, o autovacuum não pode ser usado a não ser que o parâmetro track_counts do postgresql.conf esteja habilitado.

Por padrão, a autovacumização é habilitada e seus parâmetros relativos apropriadamente configurados.

O *daemon* autovacuum atualmente consiste de múltiplos processos.

Existe um processo *daemon* persistente, chamado *autovacuum launcher*, que é responsável pela partida dos processos *autovacuum worker* para todas bases de dados.

O *launcher* distribuirá o trabalho ao longo do tempo, tentando iniciar um *worker* dentro de cada base a cada autovacuum_naptime segundos.

Portanto, se a instalação tem N bases de dados, um novo *worker* será executado a cada autovacuum_naptime/N segundos.

Há um número máximo de autovacuum_max_workers permitido para rodar ao mesmo tempo.

Se existir mais do que autovacuum_max_workers bases de dados para ser processados, a próxima base de dados será processada assim que o primeiro *worker* finalizar.

Cada processo *worker* verificará cada tabela em sua base de dados e executará VACUUM e / ou ANALYZE conforme precisar.

log_autovacuum_min_duration pode ser usado para monitorar a atividade de autovacuum.

Se muitas tabelas grandes se tornarem elegíveis para vacumização em um curto espaço de tempo, todos autovacuum *workers* podem ficar ocupados com a vacumização dessas tabelas por um longo período. Isso resultaria em outras tabelas e bases de dados não sendo vacumizados até que um worker esteja disponível. Não há limite em como muitos workers podem estar em uma única base de dados, mas *workers* fazem tentam evitar repetir o trabalho que já tem sido feito por outros *workers*.

Note que a quantidade de *workers* rodando não contam para limites max_connections ou superuser_reserved_connections.

Tabelas cujo valor pg_class.relfrozenxid é maior do que autovacuum_freeze_max_age transações antigas são sempre vacumizadas (isso também se aplica para aquelas tabelas cujo tempo máximo de congelamento tenha sido modificado via parâmetros de armazenamento. Caso contrário, se o número de tuplas obsoletas desde o último VACUUM excede o ponto de partida de vacumização "*vacuum threshold*", a tabela é vacumizada.

O ponto de partida de vacuum é definido como:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

Onde:

- **vacuum base threshold**: autovacuum_vacuum_threshold;
- **vacuum scale factor**: autovacuum_vacuum_scale_factor;
- **number of tuples**: pg_class.reltuples;

Via consulta SQL utilizando a tabela actor:

```
> SELECT (s1.setting::REAL + s2.setting::REAL + c.reltuples::REAL)
    AS "vacuum threshold"
    FROM pg_settings AS s1, pg_settings AS s2, pg_class AS c
    WHERE
        s1.name = 'autovacuum_vacuum_threshold'
        AND s2.name = 'autovacuum_vacuum_scale_factor'
        AND c.relname = 'actor';
```

16 Catálogos de Sistema

- Sobre Catálogos de Sistema
- As Tabelas do Catálogos de Sistema
- Views de Sistema

16.1 Sobre Catálogos de Sistema

Os catálogos de sistema são os locais onde o sistema gerenciador de banco de dados relacional armazena os metadados, tais como informações sobre tabelas e colunas e informações de controle interno.

O sistema de catálogos do PostgreSQL são tabelas regulares.

É possível apagar e recriar as tabelas, adicionar colunas, inserir e alterar valores, e claro, danificar seu sistema dessa forma.

Normalmente, não se deve alterar os catálogos do sistema manualmente, há sempre os comandos SQL para fazer isso. (Por exemplo, `CREATE DATABASE` insere uma linha no catálogo `pg_database` e cria o banco de dados na hora em disco).

Há algumas exceções para operações particularmente esotéricas (incomuns), como a adição de métodos de acesso de índice.

16.2 As Tabelas do Catálogos de Sistema

A maior parte dos catálogos de sistema é copiada do banco de dados modelo (*template*) durante a criação de uma base de dados e depois disso um banco de dados específico.

Alguns poucos catálogos são fisicamente compartilhados através de todas bases de dados em um *cluster*; esses são observados nas descrições dos catálogos individuais.

https://www.postgresql.org/docs/current/static/catalogs.html

Da tabela actor verificando índices e como foram criados:

```
> SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'actor';

      indexname      |                                  indexdef
---------------------+-------------------------------------------------------------------------------
 actor_pkey          | CREATE UNIQUE INDEX actor_pkey ON public.actor USING btree (actor_id)
 idx_actor_last_name | CREATE INDEX idx_actor_last_name ON public.actor USING btree (last_name)
```

Quais papéis tem o atributo SUPERUSER?:

```
> SELECT rolname FROM pg_catalog.pg_authid WHERE rolsuper;

 rolname
----------
 postgres
```

Listando todas as bases de dados que não sejam os modelos e a base padrão:

```
> SELECT datname FROM pg_database WHERE datname !~ 'postgres|^template.';

 datname
---------
 pagila
```

16.3 Views de Sistema

Em adição aos catálogos, o PostgreSQL oferece uma série de views embutidas (*built in*).

Algumas views de sistema fornecem acesso facilitado para algumas consultas em catálogos do sistema.

Outras views oferecem acesso ao estado interno do servidor.

O esquema de informações (*information_schema*) oferece um conjunto alternativo de views que sobrepõe funcionalidades de views de sistema.

Uma vez que o esquema de informações é padrão SQL ao passo que as views aqui descritas são específicas do PostgreSQL, geralmente é melhor usar o esquema de informações se ele fornece todas as informações que você precisa.

Há algumas views adicionais que fornecem acesso a resultados do coletor de estatísticas.

Exceto quando mencionado, todas as views descritas aqui são somente leitura.

https://www.postgresql.org/docs/current/static/views-overview.html

Criação de um grupo:

```
> CREATE ROLE financeiro;
```

Criação de usuário sem pertencer a nenhum grupo inicialmente:

```
> CREATE ROLE marcia LOGIN;
```

Criação de outro usuário já pertencendo ao grupo financeiro:

```
> CREATE ROLE alice IN ROLE financeiro LOGIN;
```

Criação de usuário:

```
> CREATE ROLE zeninguem LOGIN;
```

Concedendo ao grupo financeiro ao usuário marcia com opção de admin:

```
> GRANT financeiro TO marcia WITH ADMIN OPTION;
```

Listando o conteúdo de pg_auth_members:

```
> TABLE pg_auth_members;

 roleid | member | grantor | admin_option
--------+--------+---------+--------------
  36461 |  36463 |      10 | f
  36461 |  36462 |      10 | t
```

Respectivamente: ID de grupo, ID de usuário, ID do papel que concedeu o direito ao usuário de pertencer ao grupo e tem privilégios administrativos.

Consultando a view de usuários:

```
> SELECT usename FROM pg_user;

  usename
-----------
 postgres
 marcia
 alice
 zeninguem
```

Consultando view de grupos:

```
> SELECT groname FROM pg_group;

  groname
------------
 financeiro
```

Consultando sobre a definição da view:

```
> SELECT definition FROM pg_views WHERE viewname = 'pg_indexes';

SELECT n.nspname AS schemaname, c.relname AS tablename, i.relname
   AS indexname, t.spcname AS
   tablespace, pg_get_indexdef(i.oid)
   AS indexdef FROM ((((pg_index x JOIN pg_class c
   ON ((c.oid = x.indrelid))) JOIN pg_class i
   ON ((i.oid = x.indexrelid)))
   LEFT JOIN pg_namespace n ON ((n.oid = c.relnamespace)))
   LEFT JOIN pg_tablespace t ON ((t.oid = i.reltablespace)))
   WHERE ((c.relkind = 'r'::"char")
   AND (i.relkind = 'i'::"char"));
```

17 Information Schema

- Sobre Information Schema

17.1 Sobre Information Schema

Em SGBDs relacionais, o *Information Schema* (information-schema) é um *schema* (*namespace*) padrão que contém *views* e tabelas que fornecem informações de objetos sobre a base de dados atual.

https://www.postgresql.org/docs/current/static/information-schema.html

Criação de tabela de teste:

```
> CREATE TABLE tb_teste(
    id serial primary key,
    campo1 int2,
    campo2 int4,
    campo3 int8,
    campo4 text);
```

Obtendo informações da tabela criada:

```
> SELECT column_name, data_type
    FROM information_schema.columns
    WHERE table_name = 'tb_teste'
        AND table_schema = 'public';
```

```
 column_name | data_type
-------------+-----------
 id          | integer
 campo1      | smallint
 campo2      | integer
 campo3      | bigint
 campo4      | text
```

18 O Caminho de uma Consulta

- O caminho de uma consulta
- EXPLAIN

18.1 O caminho de uma consulta

Segue abaixo uma visão geral resumida dos estágios que uma consulta tem que passar para obter um resultado.

Conexão → Parser → Rewrite → Planner → Executor

18.1.1 A Conexão

Uma conexão de uma aplicação para o servidor PostgreSQL tem que ser estabelecida. A aplicação transmite uma consulta para o servidor e aguarda o resultado vindo do servidor.

18.1.2 Parser

No estágio do *parser* (analisador) faz uma checagem da consulta transmitida pela aplicação, verificando se a sintaxe está correta e cria uma árvore de consulta.

18.1.3 Rewrite System

O sistema de reescrita (*rewrite system*) toma a árvore de consulta criada pelo *parser* e procura por quaisquer regras (armazenadas em catálogos do sistema) para aplicá-las à árvore de consulta. Transformações são executadas de acordo com os corpos das regras.

Uma utilização do sistema de reescrita é na realização de *views*. Sempre que uma consulta é feita em uma view, o sistema de reescrita reescreve a consulta feita pelo usuário para uma consulta que acessa as tabelas que pertencem à *view*.

18.1.4 Planner / Optimizer

O planejador/otimizador (*planner/optimizer*) pega a árvore de consulta (reescrita) e cria um plano de consulta que será a entrada do executor.

Primeiro, ele cria todos os caminhos possíveis que levam ao mesmo resultado. Por exemplo, se houver um índice em uma tabela para ser buscado, há dois caminhos para procurar.

Um possivelmente é uma simples busca sequencial e outra possibilidade é fazer uso do índice.

O custo da execução de cada caminho é estimado e o que for mais barato é escolhido.

O caminho menos custoso é expandido dentro de um plano completo que o executor pode usar.

18.1.5 Executor

O executor percorre recursivamente a árvore do plano de consulta e traz as linhas no caminho representado pelo plano.

O executor faz uso do sistema de armazenamento enquanto busca tabelas, faz ordenações e junções, avalia qualificações e finalmente devolve as linhas derivadas.

18.2 EXPLAIN

O comando `EXPLAIN` exibe o plano de execução de um *statement* (comando).

Sintaxe:

```
EXPLAIN [ ( opção [, ...] ) ] statement

EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

A opção pode ser:

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

18.2.1 Descrição

Exibe o plano de execução que o planejador do PostgreSQL gera para o *statement* dado.

O plano de execução exibe como a(s) tabela(s) referenciada(s) pelo *statement* será(ão) buscada(s) (planos de busca sequencial, por índice, etc) e se múltiplas tabelas são referenciadas, que juntam algoritmos que serão usados para juntos trazerem as linhas requeridas de cada tabela de entrada.

A parte mais crítica de exibir o custo estimado de execução de um *statement*, que é a estimativa do planejador de quanto tempo vai demorar para rodar o *statement* (mensurado em unidades de custo que são arbitrárias, mas convencionalmente significa buscas em páginas de disco).

Atualmente dois números são mostrados: o custo de arranque (inicialização) antes da primeira linha ser retornada, e o custo total para retornar todas linhas.

Para a maioria das consultas o custo total é o que importa, mas em contextos como uma subconsulta em `EXISTS`, o planejador escolherá o menor custo de inicialização do menor custo total (desde que o executor pare após ter obtido uma linha de qualquer maneira).

Também, se o limite de número de linhas para retornar com a cláusula `LIMIT`, o planejador faz uma interpolação apropriada entre o custo de ponto final para estimar que plano é realmente mais barato.

A opção `ANALYZE` fará com que o *statement* seja realmente executado e não apenas planejado.

Então, as estatísticas de tempo de execução são adicionadas à tela, incluindo o tempo total decorrido dentro de cada nó de plano (em milissegundos) e a quantidade total de linhas que ele retornou atualmente.

Isso é útil para ver se as estimativas do planejador estão próximas da realidade.

Aviso:

Tenha em mente que o *statement* é atualmente executado quando a opção `ANALYZE` é usada.

Embora `EXPLAIN` descartará qualquer saída que um SELECT retorne, os efeitos do *statement* (colaterais ou não) acontecerão normalmente.

Se deseja usar `EXPLAIN ANALYZE` em um *statement* `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE AS` ou `EXECUTE` sem deixar afetar seus dados, use esta abordagem (dentro de uma transação e desfazendo com `ROLLBACK`):

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Apenas as opções `ANALYZE` e `VERBOSE` podem ser especificadas, e apenas nessa ordem, sem envolver a lista de opções em parênteses.

Antes do PostgreSQL 9.0 a sintaxe sem parênteses era o único jeito suportado.

É esperado que todas novas opções suportarão apenas a sintaxe com parênteses.

18.2.2 Parâmetros

- **`ANALYZE`** (Padrão `FALSE`): Executa o comando e mostra o tempo de execução atual e outras estatísticas.

- **`VERBOSE`** (Padrão `FALSE`): Exibe informações adicionais relativas ao plano. Especificamente, incluem a lista de colunas de saída de cada nó na árvore do plano, nomes qualificados de esquema (*schema-qualified*) de tabelas e funções.

- **`COSTS`** (Padrão `TRUE`): Custo estimado de inicialização; é o tempo gasto antes da fase de saída, *e. g.*, tempo pra fazer a ordenação em um nó.
  Custo estimado total; é indicado no pressuposto de que o nó do plano é executado para conclusão, i. e., todos linhas disponíveis são recuperadas.
  Número estimado de linhas; novamente, o nó é assumido para rodar até o fim.
  Largura (*width*) média estimada de produção de linhas por este nó do plano (em bytes).

- **BUFFERS** (Padrão FALSE): Inclui informações de uso de *buffer*.
  Especificamente, inclui o número de acesso a blocos compartilhados, lidos, sujos, e escritos, o número de acesso a blocos locais, lidos, sujos, e escritos, e o número de blocos temporários lidos e escritos. Um acesso significa que uma leitura foi evitada porque o bloco foi encontrado já em cache quando necessário.
  Blocos compartilhados contêm dados de tabelas regulares e índices; blocos locais contêm dados de tabelas e índices temporários; enquanto blocos temporários contêm dados de trabalho de curto prazo usados em ordenações, *hashes*, nós de planos Materialize, e casos similares.
  O número de blocos sujos indica o número de blocos previamente não modificados que foram mudados pela consulta; enquanto o número de blocos escritos indica o número de blocos sujos previamente despejados do cache por este *backend* durante o processamento de uma consulta.
  O número de blocos mostrados por um nível mais alto de nós inclui aqueles utilizados por todos seus nós filhos.
  No formato texto, apenas valores diferentes de zero são exibidos.
  Este parâmetro só pode ser usado quando ANALYZE é habilitado também.

- **TIMING** (Padrão TRUE): Inclui a hora em que iniciou e tempo gasto em cada nó na saída.
  O *overhead* de ler repetidamente o relógio do sistema pode retardar a consulta significativamente em alguns sistemas, então pode ser útil configurar este parâmetro para FALSE quando apenas linhas atuais contam, e não exatamente tempos são necessários.
  O tempo de execução de todo o *statement* é sempre medido, mesmo quando o nível de nó em tempo é desligado com esta opção.
  Este parâmetro só pode ser usado quando ANALYZE é habilitado também.

- **FORMAT** (Padrão TEXT): Especifica o formato de saída, que pode ser TEXT, XML, JSON, ou YAML.
  Saídas não TEXT contêm as mesmas informações, mas é mais fácil para programas analisarem.

- **boolean**: Especifica se a opção selecionada ser ligada ou desligada. Pode-se escrever TRUE, ON, ou 1 para habilitar a opção, e FALSE, OFF, ou 0 para desabilitar.
  O valor booleano pode também ser omitido, assumindo seu respectivo padrão.

- **statement**: Qualquer SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, DECLARE, ou CREATE TABLE AS statement, cujo plano de execução deseja ver.

Obs.:

Com o propósito de permitir que o planejador de consultas do PostgreSQL tome decisões mais razoáveis ao otimizar consultas, os dados de `pg_statistic` devem estar atualizados para todas as tabelas usadas em uma consulta.

Normalmente o *daemon* de autovacuum vai cuidar disso automaticamente.
Mas se uma tabela tem mudanças substanciais feitas recentemente, vai ser preciso fazer um `ANALYZE` manual em vez de esperar que um autovacuum para pegar essas mudanças.

Com o propósito de medir o custo em tempo real de cada nó no plano de execução, a implementação atual de `EXPLAIN ANALYZE` são adicionados perfis de *overhead* para execução de consultas.

Como resultado, executando `EXPLAIN ANALYZE` em uma consulta pode às vezes demorar mais do que executar a consulta normalmente.

A quantidade de *overhead* depende da natureza da consulta, bem como a plataforma que está sendo usada.

O pior caso ocorre para nós de plano que por si próprios precisam de pouco tempo para execução e em máquinas que têm relativamente chamadas de sistema operacional lentas para obter a hora do dia.

Criação da tabela de teste:

```
> SELECT generate_series(1, 2000000) AS campo INTO tb_teste;
```

Criação de índice

```
> CREATE INDEX idx_teste_comum ON tb_teste (campo);
```

Criação de índice parcial:

```
> CREATE INDEX idx_teste_div19 ON tb_teste (campo) WHERE campo % 19 = 0;
```

EXPLAIN:

```
> EXPLAIN SELECT campo FROM tb_teste WHERE campo = 975421;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using idx_teste_comum on tb_teste  (cost=0.00..8.44 rows=1
 width=4)
   Index Cond: (campo = 975421)
```

EXPLAIN ANALYZE:

```
> EXPLAIN ANALYZE SELECT campo FROM tb_teste WHERE campo = 975421;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
  Index Only Scan using idx_teste_comum on tb_teste  (cost=0.43..8.45 rows=1 width=4) (actual
time=0.126..0.128 rows=1 loops=1)
   Index Cond: (campo = 975421)
   Heap Fetches: 1
 Planning time: 0.097 ms
 Execution time: 0.155 ms
```

EXPLAIN de consulta para teste de índice parcial:

```
> EXPLAIN SELECT count(*) FROM tb_teste WHERE campo % 19 = 0;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=360.26..360.27 rows=1 width=0)

   ->  Index Only Scan using idx_teste_div19 on tb_teste
 (cost=0.00..335.26 rows=10000 width=0)
```

EXPLAIN ANALYZE:

```
> EXPLAIN ANALYZE SELECT count(*) FROM tb_teste WHERE campo % 19 = 0;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=360.26..360.27 rows=1 width=0) (actual time=92.027..92.027
 rows=1 loops=1)
   ->  Index Only Scan using idx_teste_div19 on tb_teste  (cost=0.00..335.26
 rows=10000 width=0) (actual time=0.168..73.659 rows=105263 loops=1)
         Heap Fetches: 105263
 Total runtime: 92.069 ms
```

EXPLAIN ANALYZE e etc...:

```
> EXPLAIN (ANALYZE, FORMAT JSON, VERBOSE, BUFFERS)
SELECT count(*) FROM tb_teste WHERE campo % 19 = 0;

                QUERY PLAN
-------------------------------------------
 [                                        +
   {                                      +
     "Plan": {                            +
       "Node Type": "Aggregate",          +
       "Strategy": "Plain",               +
       "Startup Cost": 360.26,            +
       "Total Cost": 360.27,              +
       "Plan Rows": 1,                    +
       "Plan Width": 0,                   +

. . .
```

19 Coletor de Estatísticas

- Sobre Coletor de Estatísticas
- Configuração do Coletor de Estatísticas
- Visualizando Estatísticas Coletadas

19.1 Sobre Coletor de Estatísticas

É um subsistema que suporta coleta e relatório de informações sobre atividades do servidor.

Atualmente, o coletor pode contar acessos a tabelas e índices em ambos blocos de disco e tupla individual.

Ele também controla o número total de linhas em cada tabela e informações sobre ações de `VACUUM` e `ANALYZE` para cada tabela.

Ele pode também conta chamadas a funções criadas por usuários e o total de tempo gasto em cada uma.

O PostgreSQL também suporta relatório do comando exato atualmente sendo executado por outros processos servidores. Essa facilidade é independente do processo coletor.

https://www.postgresql.org/docs/10/static/monitoring-stats.html

19.2 Configuração do Coletor de Estatísticas

Coletar estatísticas tem seu preço, tem algum *overhead* na execução de consultas. O sistema pode ser configurado para coletar ou não informações.

Isso é controlado por parâmetros de configuração que são normalmente configurados em `postgresql.conf`:

- **`track_activities`**: Habilita monitoramento de comandos que estão sendo executados por qualquer processo servidor;
- **`track_counts`**: Controla se estatísticas são coletadas sobre acessos a tabelas e índices;
- **`track_functions`**: Habilita o acompanhamento do uso de funções definidas pelo usuário;
- **`track_io_timing`**: Habilita o monitoramento de tempos de blocos lidos e escritos.

Normalmente esses parâmetros são configurados no `postgresql.conf`, então eles se aplicam a todos processos servidores, mas é possível habilitá-los ou desabilitá-los em sessões individuais usando o comando `SET`. (Para prevenir usuários comuns de esconder suas atividades do administrador, apenas superusuários são permitidos a mudar esses parâmetros com `SET`.)

O coletor de estatísticas transmite as informações coletadas para outros processos PostgreSQL através de arquivos temporários.

Esses arquivos são armazenados no diretório definido pelo parâmetro `stats_temp_directory`, cujo diretório é `pg_stat_tmp` por padrão.

Para melhor performance, `stats_temp_directory` pode ser colocado em um sistema de arquivos baseado em disco virtual em memória RAM, diminuindo requerimentos físicos de I/O.

Quando o servidor pára, uma cópia permanente de dados estatísticos é armazenada no subdiretório global, de modo que as estatísticas podem ser mantidas quando o servidor reinicializar.

19.3 Visualizando Estatísticas Coletadas

Muitas *views* predefinidas, listadas em [1], estão disponíveis para exibir os resultados de coleta de estatísticas.

Alternativamente, pode-se fazer *views* personalizadas usando funções estatísticas relacionadas.

Quando se usa as estatísticas para monitorar a atividade atual, é importante notar que as informações não são atualizadas instantaneamente.

Cada processo servidor individual transmite novas contagens estatísticas para o coletor pouco antes de ficar ocioso; então uma consulta ou transação ainda em progresso não afetam os totais exibidos.

Também, o coletor por si só emite um novo relatório no máximo uma vez por `PGSTAT_STAT_INTERVAL` ms (500 ms a não ser que tenha sido alterado durante a compilação do servidor).

Então a informação exibida fica defasada com relação à atividade atual. Porém, a informação da consulta atual coletada por `track_activities` é sempre atualizada.

Outro ponto importante é que quando um processo servidor é pedido para exibir qualquer uma dessas estatísticas, ele primeiro busca o relatório mais recente emitido pelo processo coletor e então continua a usar esse momento (*snapshot*) para todas *views* e funções de estatísticas até o fim de sua transação atual.

Então as estatísticas mostrarão informação desde que você continue a atual transação. Similarmente, informação sobre as consultas atuais de todas sessões é coletada quando qualquer informação é requisitada primeiro dentro de uma transação, e a mesma informação será exibida toda a transação.

Essa é uma funcionalidade, não um *bug*, porque permite que você faça muitas consultas nas estatísticas e correlacionar os resultados sem se preocupar que os números estão mudando debaixo de você.

Mas se você quer ver novos resultados com cada consulta, esteja certo para fazer as consultas fora de qualquer bloco de transação. Alternativamente, você pode invocar `pg_stat_clear_snapshot()`, que desfará o atual momento (*snapshot*) estatístico de transação (se houver). A próxima utilização da informação estatística fará com que um novo *snapshot* seja obtido.

A transação pode também ver suas próprias estatísticas (como ainda não transmitidas para o coletor) nas *views* `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_usuário_tables` e `pg_stat_xact_usuário_functions`. Esses números não agem como descrito acima; em vez disso eles atualizam continuamente toda a transação.

[1] https://www.postgresql.org/docs/current/static/monitoring-stats.html#MONITORING-STATS-DYNAMIC-VIEWS-TABLE

19.4 Views de Estatísticas Padrões

As estatísticas por índice são particularmente úteis para determinar que índices estão sendo usados e quão efetivos eles são.

As views `pg_statio_` são primariamente úteis para determinar a eficácia do cache de *buffer*.

Quando o número atual de leituras de disco é muito menor do que o número de buscas em *buffer*, então o cache está satisfazendo a maior parte das requisições de leitura sem invocar uma chamada de *kernel*.

Porém, essas estatísticas não fornecem a história inteira: devido à maneira que o PostgreSQL lida com I/O de disco, dados que não estão no *cache* de *buffer* do PostgreSQL podem ainda residirem no cache de I/O do kernel e podem portanto ainda serem buscados sem precisar de uma leitura física.

Usuários interessados em obter informações mais detalhadas no comportamento de I/O do PostgreSQL são aconselhados a usar o coletor de estatísticas do PostgreSQL combinado com utilidades do sistema operacional que permitem introspectar o controle de I/O do *kernel*.

19.4.1 pg_stat_activity

É uma *view* global (para todas bases de dados) muito importante para o monitoramento de um servidor PostgreSQL.

Contém informações como pid, base de dados, estado de uma conexão e etc.

Nos exercícios a seguir precisaremos de duas sessões abertas no servidor, as quais chamaremos de **backend 1** e **backend 2**:

backend 1 e **backend 2**

Utilitário cliente psql:

```
$ psql
```

backend 1

Consultar pg_stat_activity:

```
> SELECT pid, waiting, state, query FROM pg_stat_activity;

 pid | waiting | state  |               query
-----+---------+--------+-----------------------------------
 900 | f       | active | SELECT pid, waiting, state, query+
     |         |        | FROM pg_stat_activity;
 961 | f       | idle   |
```

backend 2

Iniciando uma transação:

```
> BEGIN;
```

backend 1

Nova consulta em pg_stat_activity:

```
> SELECT pid, waiting, state, query FROM pg_stat_activity;

 pid | waiting |        state        |               query
-----+---------+---------------------+-----------------------------------
 900 | f       | active              | SELECT pid, waiting, state, query+
     |         |                     | FROM pg_stat_activity;
 961 | f       | idle in transaction | BEGIN;
```

Podemos notar que o **backend 2** tem seu estado ocioso na transaction, pois teve um `BEGIN` e ainda não teve um `COMMIT` ou um `ROLLBACK`.

backend 2

Criação de uma tabela com um bilhão de registros para testes:

```
> SELECT generate_series(1, 1000000000) AS campo INTO tb_teste;
```

backend 1

Verificando as informações da view pg_stat_activity:

```
> SELECT pid, waiting, state, query FROM pg_stat_activity;

 pid | waiting | state  |                              query
-----+---------+--------+-------------------------------------------------------------------
 900 | f       | active | SELECT pid, waiting, state, query                                +
     |         |        | FROM pg_stat_activity;
 978 | f       | active | SELECT generate_series(1, 1000000000) AS campo INTO tb_teste;
```

Nota-se que o estado do segundo *backend* está ativo (*active*), pois devido ao grande volume o processo é demorado.

No **backend 2** aborte a transação com a combinação de teclas `<Ctrl> + C`:

```
^CCancel request sent
ERROR:  canceling statement due to user request


1
```

backend 1

```
> SELECT pid, waiting, state, query FROM pg_stat_activity;
```

```
 pid | waiting |             state             |                              query
-----+---------+-------------------------------+-------------------------------------------------------------------
 900 | f       | active                        | SELECT pid, waiting, state, query                                +
     |         |                               | FROM pg_stat_activity;
 978 | f       | idle in transaction (aborted) | SELECT generate_series(1, 1000000000) AS campo INTO tb_teste;
```

O estado mudou, constando inclusive que o *statement* foi abortado.

backend 2	

Uma simples consulta:
 	
`> SELECT TRUE;`
 	
`ERROR:  current transaction is aborted, commands ignored until end of transaction block`
	Finalizando a transação:
	
	`> ROLLBACK;`

backend 1

Consulta em pg_stat_activity:

```
> SELECT pid, waiting, state, query FROM pg_stat_activity;
```

```
 pid | waiting | state  |               query
-----+---------+--------+-----------------------------------
 900 | f       | active | SELECT pid, waiting, state, query+
     |         |        | FROM pg_stat_activity;
 978 | f       | idle   | ROLLBACK;
```

Segundo backend está ocioso (idle).

backend 2	**backend 1**

Outra simples consulta:
 	
`> SELECT TRUE;`
 	
`bool`
`------`
`t`
	Quantos backends estão conectados?:
	
	`> SELECT count(pid) FROM pg_stat_activity;`
	
	`count`
	`-------`
	`2`

19.4.2 pg_stat_bgwriter

View global que traz informações relevantes como a respeito de *checkpoints* e claro, do *background writer*.

Quantos checkpoints foram requeridos neste cluster?:

```
> SELECT checkpoints_req FROM pg_stat_bgwriter;

 checkpoints_req
-----------------
              11
```

Gerando uma nova tabela de teste com 7 milhões de registros:

```
> SELECT generate_series(1, 7000000) AS campo INTO tb_teste;
```

Checando novamente a quantidade de checkpoints requeridos:

```
> SELECT checkpoints_req FROM pg_stat_bgwriter;

 checkpoints_req
-----------------
              12
```

Nota-se que houve mudança, devido ao grande volume de dados inseridos.

Apagando a tabela de teste:

```
> DROP TABLE tb_teste;
```

Forçando um checkpoint pelo comando CHECKPOINT:

```
> CHECKPOINT;
```

Nova consulta de checkpoints requeridos:

```
> SELECT checkpoints_req FROM pg_stat_bgwriter;

checkpoints_req
-----------------
              13
```

19.4.3 pg_stat_database

View global que tem uma linha por base de dados no *cluster* mostrando suas estatísticas.

Quantos backends estão conectados à base postgres?:

```
> SELECT numbackends FROM pg_stat_database WHERE datname = 'postgres';

 numbackends
-------------
           2
```

Todas informações da view pg_stat_database para a base postgres:

```
> SELECT
    xact_commit, xact_rollback, blks_read, blks_hit, tup_deleted
    FROM pg_stat_database
    WHERE datname = 'postgres';

 xact_commit | xact_rollback | blks_read | blks_hit | tup_deleted
-------------+---------------+-----------+----------+-------------
         995 |            13 |    283738 |    65330 |         257
```

Criação de tabela de teste:

```
> CREATE TABLE tb_carro(
    id smallserial,
    marca varchar(15),
    modelo varchar(15),
    cor varchar(15));
```

Inserindo dados:

```
> INSERT INTO tb_carro (marca, modelo, cor) values
    ('Fiat', '147', 'amarelo'),
    ('Volkswagen', 'Fusca', 'preto');
```

Consultar a base de dados postgres em pg_stat_database:

```
> SELECT
    xact_commit, xact_rollback, blks_read, blks_hit, tup_deleted
    FROM pg_stat_database
    WHERE datname = 'postgres';

 xact_commit | xact_rollback | blks_read | blks_hit | tup_deleted
-------------+---------------+-----------+----------+-------------
        1000 |            13 |    283750 |    65937 |         257
```

Apagando uma linha da tabela tb_carro:

```
> DELETE FROM tb_carro WHERE id = 2;
```

Consultando a view pg_stat_database:

```
> SELECT
    xact_commit, xact_rollback, blks_read, blks_hit, tup_deleted
    FROM pg_stat_database
    WHERE datname = 'postgres';

 xact_commit | xact_rollback | blks_read | blks_hit | tup_deleted
-------------+---------------+-----------+----------+-------------
        1004 |            13 |    283750 |    66112 |         258
```

Houve um aumento em `tup_deleted`.

19.4.4 pg_stat_all_tables

View que tem uma linha por tabela na base de dados atual (incluindo tabelas *TOAST*), mostrando estatísticas sobre acessos a esta tabela específica.

As views *pg_stat_user_tables* e *pg_stat_sys_tables* contêm a mesma informação, mas filtradas para apenas exibirem tabelas de usuário e de sistema respectivamente.

Apagando a tabela de teste:	Criação de uma nova tabela tb_carro:

`> DROP TABLE tb_carro;`	`> CREATE TABLE tb_carro(`
	`  id smallserial,`
	`  marca varchar(15),`
	`  modelo varchar(15),`
	`  cor varchar(15));`

Consultando a view pg_stat_all_tables para tb_carro:

```
> SELECT
    n_tup_ins, n_tup_upd, n_tup_del, n_tup_hot_upd, n_dead_tup
    FROM pg_stat_all_tables
    WHERE relname = 'tb_carro';

 n_tup_ins | n_tup_upd | n_tup_del | n_tup_hot_upd | n_dead_tup
-----------+-----------+-----------+---------------+------------
         0 |         0 |         0 |             0 |          0
```

Tudo zerado...

Inserindo dados:

```
> INSERT INTO tb_carro (marca, modelo, cor) values
    ('Fiat', '147', 'amarelo'),
    ('Volkswagen', 'Fusca', 'preto');
```

Consultando a view pg_stat_all_tables para tb_carro:

```
> SELECT
    n_tup_ins, n_tup_upd, n_tup_del, n_tup_hot_upd, n_dead_tup
    FROM pg_stat_all_tables
    WHERE relname = 'tb_carro';

 n_tup_ins | n_tup_upd | n_tup_del | n_tup_hot_upd | n_dead_tup
-----------+-----------+-----------+---------------+------------
         2 |         0 |         0 |             0 |          0
```

Nem tudo está zerado agora... (vide `n_tup_ins` e `n_live_tup`)

Uma atualização de registro:

```
> UPDATE tb_carro SET cor = 'azul' WHERE id = 1;
```

Consultando a view pg_stat_all_tables para tb_carro:

```
> SELECT
    n_tup_ins, n_tup_upd, n_tup_del, n_tup_hot_upd, n_dead_tup
    FROM pg_stat_all_tables
    WHERE relname = 'tb_carro';

 n_tup_ins | n_tup_upd | n_tup_del | n_tup_hot_upd | n_dead_tup
-----------+-----------+-----------+---------------+------------
         2 |         1 |         0 |             1 |          1
```

Uma tupla morta…

Apagando um registro:

```
> DELETE FROM tb_carro WHERE id = 2;
```

Consultando a view pg_stat_all_tables para tb_carro:

```
> SELECT
    n_tup_ins, n_tup_upd, n_tup_del, n_tup_hot_upd, n_dead_tup
    FROM pg_stat_all_tables
    WHERE relname = 'tb_carro';
```

```
 n_tup_ins | n_tup_upd | n_tup_del | n_tup_hot_upd | n_dead_tup
-----------+-----------+-----------+---------------+------------
         2 |         1 |         1 |             1 |          2
```

Mais uma tupla morta…

Vacumização em tb_carro:

```
> VACUUM tb_carro;
```

Consultando a view pg_stat_all_tables para tb_carro:

```
> SELECT
    n_live_tup, n_dead_tup, n_dead_tup, autovacuum_count, analyze_count, autoanalyze_count
    FROM pg_stat_all_tables
    WHERE relname = 'tb_carro';
```

```
 n_live_tup | n_dead_tup | n_dead_tup | autovacuum_count | analyze_count | autoanalyze_count
------------+------------+------------+------------------+---------------+-------------------
          1 |          0 |          0 |                0 |             0 |                 0
```

Sem tuplas mortas!!! :D

Apagando a tabela de teste

```
> DROP TABLE tb_carro;
```

19.4.5 pg_stat_all_indexes

Conterá uma linha por índice na base de dados atual, mostrando estatísticas sobre acessos àquele índice específico.

As views `pg_stat_user_indexes` e `pg_stat_sys_indexes` contêm a mesma informação, mas filtradas para mostrar apenas índices de usuários e sistema respectivamente.

Índices podem ser usados ou via buscas simples ou por índices (*index scans*) ou buscas por índices "*bitmap*" (*bitmap index scans*).

Em uma busca *bitmap* a saída de vários índices pode ser combinada via regras *AND* ou *OR*, por isso, é difícil de associar buscas de linhas acumuladas individuais quando uma busca bitmap é usada. Portanto, uma busca *bitmap* incrementa a contagem de `pg_stat_all_indexes.idx_tup_read` para o(s) índice(s) que usa, e incrementa a contagem `pg_stat_all_tables.idx_tup_fetch` para a tabela, mas não afeta `pg_stat_all_indexes.idx_tup_fetch`.

Obs.:

As contagens `idx_tup_read` e `idx_tup_fetch` podem ser diferentes mesmo sem qualquer uso de buscas bitmap, porque `idx_tup_read` conta entradas de índice recuperadas do índice enquanto `idx_tup_fetch` conta tuplas vivas trazidas da tabela. A última será menor se houver tuplas mortas ou tuplas não ainda efetivadas forem trazidas usando o índice, ou se houver buscas acumuladas são evitadas por significar um busca apenas por índice (*index only scan*).

Criando uma tabela de teste com 2 milhões de registros:

```
> SELECT generate_series(1, 2000000) AS campo INTO tb_teste;
```

Tentativa de criação de índice:

```
> CREATE INDEX idx_teste_comum ON tb_teste (campo);
```

Criando um novo índice (índice parcial):

```
> CREATE INDEX idx_teste_div19 ON tb_teste (campo) WHERE campo % 19 = 0;
```

Consultando a view pg_stat_all_indexes:

```
> SELECT
    indexrelname, idx_scan, idx_tup_read, idx_tup_fetch
    FROM pg_stat_all_indexes
    WHERE relname = 'tb_teste';

  indexrelname   | idx_scan | idx_tup_read | idx_tup_fetch
-----------------+----------+--------------+---------------
 idx_teste_comum |        0 |            0 |             0
 idx_teste_div19 |        0 |            0 |             0
```

Uma consulta simples:

```
> SELECT campo FROM tb_teste WHERE campo = 547889;
```

Consultando a view pg_stat_all_indexes:

```
> SELECT
    indexrelname, idx_scan, idx_tup_read, idx_tup_fetch
    FROM pg_stat_all_indexes
    WHERE relname = 'tb_teste';

  indexrelname   | idx_scan | idx_tup_read | idx_tup_fetch
-----------------+----------+--------------+---------------
 idx_teste_comum |        1 |            1 |             1
 idx_teste_div19 |        0 |            0 |             0
```

Pode-se observar que o índice comum foi acionado para a consulta feita.

Quantos registros existem cujo valor é divisível por 19?:

```
> SELECT count(*) FROM tb_teste WHERE campo % 19 = 0;
```

Consultando a view pg_stat_all_indexes:

```
> SELECT
    indexrelname, idx_scan, idx_tup_read, idx_tup_fetch
    FROM pg_stat_all_indexes
    WHERE relname = 'tb_teste';

  indexrelname   | idx_scan | idx_tup_read | idx_tup_fetch
-----------------+----------+--------------+---------------
 idx_teste_comum |        1 |            1 |             1
 idx_teste_div19 |        1 |       105263 |        105263
```

Agora o planejador de consultas escolheu usar o índice próprio para valores divisíveis por 19.

19.4.6 pg_statio_all_tables

Contém uma linha por tabela na base de dados atual (incluindo tabelas *TOAST*), mostrando estatísticas sobre I/O nessa tabela específica.

As *views* `pg_statio_user_tables` e `pg_statio_sys_tables` contêm a mesma informação, mas filtradas para exibir apenas tabelas de usuários e de sistema respectivamente.

Consultando o registro de tb_teste em pg_statio_all_tables:

```
> SELECT
    heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
    FROM pg_statio_all_tables
    WHERE relname = 'tb_teste';

 heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
----------------+---------------+---------------+--------------
          35866 |          8385 |           295 |            0
```

Uma simples consulta:

```
> SELECT campo FROM tb_teste WHERE campo = 19038;
```

Consultando o registro de tb_teste em pg_statio_all_tables para observar mudança nas estatísticas:

```
> SELECT
    heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
    FROM pg_statio_all_tables
    WHERE relname = 'tb_teste';

 heap_blks_read | heap_blks_hit | idx_blks_read | idx_blks_hit
----------------+---------------+---------------+--------------
          35866 |          8386 |           297 |            1
```

19.4.7 pg_statio_all_indexes

Contém uma linha por índice na base de dados atual, mostrando estatísticas sobre I/O naquele índice específico.

As view `pg_statio_user_indexes` e `pg_statio_sys_indexes` contêm a mesma informação, mas filtradas para exibir apenas índices de usuário e de sistema respectivamente.

Todos registros de índices da tabela tb_teste:

```
> SELECT indexrelname, idx_blks_read, idx_blks_hit FROM  pg_statio_all_indexes
    WHERE relname = 'tb_teste';
```

```
  indexrelname   | idx_blks_read | idx_blks_hit
-----------------+---------------+--------------
 idx_teste_comum |             6 |            1
 idx_teste_div19 |           291 |            0
```

Consulta simples:

```
> SELECT campo FROM tb_teste WHERE campo = 598731;
```

Quantos campos são divisíveis por 19:

```
> SELECT count(*) FROM tb_teste WHERE campo % 19 = 0;
```

Todos registros de índices da tabela tb_teste:

```
> SELECT indexrelname, idx_blks_read, idx_blks_hit FROM  pg_statio_all_indexes
    WHERE relname = 'tb_teste';
```

```
  indexrelname   | idx_blks_read | idx_blks_hit
-----------------+---------------+--------------
 idx_teste_comum |             7 |            3
 idx_teste_div19 |           291 |          290
```

19.4.8 pg_statio_all_sequences

Contém uma linha por sequência na base de dados atual, mostrando estatísticas sobre I/O naquela sequência específica.

Criação de sequência:

```
> CREATE SEQUENCE sq_teste;
```

Verificando I/O de sequências:

```
> SELECT blks_read, blks_hit FROM pg_statio_all_sequences WHERE relname = 'sq_teste';
```

```
 blks_read | blks_hit
-----------+----------
         1 |        0
```

Próximo valor da sequência

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

Verificando I/O de sequências:

```
> SELECT blks_read, blks_hit FROM pg_statio_all_sequences WHERE relname = 'sq_teste';

 blks_read | blks_hit
-----------+----------
         1 |        1
```

19.4.9 pg_stat_user_functions

Contém tem uma linha por função localizada, *mostrando estatísticas sobre execuções daquela função.*

O parâmetro `track_functions` controla exatamente que funções são rastreadas.

Criando uma simples função:

```
> CREATE OR REPLACE FUNCTION fc_teste(num INT)
       RETURNS INT AS $abobrinha$
       BEGIN
           RETURN num ^ 3;
       END;
       $abobrinha$ LANGUAGE plpgsql;
```

Testando:

```
> SELECT fc_teste(3);
```

Verificando estatísticas de uso de funções criadas por usuários:

```
> SELECT calls FROM pg_stat_user_functions WHERE funcname = 'fc_teste';

(No rows)
```

Ajustando em sessão o valor do parâmetro track_functions:

```
> SET track_functions TO pl;
```

Testando a função:

```
> SELECT fc_teste(3);
```

Nova tentativa de verificar estatísticas de funções de usuários:

```
> SELECT calls FROM pg_stat_user_functions WHERE funcname = 'fc_teste';
```

```
 calls
-------
     1
```

19.4.10 pg_stat_replication

Contém uma linha por processo *WAL sender*, mostrando estatísticas sobre replicação para o servidor *standby* que está conectado ao de envio.

Apenas servidores *standby* diretamente conectados são listados; nenhuma informação está disponível sobre servidores *standby* em nível abaixo (em cascata).

Esse catálogo é visto em testes no curso de Replicação.

19.4.11 pg_stat_database_conflicts

Contém uma linha por base de dados, mostrando de toda a base de dados estatísticas sobre cancelamentos de consultas ocorrendo devido a conflitos com *recovery* em servidores *standby*.

Esta *view* conterá apenas informação em servidores standby, desde que conflitos não ocorram em servidores *master*.

Estatísticas de conflitos de bases de dados:

```
> SELECT
       datname,
       confl_tablespace,
       confl_lock,
       confl_snapshot,
       confl_bufferpin,
       confl_deadlock
       FROM pg_stat_database_conflicts;
```

```
  datname  | confl_tablespace | confl_lock | confl_snapshot | confl_bufferpin | confl_deadlock
-----------+------------------+------------+----------------+-----------------+----------------
 template1 |                0 |          0 |              0 |               0 |              0
 template0 |                0 |          0 |              0 |               0 |              0
 postgres  |                0 |          0 |              0 |               0 |              0
 pagila    |                0 |          0 |              0 |               0 |              0
```

20 Funções Estatísticas

- Sobre Funções Estatísticas

20.1 Sobre Funções Estatísticas

Outras formas de procurar nas estatísticas podem ser configuradas escrevendo consultas que usam as mesmas funções estatísticas de acesso relacionadas usadas por *views* padrões mostradas anteriormente.

Para detalhes tais como nomes de funções, consulte as definições das *views* padrões.

As funções de acesso a estatísticas por base de dados usa um OID de banco de dados como um argumento para identificar que base de dados informar sobre.

As funções por tabela e por índice funções precisam de um OID de tabela ou de índice.

As funções para estatísticas função precisam de um OID de função. Note que apenas tabelas, índices, e funções na base de dados atual podem ser vistas com essas funções.

20.1.1 Funções Estatísticas Adicionais

Função	Tipo de Retorno	Descrição
pg_backend_pid()	integer	ID de processo do processo servidor que controla a sessão atual
pg_stat_get_activity(integer)	setof record	Retorna um registro de informações sobre o backend com o PID especificado ou um registro por cada backend ativo no system se NULL estiver especificado. Os campos retornados são um subconjunto daqueles na view pg_stat_activity.
pg_stat_clear_snapshot()	void	Descarta as estatísticas atuais do snapshot
pg_stat_reset()		É feito um reset em todos contadores estatísticos da base de dados atual para zero (requer privilégios de super usuário)
pg_stat_reset_shared(text)		É feito um reset em alguns contadores estatísticos para zero de todo o cluster, dependendo do argumento (requer privilégios de super usuário). Chamando pg_stat_reset_shared('bgwriter') vai zerar todos os contadores mostrados na view pg_stat_bgwriter.
pg_stat_reset_single_table_counters(oid)		Faz um reset nas estatísticas para uma única tabela ou índice na base de dados atual para zero (requer privilégios de super usuário)
pg_stat_reset_single_function_counters(oid)		Faz um reset nas estatísticas para uma única função na base de dados atual para zero (requer privilégios de super usuário)

Tabela 3: 1.2.1 Funções Estatísticas Adicionais

`pg_stat_get_activity`, função relacionada da *view* `pg_stat_activity`, retorna um conjunto de registros contendo todas informações disponíveis sobre cada processo backend.

Às vezes deve ser mais conveniente obter apenas um subconjunto dessa informação. Em tais casos, um conjunto mais velho de funções estatísticas de acesso por *backend* podem ser usadas; que são mostradas na tabela “Funções Estatísticas por *Backend”*, a seguir. Essas funções de acesso usam um número ID de *backend*, que varia de um para o número de *backends* ativos atualmente.

A função `pg_stat_get_backend_idset` fornece uma maneira conveniente para gerar uma linha por *backend* ativo para invocar estas funções.

Exibe os PIDs e consultas atuais de todos backends:

```
> SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
       FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

20.1.2 Funções Estatísticas por Backend

Função	Tipo de Retorno	Descrição
pg_stat_get_backend_idset()	setof integer	Conjunto de número ID de *backends* ativos (de 1 ao número de *backends* ativos)
pg_stat_get_backend_activity(integer)	text	Texto da consulta mais recente do *backend*
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	Quando a consulta mais recente foi iniciada
pg_stat_get_backend_client_addr(integer)	inet	Endereço IP do cliente conectado a este *backend*
pg_stat_get_backend_client_port(integer)	integer	Número da porta TCP que o cliente está usando para comunicação
pg_stat_get_backend_dbid(integer)	oid	OID da base de dados que este *backend* está conectado
pg_stat_get_backend_pid(integer)	integer	ID de processo deste *backend*
pg_stat_get_backend_start(integer)	timestamp with time zone	Quando este processo foi iniciado
pg_stat_get_backend_userid(integer)	oid	OID do usuário logado neste *backend*
pg_stat_get_backend_waiting(integer)	boolean	`True` se este *backend* atualmente estiver esperando em um bloqueio (*lock*)
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	Quando a transação atual foi iniciada

Tabela 4: Funções Estatísticas por Backend

Listando IDs de backends:

```
> SELECT pg_stat_get_backend_idset();

 pg_stat_get_backend_idset
---------------------------
                         1
                         2
```

Verificando as atividades do backend 1:

```
> SELECT pg_stat_get_backend_activity(1);

 pg_stat_get_backend_activity
------------------------------
 TABLE pg_roles;
```

Verificando as atividades do backend 2

```
> SELECT pg_stat_get_backend_activity(2);

 pg_stat_get_backend_activity
------------------------------
 SELECT TRUE;
```

21 Funções de Administração do Sistema

- Sobre Funções de Administração do Sistema
- Tipos de Funções Administrativas

21.1 Sobre Funções de Administração do Sistema

As funções descritas aqui são usadas para controlar e monitorar uma instalação PostgreSQL.

Na documentação oficial *online* do PostgreSQL podem ser consultadas no seguinte endereço:

http://www.postgresql.org/docs/current/static/functions-admin.html

21.2 Tipos de Funções Administrativas

Funções administrativas do PostgreSQL tem os seguintes tipos:

- Definições de Configuração / Configuration Settings [1]
- Sinalização de Servidor / Server Signaling [2]
- Controle de Backup / Backup Control [3]
- Controle de Recuperação de Backup / Recovery Control [4]
- Sincronização de Snapshop / Snapshot Synchronization [5]
- Replicação / Replication [6]
- Gerenciamento de Objetos de Banco de Dados / Database Object Management [7]
- Manutenção de Índices / Index Maintenance [8]
- Acesso Genérico a Arquivos / Generic File Access [9]
- Bloqueio Consutivo / Advisory Lock [10]

[1] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-SET

[2] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-SIGNAL

[3] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-BACKUP

[4] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-RECOVERY-CONTROL

[5] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION

[6] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-REPLICATION

[7] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-DBOBJECT

[8] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-INDEX

[9] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-GENFILE

[10] https://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADVISORY-LOCKS

22 Funções de Informação de Sistema

- Funções de Informação de Sessão
- Outros subgrupos de Funções de Informação de Sistema

22.1 Sobre Funções de Informação de Sistema

Grupo majoritário de coleção de funções que extraem informações do sistema.
Na documentação oficial *online* do Postgres podem ser consultadas no seguinte endereço:

http://www.postgresql.org/docs/current/static/functions-info.html

22.2 Funções de Informação de Sessão

É o principal subgrupo das Funções de Informação de Sistema, cujas funções exibem informações a respeito de sessões.

Função	Retorno	Descrição
current_catalog	name	Nome da base de dados atual (chamada de catálogo no padrão SQL).
current_database()		Nome da base de dados atual.
current_schema[()]		Nome do schema atual.
current_user		Nome do usuário do contexto da execução.
session_user		Nome do usuário da sessão.
current_user		Equivalente a current_user.
current_schemas()	name[]	Nomes dos esquemas no caminho de busca (search path), opcionalmente incluindo schemas implícitos.
current_query()	text	Texto da consulta atual executada pelo cliente que pode conter mais de uma instrução. Muito útil ao ser usada em outras funções ou funções de trigger.
version()		Informação da versão do PostgreSQL.
pg_listening_channels()	setof text	Nomes de canais que a sessão está escutando atualmente (consultar o comando LISTEN na documentação oficial);
inet_client_addr()	inet	Endereço da conexão remota (endereço do cliente que se conecta ao servidor).
inet_server_addr()		Endereço do servidor ao qual se conectou.
inet_client_port()	int	Porta da conexão remota (porta do cliente que se conecta ao servidor).
inet_server_port()		Porta do servidor ao qual se conectou.
pg_backend_pid()		Process ID do processo do servidor da sessão atual.
pg_trigger_depth()		Nível de aninhamento atual de gatilhos PostgreSQL (0 se não for chamado, direta ou indiretamente, de dentro de um gatilho).
pg_conf_load_time()	timestamp with time zone	Data e hora em que a configuração foi carregada.
pg_postmaster_start_time()		Data e hora que o servidor iniciou;
pg_is_other_temp_schema(oid)	boolean	Retorna true se o OID dado é o OID do esquema temporário de outra sessão. (pode ser útil para excluir tabelas temporárias de outras sessões de uma exibição de catálogo).
pg_my_temp_schema()	oid	OID do esquema temporário da sessão ou 0 (zero) para nada;

Tabela 5: 1.2 Funções de Informação de Sessão

Obs.:

current_catalog, current_schema, current_user, session_user, e user têm status especial de sintaxe em SQL: elas devem ser chamadas sem parênteses.

No PostgreSQL, os parênteses podem ser opcionalmente usados com current_schema, mas não com as outras.

Desde quando o servidor está rodando?:

```
> SELECT pg_postmaster_start_time();
```

Qual o nome da base que o cliente se conectou?

```
> SELECT current_database();
```

Qual versão (servidor) do PostgreSQL está sendo usada?

```
> SELECT version();
```

22.3 Outros subgrupos de Funções de Informação de Sistema

Funções de ...	**Descrição**
Consulta a Privilégios de Acesso	Consultam se um papel tem privilégios sobre um objeto ou se pertence a outro papel.
Consulta de Visibilidade no Esquema	Consultam se um objeto está visível ao caminho de busca (*search path*).
Informação do Catálogo de Sistema	Extraem informações do catálogo de sistema.
Informação de Comentários	Funções que extraem comentários de objetos armazenados com o comando `COMMENT`.
IDs de Transação e Snapshots	Fornecem informações de transação em uma forma exportável e tem como principal finalidade é determinar que transações foram efetivadas (*committed*) entre dois *snapshots*.

Tabela 6: 1.3 Outros subgrupos de Funções de Informação de Sistema

23 ANALYZE

- Sobre ANALYZE

23.1 Sobre ANALYZE

O comando `ANALYZE` faz estatísticas sobre o conteúdo de tabelas na base de dados e armazena os resultados no catálogo de sistema `pg_statistic`.

Subsequentemente, o planejador de consultas usa essas estatísticas para auxiliar a determinar o mais eficiente plano de execução de consultas.

Sem parâmetros, `ANALYZE` examina todas as tabelas na base de dados atual. Com um parâmetro examina apenas aquela tabela.

Há também a possibilidade de fornecer uma lista de nomes de colunas, que nesse caso as estatísticas serão coletadas apenas para elas.

Sintaxe:

```
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

23.1.1 Parâmetros

- **`VERBOSE`**: Habilita a exibição das mensagens de progresso;
- **`table_name`**: O nome (possivelmente qualificado de esquema) de uma tabela específica para analisar. Se omitido, todas tabelas da base de dados (exceto as tabelas estrangeiras) serão analisadas;
- **`column_name`**: O nome de uma coluna específica para analisar. Padrão: todas as colunas.

Conectando à base pagila:

```
> \c pagila
```

Analisar a tabela actor:

```
> ANALYZE VERBOSE actor;
```

```
INFO:  analyzing "public.actor"
INFO:  "actor": scanned 2 of 2 pages, containing 206 live rows and 0 dead rows;
206 rows in sample, 206 estimated total rows
ANALYZE
```

Analisar todas as tabelas da base de dados pagila

```
> ANALYZE VERBOSE;
```

Várias mensagens de progresso...

Aviso:

Sempre que for alterada significativamente a distribuição dos dados dentro de uma tabela, rodar o comando `ANALYZE` é fortemente recomendado.

Isso inclui carga em massa de grandes quantidades de dados em uma tabela.

Rodando `ANALYZE` (ou `VACUUM ANALYZE`) garante que o planejador terá suas estatísticas atualizadas sobre a tabela.

Sem ou estatísticas obsoletas, faz com que o planejador de consultas tome decisões ruins, levando a uma performance horrível em qualquer tabela com estatísticas imprecisas ou mesmo ausentes.

Se o *daemon* `autovacuum` estiver habilitado, ele deve fazer o `ANALYZE` automaticamente.

24 Migração e Atualização

- Sobre Migração e Atualização
- Versões do PostgreSQL
- Migração por Replicação
- pg_upgrade

24.1 Sobre Migração e Atualização

Migração, palavra que segundo a definição de dicionários é o ato se mudar de um lugar para outro.

No nosso contexto pode se tratar de migrar de uma máquina/plataforma para outra ou também atualizar, ou seja, mudar de uma versão para outra.

A cada versão do PostgreSQL novos recursos são adicionados o que pode tornar muito interessante migrar a base de dados atual para uma nova versão do SGBD.

No capítulo sobre backup e restauração foram vistos conceitos parecidos com os deste, sendo que nele foi apresentado o método de *dump*, o qual também serve para fazer migração de versões do PostgreSQL, porém, dependendo do tamanho da base pode ser extremamente lento, pois cada registro é processado novamente.

Alternativamente temos a ferramenta pg_upgrade, que tem um método muito mais rápido para se fazer essa tarefa.

24.2 Versões do PostgreSQL

24.2.1 Política de Versionamento

Até a versão 9.6, o PostgreSQL adotava o modelo de versão X.Y.Z, sendo que a parte X.Y era a versão majoritária e a Z a versão minoritária.

A partir da versão 10, adotou-se o modelo X.Y, sendo X a versão majoritária e Y a versão minoritária.

É fortemente recomendado a atualização para a última versão minoritária (minor version: X.Y) para qualquer que seja sua sua versão maior (*major release*: X) em uso.

As versões maiores do PostgreSQL incluem novas funcionalidades e ocorrem uma vez por ano.

Major releases normalmente mudam o formato interno do sistema de tabelas e arquivos de dados.

Um *dump* de uma base de dados ou o uso do módulo `pg_upgrade` são necessários para major *releases*.

Versões menores (*minor releases*) são numeradas incrementando a segunda parte do número da versão, e.g. 10.0 para 10.1. O time de desenvolvimento do PostgreSQL, para *minor releases* adiciona apenas correções de *bugs*.

Todos usuários devem atualizar para a versão de lançamento menor (X.Y) assim que possível.

Apesar de atualizações terem algum risco, versões menores do PostgreSQL corrigem apenas *bugs* de segurança e corrupção de dados para reduzir o risco de atualização. A comunidade considera que não atualizar é mais arriscado do que atualizar.

Atualizando para um *minor release* não requer um *dump* e *restore*; simplesmente pare o banco de dados, instale os binários atualizados e reinicie o servidor.

Para alguns lançamentos, mudanças manuais podem ser necessárias para completar a atualização, então sempre leia as notas de lançamento antes de atualizar.

24.2.2 Política de Suporte de Lançamentos do PostgreSQL

O projeto PostgreSQL tem como objetivo suportar totalmente o major *release* por cinco anos.

Nenhum binário será produzido pelo projeto, mas o código fonte atualizado estará disponível no sistema de controle de código fonte.

Para saber sobre versões suportadas no momento, no site oficial do PostgreSQL há uma tabela no link:

http://www.postgresql.org/support/versioning/

Obs.:

Dica: No site oficial do PostgreSQL há uma parte que tem uma matriz de funcionalidades que foram adicionadas ao longo das versões:

http://www.postgresql.org/about/featurematrix/

24.3 Migração por Replicação

É possível fazer migração por meio de replicação, desde que seja replicação lógica.

A replicação nativa do PostgreSQL, atualmente é a replicação via *streaming*, que é uma replicação física.

Devido à incompatibilidade de arquivos físicos entre versões, uma migração via replicação física não é possível.

24.3.1 Slony-I

Slony-I é um sistema de replicação lógica e *open source* para o PostgreSQL.

Seu funcionamento é baseado em *triggers*, podendo ser granular, ou seja, replicar parcialmente a base de dados.

Nesse sistema de replicação uma máquina deve fazer o papel de servidor primário e outras fazendo o papel de servidores secundários e podendo até fazer cascateamento.

Esse sistema foi muito utilizado como replicação para o PostgreSQL antes da versão 9.0, quando então surgiu a replicação nativa via *streaming*.

Através do Slony-I podemos fazer migração de versões mais antigas com *downtime* (tempo de parada) ínfimo entre uma versão mais antiga e até mesmo sem suporte para a versão mais atual.

http://slony.info/

24.3.2 pglogical

O pglogical também é um sistema de replicação lógica e *open source* para o PostgreSQL, desenvolvido e mantido principalmente pela empresa 2ndQuadrant.

Está disponível desde a versão 9.4.

Diferente do Slony-I não é uma aplicação à parte, mas sim instalado em forma de extensão.

Não requer *triggers* ou programas externos.

O tipo de replicação que utiliza é chamada também de "*logical streaming replication*", replicação lógica via *streaming*.

https://2ndquadrant.com/en/resources/pglogical/

24.4 pg_upgrade

Antigamente chamado *pg_migrator*, possibilita a migração de versões majoritárias do PostgreSQL sem utilizar *dumps*.

O *pg_upgrade* suporta atualizações desde a série 8.3.X, incluindo *snapshots* e lançamentos *alpha*.

Aviso:

Os clusters devem combinar entre si a respeito de *checksum* (opção `-k` do `initdb`), ou seja, ou ambos têm, ou ambos não têm.

---

http://www.postgresql.org/docs/current/static/pgupgrade.html

24.4.1 Parâmetros e Variáveis de Ambiente do pg_upgrade

Parâmetro		Variável de Ambiente	Descrição
Curto	Longo		
-b	--old-bindir	PGBINOLD	Binários antigos
-B	--new-bindir	PGBINNEW	Binários novos
-d	--old-datadir	PGDATAOLD	Diretório de dados antigo
-D	--new-datadir	PGDATANEW	Diretório de dados novo

24.4.2 Nosso Laboratório de Aprendizado do pg_upgrade

Será utilizado o modelo de instalação via código-fonte, conforme visto no capítulo de instalação.

Padronização de diretórios faz muita diferença na hora de fazer uma migração, agilizando muito o trabalho.

Cluster	Antigo	Novo
Versão	`${PGVERSION_OLD}`	`${PGVERSION_NEW}`
Porta	5432	5433 (provisória)
PGDATA	`/var/lib/pgsql/${PGVERSION_OLD}/data`	`/var/lib/pgsql/${PGVERSION_NEW}/data`
PGBIN	`/usr/local/pgsql/${PGVERSION_OLD}/bin`	`/usr/local/pgsql/${PGVERSION_NEW}/bin`

Informar as versões (majoritárias) dos clusters (antigo e novo):

```
$ read -p 'Digite a versão antiga: ' PGVERSION_OLD

$ read -p 'Digite a versão nova: ' PGVERSION_NEW
```

Definir as variáveis de ambiente do pg_upgrade com base nas estruturas de diretórios e variáveis de ambiente para versões:

```
$ export PGDATAOLD="/var/lib/pgsql/${PGVERSION_OLD}/data"
$ export PGDATANEW="/var/lib/pgsql/${PGVERSION_NEW}/data"
$ export PGBINOLD="/usr/local/pgsql/${PGVERSION_OLD}/bin"
$ export PGBINNEW="/usr/local/pgsql/${PGVERSION_NEW}/bin"
```

Parando os clusters:

```
$ ${PGBINOLD}/pg_ctl -m i -D ${PGDATAOLD} stop
$ ${PGBINNEW}/pg_ctl -m i -D ${PGDATANEW} stop
```

Dando início à migração:

```
$ pg_upgrade

. . .
```

Ao término da execução do pg_upgrade, no final é informado que as estatísticas de otimização não foram transferidas.

No diretório de execução são deixados dois scripts: `analyze_new_cluster.sh` e `delete_old_cluster.sh`, os quais usaremos adiante.

Mudaremos a configuração de porta de escuta do cluster novo para a porta padrão:

```
$ sed -i 's/port = 5433/port = 5432/g' ${PGDATANEW}/postgresql.conf
```

Inicializando o novo cluster:

```
$ ${PGBINNEW}/pg_ctl -m i -D ${PGDATANEW} start
```

Script para estatísticas de otimização:

```
$ ./analyze_new_cluster.sh
```

Apagar o cluster antigo:

```
$ ./delete_old_cluster.sh
```

Testando com uma consulta na base que tinha no cluster antigo:

```
$ psql -c 'SELECT * FROM actor LIMIT 3;' pagila

 actor_id | first_name | last_name |     last_update
----------+------------+-----------+---------------------
        1 | PENELOPE   | GUINESS   | 2006-02-15 09:34:33
        2 | NICK       | WAHLBERG  | 2006-02-15 09:34:33
        3 | ED         | CHASE     | 2006-02-15 09:34:33
```

Desabilitando a inicialização cluster antigo:

```
$ systemctl disable postgresql-${PGVERSION_OLD}.service
```

25 Troubleshooting

- Sobre Troubloshooting
- Gerenciamento de Recursos do Kernel
- Configuração Pós Instalação
- Configurando Logs

25.1 Sobre Troubleshooting

Um dos mais importantes trabalhos de um administrador de banco de dados é o de solucionar problemas (*troubleshooting*). Esse trabalho depende muito do grau de conhecimentos e experiência que o DBA tem.

É imprescindível que o administrador de banco de dados PostgreSQL, além do próprio SGBD, que já é muito complexo por si só, também conheça bem a estrutura do sistema operacional em que está rodando.

Nessa combinação SGBD + SO podem acontecer problemas ocasionando erros, paradas e/ou comportamentos indesejados.

Há muitas situações inclusive que o DBA faz um trabalho similar ao de um detetive, investigando através de rastros as causas de um determinado problema.

Felizmente as mensagens de erro geradas pelo PostgreSQL são muito didáticas, que não raras vezes diz onde está o problema e qual providência tomar.

25.1.1 Algumas Dicas Primárias para Troubleshooting

- **Idioma inglês**: Instale seu SGBD e seu SO ambos em inglês, pois para procurar ajuda externa como listas de discussão e sites de busca, tem muito mais informações nesse idioma do que qualquer outro;

- **Monitoramento**: Prevenção é a melhor forma de evitar problemas, portanto é bom sempre estar alerta a itens que envolvem o SGBD como discos, memória e processador. Softwares de monitoramento como Zabbix [1] e Nagios [2], por exemplo, são de grande utilidade para um DBA.

- **Logs de atividades**: Ter logs de atividades do PostgreSQL bem configurados auxiliam na investigação de um problema. Olhar os logs do sistema operacional também são de grande utilidade.

[1] https://www.zabbix.com/
[2] https://www.nagios.org/

25.2 Gerenciamento de Recursos do Kernel

25.2.1 Memória Compartilhada

Memória compartilhada é aquela que pode ser simultaneamente acessada por múltiplos programas para fornecer comunicação entre eles ou evitar cópias redundantes.

É também um meio eficiente de passar dados entre programas.

Em um único programa, por exemplo, o mesmo tenha várias *threads* também pode fazer uso de memória compartilhada.

25.2.1.1 Parâmetros de Configuração do Kernel para Memória Compartilhada

- **SHMMAX** → `kernel.shmmax` (bytes)
  Tamanho máximo de segmento de memória compartilhada.
  Valores razoáveis (pelo menos):

  1 kB (se tiver mais de uma instância de servidor será necessário um valor proporcional à quantidade).

- **SHMALL** → `kernel.shmall` (páginas)
  Total de memória compartilhada disponível.
  Valores razoáveis:

  `ceil(SHMMAX/PAGE_SIZE)`
  ou o mesmo valor de SHMMAX (convertido de bytes para páginas).

  Como obter PAGE_SIZE:

  ```
  $ getconf PAGE_SIZE
  ```

- **SHMMNI** → `kernel.shmmni`
  Quantidade máxima de segmentos de memória compartilhada para todo o sistema.

25.2.2 Semáforos

Tipo de variável especial de sistema operacional para auxiliar em sincronização.

Se múltiplos processos compartilham um recurso, então é necessária uma maneira para que o uso desse recurso não cause confusão entre esses processos.

Um semáforo permite ou não o acesso a um recurso, dependendo como ele está configurado. Existem dois tipos de semáforos:

- **Semáforos Binários / *Binary Semaphores***:
  Neste tipo de semáforo tem 2 métodos associados a ele: (*up*, *down* / *lock*, *unlock*).
  Como próprio nome sugere, só pode ter 2 valores (`0` e `1`), os quais são usados para determinar uma trava (*lock*). Quando um recurso está disponível, o processo responsável ajusta o semáforo para `1`, senão `0`.

- **Semáforos de Contagem / *Counting Semaphores***:
  Um semáforo de contagem utiliza números inteiros para determinar quantos acessos um recurso pode ter.

25.2.2.1 Parâmetros de Configuração do Kernel para Semáforos

- **`SEMMSL`** → `kernel.sem`
  Quantidade máxima de semáforos por conjunto.
  Valores razoáveis (pelo menos):

  17

- **`SEMMNS`** → `kernel.sem`
  Quantidade máxima de semáforos de todo o sistema.
  Valores razoáveis:

  ```
  ceil((max_connections + autovacum_max_workers + 4) / 16) * 17
  ```
  + espaço para outras aplicações

Obs.:

O parâmetro `kernel.sem` tem um valor composto, cuja sequência é `SEMMSL`, `SEMMNS`, `SEMOPM` e `SEMMNI`.

Se for dado o seguinte comando no *shell*, *sysctl* exibe e ajusta parâmetros do kernel:

```
$ sysctl kernel.sem
```

```
kernel.sem = 250       32000   32      128
```

SEMMSL = 250, SEMMNS = 32000, SEMOPM = 32 e SEMMNI = 128.

25.2.3 Limites e Recursos

Sistemas operacionais Unix *like* forçam vários tipos de limites de recursos que podem interferir com a operação do servidor PostgreSQL.

Especialmente limites para números de processos por usuário (`maxproc`), de arquivos abertos por processo (`openfiles`) e quantidade de memória disponível para cada processo (`datasize`).

Cada um desses tem um limite "*hard*" e um "*soft*".

O limite *soft* é que atualmente conta, mas pode ser mudado pelo usuário até o limite *hard*.

O limite *hard* só pode ser mudado pelo usuário *root*.

O comando de *shell built-in* `ulimit` (*Bourne shells*) ou `limit (csh)` são usados para controlar os limites de recursos pela linha de comando.

O arquivo de configuração para esses limites em sistemas operacionais BSD costuma ser o arquivo `/etc/login.conf` e no Linux `/etc/security/limits.conf`. A documentação do sistema operacional deve ser consultada.

Os parâmetros de limites tem seu nome idêntico em **BSD**s, a seguir a tabela que demonstra como são essas configurações no **Linux**:

/etc/logon.conf (BSD)	/etc/security/limits.conf (Linux)
`maxproc`	`nproc`
`openfiles`	`nofile`
`datasize`	`data`

Obs.:

O kernel de um sistema operacional também pode oferecer opções de limites de recursos.

No kernel do Linux o parâmetro `fs.file-max`, que determina o número máximo de arquivos abertos que o kernel suportará. Para detalhes use o manual:

```
man proc
```

Obs.:

Alguns sistemas permitem que processos individuais para abrir grandes quantidades de arquivos; o que pode fazer com que o limite de todo o sistema possa ser facilmente ser excedido.

Se encontrar isso acontecendo, e não quiser alterar a configuração de limites de todo o sistema, para esse fim, o PostgreSQL disponibiliza o parâmetro `max_files_per_process`.

25.2.4 Swapiness

É a maneira que o kernel do Linux fornece para ajustar a configuração que controla a frequência que a swap é usada, cuja faixa de valores varia de 0 (zero) a 100 (cem). Cujo valor corresponde à porcentagem de memória RAM restante, que é o ponto de partida para se começar a fazer swap.

É o controle de tendência de o kernel de direcionar o conteúdo de memória para *swap*, ao invés da memória RAM.

Vale lembrar que o I/O de disco, como é feito o processo de swap, é muito mais lento do que o uso de memória RAM, portanto, impacta diretamente na performance do sistema.

Uma configuração `swappiness = 1` significa que será evitado fazer *swap* a não ser que seja absolutamente necessário, ou seja, tem 99% de RAM usada.

Verificando o valor atual de swappiness:

```
$ sysctl vm.swappiness
```

```
vm.swappiness = 60
```

Em uma máquina com 8GB de memória RAM, o consumo de memória no instante:

```
$ free -m
```

```
             total       used       free     shared    buffers     cached
Mem:          8032       2058       5973          0          1       1440
-/+ buffers/cache:        617       7415
Swap:         3814          0       3814
```

Conforme a saída do comando acima temos:

Total usado: 2058 MB = 25,623%
Restante: 5973 MB = 74,37%

Conforme a configuração de *swappiness* do exemplo, se a quantidade livre de RAM diminuir para 4819,2 MB (60%), começará a fazer swap.

25.2.5 Linux Memory Overcommit

Figura: Out Of Memory Killer

Quando um programa *user space* reserva memória (função em C `malloc()`), se tiver retorno NULL, significa que não tem memória disponível.

Por padrão o Linux aceita a maior parte das requisições por mais memória, pressupondo que muitos programas requiram mais memória do que precisam. Tal pressuposição estando certa permite que mais programas rodem na mesma memória ou pode fazer rodar um programa que exija mais memória do que está disponível.

Se mesmo através desse recurso de "achar" mais memória não for possível alocar memória para um programa que requisita, pode ter efeitos indesejados. O **OOM Killer** (***Out Of Memory Killer***) é invocado e então através de um algoritmo heurístico seleciona algum processo para matar. Há uma grande polêmica em torno da escolha da "vítima".

Pode até não ser um processo *root*, algum processo que esteja fazendo I/O de disco, algum que já esteja a muito tempo fazendo algum cálculo. Assim pode acontecer de um vi (editor) ser terminado quando alguém iniciar mais coisas do que o *kernel* pode manipular.

Resumindo, *Memory Overcommit* é um recurso que permite que ao se esgotar a memória (RAM + swap), um programa que requisitar alocação de memória após isso provavelmente será atendido utilizando a mesma memória que outro(s) programa(s), mas caso isso não for possível o *OOM Killer* matará algum processo aleatoriamente, mas seguindo um algoritmo heurístico.

25.2.5.1 Parâmetros de Memory Overcommit do Kernel

- **`overcommit_memory`** (**`vm.overcommit_memory`**): Este valor contém a flag que habilita o recurso Memory Overcommit, cujos possíveis valores são:
    - **`0`** (padrão): **Gerenciamento heurístico** de *overcommit;* faz com que o *kernel* tente estimar a quantidade de memória livre restante quando programas *user space* pedem mais memória. O usuário *root* é permitido alocar um pouco mais de memória neste modo.
    - **`1`**: ***Sempre fazer overcommit***; apropriado para algumas aplicações científicas, faz com que o *kernel* "minta" afirmando que sempre há memória suficiente até que ela realmente se esgote.

- **2**: Não fazer *overcommit*; política "*never overcommit*" que tenta prevenir qualquer *overcommit* de memória. Dependendo da quantidade de memória que for usada, na maioria das situações isso significa que um processo não será terminado enquanto acessar páginas, mas receberá erros de alocação de memória como apropriado. Útil para aplicações que deseja-se garantir que suas alocações de memória estarão disponíveis no futuro sem ter que iniciar cada página. O limite atual de *overcommit* e quantidade comprometida (*commited*) estão disponíveis em `/proc/meminfo` como *CommitLimit* e *Committed_AS* respectivamente.

Informações de memória sobre comprometimento (commit):

```
$ cat /proc/meminfo | fgrep -i Commit | \
awk '{print $1" "$2/1024 " MB"}'

CommitLimit: 7831.14 MB
Committed_AS: 3341.14 MB
```

- **overcommit_ratio (vm.overcommit_ratio)**: Ao se configurar `overcommit_memory` para `2`, o espaço comprometido (*commited*) não está permitido a exceder *swap* mais o percentual da memória RAM referida no valor deste parâmetro, conforme segue a fórmula abaixo:

  ```
  swap + (overcommit_ratio * 0.01 * RAM)
  ```

  Seu valor padrão é 50, que significa 50%.

25.2.5.2 A Relação entre Memory Overcommit e PostgreSQL

O comportamento padrão da memória virtual do Linux não é otimizada para o PostgreSQL.

Devido à maneira que o *kernel* implementa *memory overcommit*, o *kernel* pode matar o processo principal do PostgreSQL se a memória demandar ou o PostgreSQL ou outro processo fizerem o sistema ficar sem memória virtual.

Se isso acontecer, será vista uma mensagem do *kernel* como a que segue (consulte a documentação do sistema e configuração de onde procurar essa mensagem):

```
Out of Memory: Killed process 12345 (postgres).
```

Isso indica que o processo postgres foi terminado devido à pressão de memória. Embora as conexões à base de dados continuarão a funcionar normalmente, nenhuma conexão nova será aceita. Para se recuperar, o PostgreSQL precisa ser reiniciado.

25.3 Configuração Pós Instalação

Principalmente em um ambiente de produção, fazer ajustes iniciais nas configurações do PostgreSQL é necessário.

As configurações iniciais do PostgreSQL são muito conservadoras e muito limitadas.

Ajustamos configurações para fins de melhores condições do banco, desempenho, habilitar ou desabilitar recursos, segurança e etc.

Configurando swappiness para um:

```
# echo 'vm.swappiness = 1' >> /etc/sysctl.conf
```

Desabilitando overcommit_memory:

```
# echo 'vm.overcommit_memory = 2' >> /etc/sysctl.conf
```

Efetivando as mudanças:

```
# sysctl -p
```

Editar o postgresql.conf:

```
$ vim ${PGDATA}/postgresql.conf
```

Parâmetro	Valor	Descrição
max_connections	150	Imaginemos que é um servidor que tem uma previsão de atender até 140 conexões simultâneas, com uma folga ajustamos o valor máximo de conexões para 150.
shared_buffers	?	Aumentar o uso de memória compartilhada para cache de dados para ¼ da memória RAM.
work_mem	16MB	Ajuste de memória a ser usada por operações de ordenação e tabelas hash antes de escrever para arquivos temporários em disco.
max_wal_size	3GB	Também por questão de maior desempenho vamos aumentar a quantidade de arquivos de segmentos do WAL
listen_addresses	'*'	Definir o serviço escutará em todos endereços:

Testando a reinicialização de cluster:

```
$ pg_ctl restart
```

25.4 Configurando Logs

Os logs do servidor (não confundir com logs de transação) registram as atividades do sistema. São de vital importância para auditorias, consultorias e *troubleshooting*.

Conforme os passos a seguir, algumas configurações serão mudadas para uma melhor gerência de logs.

Editar o arquivo de configuração:

```
$ vim ${PGDATA}/postgresql.conf
```

Parâmetro	Valor	Descrição
log_destination	'stderr, csvlog'	Definição do tipo de saída de log para saída padrão de erro e CSV, desta forma gerando dois arquivos; um .log e outro .csv
logging_collector	on	Habilitar log em arquivo
log_directory	'/var/log/postgresql'	Diretório de arquivos de log
log_filename	'postgresql-%Y-%m-%d_%H%M%S.log'	Padrão de nomenclatura do arquivo de log
log_min_duration_statement	0	Registrar em log todos comandos (statements) executados
log_temp_files		Registra todos arquivos temporários criados, pois a partir disso, monitorando os logs, se faz necessária a alteração do parâmetro work_mem para um valor maior para evitar criação de arquivos temporários, o que resultaria em um ganho de desempenho por evitar fazer I/O de disco

Reinicializando o serviço do PostgreSQL:

```
$ pg_ctl restart
```

26 Extensões

- Sobre Extensões
- Instalação de Extensões Contrib
- Instalação de Extensões de Terceiros

26.1 Sobre Extensões

Uma das características marcantes do PostgreSQL é a sua extensibilidade.

Essa extensibilidade do PostgreSQL pode ser facilmente alcançada via instalação de extensões [1] que aumentam seu poder.

As extensões no PostgreSQL podem ser de 3 (três) tipos: nativas (*built in*), de módulos *contrib* [2] ou de terceiros.

Depois de instalada, uma extensão pode ser habilitada em uma base de dados específica com o comando `CREATE EXTENSION`.

Extensões podem trazer novos tipos, funções, *views*, FDWs [3], tabelas ou outras coisas que ampliem de alguma forma o que o PostgreSQL puro não tem.

Para habilitar extensões *built in* um simples `CREATE EXTENSION`.

Extensões que pertencem ao módulo *contrib* precisam dele instalado, o que pode ser feito via pacote ou compilação de código-fonte.

Dependendo da extensão é necessário fazer o carregamento prévio (`*_preload_libraries`) e pode ter suas próprias configurações.

[1] https://www.postgresql.org/docs/current/static/external-extensions.html
[2] https://www.postgresql.org/docs/current/static/contrib.html
[3] https://wiki.postgresql.org/wiki/Foreign_data_wrappers

Verificando extensões habilitadas na base atual:

```
> SELECT extname FROM pg_extension;

. . .
```

Verificando extensões disponíveis para serem instaladas:

```
> SELECT name, comment FROM pg_available_extensions;

. . .
```

Exibir o conteúdo de uma extension:

```
> \dx+ <nome_da_extension>

. . .
```

26.2 Instalação de Extensões Contrib

26.2.1 O Módulo pg_stat_statements

Fornece meios de acompanhar estatísticas de execução de todos comandos SQL executados no servidor.

Este módulo deve ser carregado adicionando `pg_stat_statements` ao parâmetro `shared_preload_libraries` no postgresql.conf, porque ele precisa de memória compartilhada adicional e é necessário um *restart* no serviço.

Ao carregar esse módulo ele rastreia estatísticas através de todas as bases do servidor. Para acessar e manipular essas estatísticas, é fornecida uma *view* de mesmo nome e as funções `pg_stat_statements_reset` e `pg_stat_statements`.

https://www.postgresql.org/docs/current/static/pgstatstatements.html

Editar o postgresql.conf mudando shared_preload_libraries, adicionar configurações próprias do módulo e em seguida reiniciar o serviço:

```
$ vim ${PGDATA}/postgresql.conf && pg_ctl restart

shared_preload_libraries = 'pg_stat_statements'

# pg_stat_statements Settings

pg_stat_statements.max = 10000
pg_stat_statements.track = all
pg_stat_statements.track_utility = on
pg_stat_statements.save = on
```

Criar uma base de dados para teste de performance:

```
$ createdb db_bench
```

Habilitar a extensão na base:

```
$ psql -Atqc 'CREATE EXTENSION pg_stat_statements;' db_bench
```

Dar reset nos dados de pg_stat_statements:

```
$ psql -Atqc 'SELECT pg_stat_statements_reset();' db_bench
```

Inicialização da base de teste de desempenho com o pg_bench:

```
$ pgbench -i db_bench
```

Testes com o pgbench:

```
$ pgbench -c10 -t300 db_bench
```

Conectar à base, habilitar o modo expandido do psql e consultar estatísticas:

```
$ psql db_bench

> \x on

> SELECT
    query,
    calls,
    total_time,
    rows, 100.0 * shared_blks_hit /
        nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent

    FROM pg_stat_statements
    ORDER BY total_time DESC LIMIT 5;

-[ RECORD 1 ]---------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
calls       | 3000
total_time  | 188761.810633
rows        | 3000
hit_percent | 99.9915946962533359
-[ RECORD 2 ]---------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
calls       | 3000
total_time  | 157445.243361
rows        | 3000
hit_percent | 99.9784892266876994
-[ RECORD 3 ]---------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 184.565168
rows        | 0
hit_percent | 99.9033816425120773
-[ RECORD 4 ]---------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
calls       | 3000
total_time  | 109.497158
rows        | 3000
hit_percent | 98.3849398210060693
-[ RECORD 5 ]---------------------------------------------------------------
query       | vacuum analyze pgbench_accounts
calls       | 1
total_time  | 73.787167
rows        | 0
hit_percent | 99.7611464968152866
```

26.3 Instalação de Extensões de Terceiros

Considera-se extensões de terceiros aquelas que não são *built ins* nem estão como módulos `contrib`.

Sua forma de instalação pode variar podendo ser por meio de compilação ou por um *software* gerenciador de extensões.

26.3.1 PGXN

PGXN significa PostgreSQL Extension Network é um sistema de distribuição central de bibliotecas de extensão do PostgreSQL.

Além do site em si, fornece uma API própria e um cliente (pgxnclient) para busca e instalação de extensões no PostgreSQL

26.3.2 pgxnclient

É um utilitário gerenciador de extensões providas pela PGXN escrito em Python.

Determinar qual é o executável mais recente de Python:

```
# export PYTHON="/usr/bin/`ls /usr/bin/python3* | \
sed 's:/usr/bin/::g' | egrep '\.[0-9]$'`"
```

Instalação do gerenciador de módulos Python (PIP):

```
# wget -qO- https://bootstrap.pypa.io/get-pip.py | ${PYTHON}
```

Via PIP instalar o pgxnclient:

```
# pip install pgxnclient
```

Mudando a variável de ambiente PATH adicionando o diretório onde está o binário de pg_config:

```
# export PATH="${PATH}:`dirname \`su - postgres -c 'which pg_config'\``"
```

Instalação de make e gcc que são necessários para instalação de um novo módulo que é compilado na hora da instalação:

```
# yum install -y make gcc
```

Para testar vamos instalar a extensão shacrypt [1]:

```
# pgxnclient install shacrypt
```

[1] https://github.com/dverite/postgres-shacrypt

Criar base para teste e em seguida conectar à ela:

```
$ createdb db_shacrypt && psql db_shacrypt
```

Habilitar a extensão na base corrente:

```
> CREATE EXTENSION shacrypt;
```

Testando a função de encriptação SHA256 provida pelo módulo:

```
> SELECT sha256_crypt('minha_senha', 'salt_aqui');

                        sha256_crypt
------------------------------------------------------------
 $5$salt_aqui$2OoHmQQ10G..Iw7j1Ado58eqPtvBVnxr/I.B6JOWqi3
```

27 Foreign Data Wrappers

- Sobre Foreign Data Wrappers
- postgres_fdw: Acessando outro Servidor PostgreSQL
- mysql_fdw: Acessando o MySQL ou MariaDB
- file_fdw: Acesso a Arquivos

27.1 Sobre Foreign Data Wrappers

Em 2003 houve uma nova especificação chamada SQL/MED (SQL *Management of External Data* – Gerenciamento de Dados SQL Externos), a qual foi adicionada ao padrão SQL.

É uma forma padronizada de se lidar com acesso a objetos remotos vindos de bases de dados SQL.

Em 2011, na versão 9.1 do PostgreSQL foi lançada com suporte *read-only* (somente leitura) desse padrão e em 2013 o suporte a escrita foi adicionado na versão 9.3.

Hoje há uma variedade de *Foreign Data Wrappers* (FDW) disponível que habilita um servidor PostgreSQL a acessar diferentes armazenamentos de dados remotos, desde outros bancos SQL a um simples arquivo de texto.

Para encontrar esses FDWs pode ser na Wiki do PostgreSQL [1] ou na PGXN [2]:

[1] http://wiki.postgresql.org/wiki/Foreign_data_wrappers
[2] http://pgxn.org

Aviso:

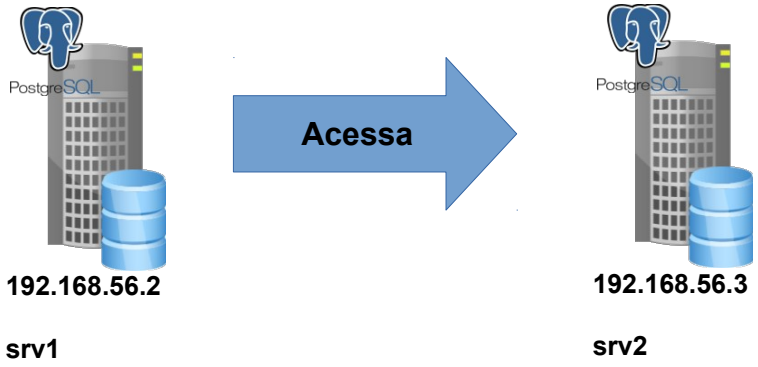
Tenha em mente que a maioria dos FDWs não são suportados oficialmente pelo PGDG (*PostgreSQL Global Development Group*) e que alguns desses projetos ainda estão em sua versão beta.

Use com cuidado!

27.2 postgres_fdw: Acessando outro Servidor PostgreSQL

A partir da versão 9.3 além de poder acessar outra base PostgreSQL também é possível fazer gravações.

Na prática a seguir um servidor (**srv1**) acessará outro (**srv2**), leitura e escrita.



srv2

Definir uma senha para o usuário postgres:

```
$ psql -c "ALTER ROLE postgres ENCRYPTED PASSWORD '123';"
```

Permitir conexão no pg_hba.conf:

```
$ echo 'host all all 192.168.56.2/32 md5' >> ${PGDATA}/pg_hba.conf
```

Recarregar as configurações:

```
$ pg_ctl reload
```

Conectar localmente com o utilitário cliente psql:

```
$ psql
```

Criação da base de dados de teste:

```
> CREATE DATABASE db_teste2;
```

Conectar à nova base criada:

```
> \c db_teste2
```

Criação de tabela de teste:

```
> CREATE TABLE tb_teste(
    id SERIAL PRIMARY KEY,
    campo_a INT,
    campo_b VARCHAR(10));
```

srv1

Conectar localmente com o utilitário cliente psql:

```
$ psql
```

Criação da base de dados de teste:

```
> CREATE DATABASE db_teste1;
```

Conectar à nova base criada:

```
> \c db_teste1
```

Habilitando a extensão postgres_fdw na base de dados atual:

```
> CREATE EXTENSION postgres_fdw;
```

Criação de servidor FDW:

```
> CREATE SERVER srv_srv2
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host '192.168.56.3', dbname 'db_teste2', port '5432');
```

Criação de mapeamento de usuários:

```
> CREATE USER MAPPING FOR postgres
    SERVER srv_srv2
    OPTIONS (user 'postgres', password '123');
```

Criação da tabela estrangeira que vai se relacionar com a tabela criada em srv2:

```
> CREATE FOREIGN TABLE tb_fdw_teste(
    id SERIAL,
    campo_a INT,
    campo_b VARCHAR(10))
    SERVER srv_srv2
    OPTIONS (table_name 'tb_teste', updatable 'true');
```

Inserção de um registro:

```
> INSERT INTO tb_fdw_teste (campo_a, campo_b) VALUES (700, 'foo');
```

Consulta todos registros na tabela estrangeira (srv1):

```
> SELECT * FROM tb_fdw_teste;

 id | campo_a | campo_b
----+---------+---------
  1 |     700 | foo
```

Consulta todos registros na tabela original (srv2):

```
> SELECT * FROM tb_teste;

 id | campo_a | campo_b
----+---------+---------
  1 |     700 | foo
```

Apagando tudo o que foi criado para o teste de FDW:

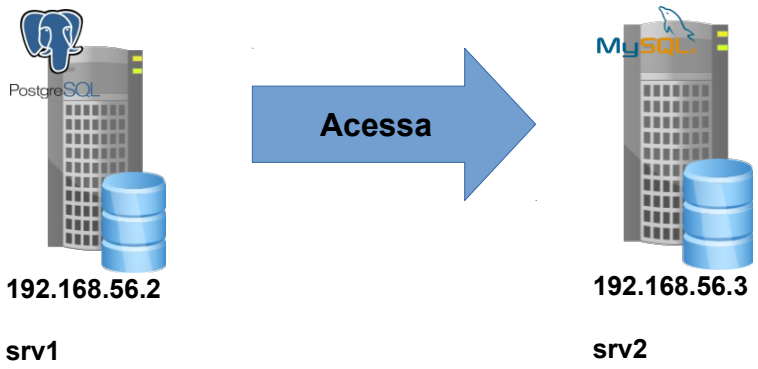
```
> DROP FOREIGN TABLE tb_fdw_teste; -- Apaga a tabela estrangeira

> DROP USER MAPPING FOR postgres SERVER srv_srv2;  -- Apaga o mapeamento de usuário

> DROP SERVER srv_srv2; -- Apaga a configuração de servidor

> DROP EXTENSION postgres_fdw; -- Desabilita a extensão na base de dados
```

27.3 mysql_fdw: Acessando o MySQL ou MariaDB

Como avisado previamente, nem todos FDWs têm suporte do PGDG e alguns estão em versão beta, na prática a seguir isso se aplica ao FDW do MySQL.

Por enquanto só é possível fazer consultas.

https://github.com/EnterpriseDB/mysql_fdw



27.3.1 Preparação no MySQL

Criar uma base de dados no MySQL:

```
# mysql -e 'CREATE DATABASE db_teste2;'
```

Acessar a nova base:

```
# mysql db_teste2
```

Criação de usuário no MySQL:

```
> CREATE USER 'postgres'@'192.168.56.2' IDENTIFIED BY '123';
```

Conceder todos privilégios a todos objetos do banco ao usuário e servidor PostgreSQL :

```
> GRANT ALL PRIVILEGES ON db_teste2.* TO 'postgres'@'192.168.56.2';
```

Criação de tabela a ser acessada via FDW:

```
> CREATE TABLE tb_teste(
        id INT AUTO_INCREMENT PRIMARY KEY,
        campo_a INT,
        campo_b VARCHAR(10));
```

Inserir uma linha na tabela:

```
> INSERT INTO tb_teste (campo_a, campo_b) VALUES (700, 'foo');
```

27.3.2 Instalação do mysql_fdw no Servidor PostgreSQL

Instalação da parte de compilação e o código-fonte do MariaDB:

```
# yum install -y make gcc mariadb-devel
```

Baixar o repositório e acessar o diretório:

```
# git clone https://github.com/EnterpriseDB/mysql_fdw.git /tmp/mysql_fdw

# cd /tmp/mysql_fdw
```

Ler as variáveis de ambiente do usuário postgres:

```
# source ~postgres/.pgvars
```

Com a variável de ambiente USE_PGXS habilitada iniciar a compilação do FDW:

```
# export USE_PGXS=1 && make && make install
```

Apagar os pacotes usados na compilação:

```
# yum erase -y make gcc mariadb-devel
```

Criar arquivo de configuração de bibliotecas MySQL/MariaDB e atualizar o cache:

```
# echo '/usr/lib64/mysql' > /etc/ld.so.conf.d/mysql.conf

# ldconfig
```

27.3.3 Configuração do mysql_fdw e Testes

De usuário root para usuário postgres:

```
# su - postgres
```

Acessar o psql:

```
$ psql
```

Criar base de dados de teste:

```
> CREATE DATABASE db_teste1;
```

Acessar nova base de dados:

```
> \c db_teste1
```

Habilitar exetensão de FDW MySQL:

```
> CREATE EXTENSION mysql_fdw;
```

Determinar um servidor a ser acessado via FDW:

```
> CREATE SERVER srv_mysql
    FOREIGN DATA WRAPPER mysql_fdw
    OPTIONS (host '192.168.56.3', port '3306');
```

Mapeamento de usuários:

```
> CREATE USER MAPPING FOR postgres SERVER srv_mysql
    OPTIONS (username 'postgres', password '123');
```

Criação de tabela estrangeira:

```
> CREATE FOREIGN TABLE ft_teste(
    id INT,
    campo_a INT,
    campo_b VARCHAR(10))
    SERVER srv_mysql
        OPTIONS (dbname 'db_teste2', table_name 'tb_teste');
```

Inserir uma linha na tabela estrangeira:

```
> INSERT INTO ft_teste (campo_a, campo_b) VALUES (500, 'bar');
```

Consultando a tabela estrangeira (srv1 - PostgreSQL):

```
> SELECT * FROM ft_teste;

 id | campo_a | campo_b
----+---------+---------
  1 |     700 | foo
  2 |     500 | bar
```

Consultando a tabela (srv2 - MySQL):

```
> SELECT * FROM tb_teste;

+----+---------+---------+
| id | campo_a | campo_b |
+----+---------+---------+
|  1 |     700 | foo     |
|  2 |     500 | bar     |
+----+---------+---------+
```

27.4 file_fdw: Acesso a Arquivos

O FDW `file_fdw` pode ser usado para acessar arquivos de dados no servidor, no sistema de arquivos ou executar programas e pegar sua saída.

O arquivo de dados ou a saída do programa deve estar em um formato que possa ser lido pelo comando `COPY FROM`.

Por enquanto o acesso ao arquivo de dados é somente leitura.

https://www.postgresql.org/docs/current/static/file-fdw.html

Criação do arquivo CSV de teste:

```
$ cat << EOF > /tmp/capitais_sudeste.csv
SP;São Paulo;12106920
RJ;Rio de Janeiro;6520266
MG;Belo Horizonte;2523794
ES;Vitória;363140
EOF
```

No psql Habilitar a extensão file_fdw:

```
> CREATE EXTENSION file_fdw;
```

Configuração de servidor de FDW:

```
> CREATE SERVER srv_srv1_file_fdw FOREIGN DATA WRAPPER file_fdw;
```

Tabela estrangeira para o arquivo:

```
> CREATE FOREIGN TABLE ft_capitais_sudeste(
    id char(2),
    nome VARCHAR(50),
    populacao INT)
    SERVER srv_srv1_file_fdw
    OPTIONS (filename '/tmp/capitais_sudeste.csv', format 'csv', delimiter ';');
```

Consulta à tabela estrangeira:

```
> SELECT id, nome, populacao FROM ft_capitais_sudeste;

 id |      nome       | populacao
----+-----------------+-----------
 SP | São Paulo       |  12106920
 RJ | Rio de Janeiro  |   6520266
 MG | Belo Horizonte  |   2523794
 ES | Vitória         |    363140
```

Via shell do sistema operacional, apagar a primeira linha:

```
> \! sed '1d' -i /tmp/capitais_sudeste.csv
```

Verificando a tabela após a alteração do arquivo:

```
> SELECT id, nome, populacao FROM ft_capitais_sudeste;

 id |      nome       | populacao
----+-----------------+-----------
 RJ | Rio de Janeiro  |   6520266
 MG | Belo Horizonte  |   2523794
 ES | Vitória         |    363140
```

28 PostgreSQL no Docker

- Sobre o Docker
- Contêineres PostgreSQL no Docker
- Docker Compose

28.1 Sobre o Docker

Docker é uma plataforma para disponibilizar uma aplicação de forma rápida.

Seu funcionamento é muito similar a tecnologias de virtualização em contêiner, tais como OpenVZ, LXC, Jails e etc.

28.2 Contêineres PostgreSQL no Docker

Os passos a seguir serão com base na imagem oficial do PostgreSQL do *Docker Hub* [1].

O objetivo é construir um contêiner Docker PostgreSQL preparado para suportar português do Brasil.

Como desafio a mais também criar volumes para o `PGDATA`, diretório de *logs* de transação e diretório de *logs*, além de um ponto de montagem em memória para o diretório de estatísticas temporárias.

[1] https://hub.docker.com/_/postgres/

Criação de rede para o contêiner:

```
$ docker network create net_curso
```

Criação de diretório para o laboratório Docker e dar propriedade ao grupo:

```
$ sudo mkdir -m 770 /srv/docker && sudo chgrp docker /srv/docker
```

Criação da estrutura de diretórios que serão usados:

```
$ mkdir -p /srv/docker/pg/{build,data,pg_wal,log,pg_stat_tmp}
```

Criação de um contêiner temporário para extrair arquivos necessários:

```
$ docker run -itd --name ct_tmp postgres
```

Dentro do contêiner criar um diretório chamado conf:

```
$ docker exec -it ct_tmp mkdir /tmp/conf
```

Dentro do contêiner copiar os arquivos de configuração para /tmp/conf:

```
$ docker exec -it ct_tmp bash -c 'cp /var/lib/postgresql/data/*.conf /tmp/conf'
```

Copiar o diretório de dentro do contêiner:

```
$ docker cp ct_tmp:/tmp/conf /srv/docker/pg/
```

Variável de ambiente com o ID de usuário postgres do contêiner:

```
$ PGUSER_ID=`docker exec -it ct_tmp id -u postgres | tr -d "\r"`
```

Dar propriedade para o diretório ao usuário postgres do contêiner (via ID) e o grupo docker do host:

```
$ sudo chown -R ${PGUSER_ID}:docker /srv/docker/pg
```

Dar permissões de leitura e escrita para o grupo:

```
$ sudo chmod -R g+rw /srv/docker/pg
```

Edição do postgresql.conf que ficará fora do contêiner:

```
$ vim /srv/docker/pg/conf/postgresql.conf
```

```
log_destination = 'stderr'
logging_collector = on
log_directory = '/var/log/postgresql'
log_filename = 'postgresql-%Y%m%d_%H%M.log'
stats_temp_directory = '/var/lib/postgresql/pg_stat_tmp'
```

Apagar o contêiner temporário:

```
$ docker rm -f ct_tmp
```

Mudar para o diretório build:

```
$ cd /srv/docker/pg/build
```

Edição do arquivo Dockerfile:

```
$ vim Dockerfile


FROM postgres:latest

ENV POSTGRES_INITDB_ARGS="-k \
-D /var/lib/postgresql/data \
-E utf8 -U postgres \
--locale=pt_BR.utf8 \
--lc-collate=pt_BR.utf8 \
--lc-monetary=pt_BR.utf8 \
--lc-messages=en_US.utf8 \
-T portuguese"\
    POSTGRES_INITDB_WALDIR='/var/lib/postgresql/pg_wal'\
    LANG="en_US.UTF-8"

RUN localedef -i pt_BR -c -f UTF-8 \
-A /usr/share/locale/locale.alias pt_BR.UTF-8 && \
usermod -s /bin/bash -d /var/lib/postgresql \
-c 'PostgreSQL System User' postgres && \
echo "\x auto\n\
\set HISTCONTROL ignoreboth\n\
\set COMP_KEYWORD_CASE upper" >> ~postgres/.psqlrc && \
chown -R postgres: ~postgres
```

Processo de build da imagem com o nome postgres_br e versão v1.0:

```
$ docker build -t postgres_br:v1.0 .
```

Criação do contêiner usando a imagem criada:

```
$ docker run -itd \
    --memory 1g \
    --memory-swappiness 1 \
    --memory-swap 256m \
    --restart=always \
    --name pg \
    --hostname pg.local \
    -p 5433:5432 \
    --network net_curso \
    --volume /srv/docker/pg/conf:/etc/postgresql:ro \
    --volume /srv/docker/pg/data:/var/lib/postgresql/data \
    --volume /srv/docker/pg/pg_wal:/var/lib/postgresql/pg_wal \
    --volume /srv/docker/pg/log:/var/log/postgresql \
    --tmpfs /var/lib/postgresql/pg_stat_tmp:size=32M,mode=0770 \
    postgres_br:v1.0 postgres -c 'config_file=/etc/postgresql/postgresql.conf'
```

No host, via netstat, verificando a porta 5433:

```
$ sudo netstat -nltp | fgrep 5433
```

```
tcp6       0      0 :::5433                 :::*                    LISTEN      29368/docker-proxy
```

Após a criação do contêiner fazer a exclusão de arquivos e diretórios desnecessários:

```
$ docker exec -itu postgres pg rm -fr \
    /var/lib/postgresql/data/{pg_stat_tmp,log,*.conf}
```

Testando o contêiner:

```
$ docker exec -itu postgres pg psql -Atqc 'SELECT version();'

. . .
```

28.3 Docker Compose

Compose é uma ferramenta para rodar aplicações de múltiplos contêineres.

Com o Compose, usa-se um arquivo YAML para configurar uma aplicação com vários serviços. Com um único comando pode-se criar e iniciar serviços para uma aplicação.

Para o nosso laboratório, nossa aplicação terá apenas um contêiner, que é o do PostgreSQL.

https://docs.docker.com/compose

Apagar o contêiner pré existente:

```
$ docker rm -f pg
```

Apagar a rede pré existente:

```
$ docker network rm net_curso
```

Mudar para o diretório build:

```
$ cd /srv/docker/pg/build
```

Criar o arquivo docker-compose.yml:

```
$ vim docker-compose.yml

version: '2.3'
services:
    db:
        container_name: pg
        hostname: pg.local
        build: /srv/docker/pg/build
        restart: always
        command: postgres -c 'config_file=/etc/postgresql/postgresql.conf'
        mem_limit: 1G
        memswap_limit: 256M
        mem_swappiness: 1

        ports:
            - "5433:5432"
        volumes:
            -  /srv/docker/pg/conf:/etc/postgresql:ro
            -  /srv/docker/pg/data:/var/lib/postgresql/data
            -  /srv/docker/pg/pg_wal:/var/lib/postgresql/pg_wal
            -  /srv/docker/pg/log:/var/log/postgresql

        tmpfs:
            - /var/lib/postgresql/pg_stat_tmp:size=32M,mode=0770

        networks:
            - net_curso

networks:
    net_curso:
        name: net_curso
        driver: bridge
```

Mudar para o diretório build:

```
$ cd /srv/docker/pg/build
```

Testando o contêiner:

```
$ docker exec -itu postgres pg psql -Atqc 'SELECT version();'

. . .
```