PostgreSQL Administração

AVISO DE RESPONSABILIDADE

As informações contidas neste material de treinamento são distribuídas “NO ESTADO EM QUE SE ENCONTRAM”, sem qualquer garantia, expressa ou implícita. Embora todas as precauções tenham sido tomadas na preparação deste material, a Processor Alfamídia LTDA. não têm qualquer responsabilidade sobre qualquer pessoa ou entidade com respeito à responsabilidade, perda ou danos causados, ou alegadamente causados, direta ou indiretamente, pelas instruções contidas neste material ou pelo software de computador e produtos de hardware aqui descritos.

Conteúdo retirado do site do grupo de desenvolvimento do PostgreSQL, traduzido e adaptado por Processor Alfamídia LTDA.

Novembro de 2004

Processor Alfamídia LTDA
Hilário Ribeiro 202
Porto Alegre, RS
(51) 3346-7300
http://www.alfamidia.com.br

PostgreSQL - Administração

Sobre o Curso

O Curso PostgreSQL Administração oferece aos alunos uma visão ampla da estrutra do física e lógica do banco de dados , bem como dos aplicativos operacionais do mesmo.

Ao final do curso os alunos serão capazes de instalar e configurar um banco de dados para uso em aplicações comerciais.

Formato do Curso

Esse curso é dividido em 11 unidades e um manual de referência, a maioria delas apresentando novas informações e contendo demonstrações de uso do banco de dados.

Objetivos do Curso

Após completar esse curso, você será capaz de

- Instalar e configurar um servidor de Banco de Dados
- Administrar e gerenciar o PostgreSQL

Pré-requisitos do Curso

Para obter o máximo desse treinamento, você deve já estar familiarizado com:

- Os sistemas operacionais de mercado.
- Linguagem SQL .

Unidade 1:
Instalação e configuração em Linux

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Obter o PostgreSQL.
- Instalar o PostgreSQL
- Inicializar o Banco de Dados

Para instalação do Banco de Dados PostgreSQL é interessante uma básica noção do sistema operacional LINUX. Nesta apostila veremos passo a passo como instalar e inicializar o Banco de Dados.

Tópicos da Unidade

- Obtendo o PostgreSQL
- Instalando o PostgreSQL

Obtendo o software de instalação

No site do grupo de desenvolvimento do PostgreSQL (**www.postgresql.org**) podemos fazer o download do software sem custos. O software pode ser obtido em duas formas:

Fonte: baixando o fonte do PostgreSQL talvez seja possível instalar o Banco de Dados em um sistema LINUX que não se encontra na lista dos suportados.

Pacote: em uma segunda fase o grupo disponibiliza os pacotes (RPM's) de instalação do PostgreSQL.

Requerimentos

Em geral, uma plataforma moderna tipo Unix-compatível deveria ser capaz de rodar o PostgreSQL. O resumo de plataformas que receberam testes específicos e as outras não suportadas estão listadas logo abaixo nesta apostila. No diretório doc da distribuição que você estiver usando você encontra diversos FAQ específicos de várias plataformas caso você necessite de ajuda.

Os pré-requisitos que existem para a compilação do PostgreSQL:

• O GNU make é requerido; outros pragramas make podem *não funcionar*. O GNU make é frequentemente instalado com o nome de gmake; essa apostila irá sempre se referir à ele por gmake. (Em alguns sistemas GNU make é a ferramente default com o nome make.) Para testar se você possui o GNU make instalado, digite

gmake --version

É recomendado o uso da versão 3.76.1 ou posterior.

• Você precisa um compilador C ISO/ANSI. As versões recentes do GCC são recomendáveis, mas o PostgreSQL é conhecido por poder ser construído por uma enorme variedade de compiladores de diferentes empresas.

• gzip é também necessário para se deszipar a própria distribuição (caso você baixe os fontes).

• A Readline library do GNU (para edição confortável de linhas e busca de comandos no arquivo de history) será automaticamente usada se for encontrada. Você pode desejar instalá-la antes de proceder, mas isso não é essencial. (No NetBSD, a library libedit é Readline compatível e será usada se libreadline não for encontrada.)

• Os programas GNU Flex e Bison são necessários para a compilação quando se for começar do zero, mas eles *não* são necessários quando ela for feita apartir de um pacote de fontes pois os arquivos pré-gerados de saída já estão incluidos no pacote.

Verifique se você tem espaço livre suficiente em disco. É necessário ter pelo menos 30 MB de espaço para a árvore dos fontes durante a compilação e cerca de 10 MB para o diretório de instalação. Um cluster vazio de bancos de dados usam cerca de 20 MB, os bancos de dados usam mais ou menos cinco (5) vezes mais espaço em disco do que arquivos texto com o mesmo conteúdo em dados. Se você for fazer alguns testes de regressão, você irá necessitar temporáriamente de uns extra 20 MB. Use o comando **df -h** para checar o espaço em disco.

Antes de instalar

Você precisa estar logado com super-usuário (root) para instalar o PostgreSQL. Verifique se já existe uma versão anterior do Banco de Dados rodando em seu sistema. Caso exista , pare o serviço e remova o aplicativo.

Para verificar a existência de uma versão anterior digite os seguintes comandos :

rpm –qa | grep –i postgre

ou

ps –aux | grep –i postmaster

Nota : caso exista uma versão anterior esta deve ser removida.

Instalação pelo pacote RPM

Seguindo regras a instalação através de pacote é muito simples, o cuidado que devemos ter é a confirmação de que o pacote adquirido é o suportado para nossa versão do LINUX. Feito esta verificação para instalar utilize o comando seguinte :

root# rpm –iv <nome do pacote.rpm>

onde nome do pacote equivale normalmente a versão do banco de dados.

Exemplo :

root# rpm –iv postgresql-7.2.1-i386.rpm

Instalação rápida pelo fonte

Para se instalar a partir do código fonte é necessário que se tenha instalado o compilador "GNU gcc", e utilitários de desenvolvimento, como o "GNU make". Esses programas fazem parte da distribuição original do LINUX. Para a instalação através do fonte siga os passos a seguir:

1. Faça login como root e mude para o diretório onde baixou o fonte.

2. Instale os fonte com o seguinte comando:

   root# tar –xzvf postgresql-7.2.1.tar

3. Mude para o diretório do PostgreSQL que foi criado:

   root# cd postgresql-7.2.1/

4. Execute o programa de configuração :

   root# ./configure

5. Execute o programa make e make install :

   root# gmake
   root# gmake install

Pronto, o software do banco de dados PostgreSQL está instalado.

Se você está fazendo um Upgrade

O formato interno de gravação dos dados do PostgreSQL mudam com os novos releases. Portanto, se você está fazendo um upgrade de uma instalação existente que não tem o número da versão "7.2.x", você deve fazer um backup e restaurar seus dados como mostrado aqui. Essas instruções assumem que sua instalação existente está no diretório `/usr/local/pgsql`, e que a área de dados está no `/usr/local/pgsql/data`. Substitua seus paths apropriadamente.

1. Tenha certeza de que o seu banco de dados não está sofrendo updates durante ou após o backup. Isso não afeta a integridade do backup, mas com certeza os dados alterados durante os updates não estarão inclusos. Se necessário, edite as permissões no arquivo `/usr/local/pgsql/data/pg_hba.conf` (ou equivalente) para disabilitar o accesso ao banco de todos, exceto você.

2. Para dumpear a sua instalação, digite:

pg_dumpall > *arquivo_de_saída*

Se você necessita preservar os OIDs (por exemplo quando você os estiver usando como foreign keys), use então a opção `-o` quando for rodar o **pg_dumpall**. O **pg_dumpall** não salva objetos largos. Cheque a seção XXXXXXX se você precisar fazer isso. Tenha certeza também de estar usando o comando **pg_dumpall** da versão que você está atualmente usando. O **pg_dumpall** não da versão 7.2 não deveria ser usando em bancos de dados de versõs anteriores a 7.2.

3. Se você está instalando uma nova versão na mesma localização da anterior então dê um shut down no antigo servidor antes de instar os novos arquivos :

kill -INT `cat /usr/local/pgsql/data/postmaster.pid`

Versões anteriores à 7.0 não têm o arquivo `postmaster.pid`. Se você está usando uma versão que não têm esse arquivo tente achar o id do processo usando o comando **ps ax | grep postmaster**, e usar o comando **kill** após.

Em sistemas que tem o PostgreSQL starteado na hora do boot, existe provávelmente um arquivo de start-up que irá fazer a mesma coisa. Por exemplo, no Red Hat Linux o comando **/etc/rc.d/init.d/postgresql stop** deve funcionar. Outra possibilidade é **pg_ctl stop**.

4. Se você está instalando uma nova versão na mesma localização da anterior é também uma boa idéia manter a anterior, em caso de você ter problemas e necessite reverter o processo. Use este comando para renomear a antiga instalação :

mv /usr/local/pgsql /usr/local/pgsql.old

Após você ter instalado o PostgreSQL 7.4, crie um novo diretório para a base de dados e starte o novo servidor. Lembre que você de executar esses comandos estando logado como postgres (o qual você já deve ter já que isso é um upgrade).

/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data

Finalmente, restore seus dados com

/usr/local/pgsql/bin/psql -d template1 -f *outputfile*

usando o *novo* psql.

Você pode também instalar a nova versão em paralelo com a velha para diminuir seu tempo com o banco parado.

Instalação normal pelo fonte

1. Configuração

O primeiro passo do procedimento de instalação é o de configurar árvore dos fontes para o seu sistema e escolher as opções que você gostaria. Isso é feito rodando o script chamado `configure`.

Para uma instalação default simplesmente digite

```
./configure
```

Esse script rodará uma série de testes para descobrir os valores das variáveis dependentes do sistema e detectar algumas peculiaridades do seu sistema operacional, finalmente ele irá criar diversos arquivos na árvore para gravar o que ele achou.

A configuração default irá construir o servidor e os utilitários, assim como todas as aplicações clientes e as interfaces que requerem apenas um compilador C. Todos os arquivos serão instalados por default no diretório `/usr/local/pgsql`.

Você pode customizar o processo de instalação e construção informando um ou mais opções de linha de comando para o `configure`:

`--prefix=`*`PREFIXO`*

Instala todos os arquivos dentro do diretório *`PREFIXO`* ao invés de `/usr/local/pgsql`. Na verdade, os arquivos serão instalados dentro de vários subdiretórios; nunca nenhum arquivo será instalado diretamente dentro do diretório *`PREFIXO`*.

Se você tem necessidades especiais, você pode também customizar subdiretórios individuais com as seguintes opções.

`--exec-prefix=`*`EXEC-PREFIXO`*

Você pode instalar arquivos que são independentes da estrutura dentro de um diretório diferente, *`EXEC-PREFIXO`*, do que *`PREFIXO`* foi setado. Isso pode ser útil para compartilhar arquivos independentes da estrutura entre vários hosts.

Se você omitir isso, então *EXEC-PREFIXO* é setado igual à *PREFIXO* e ambos os arquivos que são dependentes e independentes da estrutura serão instalados dentro da mesma árvore, o que provávelmente será isso que você quer.

--bindir=*DIRETORIO*

Especifica o diretório para os programas executáveis. O diretório default é *EXEC-PREFIXO*/bin, o que normalmente significa /usr/local/pgsql/bin.

--datadir=*DIRETORIO*

Seta o diretório para os arquivos de dados read-only usados pelos programas instalados. O default é *PREFIXO*/share. Note que isso não tem nada a ver com o local em que os arquivos do seu banco de dados serão colocados.

--sysconfdir=*DIRETORIO*

É o diretório para os vários arquivos de configuração, *PREFIXO*/etc por default.

--libdir=*DIRETORIO*

É a localização onde serão instaladas as libraries e os módulos dinamicamente lidos. O default é *EXEC-PREFIXO*/lib.

--includedir=*DIRETORIO*

É o diretório para a instalação dos arquivos header do C e do C++. O default é *PREFIXO*/include.

--docdir=*DIRETORIO*

Os arquivos de documentação, exceto as “man pages”, serão instalados nesse diretório. O default é *PREFIXO*/doc.

`--mandir=`*`DIRETORIO`*

As man pages que vêm com o PostgreSQL serão instaladas nesse diretório, nos seus respectivos `man`*`x`* subdiretórios. O default é *`PREFIXO`*`/man`.

Note: Todo o cuidado foi tomado para tornar possível instalar o PostgreSQL dentro de shared locations (tal como `/usr/local/include`) sem interferência com o resto do sistema. Primeiro, a string "`/postgresql`" é automaticamente adicionada à `datadir`, `sysconfdir`, e `docdir`, à menos que o nome do diretório já contenha a string "`postgres`" ou "`pgsql`". Por exemplo, se você escolher `/usr/local` como prefixo, a documentação será instalada em `/usr/local/doc/postgresql`, mas se o prefixo for `/opt/postgres`, então ela será colocada em `/opt/postgres/doc`. Segundo, o layout de instalação dos arquivos headers do C e do C++ foram reorganizados no release 7.2. Os arquivos headers públicos das interfaces dos clientes serão instalados dentro do `includedir`. Os arquivos headers internos e do servidor serão instalados dentro de diretórios privados dentro do `includedir`. Finalmente, um subdiretório privado será também criado, se apropriado, abaixo do `libdir` para os módulos dinamicamente lidos.

`--with-includes=`*`DIRETORIOS`*

`DIRETORIOS` é uma lista de diretórios separados por dois pontos (:) que serão adicionados à lista que o compilador usa para procurar por arquivos de header. Se você tem pacotes opcionais (tipo o GNU Readline) instalado em uma localização não standard, você deve usar essa opção e provávelmente a opção correspondente `–withlibraries`.

Exemplo: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`DIRETORIOS`*

`DIRETORIOS` é uma lista de diretórios separados por dois pontos (:) para a procura de libraries. Você precisará usar essa opção (e a opção correspondente `--with-includes`) se você tiver packages instaladas em localizações não standard.

Exemplo: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-locale`

Habilita o suporte à locais. Há uma penalização de performance associada ao puporte à locias, mas se você estiver em um ambiente fora do Inglês, talvez essa seja uma boa opção.

`--enable-recode`

Habilita o suporte à caracteres single-byte.

`--enable-multibyte`

Habilita o suporte à codificação de caracteres multibyte (includindo o Unicode) e conversão de caracter sets. Note que algumas interfaces (tipo Tcl ou Java) esperam que todas as strings de caracter estejam em Unicode, essa opção é então required para o correto suporte à essas interfaces.

`--enable-nls[=`*`LIGUAGENS`*`]`

Habilita o Native Language Support (NLS), isso é, a habilidade de display de mensagens em uma outra língua que o Inglês. *`LINGUAGENS`* é uma lista separada por espaços de linguas que você quer obter suporte, por exemplo `--enable-nls='de fr'`. (A interseção entre a sua lista e o set das já atualmente suportadas será computada automáticamente.) Se você não especificar uma lista, então todas as traduções disponíveis serão instaladas.

Para usar essa opção, você precisará da implementação gettext API. Alguns sistemas operacionais já tem isso built-in (exemplo, Linux, NetBSD, Solaris), para outros sistemas você pode fazer um download um pacote add-on daqui:

http://www.postgresql.org/~petere/gettext.html.

Se você está usando a implementação gettext do GNU C library então adicionalmente você irá precisar do pacote GNU gettext para alguns programas utilitários. Para qualquer outra implementação você não precisará disso.

`--with-pgport=`*`NUMERO`*

Selecione *`NUMERO`* como a porta default para o servidor e os clientes. A porta default é 5432. A porta pode sempre ser modificada depois, mas se você especifica isso aqui, então ambos os clientes e o servidor terão o mesmo número default compilado, o que pode ser muito conveniente. Geralmente a única boa razão para não selecionar um valor default é se você pretende rodar múltiplos servidores PostgreSQL em uma mesma máquina.

`--with-CXX`

Monta a C++ interface library.

`--with-perl`

Monta o módulo de interface Perl. Ele será instalado no local usual para os Perlmodules (tipicamente dentro do `/usr/lib/perl`), você deve ter acesso do root para executar o processo de instalação (veja o passo 4). Você necessita ter o Perl 5 instalado para usar essa opção.

`--with-python`

Monta o módulo de interface Python. Você deve ter acesso do root para executar o processo de instalação do Python no seu diretório default (`/usr/lib/python`*`x.y`*). Você necessita ter o Python instalado para usar essa opção e o seu sistema necessita suportar shared libraries. Se ao invés de usar o Python você deseja montar um completamente novo interpretador binário, você deve fazer isso manualmente.

`--with-tcl`

Monta os componentes que requerem Tcl/Tk, que são os libpgtcl, pgtclsh, pgtksh, PgAccess, e PL/Tcl. Mas veja abaixo sobre `--without-tk`.

`--without-tk`

Se você especificar `--with-tcl` e essa opção, então os programas que requerem Tk (pgtksh e PgAccess) serão excluídos.

`--with-tclconfig=`*`DIRETORIO`*

`--with-tkconfig=`*`DIRETORIO`*

Tcl/Tk instala os arquivos `tclConfig.sh` e `tkConfig.sh`, os quais contém informações de configuração necessárias para montar os módulos de interface para o Tcl ou Tk. Esses arquivos são normalmente encontrados automáticamente nas suas conhecidas localizações, mas se você quiser usar uma versão diferente de Tcl ou Tk você pode especificar o diretório onde encontrá-los.

`--enable-odbc`

Monta o driver ODBC. Por default, ele será independente de um gerenciador de drivers. Para se trabalhar melhor com um gerenciador de drivers já instalado no seu sistema, use uma das próximas opções em conjunto com essa.

`--with-iodbc`

Monta o driver ODBC para uso com iODBC.

`--with-unixodbc`

Monta o driver ODBC para uso com unixODBC.

`--with-odbcinst=`*`DIRETORIO`*

Especifica o diretório onde o driver ODBC irá encontrar o seu arquivo de configuração : `odbcinst.ini`. O default é `/usr/local/pgsql/etc` ou outro qualquer que você especificou com `--sysconfdir`. Isso deveria estar arranjado de forma que o driver lê o mesmo arquivo que o gerenciador de drivers lê.

Se `--with-iodbc` ou `--with-unixodbc` são usados, essa oção será ignorada porque nesse caso o gerenciador de drivers cuida da localização do arquivos de configuração.

`--with-java`

Monta o driver JDBC e as packages associadas. Essa opção requere que o programa Ant também seja instalado (bem como a JDK, claro).

`--with-krb4[=`*`DIRETORIO`*`]`

`--with-krb5[=`*`DIRETORIO`*`]`

Monta o suporte para a autenticação Kerberos. Você pode usar tanto o Kerberos versão 4 ou 5, mas não ambos. O argumento *`DIRETORIO`* especifica o diretório root da instalação Kerberos; `/usr/athena` é assumido como default. Se os arquivos header importantes e as libraries não forem ser encontradas abaixo de um diretório pai em comum, então você deve usar as opções `--with-includes` e `--with-libraries` em conjunto com essa opção. Se, por outro lado, os arquivos requeridos se encontrarem em uma localização

que é varrida por default (exemplo, `/usr/lib`), então você pode esquecer o argumento. `configure` irá checar pelos arquivos header files e libraries para ter certeza de que a sua instalação Kerberos é suficiente antes de proceder.

`--with-krb-srvnam=`*`NOME`*

O nome do serviço principal do Kerberos. `postgres` é o default. Não há provávelmete razão para mudar isso.

`--with-openssl[=`*`DIRETORIO`*`]`

Monta o suporte para conexões SSL encriptadas. Isso requer o pacote OpenSSL instalado. O argumento *`DIRECTORIO`* especifica o diretório root da instalação do OpenSSL; o local default é `/usr/local/ssl`. `configure` irá checar pelos arquivos header files e libraries para ter certeza de que a sua instalação OpenSSL é suficiente antes de proceder.

`--with-pam`

Monta o suporte para PAM (Pluggable Authentication Modules).

`--enable-syslog`

Habilita o servidor PostgreSQL para usar o syslog. (O uso essa opção não significa que você tenha que se logar usando o syslog ou que isso será feito por default, isso simplesmente torna possível a opção de torná-lo on a qualquer hora.)

`--enable-debug`

Compila totdos os programas e libraries com os síbolos de debug. Isso que dizer que você pode rodar programas através de um debugger para analizar problemas. Isso faz crescer o tamanho dos executáveis considerávelmente, e para os compiladores não-GCC isso geralmente disabilita a otimização do compilador, causando travamentos e paradas no sistema. Entretanto, ter os símbolos disponível é extremamente bom quando surgir algum problema. Atualmente, essa opção é recomendada para instalaçoes que irão entrar em produção apenas se você usa GCC. Você deveria habilitar essa opção sempre que for fazer trabalho de desenvolvimento ou rodando uma versão beta do banco.

`--enable-depend`

Habilita a checagem automática de dependências. Com essa opção, os makefiles serão setados de modo que todos os objetos afetados irão ser recompilados quando qualquer arquivo header for modificado. Isso é muito útil quando você estiver fazendo algum desenvolvimento, mas isso causará um grande aumento de código se você pretende apenas compilar uma vez e instalar. Atualmente essa opção só funciona se você usar GCC. Se você prefere um compilador C ou C++ diferente dos que o `configure` usa, sete as variáveis de ambiente `CC` ou `CXX`, respectivamente, para o programa de sua escolha. Similarmente, você pode sobrescrever as variáveis flag default do compilador com as variáveis `CFLAGS` e `CXXFLAGS`. Por exemplo:

```
env CC=/opt/bin/gcc CFLAGS='-O2 -pipe' ./configure
```

2. Compilação

Para começar a compilação, digite

```
gmake
```

(Lembre de usar o GNU make.) A compilação pode levar de 5 minutos até meia-hora dependendo do seu hardware. A última linha mostrada deveria ser

```
All of PostgreSQL is successfully made. Ready to install.
```

3. Testes de Regressão

Se você quiser testar esse novo servidor compilado antes de instalá-lo, você pode rodar testes de regressão apartir de agora. Eles servem para verificar se o PostgreSQL rodará na sua máquina do jeito que os desenvolvedores planejaram. Digite

```
gmake check
```

(Isso não rodará se você estiver logado como root; faça isso como um usuário sem privilégios.) É possível que algum teste falhe, devido à diferenças de interpretação de mensagens de erro ou resultados de pontos flutuantes. Você pode repetir esse teste a qualquer hora mais tarde apenas repetindo esse comando.

4. Instalando os arquivos

Nota: Se você está fazendo upgrade de um sistema existente e está instalando os novos arquivos em cima dos anteriores, à esta altura, você já deveria ter feito um backup dos seus dados e dado um shutdown no servilor antigo, como já explicado na Seção anterior.

Para instalar o PostgreSQL digite

```
gmake install
```

Isso irá instalar os arquivos dentro dos diretórios onde foram especificados no passo 1. Tenha certeza de ter as permissões adequadas para gravar naquela área. Normalmente você precisa executar esse passo logado como root. Alternativamente, você poderia criar os diretórios destino antecipadamente e garantir que as permissões já estivessem sido dadas.

Se você compilar as interfaces do Perl ou do Python e você não foi o usuário root quando executou o comando acima, então parte da sua instalação provávelmente falhou. Nesse caso, logue como root e digite

```
gmake -C src/interfaces/perl5 install
```

```
gmake -C src/interfaces/python install
```

Se você não tem acesso como superusuário, então você está sozinho nessa: você ainda pode colocar os arquivos requeridos em outros diretórios onde o Perl ou o Python possam achá-los, mas como fazer isso é deixado apenas como um exercício.

A instalação standard provê apenas os arquivos header necessários para o desenvolvimento de aplicações clientes. Se você planeja em fazer algum desenvolvimento do lado do servidor (tipo funções customizadas ou tipos de dados escritos em C), então você deverá instalar toda a árvore include do PostgreSQL no seu diretório. Para fazer isso, digite

```
gmake install-all-headers
```

Isso irá gerar um ou dois megabytes a mais na sua instalação.

Instalação apenas do Cliente : se você quiser instalar apenas as aplicações clientes e as libraries de interface, use então os seguintes comandos :

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
```

```
gmake -C doc install
```

Para desfazer a instalação use o comando **gmake uninstall**. Entretanto, isso não irá remover qualquer diretório novo criado.

Após a instalação você pode abrir algum espaço removendo os arquivos compilados da árvore de fontes com o comando **gmake clean**. Isso irá preservar os arquivos feitos pelo programa configure, desse jeito você pode recompilar tudo novamente com **gmake** quando quiser. Para fazer voltar toda a árvore de fontes ao estado original em que ela foi distribuída, use **gmake distclean**. Se você for compilar o PostgreSQL para diversas plataformas usando a mesma árvore de fontes, você deve fazer sempre isso e reconfigurar para cada compilação.

Se você executar uma compilação e descobrir que suas opções estavam erradas, ou se você mudar alguma coisa que o configure investigue (por exemplo, você instala o GNU Readline), então é uma boa idéia rodar o **gmake distclean** antes de reconfigurar e recompilar. Sem isso, as suas mudanças de configuração podem não chegar onde elas devem.

Após a Instalação

Shared Libraries

Em alguns sistemas que têm shared libraries (a maioria deles têm) você deve dizer ao seu sistema como achar as shared libraries recentemente instaladas. Os sistems no qual isso *não* é necessário incluem BSD/OS, FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (conhecido antigamente como Digital UNIX), and Solaris.

O método para setar o path de pesquisa para as shared library varia entre as plataformas, mas o método mais usado é o setamento da variável de ambiente `LD_LIBRARY_PATH` mais ou menos assim:

Em Bourne shells (**sh**, **ksh**, **bash**, **zsh**)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib

export LD_LIBRARY_PATH
```

ou em **csh** ou **tcsh**

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Troque `/usr/local/pgsql/lib` com o valor que você setou em `--libdir` no passo 1. Você deveria colocar esses comandos dentro de um arquivo shell de startup tipo o `/etc/profile` ou o `~/.bash_profile`. Você pode obter mais alguma boa informação associada com esse método em http://www.visi.com/~barr/ldpath.html.

Em alguns sistemas é preferível que você sete a variável de ambiente `LD_RUN_PATH` *antes* da compilação. Em caso de dúvida, dê uma olhada no manual do seu sistema (talvez **ld.so** ou **rld**). Se mais tarde você receber uma mensagem tipo

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or
directory
```

então talvez esse passo tivesse sido necessário. Simplesmente faça-o antes de prosseguir.

Se você está usando BSD/OS, Linux, ou SunOS 4 e você tem acesso como root execute o seguinte comando

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(ou o equivalente diretório) após a instalação para habilitar o run-time linker e fazê-lo encontrar as shared libraries mais rápido.

Dê uma olhada na página **ldconfig** do manual para mais informação. No FreeBSD, NetBSD, e Open BSD o comando é

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Outros sistemas não são conhecidos de terem um comando equivalente.

Variáveis de Ambiente

Se você instalou o PostgreSQL dentro do diretório `/usr/local/pgsql` ou algum outro que não é procurado pelos programas por default, adicione então `/usr/local/pgsql/bin` (ou o que você setou em

`--bindir` no passo 1) dentro do seu `PATH`. Para fazer isso, adicione a seguinte linha no seu arquivo de startup shell, tipo o `~/.bash_profile` (ou o `/etc/profile`, se você quiser passar isso para todos os usuários):

```
PATH=/usr/local/pgsql/bin:$PATH
```

Se você estiver usando o **csh** ou o **tcsh**, use então esse comando:

```
set path = ( /usr/local/pgsql/bin $path )
```

Para liberar o seu sistema a encontrar a documentação man, adicione a seguinte linha no seu arquivo de startup shell:

```
MANPATH=/usr/local/pgsql/man:$MANPATH
```

As variávies de ambiente `PGHOST` e `PGPORT` especificam para as aplicações clientes o host e a porta do servidor de banco de dados, sobrescrevendo as defaults compiladas. Se você indo rodar aplicações clientes remotamente então é conveniente que cada usuário que planeja usar o banco de dados sete o seu `PGHOST`. Isso não é requerido, entretanto:

os setups podem ser comunicados via linha de comando para a maioria dos programas cliente.

Plataformas Suportadas

O PostgreSQL foi verificado pelos seus desenvolvedores trabalhando nas plataformas listadas abaixo. Uma plataforma suportada geralmente significa que o PostgreSQL compila e instala de acordo com essas instruções e passa nos testes de regressão.

Nota: Se você está tendo problemas com a instalação em uma plataforma suportada, por favor, escreva para <pgsql-bugs@postgresql.org> ou <pgsql-ports@postgresql.org>, e não para as pessoas listadas aqui.

Inicializando a área de dados

O banco de dados padrão não foi pré-inicializado, sendo necessário efetuar este procedimento manualmente com os seguintes passos :

Neste momento subentende-se que já exista um usuário postgres do grupo postgres.

1. Crie um diretório para armazenar seus dados :

   root# mkdir “path do diretório”

2. Altere as permissões do diretório:

   root# chown postgres.postgres “path do diretório”

3. Mude para o usuário “postgres”

   root# su postgres

4. Posicione-se no diretório onde o postgreSQL foi instalado:

   postgres# cd /usr/local/pgsql/bin

5. Inicialize o diretório para receber os dados

   ./initdb –d “path do diretório”

Inicializando o banco de dados

Utilizando o editor vi inclua no .bash_profile do usuário postgres a linha seguinte:

```
PATH=/usr/local/pgsql/bin:$PATH
```

Faça um novo login com o usuário postgres.

Após ter concluído com sucesso a instalação do PostgreSQL e da área de dados inicial, você deve inicializar o servidor digitando:

pg_ctl –D “path_do_diretório” start

ou

postmaster –D “path_do_diretório”

Resumo da Unidade

- O PostgreSQL é 100% Open Source.
- Existem empresas especializadas no suporte do PostgreSQL.

Revisão da Unidade

1. Onde conseguimos o código fonte do PostgreSQL ?

2. Quanto custa a versão mais atualizada do PostgreSQL ?

Laboratório 1:
Instalação e configuração em Linux

Acompanhe o instrutor na execução do laboratório.

Unidade 2:Autenticação de clientes

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Descrever a Autenticação de Clientes do PostgreSQL
- Mostrar o arquivo pg_hba.conf
- Métodos de autenticação
- Resolver alguns problemas de autenticação

Tópicos da Unidade

- Liberando acesso ao PostgreSQL
- Os métodos de autenticação

Autenticação de Clientes

Quando uma aplicação cliente conecta em um servidor de banco de dados, ela especifica com qual nome de usuário PostgreSQL ela quer se conectar, do mesmo jeito que uma outra aplicação cliente se loga em um computador UNIX como um usuário em particular. Dentro do ambiente SQL o nome do usuário ativo determina os privilégios de acesso aos objetos do banco de dados. Portanto, é óbvio que precisamos restringir com qual nome de usuário de banco de dados cada aplicação pode se conectar.

Autenticação é o processo no qual o servidor de banco de dados estabelece a identidade do cliente, e por consequência determina se a aplicação cliente (ou o usuário que está rodando essa aplicação) pode se conectar ou não no banco de dados com o nome de usuário que ela está usando. O PostgreSQL oferece diferentes métodos de autenticação de clientes. O método a ser utilizado pode ser selecionado com base de (client) host e banco de dados. Alguns métodos permitem que você restrinja o acesso pelo nome do usuário também.

Os nomes de usuários do banco de dados PostgreSQL estão logicamente separados dos nomes dos usuários do sistema operacional no qual o banco está rodando. Se todos os usuários de um banco de dados PostgreSQL também são usuários registrados no sistema operacional da máquina que está rodando o banco, então faz sentido nomear os usuários do PostgreSQL com os já registrados no sistema operacional. Entretanto, um servidor que aceite conexões remotas pode ter muitos usuários que não tem contas locais no sistema operacional, e nesses casos, não há necessidade de conexão entre o nome do usuário no banco e um nome de usuário no sistema operacional.

O arquivo pg_hba.conf

A autenticação de clientes é controlada pelo arquivo `pg_hba.conf` (hba – host-based authentication – quer dizer : autenticação baseada em host) que está localizado no diretório `/usr/local/pgsql/data/`. Um arquivo pg_hba.conf default é instalado quando a área de dados é inicializada pelo **initdb**.

O formato geral do pg_hba.conf é um set de registros, um por linha. Linhas em branco ou linhas começadas pelo caracter grade ("#") são ignoradas. Um registro é feito por um número de campos os quais são separados por espaços e/ou tabs. Os registros não podem ser escritos em mais de uma linha. Cada registro especifica um tipo de conexão, uma faixa de endereços IPS de clientes (se for importante para o tipo de conexão), o nome ou os nomes do(s) banco(s) de dado(s) e o método de autenticação que será usado para as conexões que validarem com esses parâmetros. O primeiro registro que validar com o tipo, o endereço do cliente e o nome do banco de dados requisitado para essa tentativa de conexão, é passado para o passo de autenticação. Não há novas tentativas de achar outro registro válido no pg_hba.conf caso um registro seja escolhido e a conexão falhe, os outros registros são desconsiderados. Se nenhum registro validar com os parâmetros, o acesso será negado.

Um registro pode ter um dos 3 formatos a seguir :

```
local banco-de-dados método-de-autenticação [opção-método-de-autenticação]


host banco-de-dados endereço-IP máscara-IP método-de-autenticação [opção-método-de-autenticação]


hostssl banco-de-dados endereço-IP máscara-IP método-de-autenticação [opção-método-de-autenticação]
```

O Significado dos campos está descrito a seguir :

local

Esse registro pertence às tentativas de conexão usando soquetes de domínio no ambiente Unix.

host

Esse registro pertence às tentativas de conexão nas redes TCP/IP. Observe que as conexões TCP/IP estão desabilitadas até que você inicialize o PostgreSQL_com a opção **–i**.

hostssl

Esse registro pertence às tentativas de conexão nas redes TCP/IP usando o SSL. Para fazer uso dessa opção, o servidor deve estar com o suporte ao SSL ativo. O SSL pode ser ativado com a opção **–l**.

Observação: os registros de **host** podem validar tanto com tentativas de conexão SSL ou não-SSL, mas registros **hostssl** só validam com conexões SSL.

banco-de-dados

Especifica o banco de dados a que o registro se refere. O valor **all** especifica que o registro se aplica a todos os bancos de dados, enquanto o valor **sameuser** identifica o banco de dados com o mesmo nome que o usuário está usando para se conectar. Caso seja usado outro nome, será o de um banco de dados PostgreSQL específico.

endereço-IP
máscara-IP

Esses dois campos especificam para qual máquina cliente o registro se refere baseado no endereço IP. A lógica é : **(endereço-IP-atual *xor* campo-endereço-IP) e máscara-IP** tem que ser **ZERO** para que o registro valide.

método-de-autenticação

Especifica o método que o usuário deve usar para se autenticar. As formas de autenticação serão apenas descritas agora, e serão vistas em detalhe mais adiante.

trust

A conexão é permitida incondicionalmente. Esse método permite que qualquer usuário que tenha acesso de login no host do cliente possa conectar como qualquer usuário PostgreSQL.

reject

A conexão é rejeitada incondicionalmente. Esse método é útil para "retirar filtrando" alguns hosts de um grupo.

password

O cliente é requisitado a fornecer uma senha, a qual é checada no banco de dados e deve bater com a senha do usuário no banco. As senhas são enviadas para o servidor em texto puro.

Um nome de arquivo opcional pode ser especificado após a palavra **password**.

Esse arquivo deve conter uma lista de usuários que podem se conectar usando esse registro, e pode conter opcionalmente senhas para eles. Para uma melhor proteção, use os métodos **md5** ou **crypt**.

md5

Funciona como o método **password**, mas envia as senhas encriptadas usando um protocolo simples de encriptação. Isso protege a autenticação de programas rastreadores de conexões (wire-sniffing). Esse método é o recomendado para autenticações baseadas em senhas. O nome do arquivo pode ser acrescentado após a palavra chave **md5**. Esse arquivo deve conter uma lista de usuários que podem se conectar usando esse registro.

crypt

Funciona como o método **md5**, mas usa um antigo protocolo de encriptação, o qual é necessário para clientes pré versão 7.2. O método **md5** é preferido para clientes com versão 7.2 ou maior. O método **crypt** não é compatível com senhas encriptadas no **pg_shadow**, e pode falhar se o cliente e o servidor tiverem diferentes implementações da rotina de library **crypt()**.

krb4

Kerberos V4 é usado para autenticar o usuário. Ele é apenas disponível para conexões TCP/IP.

Krb5

Kerberos V5 é usado para autenticar o usuário. Ele é apenas disponível para conexões TCP/IP.

ident

A identidade do usuário como determinada no login do sistema operacional é usada pelo PostgreSQL para determinar se o usuário tem permissão de conexão. Para conexões TCP/IP, a identidade do usuário é determinada através do contato ao servidor ***ident*** no host do cliente. (Note que isso é tão confiável quanto um servidor ***ident*** remoto; a autenticação ***ident*** nunca deveria ser usada por hosts remotos os quais seus administradores não são confiáveis). Em sistemas operacionais que suportam requisições **so_peercred** para soquetes de domínio UNIX, a autenticação ***ident*** é possível para conexões locais. O sistema operacional é então perguntado sobre a identidade do usuário. Em sistemas operacionais que não recebem requisições **so_peercred**, a autenticação ***ident*** é possível para conexões TPC/IP. Como alternativa, é possível especificar o endereço do localhost como 127.0.0.1 e fazer conexões nesse endereço. A opção de autenticação que segue a palavra chave ***ident*** especifica o nome de um ***mapa de identidades*** o qual especifica que usuários do sistema operacional casam com qual usuários do banco de dados. Veja os detalhes abaixo :

pam

Esse tipo de autenticação opera de maneira similar ao método **password**, com a maior diferença em que esse método usará o PAM (Módulos Plugáveis de Autenticação) como mecanismo de autenticação. A ***opção-de-autenticação*** após a palavra chave ***pam*** especifica o nome do serviço que será passado ao PAM. O nome default do serviço é **postgresql**.

Para maiores informações sobre o método PAM, leia : as páginas do manual Linux-PAM (http://www.kernel.org/pub/linux/libs/pam/) e/ou as páginas do manual Solaris-PAM (http://www.sun.com/software/solaris/pam/).

opção-de-autenticação

Esse campo é interpretado diferentemente dependendo do método de autenticação, como descrito acima.

Como os registros do pg_hba.conf são examinados sequencialmente para cada tentativa de conexão, a ordem dos registros é muito importante. Tipicamente, os primeiros registros terão um casamento de parâmetros de conexão forte e fracos métodos de autenticação, enquanto os últimos registros terão um fraco casamento de parâmetros de conexão e fortes métodos de autenticação. Por examplo, alguém pode querer usar o método de autenticação trust para conexões TCP locais mas requerer uma senha para conexões TCP remotas. Nesse caso um registro especificando o método de autenticação trust para conexões de 127.0.0.1 deveria aparecer antes de um registro especificando autenticação com senha para uma gama maior de endereços IP de clientes permitidos.

O arquivo pg_hba.conf é lido no startup e quando o postmaster recebe um sinal SIGHUP. Se você editar o arquivo com o sistema ativo, você terá que sinalizar o postmaster (usando pg_ctl reload or kill -HUP) para fazer o postmaster reler o arquivo.

Veja a seguir um exemplo de um arquivo pg_hba.conf :

Exemplo de arquivo `pg_hba.conf`

```
# TYPE   DATABASE   IP_ADDRESS           MASK
AUTHTYPE     MAP

# Allow any user on the local system to connect to any
# database under any username, but only via an IP connection:

  host   all         127.0.0.1       255.255.255.255    trust

# The same, over Unix-socket connections:

  local  all                                            trust

# Allow any user from any host with IP address 192.168.93.x
to
# connect to database "template1" as the same username that
ident on that
# host identifies him as (typically his Unix username):

  host   template1   192.168.93.0    255.255.255.0      ident
sameuser

# Allow a user from host 192.168.12.10 to connect to database
"template1"
# if the user's password in pg_shadow is correctly supplied:

  host   template1   192.168.12.10   255.255.255.255    md5

# In the absence of preceding "host" lines, these two lines
will reject
# all connection attempts from 192.168.54.1 (since that entry
will be
```

```
# matched first), but allow Kerberos V5-validated connections
from anywhere
# else on the Internet. The zero mask means that no bits of
the host IP
# address are considered, so it matches any host:

  host   all         192.168.54.1    255.255.255.255
reject
  host   all         0.0.0.0         0.0.0.0            krb5

# Allow users from 192.168.x.x hosts to connect to any
database, if they
# pass the ident check. If, for example, ident says the user
is "bryanh"
# and he requests to connect as PostgreSQL user "guest1", the
connection
# is allowed if there is an entry in pg_ident.conf for map
"omicron" that
# says "bryanh" is allowed to connect as "guest1":

  host   all         192.168.0.0     255.255.0.0        ident
omicron

# If these are the only two lines for local connections, they
will allow
# local users to connect only to their own databases
(database named the
# same as the user name), except for administrators who may
connect to
# all databases. The file $PGDATA/admins lists the user names
who are
# permitted to connect to all databases. Passwords are
required in all
# cases. (If you prefer to use ident authorization, an ident
map can
# serve a parallel purpose to the password list file used
here.)

  local  sameuser                                          md5
```

```
  local  all                                                md5
admins
```

Métodos de autenticação

A seguir descreveremos os métodos de autenticação em maiores detalhes.

Autenticação Trust

Quando o método de autenticação trust é especificado, o PostgreSQL assume que qualquer um que pode se conectar ao postmaster está autorizado a acessar o banco de dados assim como qualquer base que o usuário especificar (inclusive a base do superusuário). Esse método deveria ser usado apenas quando há um sistema adequado de proteção para as conexões na porta do postmaster. O método de autenticação trust é apropriado e muito conveniente para conexões locais e com apenas um usuário. Ela geralmente *não* é apropriada em uma máquina multiusuário. Entretanto, você pode usar o método trust até com máquinas multiusuário, isso se você restringir o accesso ao soquete do postmaster usando um sistema de permissões baseado no sistema operacional.

Para fazer isso, use o parâmetetro unix_socket_permissions (e possivelmente unix_socket_group) no postgresql.conf. Ou você pode setar unix_socket_directory e colocar o arquivo de soquete em um diretório restrito.

Conexões locais TCP não estão restritas às permissões baseadas no sistema operacional; portanto, se você quer usar permissões para segurança local, remova a linha host ... 127.0.0.1 ... do pg_hba.conf, ou use outro método de aytenticação que não seja trust.

Autenticação Password

Métodos de autenticação baseados em senhas incluem md5, crypt, and password. Esses métodos operam de maneira similar exceto pelo jeito em que a senha é enviada através da conexão. Se você está muito preocupado com ataques "sniffing" de senhas, então seu método preferido será o md5, o crypt será sua segunda escolhas, isso se você tiver que manter suporte à clientes obsoletos. O método puramente password deve ser evitado para as conexões diretas vindas da Internet (a menos que você use SSL, SSH, ou outro tipo de segurança na conexão).

As senhas do PostgreSQL são separadas das do sistema operacional. Ordinariamente, a senha para cada usuário do banco de dados é gravada na tabela do banco pg_shadow. As senhas podem ser manipuladas com os comandos da linguagem plpgsql **CREATE USER** e **ALTER USER**, por exemplo., **CREATE USER teste**

WITH PASSWORD 'senhasecreta';. Por default, se nenhuma senha foi ainda setada, a senha gravada para o usuário estará NULL e a autenticação tipo password irá sempre falhar para aquele usuário.

Para restringir um grupo de usuários que tem permissão para se conectar em certos bancos

de dados, coloque o grupo em um arquivo separado (um usuário por linha) no mesmo diretorrio do pg_hba.conf, e mecione o nome do arquivo após a palavra chave password, md5, ou crypt, respectivamente, no pg_hba.conf. Se você não usar essa característica, então qualquer usuário que é conhecido do sistema poderá se conectar a qualquer banco de dados (contanto que ele informe a senha correta, é claro).

Esses arquivos podem também serem usador para aplicar um grupo diferente de senhas à um banco de dados em particular. Nesses casos, os arquivos tem um formato similar ao arquivo standard de senhas do Unix, o /etc/passwd, isso é, *username*:*password*

Qualquer outro campo separado por : (dois pontos) após a senha será ignorado. É esperado que a senha esteja encriptada usando a a função do sistema crypt(). O utilitário pg_passwd que é instalado com o PostgreSQL pode ser usado para gerenciar esses arquivos de senhas.

Linhas com e sem senhas podem ser misturadas em arquivos secundários. Linhas sem senha indicam o uso de senha principal da tabela pg_shadow que é gerenciada por **CREATE USER** e **ALTER USER**. Linhas com senhas indicam o uso da senha do arquivo de senhas. Uma senha com "+" também significa o uso da tabela pg_shadow.

Senhas alternativas não podem ser usadas quando os métodos md5 ou crypt forem usados. O arquivo será lido normalmente, mas o campo da senha será ignorado e a senha da tabela pg_shadow será sempre usada.

Note que o uso de senhas alternativas faz com que a manutenção de senhas via **ALTER USER** perca o efeito. O comando alterará a senha da tabela pg_shadow mas essa não será a senha que o sistema acabará validando e usando.

Autenticação Kerberos

Kerberos é um sistema standard de segurança e autenticação apropriado para a computação de redes distribuidas e públicas. A descrição do sistema Kerberos está além do escopo desse curso; descrição essa geralmente bem complexa e poderosa. O Kerberos FAQ (http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html) ou MIT Project Athena (ftp://athena-dist.mit.edu") pode ser um bom ponto de começo para exploração. Existem muitas fontes de distribuição Kerberos.

Para você fazer uso do Kerberos, você deve habilitar seu uso na hora de configurar o executável do PostgreSQL. Ambos Kerberos 4 e 5 são suportados (`./configure --with-krb4` ou `./configure --with-krb5` respectivamente), embora apenas uma única versão pode ser suportada para cada executável do PostgreSQL.

O PostgreSQL opera como um serviço Kerberos normal. O nome do serviço principal é *`servicename/`*

`hostname@dominio`, aonde *`servicename`* é `postgres` (a menos que um nome diferente de serviço foi selecionado na hora da configuração com `./configure --with-krb-srvnam=qualquer_nome`). *`hostname`* é o nome de domínio qualificado e completo da máquina servidora. O domínio do serviço é o mesmo do servidor.

Os clientes do PostgreSQL devem usar seus nomes de usuário do banco de dados como o primeiro componente, por exemplo *nomedousuariopg/outrascoisas@dominio*. Nesse momento o domínio do cliente não é checado pelo PostgreSQL; se você tem o método de autenticação cross-realm disponível, então qualquer cliente com qualquer domínio que possa se comunicar com o seu será aceito.

Tenha certeza que o arquivo chave do seu servidor está disponível para leitura (de preferencia apenas para leitura) pelo servidor PostgreSQL. A localização do arquivo chave é especificada com o arquivo de configuração run time `krb_server_keyfile`.

O default é `/etc/srvtab` se você está usando o Kerberos 4 e `FILE:/usr/local/pgsql/etc/krb5.keytab` (ou qualquer outro diretório que foi especificado como `sysconfdir` na hora da configuração do executável) com Kerberos 5.

Para gerar o arquivo keytab, use por examplo (com a versão 5)

kadmin% **ank -randkey postgres/server.meu.dominio.org**

kadmin% **ktadd -k krb5.keytab postgres/server.meu.dominio.org**

Leia a documentação do Kerberos para mais detalhes.

Quando for conectar à um banco de dados tenha certeza de ter um ticket para casar com o nome de usuário do banco de dados requisitado. Um examplo: Para o usuário do banco chamado `fred`, ambos `fred@EXEMPLO.COM` e `fred/usuario.examplo.com@EXAMPLO.COM` podem ser usados para autenticar com o servidor do banco de dados.

Se você usa mod_auth_krb e mod_perl no seu servidor web Apache, você pode usar `AuthType KerberosV5SaveCredentials` junto com um script mod_perl. Isso cria um acesso seguro através da web sem senhas extras requeridas.

Autenticação Ident-based

Virtualmente cada sistema operacional tipo Unix vêm com um servidor de identificação que escuta TCP na porta 113 por default. A funcionalidade básica de um servidor de identificação é responder à questões como "Que usuário iniciou uma conexão que sai pela porta *X* e se conecta na minha porta *Y*?". Desde de que o PostgreSQL conheça ambas portas *X* e *Y* quando uma conecão física é estabelecida, ele pode interrogar o servidor de identificação no host da conexão cliente e poderia teóricamente determinar o sistema operacional do cliente desse jeito.

O problema desse tipo de procedimento é que ele depende da integridade do cliente: um hacker poderia rodar um programa na porta 113 e retornar qualquer nome desejado. Esse método de autenticação é apenas apropriado em redes fechadas onde cada cliente estivesse sobre um rigoroso controle de segurança. Em outras palavras, você deve confiar na

máquina que está tentando se autenticar. Preste bastante atenção no aviso:

RFC 1413 The Identification Protocol is not intended as an authorization or access control protocol.

Em sistemas que suportam requisições `SO_PEERCRED` para soquetes Unix, o método de autenticação ident também pode ser aplicado para conexões locais. Nesse caso, nenhum risco é adicionado ao usar o método ident de autenticação; essa é a escolha preferida para conexões locais em um sistema como esse.

Após ter determinado o nome do sistema operacional do usuário que iniciou a conexão, o PostgreSQL checa se o usuário está autorizado a conectar no banco de dados que ele solicitou. Isso é controlado pelo argumento que segue a palavra chave `ident` no arquivo `pg_hba.conf`. Há um mapa ident predefinido `sameuser`, que permite que qualquer usuário de qualquer sistema operacional se conecte ao banco de dados com mesmo nome (se o último existir). Outros mapas podem ser criados manualmente.

Outros mapas ident diferentes de `sameuser` são definidos no arquivo `pg_ident.conf` no diretório de dados, os quais contém geralmente linhas desse formato:

nome-do-mapa usuario-ident usuario-do-banco-de-dados

Comentários e espaços em branco são tratados de maneira usual. O *`nome-do-mapa`* é o nome que será usado para referência à esse mapeamento no `pg_hba.conf`. Os outros dois campos especificam qual sistema operacional o usuário é permitido se conectar assim como com qual nome de usuário do banco de dados ele está tentando se conectar. O mesmo *`nome-do-mapa`* pode ser usado repetidamente para especificar mais mapeamentos de usuários dentro de um único mapa. Não há restrições quanto ao número de usuários de banco de dados de um dado sistema operacional pode corresponder à um usuário e vice versa.

O arquivo `pg_ident.conf` é lido no startup e quando o postmaster recebe um sinal SIGHUP. Se você editar o arquivo com o sistema ativo, você terá que sinalizar o postmaster (usando `pg_ctl reload` or `kill -HUP`) para fazer o postmaster reler o arquivo.

Um arquivo `pg_ident.conf` éque poderia ser usado em conjunto com o arquivo `pg_hba.conf` fdo exemplo acima é mostrado no exemplo abaixo. Nesse examplo, qualquer um logado em uma máquina com rede IP 192.168 que não tenha seu nome no Unix bryanh, ann, ou robert não poderiam ter seu acesso permitido. O usuário Unix robert somente poderia ter seu acesso permitido quando ele tentasse se conectar no PostgreSQL como usuário bob, não como roberto nem com outro nome qualquer. ann teria seu acesso permitido quando ela se conectasse como ann. O usuário bryanh teria permissão para se conectar tanto como bryanh ou como guest.

Um examplo de arquivo `pg_ident.conf`

```
#  MAP       IDENT-NAME    POSTGRESQL-NAME

   omicron   bryanh        bryanh
   omicron   ann           ann

# bob has username robert on these machines

   omicron   robert        bob

# bryanh can also connect as guest

   omicron   bryanh        guest
```

Problemas de autenticação

Falhas genuínas de autenticação e problemas relationados geralmente se manifestam através de mensagens de error como essa :

No pg_hba.conf entry for host 123.123.123.123, user joeblow, database testdb

Esse tipo de mensagem é o que provavelmente você irá receber se sua conexão teve sucesso contactando o servidor, mas ele não quer conversar com você. Como a mensagem sugere, o servidor recusou a requisição de conexão porquê não encontrou nehuma forma de autorização no arquivo de configuração `pg_hba.conf`.

Password authentication failed for user 'joeblow'

Mensagens como essa indicam que o servidor foi contatado, e ele está pronto para atender, mas não até você passar o método específico no arquivo `pg_hba.conf`. Cheque a senha que você está passando, ou cheque o seu Kerberos ou até o software ident se as mensagens fizerem menção à um desses tipos de autenticação.

FATAL 1: user "joeblow" does not exist

O nome indicado do usuário não foi encontrado.

FATAL 1: Database "testdb" does not exist in the system catalog.

O banco de dados que você está tentando se conectar não existe. Note que se você não especificar o nome do banco de dados, o PostgreSQL usa o nome do usuário como banco de dados deefault, o qual pode ou não pode ser a coisa certa.

Observe que o log do servidor pode conter mais informações sobre as falhas de autenticação do que o reportado ao cliente. Se você ficar confuso sobre a razão da falha, cheque o log.

Resumo da Unidade

- O arquivo pg_hba.conf é o responsável pela liberação de acesso ao banco de dados.
- Alterações nas permissões de acesso não necessitam de um re-start do banco de dados.

Revisão da Unidade

1. Qual parâmetro de inicialização libera o acesso externo ao PostgreSQL ?

2. Que tipo de validação não solicita senha de conexão ?

Laboratório 2:
Autenticação de clientes

Acompanhe o instrutor na execução do laboratório.

Unidade 3:
Gerenciando bancos de dados

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Criar um Banco de Dados no PostgreSQL
- Usar Templates de Bancos de Dados
- Utilizar diferentes localizações para os Bancos de Dados
- Destruir um Banco de Dados no PostgreSQL

Tópicos da Unidade

- Criação de um banco de dados
- Uso de templates
- Removendo um banco de dados

Gerenciando Bancos de Dados

Um banco de dados é uma chamada coleção de objetos SQL (“objetos do banco de dados”). Geralmente, cada objeto do banco de dados (tabelas, funções, etc.) pertencem a um e apenas um banco de dados. (Mas existem uns poucos catálogos de sistema, por exemplo `pg_database`, que pertence à toda uma instalação e estão acessíveis a cada banco de dados que fazem parte dessa instalação.) Uma aplicação que se conecta no servidor de banco de dados especifica na sua requisição de conexão o nome do banco de dados que ela quer se conectar. Não é possível acessar mais de um banco de dados por conexão. (Mas uma aplicação não está restrita ao número de conexões que ela abre para o mesmo ou até outro banco de dados.)

Nota: O SQL chama os bancos de dados de “catálogos”, mas não há diferença na prática.

Para se criar ou dropar um banco de dados, o postmaster do PostgreSQL deve estar rodando.

Criando um Banco de Dados

Os bancos de dados são criados com o comando sql **CREATE DATABASE**:

CREATE DATABASE *nome*

onde *name* segue as regras usuais de identificadores SQL. O usuário corrente automaticamente se torna owner do novo banco de dados. São os privilégios do owner de um banco de dados que o deixam removê-lo mais tarde quando desejado (o qual também remove todos os objetos que estão dentro desse banco de dados, mesmo que eles tenham um owner diferente).

A criação de bancos de dados é uma operação restrita. Veja mais tarde como obter permissão para isso (grants).

Dica : Você precisa estar conectado à um servidor de banco de dados para poder executar o comando **CREATE DATABASE**, a questão permanece em como o *primeiro* banco de dados de um dado site pode ser criado. O primeiro banco de dados é sempre criado pelo comando **initdb** quando a área de storage é inicializada. Por convenção esse banco é chamado de `template1`. Então para criar o primeiro banco de dados "real" você pode se conectar no banco `template1`.

O nome "template1" não é acidente: Quando um novo banco de dados é criado, o banco de dados template é essencialmente clonado. Isso significa que qualquer mudanças que você fizer no `template1` serão propagadas para todos os banco de dados subsequentemente criados. Isso implica que você não deveria usar o banco de dados template para produção, mas quando isso for feito com bom senso essa característica pode ser até conveniente. Como um extra, há também um programa que você pode execute apartir do shell para criar novos bancos, `createdb`.

createdb *nome*
O programa **createdb** não é mágico. Ele conecta no banco template1 e executa o comando **CREATE DATABASE**, exatamente como descrito acima. Ele usa o programa psql internamente. Note que o programa **createdb** sem nenhum argumento irá criar um banco de dados com o nome do usuário corrente, o que pode não ser exatamente o que você deseja.

Banco de Dados Template

O que o comando **CREATE DATABASE** faz realmente é copiar um banco de dados já existente. Por default, ele copia o banco de dados padrão do sistema chamado `template1`. Ele serve de "template" no qual os novos bancos são criados. Se você adicionar objetos no `template1`, eles serão copiados também para os próximos bancos criados. Esse comportamento permite modificações in-site no modelo de objetos standard dos bancos de dados. Por exemplo, se você instalat a linguagem procedural `plpgsql` no `template1`, ela estará automaticamente disponível nos bancos de dados dos usuários sem qualquer ação extra apartir do momente de criação deles.

Há também um segundo sistema standard de banco de dados chamado `template0`. Esse banco de dados contém o mesmo conteúdo do `template1`, isto é, apenas os objetos standard predefinidos pela sua versão do PostgreSQL. `template0` nunca deveria sofrer mudanças após o `initdb`. Se você rodar o comando **CREATE DATABASE** para copiar o `template0` ao invés do `template1`, yvocê pode criar um banco de dados "virgem" que não contém nenhuma particularidade já instalada e/ou modificada do `template1`. Isso é particularmente útil quando você precisar dar um restore em um `pg_dump` dump: o script de dump deveria sempre ser restaurado em um banco de dados virgem para garantir a correta recriação dos conteúdos do banco de dados dump, sem qualquer conflito com as modificações presentes no `template1`.

É também possível criar templates de bancos de dados adicionais, e é claro que você pode copiar qualquer banco de dados em uma instalação apenas especificando seu nome como template no comando **CREATE DATABASE**. O que é importante entender, entretanto, que isso não é (ainda) por objetivo um "GERADOR GENÉRICO" de cópias de bancos de dados. Em particular, é também essencial que o banco de dados modelo esteja idle (onde não hajam transações em progresso) enquanto a cópia é feita. O comando **CREATE DATABASE** irá checar para que não existam processos em background (outros que ele mesmo) conectados no banco de dados que está sendo copiado no início da operação, isso não garante que mudanças não possam ser feitas enquanto a cópia está em andamento, o qual poderia resultar em um banco de dados inconsistente.

Nós recomendamos portanto que os bancos de dados usados como templates sejam tratados apenas como read-only. Dois flags muito úteis existem no `pg_database` para cada banco de dados: `datistemplate` e `datallowconn`.`datistemplate` pode ser setado para indicr que determinado banco de dados é apenas usado como template para o **CREATE DATABASE**. Se esse flag está setado, o banco de dados pode ser clonado por qualquer usuário com privilégio `CREATEDB`; se não está setado, apenas os superusuários e o owner do banco poderá cloná-lo. Se `datallowconn` é false, então nenhuma nova conexão para aquele banco de dados será permitida (mas as sessões que estiverem ativas não serão desativadas apenas pelo flag ter sido setado como false). O banco de dados `template0` é normalmente marcado como `datallowconn = false` para prevenir modificações nele. Ambos os bancos de dados `template0` e `template1` deveriam sempre ficarem marcados como `datistemplate = true`.

Após quaisquer mudanças ou preparações nos templates, seria uma boa idéia executar um **VACUUM FREEZE** ou **VACUUM FULL FREEZE** neles. Se isso for executado quando não houverem transações sendo executadas no banco de dados, então fica garantido que todas as tuplas do banco de dados estão "congeladas" e que elas não estarão sujeitas a problemas de ID wraparound. Isso é particularmente importante para um banco de dados que terá o flag `datallowconn` setado como false, tornando manutenções de **VACUUM** rotineiras nesses bancos de dados impossível.

Nota : `template1` e `template0` não tem nenhum status especial além do fato de que o nome `template1` é o banco de dados fonte para o comando **CREATE DATABASE** e é o banco da dados default de conexão de vários scripts tipo o `createdb`. Por exemplo, alguém poderia dropar o `template1` e recriá-lo apartir do `template0` sem qualquer problema. É claro que qualquer nova funcionalidade do `template1` seria perdida. Isso poderia ser feito também no caso de alguém ter adicionado um monte de lixo dentro do `template1`.

Diferentes Localizações para os Bancos de Dados

É possível criar bancos de dados em um local diferente da localização default da instalação. Lembre que qualquer acesso aos bancos de dados ocorrem através do servidor de banco de dados, portanto qualquer localização nova deve estar acessível para o servidor.

Localizações alternativas para os bancos de dados são referenciados em uma variável de ambiente que guarda o caminho absoluto para o armazenamento dos bancos de dados. Essa variável de ambiente deve estar presente no ambiente do servidor, o que significa de ela deve ser definida antes do servidor ser startado. (o grupo de variáveis de localização alternativa são controladas pelos administradores do site, usuários normais não podem mexer nessas variáveis). Qualquer nome válido para variáveis de ambiente podem ser usadas sem problema, nossa recomendação é que você use o prefixo `PGDATA` para evitar confusão e conflito com outras variáveis de ambiente.

Para a criação de uma variável de ambiente para um servidor que estiver ativo, você primeiro deve dar um shutdown nele, definir a variável, inicializar a área de dados e finalmente restartar o servidor.

Para setar uma variável de ambiente, digite :

PGDATA2=/home/postgres/data

export PGDATA2

no Bourne shell, ou

setenv PGDATA2 /home/postgres/data

no csh ou tcsh.

Você deve ter certeza que essa variável de ambiente seja sempre definida no ambiente do servidor, caso contrário, você não poderá acessar o banco de dados. Tente sempre setar as suas variáveis de ambiente em algum tipo de start-up shell ou start-up script no servidor.

Para criar uma área de armazenamento em `PGDATA2`, tenha certeza de que o diretório (nesse caso, `/home/postgres`) já exista e possa ser gravada pela conta do usuário que roda o servidor.

Então apartir da linha de comando, digite :

initlocation PGDATA2

Agora você pode restartar o servidor.

Para criar um banco de dados dentro de uma nova localização, use o comando

CREATE DATABASE *nome* WITH LOCATION = '*localização*'

Onde `localização` é a variável de ambiente que você usou, `PGDATA2` nesse exemplo.

O comando **createdb** tem a opção `-D` para esse propósito.

Os bancos de dados criados em locais alternativos podem ser acessados e dropados como qualquer outro banco de dados.

Nota : Também é possível especificar localizações absolutas diretamente para o comando **CREATE DATABASE** sem a necessidade de definirmos variáveis de ambiente. Isso é disabilitado por default pois é um risco à segurança. Para permitir isso, você deve compilar o PostgreSQL com a macro do preprocessador C `ALLOW_ABSOLUTE_DBPATHS` definida. Na hora da compilação use :

gmake CPPFLAGS=-DALLOW_ABSOLUTE_DBPATHS all

Destruindo Bancos de Dados

Bancos de Dados são destruídos com o comando **DROP DATABASE**:

DROP DATABASE *nome*

Apenas o owner do banco (isto é.,o usuário que o criou), ou um superusuário, pode dropar um banco de dados. O ato de dropar um banco de dados implica na remoção de todos os objetos que estavam contidos dentro daquele banco. A destruição de um banco de dados não pode ser desfeita.

Você não pode executar o comando **DROP DATABASE** estando conectado no próprio banco que será destruído. Você pode, entretanto, estar conectado em qualquer outro banco, inclusive o banco `template1`, o qual seria sua única opção ao dropar o último banco de dados de um determinado cluster.

Para sua conveniência, há também um programa shell que dropa bancos de dados:

dropdb *nome*

(Diferentemente do **createdb**, essa não é a ação default para se dropar o banco de dados do usuário corrente)

Schemas

Schemas podem ser entendidos como instâncias diferentes de um mesmo banco de dados. Utilizamos schemas para permitir que usuários diferentes, acessando schemas diferentes, possam ter acesso à objetos (tabelas, funções, etc...) que possuam o mesmo nome sem que haja o risco de conflito entre eles.

Razões para utilizar schemas:

• Permitir que usuários diferentes usem o mesmo banco de dados sem interferir uns com os outros

• Organizar objetos de um banco de dados em grupos lógicos de forma à torná-los mais organizados

• Aplicações desenvolvidas por terceiros podem ser direcionadas para schemas diferentes, prevenindo o conflito com nomes de objetos pré-existentes no banco de dados.

Manipulação de Schemas

Criando Schemas

Para criar um Schema utilizamos o comando CREATE SCHEMA. Veja o exemplo:

```
CREATE SCHEMA meu_schema;
```

Para criar ou acessar objetos dentro de um schema, utilizamos querys comuns, com a diferença que utilizamos o nome do schema e o nome da tabela separados por um ponto:

```
CREATE TABLE meu_schema.usuarios
(
   definição das colunas e características da tabela
);
```

O schema public

O schema public é o schema default de qualquer banco de dados Postgresql. Ele é criado automaticamente quando um banco é criado e é assumido como o schema padrão para qualquer query que não explicite o nome do schema, ou seja a query:

```
SELECT * FROM usuarios;
```

é equivalente à:

```
SELECT * FROM public.usuarios;
```

Destruindo Schemas

Para destruir um schema deve-se utilizar o comando DROP SCHEMA. Note que para destruir um schema que ainda contenha objetos, destruindo-os automaticamente deve-se utilizar o modificador CASCADE:

```
DROP SCHEMA meu_schema CASCADE;
```

Resumo da Unidade

- O default na criação de um novo banco de dados é usar o template1 como exemplo.
- Para usar um local diferente para criação de um banco de dados devemos utilizar o “initlocation”.
- Não conseguiremos remover um banco de dados se o mesmo estiver em uso por algum usuário.

Revisão da Unidade

1. Qual a finalidade do template0 ?

2. Quais usuários podem criar banco de dados ?

3. Quando o comando “DROP DATABASE” não funcionará ?

Laboratório 3:
Gerenciando bancos de dados

Acompanhe o instrutor na execução do laboratório.

Unidade 4:
Gerenciando usuários e permissões

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Criar usuários
- Definir atributos de usuários
- Criar grupos
- Atribuir Permissões
- Ter visão rápida sobre Funções e Triggers

Tópicos da Unidade

- Criando usuários
- Fornecendo direitos aos usuários
- Criando grupos de acesso

Gerenciando Usuários e Permissões

O gerenciamento de usuários dos bancos de dados e seus privilégios tem um conceito similar ao gerenciamento de usuários de um sistema operacional Unix por exemplo, mas os detalhes não são idênticos.

Usuários de Bancos de Dados

Os usuários de bancos de dados estão conceitualmente e completamente separados dos usuários do sistema operacional. Em prática isso pode parecer conveniente mantêr uma certa correspondência, mas isso não é requerido. Os nomes dos usuários de bancos de dados são globais através de uma instalação de um cluster de bancos de dados (e não apenas de um banco só).

Para criar um usuário use o comando SQL **CREATE USER** :

CREATE USER *nome*

nome segue as regras dos identificadores SQL identifiers: tanto escrito sem qualquer adorno e caracteres especiais, quanto entre aspas duplas.

Para remover um usuário use o comando SQL análogo **DROP USER**.

Para sua conveniência, os scripts shell `createuser` e `dropuser` são providenciados como auxiliares desses comandos SQL.

Em ordem de poder startar pela primeira vez, um recém inicializado banco de dados já contém um usuário prédefinedo.

Esse usuário terá o id 1 fixo, e por default (a menos que alterado quando for rodar o **initdb**) ele terá o mesmo nome de usuário do sistema operacional que inicializou a área (e está presumidamente sendo usado como o usuário que roda o servidor). Geralmente, esse usuário será chamado de postgres. Para você criar novos usuários é necessário estar conectado como esse usuário inicial.

O nome de usuário para uma conexão à um banco de dados em particular é indicado pelo cliente que está inicializando a requisição de conexão ao banco de dados. Por exemplo, o programa **psql** usa a opção de linha de comando `-U` para indicar como o usuário vai se conectar. O nome do grupo de usuários que um dado cliente vai usar para se conectar é determinado pelo setup de autenticação de clientes, como explicado no captítulo 1 - Autenticação de Clientes. (Portanto, um cliente não está necessariamente limitado a se conectar com o seu nome de sistema operacional, do mesmo jeito que uma pessoa não é obrigada a efetuar seu login usando seu nome verdadeiro.)

Atributos de Usuários

Um usuário de banco de dados pode ter um número de atributos que definem seus privilégios e interagem com o sistema de autenticação de clientes.

Superusuário

Um superusuário de banco de dados ignora todas as regras de permissões e ele também é o único que pode criar novos usuários.

Para criar um superusuário, use o comando `CREATE USER nome CREATEUSER`.

Para se revogar o direito desse usuário em criar novos usuários use o comando :

ALTER USER nome NOCREATEUSER.

Criação de bancos de dados

Para um banco de dados ser criado, o usuário que o está tentando criar deve ter recebido explicitamente permissão para criar bancos (exceto pelos superusuários, já que ele ignora todas as regras de permissões).

Para criar um usuário que possa criar bancos de dados, use o comando

CREATE USER nome CREATEDB.

Para se revogar o direito desse usuário em criar novos bancos de dados use o comando :

ALTER USER nome NOCREATEDB.

Password

A senha só tem importância se o método de autenticação password for usado. As senhas de um banco de dados são separadas das senhas do sistema operacional.

Especifique uma senha para o uuário na hora de sua criação com o comando

CREATE USER nome PASSWORD 'senha'.

Para se alterar a senha desse usuário use o comando :

ALTER USER nome PASSWORD 'nova_senha'.

Grupos de Usuários

Como no Unix, grupos são um jeito de se agrupar logicamente usuários para facilitar o gerenciamento de permissões.

As permissões podem ser atribuidas para (granted) ou revogadas de (revoked) um grupo como um todo.

Para a criação de um grupo de usuários, use o comando

CREATE GROUP *nome*

Para adicionar usuários ou removê-los de um grupo, use

ALTER GROUP *nome* ADD USER *nomeusuario1*, ...

ALTER GROUP *nome* DROP USER *nomeusuario1*, ...

Permissões

Quando um objeto de banco de dados é criado, a ele um owner (dono, proprietário) é atribuido. O owner do objeto é o usuário que executou a instrução de criação daquele objeto. Não há atualmente uma agradável interface para mudanças de owner de um objeto de banco de dados. Por default, apenas um usuário (ou um superusuário) pode fazer o que quiser com o objeto. De maneira a permitir que outros usuários possar fazer isso também, *permissões* devem ser atribuidas (granted) aos usuários.

Existem diferentes tipos de permissões: `SELECT` (leitura), `INSERT` (insersão), `UPDATE` (gravação), `DELETE`, `RULE`, `REFERENCES` (foreign key), and `TRIGGER`.

O direito de modificar ou destruir um objeto é sempre o privilégio apenas do owner. Para atribuir privilégios, o comando **GRANT** é usado. Então, se `joe` é um usuário ativo, e `accounts` é uma tabela existente no banco, acesso de gravação à essa tabela pode ser granted com o comando :

GRANT UPDATE ON accounts TO joe;

O usuário que está executando esse comando deve ser o owner da tabela. Para atribuir um privilégio a um grupo, use :

GRANT SELECT ON accounts TO GROUP nome_do_grupo;

Existe um nome de "usuário" especial chamado `PUBLIC` que pode ser usado para atribuir um privilégio a todos os usuários do sistema.

Se você colocar a palavra `ALL` no local de um privilégio específico, você estará atribuindo todos os tipos de privilégios aquele determinado objeto.

Para revogar (revoke) um privilégio, use o apropriadamente chamado comando **REVOKE** :

REVOKE ALL ON accounts FROM PUBLIC;

Os privilégios especiais do owner de uma tabela (isto é, o direito de **DROP**, **GRANT**, **REVOKE**, etc) são sempre implícitos porque ele é o criador da própria tabela, e não podem ser revogados. Mas o próprio owner da tabela pode decidir que quer revogar seus próprios privilégios sobre ela, por exemplo fazer a table read-only para ele mesmo e

também para os outros.

Permissões de Schemas

Por padrão, usuários não podem acessar schemas que são de propriedade de outro usuário. Para permitir o acesso utilizamos o comando **GRANT** e a permissão **USAGE** (uso):

GRANT USAGE ON SCHEMA meu_schema TO usuario

Podemos também permitir que um usuário crie novos objetos dentro do nosso schema, utilizando a permissão CREATE:

GRANT CREATE ON SCHEMA meu_schema TO usuario

Funções e Triggers

Funções e triggers permitem que usuários insiram código no servidor e fazer com que outros usuários executem esse código sem saber disso. Por essa razão, ambos os mecanismos permitem usuários a sabotarem outros com relativa impunidade. A única proteção real é um rigoroso controle sobre quem pode ou não definir funções (por exemplo, fazer relatções entre campos de tabelas usando SQL) e triggers. As trilhas de auditoria e colocação de alertas nas tabelas do sistema tipos `pg_class`, `pg_shadow` e `pg_group` também são possíveis.

Funções escritas em qualquer linguagem exceto SQL rodam dentro dos processos backend do servidor com as permissões do processo daemon do servidor de banco de dados. É possível mudar as estruturas internas de dados do servidor de dentro de funções consideradas verdadeiras (trusted functions) pelo banco de dados. Consequentemente, dentro de outras coisas, esse tipo de funções podem enganar qualquer sistema de controle de accesso. Esse é um problema característico de funções de usuários escritas em linguagem C.

Resumo da Unidade

- A adminstração de usuários é de reponsabilidade do DBA.
- Criando grupos de usuários o DBA pode ter seu trabalho facilitado.
- O uso de SCHEMAS pode ser bantante útil no gerenciamento das tabelas do banco de dados.

Revisão da Unidade

1. Qual comando fornece direitos aos usuários ?

2. Qual a função do grupo de usuários ?

3. Se removermos um usuário seus objetos também serão removidos ?

Laboratório 4:
Gerenciando usuários e permissões

Acompanhe o instrutor na execução do laboratório.

Unidade 5:
Backup e restore

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Fazer Backups e Restores no PostgreSQL

Tópicos da Unidade

- Utilizando o backup lógico
- Implementando backup físico do banco de dados
- Restaurando cópias de segurança

Backup e Restore

Como tudo que contém informações importantes, os bancos de dados do PostgreSQL deveriam ser backupeados regularmente. Mesmo que esse procedimento seja essencialmente simples, é importante ter um básico entendimento das técnicas fundamentais de backup.

Existem dois tipos fundamentais de backup no PostgreSQL :

- SQL dump
- backup a nível de sistema operacional

SQL Dump

A idéia atrás do método de backup via SQL-dump é gerar um arquivo texto com comandos SQL que, quando devolvidos ao servidor, irão recriar o banco de dados no mesmo estado em que ele estava no momento do dump.

O PostgreSQL provê o programa utilitário pg_dump para esse propósito. O uso básico dele é:

pg_dump ***nome-do-banco-de-dados*** **>** ***arquivo_saida***

Como você vê, o programa pg_dump escreve seus resultados em uma saída standard. Nós veremos logo abaixo como isso pode nos ser útil.

O programa pg_dump é uma aplicação cliente normal do PostgreSQL (embora ela seja particularmente inteligente). Isso significa que você pode rodar esse procedimento remotamente de qualquer host remoto que tenha acesso ao banco de dados. Mas lembre que o pg_dump não operate com permissões especiais. Em particular, vecê deve ter acesso para leitura de todas as tabelas que você quer fazer back up, em pratica você deve sempre ser um superusuário do banco.

Para especificar qual servidor de banco de dados que o pg_dump deve contactar, use as opções de linha de comando `-h` *`host`* e `-p` *`port`*. O host default é o local host ou qualquer outro que sua variável de ambiente `PGHOST` especificar. De maneira semelhante, a porta default é indicatda pela variável de ambiente `PGPORT` ou, falhando essa, pela variável default definida na hora da compilação. (Convenientemente, o servidor terá normalmente a mesma default definida na hora da compilação.)

Como qualquer outra aplicação cliente do PostgreSQL, o pg_dump por default irá se conectar usando o nome do usuário corrente do Unix. Para passar por cima disso, especifique a opção `-U` ou sete a variável de ambiente `PGUSER`. Lembre que as conexões do pg_dump estão sujeitas aos mecanismos de autenticação (aqueles já descritos no capitulo de Autenticações de Clientes).

Os arquivos dumps criados por by pg_dump são internamente consistentes, isto é, updates no banco de dados enquanto o pg_dump estiver rodando não estarão no dump. O pg_dump não bloqueia as operações que estão ocorrendo no banco de dados durante o seu trabalho. (As exceções são aquelas operações que requerem lock exclusivo do banco de dados, tipo **VACUUM FULL**.)

Importante : quando seu schema de banco de dados confia nas OIDs (por exemplo as foreign keys) você deve instruir o pg_dump para exportar as OIDs também. Para fazer isso, use a opção `-o` na linha de comando. “Objetos Largos” não são exportados por default também.

Restaurando apartir do SQL Dump

Os arquivos texto criados pelo pg_dump são feitos para serem lidos pelo programa psql. A forma geral do comando para se restaurar um dump é :

```
psql nome-do-banco-de-dados < arquivo_entrada
```

onde *arquivo_entrada* é o mesmo que você usou no comando pg_dump como *arquivo_saida*. O banco de dados *nome-do-banco-de-dados* não será criado por este comando, você deve criá-lo apartir do template0 antes de executar o psql (exemplo, `createdb -T template0 nome-do-banco-de-dados`). O psql suporta opções similares para que o pg_dump controle a localização do servidor do banco de dados e os nomes dos usuários. Se os objetos do banco de dados original pertencerem à usuários diferentes, então o arquivo dump irá instruir o psql para se conectar como cada usuário necessário para conseguir criar os seus objetos. Desse jeito, os direitos e donos dos objetos são mantidos como no banco de dados original. Isso também significa, entretanto, que todos esses usuários já devam existir, e mais ainda, que você possa se conectar como cada um deles. Pode ser portanto necessário relaxar temporáriamente com os setups de autenticação de clientes.

A habilidade do pg_dump e do psql em gravar ou ler dos pipes tornam possíveis gerar um dump de um banco de dados diretamente de um servidor para outro, por exemplo

```
pg_dump -h host1 banco-de-dados | psql -h host2 banco-de-dados
```

Importante : Os dumps produzidos pelo pg_dump são relativos ao template0. Isso significa que quaisquer línguas, procedures, etc. adicionadas ao template1 irão também ser dumpeadas pelo pg_dump. Como resultado, quando restaurando, se você estiver usando um template1 customizado, você deve criar um banco de dados vazio apartir do template0, como no exemplo acima.

Usando o pg_dumpall

O mecanismo acima é meio complicado e talvez até inapropriado quando o backup envolve todo o cluster dos bancos de dados. Por essa razão que o programa pg_dumpall foi criado. O programa pg_dumpall faz backup de cada banco de dados de um determinado cluster e também garante que o estado de dados globalis tipo usuários e grupos são preservados. A sequência para rodar o pg_dumpall é simples

```
pg_dumpall > arquivo_de_saida
```

O dump resultante pode ser restaurado com o psql como demonstrado acima. Mas nesse caso é definitivamente necessário que você tenha acesso ao banco de dados como superusuário, pois é necessário também restaurar as informações de usuários e de grupos.

Grandes Bancos de Dados

Originalmente escrito por Hannu Krosing (<hannu@trust.ee>) em 19/06/1999

Desde que o PostgreSQL permite tabelas maiores que o tamanho máximo do seu sistema, isso pode ser problemático dumpear a tabela para um arquivo, pois o arquivo resultante será provávelmente maior que o permitido pelo seu sistema operacional. Como o pg_dump escreve para a saída standar, você pode usar então as ferramentas standard *nix para dar uma volta nesse possível problema.

Use dumps comprimidos. Use o seu programa favorito de compressão, por exemplo o gzip.

```
pg_dump banco_de_dados | gzip > arquivo_zipado.gz
```

Restaure com

```
createdb banco_de_dados
gunzip -c arquivo_zipado.gz | psql banco_de_dados
```

ou

```
cat arquivo_zipado.gz | gunzip | psql banco_de_dados
```

Use split. Isso permite que você parta o arquivo de saída em várias peças que são aceitáveis em tamanho para o seu sistema operacional. Por exemplo, para fazer pedaços de 1 megabyte:

```
pg_dump banco_de_dados | split -b 1m - arquivo_partido
```

Restaure com

```
createdb banco_de_dados
cat arquivo_partido* | psql banco_de_dados
```

Use o formato padrão de dump. Se o PostgreSQL foi compilado em um sistema com a library de compressão zlib instalada, o dump irá comprimir dados no arquivo de saída enquanto ele o grava. Para grandes bancos de dados, isso irá produzir dumps com tamanhos similares aos dumps que você faria usando o **gzip**, mas têm ainda a vantagem

que as tabelas podem ser restauradas seletivamente. O seguinte comando gera um dump de um banco de dados usando o formato padrão:

```
pg_dump -Fc banco_de_dados > arquivo_dump
```

Avisos

O pg_dump (e por implicação o pg_dumpall) tem algumas pequenas limitações as quais relacionadas com dificuldades em reconstruir certas informações das tabelas de catálogo do sistema.

Especificamente, a ordem na qual o pg_dump escreve os objetos não é muito sofisticada. Isso pode gerar alguns problemas por exemplo quando funções são usadas como valores default de colunas. A única resposta é reordenar manualmente o dump. Se você criou dependências circulares em seu schema, então você terá mais trabalho a fazer.

Por razões de compatibilidade, o pg_dump não gera dumps de grandes objetos por default. Para gerar dumps de grandes objetos você deve usar o dump padrão ou o de saída no padrão TAR, e use a opção -b no pg_dump. Olhe as páginas de referência para detalhes. The directory `contrib/pg_dumplo` of the PostgreSQL source tree also contains a program that can dump large objects.

1. Please familiarize yourself with the pg_dump reference page.

Backup a nível de Sistema Operacional

Uma estratégia alternativa de backup é copiar diretamente os arquivos que o PostgreSQL usa para guardar os dados de um banco de dados. Você pode usar qualquer método que você prefira para fazer seus backups a nível de sistema, por exemplo

```
tar -cf backup.tar /usr/local/pgsql/data
```

Existem duas restrições, entretanto, as quais podem tornar esse método impraticável, ou ao menos inferior ao método do pg_dump :

1. O servidor de banco de dados *deve* estar com o shut down executado para que se possa conseguir um backup decente. Meias medidas tipo disabilitar todas as conexões não irão funcionar, pois haverá sempre alguma atividade em buffer acontecendo. Por esta razão não é também aconselhável acreditar em sistemas de arquivos que pregam suporte à "consistent snapshots". Também não há necessidade em dizer que você precisa tirar o servidor do ar para restaurar os dados.

2. Se você já foi fundo nos detalhes de layout dos arquivos de sistema já deve ter sido tentado a backupear ou restaurar apenas tabelas individuais ou bancos de dados dos seus respectivos arquivos ou diretórios. Isso também *não* irá funcionar, pois as informações contidas nesses arquivos contém apenas metade da verdade. A outra metade está nos arquivos de logs de commit, os `pg_clog/*`, os quais contém os status de commit de todas as transações. Um arquivo de tabela só pode ser aproveitado com essas informações. É claro que também é impossível restaurar apenas uma tabela e seu associado `pg_clog` pois isso irá tornar todas as outras tabelas do cluster de banco de dados inúteis.

Saiba também que um backup a nível de sistema não é necessáriamente menor que o SQL dump. Pelo contrário, ele provávelmete será maior. (o programa pg_dump não precisa gerar dump dos índices por exemplo, apenas os comandos para recriá-los.)

Migração entre releases

Como regra geral, o formato interno de armazenamento de dados é sujeito à mudanças entre os releases do PostgreSQL.

Isso não se aplica à diferentes “níveis de patches”, eles são sempre compátiveis em termos de formatação interna. Por exemplo, os releases 7.0.1, 7.1.2, e 7.2 não são compatíveis, enquanto os 7.1.1 e o 7.1.2 são. Quando você fizer um update entre versões compatíveis, você pode simplesmente reusar a área de dados em disco paro os novos executáveis.

Caso você troque de versão você obrigatóriamente deve backupear seus dados e restaurá-los em um novo servidor, usando o pg_dump. (Existem testes que previnem você de fazer a coisa errada, havendo então nenhum risco de confundir as coisas.) A completa rotina para isso já foi vista na seção de intalação.

Para parar o servidor com o mínimo de perda de tempo, faça a instalação do novo servidor em um novo diretório, usando portas diferentes, sem ter que parar o antigo. Você pode usar alguma coisa tipo

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

para transferir seus dados, ou usar um arquivo intermediário se você quiser. Você pode então executar um shut down no servidor antigo e startar o novo servidor na porta ondo o antigo estava rodando. Você deve ter certeza de que o banco de dados antigo não foi alterado após o uso do pg_dumpall, caso contrário, você obviamente perdeu esses dados. Dê uma olhada no capítulo que fala em Autenticação de Clientes para mais informações em como restringir o acesso ao banco de dados. Em prática você provávelmete vai querer testar suas aplicações cliente no novo setup antes de desativar o antigo.

Se você não pode ou não quer rodar dois servidores em paralelo, você pode rodar o backup antes de instalar a nova versão, dar o shutdown no servidor, mover a versão antiga, instalar a nova versão, startar o novo servidor, restaurar os dados. Por exemplo:

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd /usr/src/postgresql-7.2
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql < backup
```

Nota : Quando você “move a antiga instalação” ela se torna não mais perfeitamente usável. Algumas partes da nova instalação contém informações sobre onde as outras

partes estão localizadas. Isso não é geralmente um grande problema, mas se você estiver planejando usar duas instalações em paralelo por um tempo, então você deve executar a nova instalação usando diretórios diferentes na hora da compilação.

Resumo da Unidade

- O backup do banco de dados é de responsabilidade do DBA.
- Testes de restauração garantem a funcionalidade dos backups.
- Deve existir um backup externo ao prédio da empresa para uma maior garantia da segunrança dos dados.

Revisão da Unidade

1. Qual backup ocupará mais espaço, o lógico ou o fisíco ?

2. Qual é uma das maiores reponsabilidades do DBA ?

3. O que fazer para se ter um backup confiável ?

Laboratório 5:
Backup e restore

Acompanhe o instrutor na execução do laboratório.

Unidade 6:
O ambiente do servidor em tempo de

execução

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Mostrar como setar e rodar um servidor de banco de dados e suas interações com o sistema operacional.

Tópicos da Unidade

- Inicializando uma área de dados
- Colocando o banco de dados online
- Parando o banco de dados

A conta de usuário PostgreSQL

Como qualquer outro daemon servidor que está conectado com o mundo, é indicado rodar o PostgreSQL usando uma conta separada de usuário. Essa conta deveria apenas possuir os dados que estão sendo gerenciados pelo servidor, e não deveria compartilhá-la com outros daemons. Não é aconselhado instalar os executáveis como propriedade desse usuário pois existe o risco de perda de funções de usuário ou o uso indevido dessa conta comprometer os programas executáveis.

Para adicionar uma conta de usuário no seu sistema, procure pelos comandos **useradd** ou **adduser**. O nome de usuário postgres é frequentemente usado mas não é requerido.

Criando um cluster de bancos de dados

Antes que você possa fazer qualquer coisa, você deve initializar uma área de armazenamento de bancos de dados no disco. Nós chamamos isso de *cluster de bancos de dados*. (O SQL fala em um cluster de catálogos.) Um cluster de bancos de dados é uma coleção de bancos de dados que serão acessados através de uma única instância de um servidor de bancos de dados rodando. Após a initialização, um cluster de bancos de dados deve conter um banco de dados chamado `template1`. Como o nome sugere, ele será usado de template para os próximos bancos de dados criados; ele não deveria ser usado para trabalho em produção.

Em termos de arquivos de sistema, um cluster de bancos de dados será um único diretório no qual todos os dados serão armazenados. Nós chamamos isso de *diretório de dados* ou de *área de dados*. É completamente com você escolher o local aonde armazenar seus dados.

Não existe local default, embora os locais tipo `/usr/local/pgsql/data` ou `/var/lib/pgsql/data` sejam muito populares. Para initializar um cluster, use o comando **initdb**, o qual está instalado com o PostgreSQL.

O local desejado para armazenar seus dados é indicado pela opção `-D`, por exemplo

```
$ initdb -D /usr/local/pgsql/data
```

Note que você deve executar esse comando estando logado com a conta do usuário PostgreSQL, a qual foi descrita na seção anterior.

Tip: Uma alternativa para a opção `-D` seria você setar a variável de ambiente `PGDATA`.

O **initdb** tentará criar o diretório que você especificou caso ele ainda não exista. É provável que ele não tenha permissão para fazê-lo (se você seguiu nosso conselho e criou uma conta de usuário sem privilégios). Nesse caso você deveria criar o diretório como root e transferir sua propriedade para a conta do usuário PostgreSQL.

Tente algo assim :

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

O **initdb** irá se recusar a rodar caso o diretório de dados lhe pareça pertencer à uma instalação já inicializada.

Porque o diretório de dados contém todos os dados armazenados de um banco de dados, é essencial que ele esteja bem seguro de acessos não autorizados. O **initdb** portanto revoga a permissão de acesso de todos, menos da conta do usuário PostgreSQL.

Entretanto, enquanto os conteúdos do diretório estão seguros, o método default de autenticação do arquivo `pg_hba.conf` chamado `trust` permite que qualquer usuário local se conecte ao banco de dados e ainda de torne um superusuário. Se você não confia nos outros usuários locais, nós recomendamos que você use a opção do **initdb,** `-W` ou `--pwprompt` para determinar uma senha para o superusuário do banco de dados. Após o **initdb**, modifique o arquivo `pg_hba.conf` para usar os métodos de autenticação `md5` ou `password`, ao invés de `trust`, *antes* de startar o servidor pela primeira vez. (Outra possibilidade talvez mais conveniente ainda seria usar o método de autenticação `ident` ou o arquivo de permissões do sistema para restringir conexões.

Uma surpresa seria você receber uma mensagem igual a essa quando rodar o **initdb** :

```
NOTICE: Initializing database with en_US collation order.
This locale setting will prevent use of index optimization
for
LIKE and regexp searches. If you are concerned about speed of
such queries, you may wish to set LC_COLLATE to "C" and
re-initdb. For more information see the Administrator's
Guide.
```

Essa mensagem é de alerta para que você saiba que o local selecionado corrente (locale) irá causar os índices serem sorteados em uma ordem que os impede de serem usados em expressões do tipo LIKE e outras procuras com expressões regulares. Se você precisa de boa performance nesses tipos de procuras, você deveria setar seu local corrente para “c” e rodar novamente o **initdb**. Na maioria dos sistemas, isso é feito com a troca do valor da variável de ambiente `LC_ALL` ou `LANG`. A ordem dos sorts usados dentro de um banco de dados em particular é setado pelo **initdb** e não pode mais ser mudado, tipo nos obrigando a fazer um dump de todos os dados, rodar novamente o **initdb**, e recarregando os dados novamente prá dentro do banco. É muito importante fazer essa escolha certa agora.

Startando o servidor de bancos de dados

Antes que qualquer um possa acessar o banco de dados você deve startar o servidor. O servidor de bancos de dados é chamado de *postmaster*. O postmaster deve saber onde encontrar os dados no qual ele é suposto a ter de trabalhar. Isso é feito com a opção `-D`. Portanto, a maneira mais simple de se startar o servidor é, por exemplo,

```
$ postmaster -D /usr/local/pgsql/data
```

o qual irá deixar o servidor rodando em foreground. Isso novamente deve ser feito estando-se logado com a conta do usuário PostgreSQL. Sem o `-D`, o servidor irá tentar usar o diretório de dados que está na variável de ambiente `PGDATA`; se nenhum dos dois jeitos der certo ele irá falhar.

Para se startar o postmaster em background, use a sintaxe usual do shell :

```
$ postmaster -D /usr/local/pgsql/data > logfile 2>&1 &
```

É uma idéia extremamente boa manter as saídas standard do servidor ligadas (stdout e stderr), como sugerido aqui. Isso ajudará sempre na necessidade de auditorias e no diagnóstico de problemas.

O postmaster também aceita outras opções de linha de comando. Em particular, para que o servidor aceite conexões TCP/IP (além das vindas através dos soquetes de domínio Unix), você deve também especificar a opção `-i`.

Essa sintaxe de shell nos faz cansar rápidamente. Foi pensando nisso que o script shell chamado pg_ctl foi criado. Ele encapsula algumas dessas tarefas. O exemplo,

```
pg_ctl start -l logfile
```

irá startar o servidor em background e ainda colocar a saída para o arquivo chamado logfile. A opção `-D` tem o mesmo significado com o pg_ctl que tem com o próprio postmaster. O pg_ctl também implementa a operação simétrica de “stop”.

Com certeza você irá querer startar o servidor de bancos de dados quando o seu computador é bootado. Isso não é requerido entretanto, pois o servidor do PostgreSQL pode rodar com sucesso por contas não privilegiadas de usuários sem qualquer intervenção do root.

Diferentes sistemas tem diferentes convenções para startar daemons na hora em que a máquina é ligada, familiarize-se com as convensões de seu sistema operacional. Muitos sistemas tem um arquivo no `/etc/rc.local` ou `/etc/rc.d/rc.local`. O que quer que você faça, o servidor deve começar a rodar pela conta de usuário PostgreSQL *e não pelo root* ou qualquer outro usuário. Portanto você irá formar suas linhas de comando junto com as linhas de comando `su -c '...' postgres`, por exemplo:

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog' postgres
```

Aqui vão algumas sugestões especícias de alguns sistemas operacionais. (Troque sempre o diretório de instalação com o seu diretório correto e o nome de usuário correto também.)

• Para o FreeBSD, dê uma olhada no arquivo `contrib/start-scripts/freebsd`.

• No OpenBSD, adicione as seguintes linhas no arquivo `/etc/rc.local`:

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postmaster ]; then
   su - -c '/usr/local/pgsql/bin/pg_ctl start -l /var/postgresql/log -s' postgres
   echo -n ' postgresql'
fi
```

• Nos sistemas Linux adicione :

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

no `/etc/rc.d/rc.local` ou olhe dentro do arquivo `contrib/start-scripts/linux` na distribuição fonte do PostgreSQL para integrar o start e o shutdown dentro do run level system.

• No Solaris, crie um arquivo chamado `/etc/init.d/postgresql` tpara conter a seguinte linha:

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l
logfile -D /usr/local/pgsql/data"
```

Após isso, crie um link simbolico para isso no `/etc/rc3.d` como `S99postgresql`.

Enquanto o postmaster está rodando, o seu PID está no arquivo `postmaster.pid` no diretório de dados. Isso é usado como um lock interno para previnir que múltiplos postmasters rodem no mesmo diretório de dados, e pode ser usado também para dar o shutdown no postmaster.

Falhas de start-up no servidor de bancos de dados

Existem várias razões comuns para que o postmaster falhe no startup. Cheque o arquivo de log do postmaster, ou starte ele à mão (sem o redirecionamento standard de saída ou de erros) para que você possa ver diretamente as menssagens que aparecem. Algumas das mensagens de erros são auto-explicativas, temos aqui algumas que não são.

```
FATAL: StreamServerPort: bind() failed: Address already in
use
Is another postmaster already running on that port?
```

Isso geralmente significa o que isso sugeres: você tentou startar um segundo postmaster na mesma porta onde já havia um outro rodando. Entretanto, se a mensagem de erro do kernel não é `Address already in use` ou alguma variante dessas palavras, pode então ser um problemas diferente. Por exemplo, ao tentar startar o postmaster em uma porta reservada podemos receber uma mensagem tipo essa :

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
Is another postmaster already running on that port?
```

Uma mensagem tipo :

```
IpcMemoryCreate: shmget(key=5440001, size=83918612, 01600)
failed: Invalid argument
FATAL 1: ShmemCreate: cannot create region
```

Provávelmente significa que o limite de memória compartilhada do seu kernel é menor que a área de buffer que o PostgreSQL está tentando criar (83918612 bytes nesse exemplo). Ou poderia significar que você não tem suporte configurado de memória compartilhada estilo System-V no seu kernel de jeito nenhum. Como uma alternativa temporária, você pode tentar startar o postmaster com um número de buffers menor que o normal usando o switch `-B`. Você pode eventualmente querer reconfigurar seu kernel para aumentar o tamanho de memória compartilhada. Você pode ainda ver essa mensagem quando for startar múltiplos postmasters na mesma máquina, se o espaço total de requisições exceder o limite do kernel.

Um erro do tipo

```
IpcSemaphoreCreate:   semget(key=5440026,   num=16,   01600)
failed: No space left on device
```

não significa que você está sem espaço em disco; isso significa que o limite de semáforos do tipo System V em seu kernel é menor do que o número que o PostgreSQL quer criar. Como acima, você pode eventualmente querer reconfigurar seu kernel para aumentar o tamanho de semáforos que ele pode suportar, ou resolva temporáriamente seu problema startando o postmaster com um número reduzido de processos backend usando o switch `-N`.

Se você receber um erro do tipo “illegal system call”, então é provável que memória compartilhada ou semáforos não são de jeito nenhum suportados pelo seu kernel. Nesse caso, sua única opção será a de reconfigurar o kernel para torná-las disponíveis.

Problemas de conexão dos clientes

Embora as possíveis condições de erro no lado dos clientes serem ambas virtualmente infinitas e independentes de aplicações, umas poucas podem ser diretamente relacionadas ao modo de como o servidor foi starteado. Condições outras das quais iremos mostrar abaixo deveriam ser documentadas com a respectiva aplicação cliente.

```
psql: could not connect to server: Connection refused
Is the server running on host server.joe.com and accepting
TCP/IP connections on port 5432?
```

Essa é uma falha genérica de "Não achei um servidor para conversar". Parece que acima tivemos uma tentativa de comunicação TCP/IP. Um erro muito comum é esquecer de usar a opção `-i` para permitir o postmaster de aceitar conexões TCP/IP.

Alternativamente, você receberá essa mensagem quando tentar uma conexão local ao postmaster no ambiente Unix :

```
psql: could not connect to server: Connection refused
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

A última linha é útil para verificarmos aonde o cliente está tentando se conectar. Se de fato não tivermos um postmaster rodando lá, a mensagem do kernel será tipicamente `Connection refused` ou `No such file or directory`, como ilustrado. (é particularmente importante se dar conta de que `Connection refused` nesse contexto *não* significa que o postmaster recebeu a requisição de conecção e a rejeitou – esse caso irá gerar uma mensagem diferente, como mostrado na seção Problemas de Autenticação.) Outras mensagens de erro tipo `Connection timed out` podem indicar problemas mais fundamentais, como falta de conectividade da rede.

Configuração em modo de execução

Existem vários parâmetros de configuração que afetam o comportamento do sistema de banco de dados em algum jeito ou outro. Iremos aqui descrever como setá-los.

Todos os nomes de parâmetros não são case-sensitivos. Cada parâmetro recebe um valor de um dos quatro tipos Booleano, inteiro, ponto flutuante e string como descrito abaixo. Os valores booleanos são `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (não case-sensitivo).

Um jeito de setar essas opções é em editar o arquivo `postgresql.conf` no diretório de dados. (Existe um arquivo default instalado lá.) Um exemplo de como esse arquivo pareceria é :

```
# Esse é um comentário
log_connections = yes
syslog = 2
```

Como você vê, as opções são uma por linha. O sinal de igual entre o nome e o valor são opcionais. O espaço em braco é insignificante, linhas brancas são ignoradas. As marcas hash (“#”) introduzem comentários em qualquer lugar.

O arquivo de configuração é relido sempre que o postmaster recebe um sinal SIGHUP (o que é fácilmente enviado pelo `pg_ctl reload`). O postmaster também propaga esse sinal à todos os processos já rodando em backend, para que as sessões já existentes peguem o novo default. Alternativamente, você pode enviar o sinal diretamente para apenas um processo.

Um segundo jeito de setar esses parâmetros de configuração é colocá-los nas opções de linha de comando do postmaster, tipo assim

```
postmaster -c log_connections=yes -c syslog=2
```

o que teria o mesmo efeito que o exemplo anterior. As opções de linha de comando sobrescrevem qualquer conflito de configuração que exista no `postgresql.conf`.

Ocasionalmente é útil enviar uma opção de linha de comando para um processo em particular. A variável de ambiente `PGOPTIONS` pode ser usada para esse propósito no lado cliente:

```
env PGOPTIONS='-c geqo=off' psql
```

(Isso funciona para qualquer aplicação cliente, não paenas para o psql.) Note que isso não funcionará para opções que são fixas quando o servidor é startedo, tipo o número da porta.

Finalmente, algumas opções podem ser alteradas em sessões SQL individuais usando-se o comando **SET**, por exemplo

⇨ **SET ENABLE_SEQSCAN TO OFF;**

Limitação de recursos

Os sistemas operacionais do tipo Unix impõem vários tipos de limitações de recursos que podem interferir na operação do seu servidor PostgreSQL. Os de real importância são especialmente os limites de número de processos por usuário, o número de arquivos abertos por processo, e o total de memória disponível para um processo.

Cada um desses tem um limite chamado “duro” e um limite chamado “suave”. O limite suave é o que realmente conta mas isso pode ser mudado pelo usuário até o limite duro. O limite duro pode apenas ser modificado pelo usuário root. A chamade de sistema `setrlimit` é responsável por setar esses parâmetros. O comando built-in do shell **ulimit** (Bourne shells) ou **limit** (csh) é usado para o controle de limites de recursos apartir da linha de comando. Em derivações do sistema BSD o arquivo `/etc/login.conf` controla os valores de vários limites por login. Veja o login.conf para mais detalhes. Os parâmetros relevantes são `maxproc`, `openfiles`, e `datasize`. Por exemplo:

```
default:\
...
:datasize-cur=256M:\
:maxproc-cur=256:\
:openfiles-cur=256:\
...
```

(`-cur` é o limite suave. `-max` setaria o limite duro.)

Os Kernels genermente tem implementações de limites para todo o sistema em alguns recursos.

• No Linux, o `/proc/sys/fs/file-max` determina o número máximo de arquivos abertos que o kernel irá suportar. Isso pode ser mudado trocando-se esse número ou adicionando uma atribuição no `/etc/sysctl.conf`. O limite máximo de arquivos por processo é fixado no momento em que o kernel é compilado; veja `/usr/src/linux/Documentation/proc.txt` para mais informações.

O servidor PostgreSQL usa apenas um processo por conexão e você deveria ao menos garantir que possam existir tantos processos quanto o número máximo de conexões permitidas, em adição ao que mais for necessário para o resto do seu sistema. Isso não é geralmente um problema mas se você rodar vários servidores em uma mesma máquina, isso talvez faça as coisas ficarem um pouco apertadas.

O limite default de fábrica para os arquivos abertos é geralmente setado com valores "socialmente amigáveis" os quais permitirão vários usuários coexistir em uma só máquina usando uma fração apropriada dos limites de recursos do sistema. Se você for rodar vários servidores em uma mesma máquina, talvez seja isso mesmo o que você quer, mas em servidores dedicados talvez você queira aumentar esse limite de fábrica.

Do outro lado da moeda, alguns sistemas permitem que processos individuais abram uma grande quantidade de arquivos; se vários processos começarem a fazer isso, os limites do sistema podem ser excedidos rápidamente. Se isso acontecer com você, e você não quer alterar o limite do próprio sistema operacional, você pode setar no PostgreSQL o parâmetro de configuração `max_files_per_process` para limitar o seu próprio consumo de arquivos abertos.

Tirando o servidor do ar (Shutdown)

Dependendo das susas necessidades, existem vários jeitos de se dar um shutdown no servidor de bancos de dados quando o seu trabalho estiver pronto. A diferença é feita pelo sinal enviado ao processo servidor.

SIGTERM

Após receber o SIGTERM, o postmaster disabilita novas conexões, mas ele deixa os processos backend existentes terminares seu trabalho normalmente. Ele encerra o servidor somente quando todos os backends terminem as requisições de seus clientes. Esse é o *Smart Shutdown*.

SIGINT

O postmaster disabilita novas conexões e e envia a todos os backends existentes o SIGTERM, o que causará que as suas transações correntes devem ser abortadas e que eles saiam na hora. O postmaster então espera que os backends terminem e encerra o servidor. Esse é o *Fast Shutdown*.

SIGQUIT

Esse é o *Immediate Shutdown* que fará com que o postmaster envie o sinal SIGQUIT para todos os processos backends e saia imediatamente (sem mesmo ter efetuado um shutdown decente no banco de dados). Os processos backend param de trabalhar imediatamente e saem logo após terem recebido o sinal SIGQUIT. Isso irá fazer o servidor tentar um recovery automático (aplicando o WAL log) no próximo startup. Só é recomendado em emergências.

Importante : Existe ainda o sinal SIGKILL para fazer parar o postmaster, mas ele é extremamente não recomendado. Ele impedirá o postmaster de liberar memória compartilhada e semáforos, o qual você terá que fazer manualmente.

O PID do processo postmaster pode ser encontrado usando-se o programa os, ou abrindo-se o arquivo `postmaster.pid` no diretório de dados. Por exemplo, para se fazer um fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

O programa pg_ctl é um script shell que lhe provê uma interface mais conveniente de se fazer shutdown no postmaster.

Conexões TCP/IP seguras com SSL

O PostgreSQL tem suporte nativo para conexões SSL que encriptam as comunicações cliente/servidor para aumento de segurança. Isso requere que o OpenSSL seja instalado tanto no lado cliente quanto no sistema do servidor e tenha seu suporte habilitado na hora da construção do servidor (compilação).

Com o suporte SSL compilado, o servidor PostgreSQL pode ser startado com o argumento `-l` (ele) para habilitar conexões SSL. Quando startado em modo SSL, o servidor irá procurar pelos arquivos `server.key` e `server.crt` no diretório de dados. Esses arquivos deveriam conter a chave privada e a chave certificada do servidor respectivamente.

Eles devem ser setados corretamente antes do servidor SSL-enabled possa ser startado. Se a chave privada é protegida por uma frase de senha (passphrase), o servidor a pedirá e não startará enquanto não lhe for dada a chave correta.

O servidor irá atender ambas conexões standard e SSL na mesma porta TCP/IP, e irá negociar com qualquer cliente, usando ou não o SSL. Olhe o capítulo de Autenticação de Clientes para ver como forçar o servidor a usar o SSL para certas conexões.

Para detalhess em como criar as chaves privadas e certificadas para o seu servidor, refira-se a documentação do OpenSSL.

Um certificado simples e auto_assinado pode ser usado para testes, mas um certificado assinado por uma AC (Aplicação Cliente), mesmo uma global ou local deveria ser usada em produção para que o cliente possa verificar a identidade do servidor. Para criar um rápido certificado auto-assinado, use o seguinte comando OpenSSL :

```
openssl req -new -text -out cert.req
```

Preencha a informação que o **openssl** lhe pergunta. Tenha certeza de informar o nome local do host na pergunta do Common Name; a “challenge password” pode ser deixada em branco. O script irá gerar uma chave que é protegida por uma frase de senha (passphrase); ele não aceitará uma frase com menos de 4 caracteres de tamanho.

Para remover a frase (e você deve fazer isso caso queira startar automaticamente o servidor), rode o comando

```
openssl rsa -in privkey.pem -out cert.pem
```

Entre com a velha frase de senha para destrancar a chave existente. Agora faça

```
openssl req -x509 -in cert.req -text -key cert.pem -out
cert.cert
cp cert.pem $PGDATA/server.key
cp cert.cert $PGDATA/server.crt
```

para tornar o certificado em um certificado auto-assinado e para copiar a chave e o certificado para um local o qual o servidor irá procurar por eles.

Conexões TCP/IP seguras com SSL com túneis SSH

Agradecimento : Idéia tirada de um email de Gene Selkov Jr. (<selkovjr@mcs.anl.gov>) escrito em 08/09/1999 em resposta a uma questão de Eric Marsden.

Pode-se usar ssh para encriptar a conexão de rede entre clientes e um servidor PostgreSQL. Feito propriamente, isso deveria nos prover uma segurança adequada para conexões de redes.

Primeiro, tenha certeza de que um servidor ssh esteja rodando propriamente na mesma máquina que o PostgreSQL e que você pode se logar usando o **ssh** como um usuário. Você pode então estabelecer um túnel seguro com um comando tipo esse apartir de uma máquina cliente :

```
$ ssh -L 3333:company.com:5432 guilhermecosta@company.com
```

O primeiro número no argumento `-L`, o 3333, é o número da porta do final do túnel; isso pode ser escolhido livremente. O segundo número, o 5432, é o fim remoto do túnel – o número de porta que o servidor está usando.

O nome ou o endereço entre os números das porta é o host com o servidor de bancos de dados que você está se conectando. Para se conectar no servidor usando esse túnel, você deve ser conectar na porta 3333 da máquina local :

```
psql -h localhost -p 3333 template1
```

O servidor irá então pensar que você é realmente o usuário `guilhermecosta@company.com` e irá usar o procedimento de autenticação que foi setado para esse usuário. Para que o túnel tenha sucesso você deve ter permissão de se conectar via **ssh** como guilherme@company.com, da mesma forma como se você tivesse tentado usar o **ssh** para abrir uma sessão de terminal.

Dica : Existem muitos outros produtos que provêem túneis de segurança usando um procedimento similar em conceito desse que acabamos de descrever.

Resumo da Unidade

- Colocando o banco de dados online.
- Analisando os logs do banco de dados.
- Parando o banco de dados sem perder informações das seções online.

Revisão da Unidade

1. O banco de dados está online , mas ninguém consegue acesso. O que está acontecendo ?

2. Efetivamos o comando para desligar o banco de dados e o STOP não funcionou. Por quê?

3. O banco de dados não inicializa. Onde verifico o problema ?

Laboratório 6:
O ambiente do servidor em tempo de execução

Acompanhe o instrutor na execução do laboratório.

Unidade 7:
Manutenção do Banco de Dados

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Verificar que mesmo o PostgreSQL sendo um sistema gerenciador de bancos de dados leve, ele merece ser cuidado com algumas rotinas básicas de manutenção para obtermos dele o máximo proveito.

Tópicos da Unidade

- O uso do Vacuum
- Atualizando as estatísticas do banco de dados
- Recuperando espaço em disco

Considerações Gerais

Existem algumas poucas rotinas de manutenção que devem ser executadas regularmente para manter a instalação do PostgreSQL rodando redonda. As tarefas discutidas aqui são repetitivas por natureza e podem ser fácilmente automatizadas usando-se ferramenteas standard do Unix, tipo os scripts `cron`. É responsabilidade do administrador do banco de dados setar os scripts apropriadamente, e checar se eles executam com sucesso.

Uma tarefa óbvia de manutenção seria a criação regular de backups dos dados. Sem uma cópia recente do banco de dados, você não teria chance alguma de recuperação após uma catástrofe do tipo : falha de disco, incêndio, dropar por acidente uma tabela crítica, etc. Os mecanismos de backup e recovery disponíveis no PostgreSQL são discutidos nas seção Backup e Recovery.

Outra importante categoria de manutenção periódica seria a de executar o “vacuum” no banco de dados. Essa atividade é discutida com mais detalhe daqui a pouco.

Mais um ponto que merece atenção é o gerenciamento do arquivo de log do banco. Isso também será discutido mais adiante.

O PostgreSQL é considerado um produto com baixa manutenção se comparado a outros gerenciadores de bancos de dados. Entretanto, a devida atenção a essas tarefas lhe garantirão uma experiência tanto produtiva quanto prazerosa com o sistema.

A rotina de Vacuum

O comando **VACUUM** do PostgreSQL deve ser rodado regularmente por várias razões:

1. Para a recuperação de espaço ocupado em disco por linhas alteradas ou deletadas.
2. Atualizar os dados estatísticos usados pelo query planner do PostgreSQL.
3. Proteção contra a perda de dados muito antigos devido ao *transaction ID wraparound*.

A frequência e o escopo dos **VACUUM**s executados para cada uma dessas razões irá variar dependendo da necessidade de cada instalação. Portanto, os administradores de bancos de dados devem entender esses pontos e desenvolver uma apropriada estratégia de manutenção. Essa seção se concentra em explicar esses pontos em um nível resumido; para detalhes sobre as sintaxes dos comandos ou outros detalhes, veja a página de referência do comando **VACUUM**.

Começando no PostgreSQL 7.2, a forma standard de **VACUUM** pode rodar em paralelo com as operações normais do banco (selects, inserts, updates, deletes, mas não com mudanças nos schemas de tabelas). A rotina de vacuum é portanto não mais tão importuna quanto ela era nos primeiros releases, e também não vemos problemas em você rodá-la nas horas de menos pico de acessos ao seu sistema durante o dia.

O pg_autovacuum

O pg_autovacuum é uma das tantas contribuições da comunidade para o desenvolvimento do Postgresql. Ele faz com que a rotina de vacuum seja rodada automaticamente no servidor Postgres, eliminando a necessidade da execução manual da mesma.

Instalando o pg_autovacuum

Para instalar o pg_autovacuum dve-se utilizar as instruções **make** e **make install**, padrão de instalação em máquinas Linux. Para máquinas Windows o pg_autovacuum já vem pré-compilado, ou seja, “pronto” para usar.

Usando o pg_autovacuum

Tanto para máquinas Linux – após a compilação - quanto para máquinas Windows, o pg_autovacuum nada mais é do que um binário (“executável”), ou seja, para habilitar sua execução, devemos incluí-lo na rotina de inicialização do servidor de banco de dados.

O formato de utilização do pg_autovacuum é o que segue:

***caminho*/pg_autovacuum -D -s *base_de_tempo_para_sleep* -S**

***escala_de_tempo_para_sleep* ... [*outros argumentos*]**

os parâmetros **–s** e **–S** informam o tempo que o pg_autovacuum deve "dormir" após cada processamento. Este tempo é calculado da seguinte forma:

base_de_tempo + escala_de_tempo * duração_da_última_execução

Recomenda-se um bom tempo de sleep para que não haja um consumo exacerbado dos recursos da máquina.

O parâmetro **–D** informa o pg_autovacuum que este deve rodar como um processo em segundo plano.

Para mais informações sobre o pg_autovacuum consulte o arquivo README do próprio.

Recuperando espaço em disco

Em operação normal, no PostgreSQL, um **UPDATE** ou **DELETE** de uma linha não imediatamente remove a velha *tupla* (versão de uma linha). Esse caráter é necessário para se obter os ganhos e benefícios do controle de concorrência : a tupla não deve ser deletada enquanto ela estiver potencialmente visível a outras transações. Mas eventualmente, uma obsoleta ou deletada tupla não é mais interessante para uma transação.

O espaço ocupado deve ser então reclamado para ser reusado por novas tuplas, para se evitar o crescimento infinito dos requerimentos de espaço em disco. Isso é feito rodando o **VACUUM**.

Claramente, uma tabela que recebe frequentes updates ou deletes irá necessitar sofrer o vacuum muito mais frequentemente do que outras tabelas que são raramente atualizadas. Pode ser útil setar tarefas periódicas no cron para rodar o vacuum em apenas algumas tabelas, pulando as que são conhecidas de não sofrerem atualizações frequentes.

A forma standard do **VACUUM** é melhor usada se sua meta for apenas recuperar algum espaço em seus discos. Ela apenas acha velhas tuplas e libera seu espaço para reuso dentro da tabela, mas ele não tenta realmente diminuir o tamanho da tabela e retornar esse espaço ao sistema operacional.

Se você precisa desse espaço para o sistema operacional você pode usar o **VACUUM FULL** --- mas qual é o ponto em liberar espaço em disco se ele será logo alocado novamente ? Moderados e frequentes **VACUUM**s standard são melhores que não frequentes **VACUUM FULL**s para se manter tabelas pesadamente atualizadas.

A prática recomendada para a maioria dos servidores é a de agendar um **VACUUM** ao nível de todo o banco de dados uma vez por dia quando o servidor estiver num baixo nível de acessos complementado por mais frequentes vacuums de tablelas grandemente atualizadas se necessário. (Se você tem múltiplos bancos de dados em uma mesma instalação, não esqueça de rodar o vacuum em cada uma; a criação de um script tipo um `vacuum_all` seria o ideal.) Use o plain **VACUUM**, e não o **VACUUM FULL**, para a sua rotina de recuperação de espaço.

VACUUM FULL é recomendado para casos onde você sabe que têm a maioria das tuplas de uma tabela deletadas, e nesse caso o tamanho da tabela será razoavelmente diminuído.

Se você tem uma tabela na qual seu conteúdo é deletado completamente de vez enquando, considere dar um **TRUNCATE** na tabela do que usar o **DELETE** seguido de um **VACUUM**.

Prevenindo falhas de transaction ID wraparound

A semântica de transações do PostgreSQL dependem da habilidade de comparar os números dos IDs de transação (*XID*) : uma tupla com um XID de inserção mais novo do que um XID de uma transação corrente está "no futuro" e portanto não deveria ser visível para a transação corrente. Mas desde que os IDs de transação tem um tamanho limitado (32 bits até então) uma instalação que rodar por um longo tempo (por exemplo com mais de 4 bilhões de transações) irá sofrer um *transaction ID wraparound*: o contador XID chega ao seu total e volta ao zero, e de uma hora para outra todas as transações que estavam no passado parecem agora que estão no futuro --- o que significa que as suas saídas se tornam invisíveis. Em resumo, uma perda catastrofica de dados.

(Na verdade os dados ainda estão lá, mas isso é terrível, pois não há como alcançá-los.)

Antes do PostgreSQL 7.2, a única defesa contra o XID wraparound era dar um re-**initdb** ao menos a cada 4 bilhões de transações. Isso claro não era muito satisfatório para sites com altos níveis de tráfego, então uma solução melhor tem sido planejada. Esse novo planejamento permite que uma instalação permaneça no ar indefinidamente, sem precisar do **initdb** ou qualquer outro tipo de restart. O preço disso é : *cada tabela do banco de dados deve sofrer um vacuum ao menos a cada 1 bilhão de transações*.

Em prática isso não é um requerimento tão oneroso, mas desde que as consequências em não se fazendo isso pode significar a perda completa dos dados (sem falar do espaço em disco disperdiçado e baixa performance), algumas providências tem sido tomadas para ajudar o administrador a ter o controle da última vez desde o último **VACUUM**. O resto dessa seção lhe dará os detalhes.

O novo caminho da comparação do XID distinguem dois especiais XIDs, números 1 e 2 (`BootstrapXID` e `FrozenXID`). Esses dois XIDs são sempre considerados mais velhos que cada XID normal. Os XIDs normais (aqueles maiores que 2) são comparados usando-se a aritmética do modulo-2. Isso significa que para cada XID normal, existirão dois bilhões de XIDs que serão "mais velhos" e dois bilhões que serão "mais novos"; outro jeito de dizer isso seria que o espaço de um XID normal é circular sem ponto final. Portanto, uma vez que uma tupla tenha sido criada com um particular XID normal, a tupla irá parecer estar "no passado" pelos próximos dois bilhões de transações, não importa qual XID normal nós estamos falando. Se essa tupla ainda existir após mais de dois bilhões de transações, ela de repente parecerá estar no futuro. Para a prevenção de perda de dados, as velhas tuplas devem ter seus `FrozenXID` setados em algum momento antes que elas alcancem a sua marca dos dois bilhões de transações. Uma vez setadas com esse XID especial, elas parecerão estar "no passado" para todas as transações normais independentes de qualquer problema de wraparound, essas tuplas estarão boas até serem deletadas, não importa quanto tempo isso levar. Essa designação de XIDs é manuseado pelo **VACUUM**.

A política normal do **VACUUM** é setar o `FrozenXID` para qualquer tupla com um XID com mais de um bilhão de transações no passado. Essa política preserva o XID original da inserção até a provável falta de interesse nessa tupla (em fato, a maioria das tuplas irão provavelmente nascer e morrer sem nunca terem sido "frozen"). Com essa política, o intervalo máximo de segurança entre os **VACUUM**s de uma tabela é exatamente um bilhão de transações: se você esperar mais do que isso, é possível que a tupla que não estava velha o suficiente para ser setada da última vez está agora com mais de dois bilhões de transações e foi jogada agora para o futuro --- isto é, é perda para você. (É claro, ela irá reaparecer novamente após mais dois bilhões de transações, mas isso não ajuda muito.)

Desde que **VACUUM**s periódicos são necessários de qualquer jeito pelas razões descritas anteriormente, é improvável que alguma tabela não tenha recebido um vacuum a cada bilhão de transações. Mas para ajudas os administradores a garantir que isso aconteça, o **VACUUM** guarda estatísticas dos IDs de transações na tabela de sistema `pg_database`. Em particular, o campo `datfrozenxid` da tabela `pg_database` é atualizado a cada operação de vacuum do banco de dados inteiro (isto é, **VACUUM** que não especifica nome de tabela). O valor desse campo é o XID que foi usado pelo comando **VACUUM**. Todos os XIDs normais mais velhos que esse XID estão garantidos de terem sido atualizados pelo `FrozenXID` dentro daquele banco de dados. Um bom jeito de examinar essa informação é executar a query

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

A coluna `age` mede o número de transações do XID do vacumm até o XID da transação corrente.

Com a política standard do freezing, a coluna `age` irá começar em um bilhão para um banco de dados no qual o vacuum acabou de ser executado. Quando `age` se aproxima a dois bilhões, o banco de dados deve sofre novo vacuum para evitar uma falha de wraparound. A prática recomendada é executar o vacuum no banco de dados a cada meio bilhão (500 milhões) de transações, para se ter uma margem segura. Para ajudá-lo a não esquecer, a cada **VACUUM** dado em todos os bancos de dados ele automaticamente emite um warning caso existam entradas na `pg_database` mostrando uma `age` de mais de 1.5 bilhões de transações, por exemplo:

```
play=# vacuum;
NOTICE: Some databases have not been vacuumed in 1613770184 transactions.
Better vacuum them within 533713463 transactions,
or you may have a wraparound failure.
VACUUM
```

O **VACUUM** com a opção **FREEZE** usa uma política mais agressiva de freezing: tuplas são congeladas se elas são velhas demais para serem boas por todas as transações abertas. Em particular, se um **VACUUM FREEZE** é executado em um banco de dados inativo, é garantido que *todas as* tuplas do banco de dados serão congeladas.

Portanto, enquanto o banco de dados não é modificado em alguma maneira, ele não irá necessitar subsequentes operações de vacuum para evitar problemas de ID wraparound. Essa técnica é usada pelo `initdb` para preparar o banco de dados `template0`. Isso também deveria ser usado para o preparo de qualquer banco de dados criados por usuários que serão marcadas com `datallowconn = false` na `pg_database`, já que não existe nenhum jeito conveniente de se rodar o vacuum em um banco de dados que não se pode conectar. Note que o warning automático do **VACUUM** sobre bancos de dados que precisam ter o vacuum aplicado irá ignorar as entradas da tabela `pg_database` que tiverem o `datallowconn = false`, para evitar de ficar dando falsos warnings sobre esses bancos; é portanto com você a tarefa de garantir que sees bancos de dados estejam congelados corretamente.

Atualizando o statistic query planner

O query planner do PostgreSQL conta com informações estatísticas sobre o conteúdo das tabelas para poder gerar bons planos de leitura para suas queries. Essas estatísticas são conseguidas através do comando **ANALYZE**, o qual pode ser invocado ele mesmo ou como um passo opcional do **VACUUM**. É muito importante do ponto de vista de performance do banco de dados ter um valor preciso de estatísticas, caso contrário escolhas ruins de planos de execução podem ser feitas pelo PostgreSQL degradando assim a performance do banco.

Assim como o vacuum atua na recuperação de espaço, as frequentes atualizações das estatísticas são mais úteis para as tabelas que sofrem pesadas atualizações. Uma simples regra é pensar em quanto os valores mínimos e máximos das colunas mudam em uma tabelas.

Por exemplo, uma coluna do tipo `timestamp` que contém o tempo levado de atualização de linhas em uma tabela terá um valor máximo em constante crescimento quanto mais linhas são inseridas e atualizadas nessa tabela; uma coluna tipo essa irá provávelmete necessitar mais atualizar suas estatísticas que, digamos, uma coluna que contenha as URLs de páginas acessadas em um website. A coluna de URLs pode receber alterações tão frequentes quanto a outra, mas a distribuição estatística dos seus valores provávelmente irão variar muito lentamente.

É possível rodar o **ANALYZE** em tabelas específicas e até em colunas específicas de uma tabela. Essa flexibilidade existe em poder se atualizar algumas estatísticas mais frequentemente que outras em caso de sua aplicação o requerer. Na prática, entretanto, a utilidade dessa característica é um tanto duvidável. Iniciando no PostgreSQL 7.2, o **ANALYZE** é uma rápida operação mesmo em grandes tabelas, isso porque ele usa uma forma randômica de exemplificação das linhas de uma tabela ao invés de ler linha por linha. É provávelmente muito mais simples rodar o comando em todo o banco de dados de vez em quando.

A prática recomendada para a maioria dos servidores é a de agendar um **ANALYZE** ao nível de todo o banco de dados uma vez por dia quando o servidor estiver num baixo nível de acessos; complementado por um **VACUUM** rodado à noite.

Manutenção do Arquivo de Log

É uma boa idéia salvar o log de saída do servidor em algum lugar, melhor do que simplesmente o jogar no `/dev/null`. O log de saída tem um valor incalculável quando você tiver que diagnosticar algum problema. Entretanto, o log tende a ser bem volumoso (especialmente quando o nível solicitado de debug do banco é alto) e você não quer salvá-lo indefinidamente. Você precisa "rodar" os arquivos de log de modo que novos arquivos sejam startados e os velhos sejam jogados fora de vez em quando.

Você simplesmente direciona o stderr do postmaster para um arquivo, o único jeito de truncar um arquivo de log é parar e restartar o postmaster. Isso está bom para os setups de desenvolvimento, mas em um servidor de produção isso não é uma boa idéia.

O jeito mais simples de gerenciar os logs de saída para um servidor em produção é enviá-lo para o syslog e deixar com que ele lide com a rotação dos logs. Para fazer isso, tenha certeza de que o PostgreSQL foi construído com a opção de configuração `--enable-syslog`, e o `syslog` setado para 2 (log direcionado para o syslog) no `postgresql.conf`. Você pode então enviar um sinal `SIGHUP` para o daemon syslog daemon sempre que quiser forçá-lo a começar um novo arquivo de log.

Em muitos sistemas, entretanto, o syslog não é muito confiável, particularmente com grandes mensagens de log; ele pode truncar ou até dropar mensagens exatamente quando você as precisar. Você pode achar muito mais útil usar o pipe juntamente com o postmaster para jogar o stderr diretamente para um tipo de script que controle a rotatividade dos seus logs. Se você startar o postmaster com o pg_ctl, então o stderr do postmaster é redirecionado diretamente para o stdout, então você apenas necessita do comando pipe :

```
pg_ctl start | logrotate
```

A distribuição do PostgreSQL não inclui um programa de rotação de logs, mas existem muitos deles disponíveis na internet; existe um que é incluído na distribuição do Apache, por exemplo.

Resumo da Unidade

- O vacuum é fundamental para a rotina diária do PostgreSQL.
- Recuperando espaço em disco.
- Mensagens de log do banco de dados podem facilitar seu dia a dia.

Revisão da Unidade

1. Como liberamos espaço em disco ?

2. Qual a função do “VACUUM ANALYZE” ?

3. A estrutura de “ROLLBACK” do PostgreSQL é segura ?

Laboratório 7:
Manutenção do Banco de Dados

Acompanhe o instrutor na execução do laboratório.

Unidade 8:
Monitorando a atividade do banco de dados

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Monitorar as atividade do PostgreSQL em tempo de execução e analizar sua performance.

Tópicos da Unidade

- Localizando leituras seqüênciais em tabelas do banco de dados
- Removendo índices que não tiveram acessos
- Ligando o coletor de informações

Considerações Gerais

Um administrador de banco de dados frequentemente se pergunta “o que será que o sistema está fazendo neste exato momento ?”

Esse capítulo mostra como se descobrir isso.

Várias ferramentes estão disponíveis para o monitoramento da atividade do banco de dados e analizar sua performance. A maioria desse capítulo é devotado em descrever o coletor de estatísticas do PostgreSQL, mas também não podemos esquecer de regularmente monitorar a atividade do sistema usando as ferramentas regulares do Unix, tipo **ps** e **top**. Também, uma vez identificada uma query com pobre performance, posterior investigação pode ser necessária usando-se o comando do PostgreSQL : **EXPLAIN**.

Ferramentas standard do Unix

Na maioria das plataformas, o PostgreSQL modifica seu título quando em execução como aqui mostrado pelo comando **ps**, isso para que processos servidores individuais possam ser rápidamente identificados. Um exemplo disso seria

```
$ ps auxww | grep ^postgres
postgres 960 0.0 1.1 6104 1480 pts/1 SN 13:17 0:00 postmaster
-i
postgres 963 0.0 1.1 7084 1472 pts/1 SN 13:17 0:00 postgres:
stats buffer process
postgres 965 0.0 1.1 6152 1512 pts/1 SN 13:17 0:00 postgres:
stats collector
process
postgres 998 0.0 2.3 6532 2992 pts/1 SN 13:18 0:00 postgres:
tgl runbug
127.0.0.1 idle
postgres 1003 0.0 2.4 6532 3128 pts/1 SN 13:19 0:00 postgres:
tgl regression
[local] SELECT waiting
postgres 1016 0.1 2.4 6532 3080 pts/1 SN 13:19 0:00 postgres:
tgl regression
[local] idle in transaction
```

(Esse exemplo foi tirado de um sistema Linux.)

O primeiro processo listado aqui é o do *postmaster*, o processo master do servidor. Os argumentos do comando mostrados para ele são exatamente os que lhe foi dado quando foi executado.

Os próximos dois processos implementam os coletores de estatísticas, os quais descreveremos na próxima seção. (Eles não estarão presentes se você não setou o sistema para startar o coletor de estatísticas.) Cada um dos processos servidores restantes está atendendo a conexão de um cliente. Cada processo seta seu próprio display na forma

```
postgres: usuário banco_de_dados host atividade
```

O usuário, banco de dados, e host permanecem os mesmos durante a vida da conexão do cliente, mas o indicador de atividade pode mudar. A atividade pode ser `idle` (isto é,

esperando por um comando do cliente), `idle in transaction` (esperando pelo cliente dentro de um bloco BEGIN), ou até um nome de comando tipo `SELECT`.

Caso o servidor esteja nesse momento esperando por causa de um lock de outro processo servidor, a palavra `waiting` vai aparecer. No exemplo acima podemos perceber que o processo 1003 está esperando o processo 1016 completar sua transação e portanto liberar seu lock ou algo parecido.

O coletor de estatísticas

O coletor de estatísticas do PostgreSQL é um subsistema que suporta a coleção e gera reporta informações sobre as atividades do servidor. Atualmente, o coletor pode contar acessos a tabelas e indices em ambos os termos : blocos de disco e linhas individuais. Ele também suporta determinar uma query sendo executada por outro processo servidor.

Configuração do coletor

Desde que a coleta de estatísticas adiciona um extra esforço para a execução de queries, o sistema pode ser configurado para coletar ou não coletar informações. Isso é controlado pela configuração de variáveis que são normalmente setadas no `postgresql.conf`.

A variável `STATS_START_COLLECTOR` deve ser setada como `true` para que o coletor de estatísticas possa ser iniciado. Esse é o default e o atualmente recomendado, mas ele pode ser desligado se você não tem interesse nas estatísticas ou você quer diminuir ao máximo qualquer esforço extra do seu servidor. (Entretando, você não ganhará muita coisa fazendo isso.)

Note que você não pode mudar essa configuração com o servidor rodando.

As variáveis `STATS_COMMAND_STRING`, `STATS_BLOCK_LEVEL`, e `STATS_ROW_LEVEL` controlam o quanto de informação é realmente enviado ao coletor, e portanto determinam o quanto a mais de esforço seu servidor estará fazendo para coletar as estatísticas.

Eles respectivamente determinam quando um processo servidor envia sua string de comando, sua estatística de acesso a nível de blocos, e sua estatística de acesso a nível de linhas para o coletor. Normalmente essas variáveis são setadas no `postgresql.conf` para que elas possam ser aplicadas em todos os precessos servidores, mas é possível ligá-las e desligá-las em processos individuais usando o comando **SET**. (Para prevenir que usuários normais escondam suas atividades do administrador, apenas os superusuários tem permissão de usar essas variáveis com o comando **SET**.)

Importante : As variáveis `STATS_COMMAND_STRING, STATS_BLOCK_LEVEL, e STATS_ROW_LEVEL`

tem seu default setado como `false`! Você deve setá-las como `true` Ypara começar realmente a coleta de

estatísticas pelo sistema.

Vendo as estatísticas coletadas

Existem várias views pré-definidas disponíveis para mostrar os resultados das estatísticas coletadas. Alternativamente, você pode montar views customizadas usando as funções de estatísticas subjacentes.

Quando você monitorar as estatísticas da atividade corrente, é importante saber que a informação não é atualizada instantaneamente. Cada processo servidor individualmente transmite novos contadores de acessos ao coletor antes de ficar esperando por outro comando do cliente; isso quer dizer que uma query ainda em progresso não afetará os totais mostrados.

Também, o próprio coletor emite novos totais a cada PGSTAT_STAT_INTERVAL (500 milisegundos por default). Os totais mostrados então ficam um pouquinho atrás da atividade atual.

Outro ponto importante é que quando um processo servidor é requisitado a informar qualquer estatística sua, ele primeiro lê os totais mais recentes emitidos pelo processo coletor. Depois então ele continua a usar esse snapshot para todas as views de estatística e funções até o fim da sua transação corrente. Pode parecer então que as estatísticas não estão mudando enquanto você continua a transação. Isso é uma característica, não um bug, pois isso permite que você faça várias queries nas estatísticas e cruze os resultados sem se preocupar se os números estão mudando abaixo de você. Mas se você quiser ver novos resultado a cada query, tenha certeza de rodá-las fora que qualquer bloco de transação.

Views de estatísticas standard

Descrição do nome da view

`pg_stat_activity`

Uma linha por processo servidor, mostrando o PID do processo, banco de dados, usuário, e a query corrente. A coluna da query corrente é apenas disponível aos superusuários; para os outros o valor será NULL.

(Note que por causa do delay de retorno do coletor, a query corrente só será atual se ela for de longa duração.)

`pg_stat_database`

Uma linha por banco de dados, mostrando o número de backends ativos, o total de transações comitadas e o total de rollback desse banco de dados, o total de blocos lidos do disco, e o número total de buffer hits (isto é, requisição de leitura de blocos rejeitada pois o bloco foi achado no buffer cache).

`pg_stat_all_tables`

Para cada tabela no banco de dados corrente, número total de scans sequenciais e de índices, número total de tuplas retornadas por cada tipo de scan, e os totais de inserções, updates, e deletes de tuplas.

`pg_stat_sys_tables`

O mesmo que pg_stat_all_tables, exceto que apenas as tabelas de sistemas são mostradas.

`pg_stat_user_tables`

O mesmo que pg_stat_all_tables, exceto que apenas as tabelas de usuários são mostradas.

`pg_stat_all_indexes`

Para cada índice do banco de dados corrente, o número total de scans que usaram aquele

índice, o número de tuplas do índice lidas.

`pg_stat_sys_indexes`

O mesmo que pg_stat_all_indexes, exceto que apenas os índices das tabelas de sistema são mostrados.

`pg_stat_user_indexes`

O mesmo que pg_stat_all_indexes, exceto que apenas os índices das tabelas de usuários são mostrados.

`pg_statio_all_tables`

Para cada tabelas do banco de dados corrente, o número total de blocos de disco lidos daquela tabela, o número de buffer hits, o número total de blocos de disco lidos e o número de buffer hits para cada índice daquela tabela.

`pg_statio_user_tables`

O mesmo que pg_statio_all_tables, exceto que apenas as tabelas de usuários serão mostradas.

`pg_statio_all_indexes`

Para cada índice do banco de dados corrente, o número de blocos de disco lidos e os buffer hits daquele índice.

`pg_statio_user_indexes`

O mesmo que pg_statio_all_indexes, exceto que apenas os índices das tabelas dos usuários serão mostrados.

Resumo da Unidade

- O domínio do sistema operacional é fundamental para um bom DBA.
- Analisando as estatísticas coletadas pelo banco de dados.
- Cobrando soluções dos desenvolvedores de sistemas.

Revisão da Unidade

1. O que fazer para coletar estatisticas de acessos as tabelas do banco de dados ?

2. Qual visão do banco de dados nos mostra as tabelas com acesso seqüenciais e o número de acessos ?

3. Onde verificamos os índices criados e não utilizados ?

Laboratório 8:
Monitorando a Atividade do Banco de Dados

Acompanhe o instrutor na execução do laboratório.

Unidade 9:
Write-Ahead Logging (WAL)

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Descobrir os benefícios de uso do WAL, o modo como o PostgreSQL gerencia seus logs de transação.

Tópicos da Unidade

- O backup incremental vai ser implementado
- Os redologs do PostgreSQL

Considerações Gerais

Write Ahead Logging (WAL) é uma abordagem standard para logs de transações. Sua descrição detalhada pode ser encontrada na maioria dos (se não em todos os) livros sobre processamento de transações. Em síntese, o conceito central do WAL é que as mudanças dos arquivos de dados (onde as tabelas e índices residem) devem ser escritas apenas após essas mudanças terem sido arquivadas ou logadas – isto é, quando os registros de log tiverem sido descarregados para um armazenamento permanente. Quando nós seguimos esse procedimento, nós não precisamos descarregar páginas de dados para o disco em cada commit de transação, porque nós sabemos que no caso de uma falha, nós estaremos habilitados a recuperar os banco de dados usando os logs : quaisquer mudanças que não tiverem sido aplicadas nas páginas de dados serão primeiramente refeitas nos registros de log (essa é a chamada recuperação roll-forward, também conhecida como REDO) e então as mudanças feitas por transações não committadas serão removidas das páginas de dados (recuperação roll-backward, ou UNDO).

Benefícios imediatos do WAL

O primeiro benefício óbvio de se usar o WAL é a redução significante do número de gravações em disco, já que apenas os arquivos de log necessitam ser descarregados no disco na hora do commit de uma transação; em ambientes multiusuário, os commits de várias transações podem ser completados com apenas um único `fsync()` do arquivo de log. Além disso, o arquivo de log é escrito sequencialmente, então o custo de sincrinização do log é muito menor do que o custo de descarregar uma página de dados.

O próximo benefício é a consistência das páginas de dados. A verdade é que, sem o WAL, o PostgreSQL nunca foi capaz de garantir essa consistência no caso de uma falha. Antes do WAL, qualquer falha durante uma escrita poderia resultar em :

1. tuplas de índices apontando para linhas inexistentes de tabelas
2. tuples de índices duplicadas
3. total corrupção de tabelas ou índices, por causa de escritas parciais das páginas de dados

Os problemas com os índices (problemas 1 e 2) poderiam ser possivelmete acertados com chamadas adicionais do `fsync()`, mas não é tão óbvio lidar com o último caso sem o WAL.

O WAL salva o conteúdo inteiro da página de dados no log se isso for requerido para garantir consistência de páginas para recuperações de falhas.

Benefícios Futuros

Nesse primeiro release do WAL, as operações de UNDO não estão implementadas, devido à falta de tempo. Isso significa que as mudanças feitas por transações abortadas irão ainda ocupar espaço em disco e que iremos ainda necessitar de um arquivo permanente `pg_clog` para guardar o status das transações, já que nós não somos capazes de reutilizar os identificadores das transações.

Uma vez implementado o UNDO, o arquivo `pg_clog` não será mais necessário permanentemente; será possível remover o `pg_clog` a cada shutdown. Com o UNDO, será também possível implementar *savepoints* para o rollback parcial de transações inválidas com a habilidade de se continuar ou commitar operações válidas feitas pela transação antes do erro.

Atualmente, qualquer erro irá invalidar toda a transação e ela será totalmente abortada.

O WAL oferece a oportunidade de um novo método para on-line backup e restore (BAR) do banco de dados. Para usar esse método, alguém teria que salvar os arquivos de dados periodicamente em outro disco, tape ou outro host e ainda arquivar os arquivos de log do WAL. As cópias dos arquivos de dados e os arquivos de log arquivados poderiam ser usados para restaurar o banco de dados. Cada vez que uma nova cópia dos arquivos de dados fosse feita, os velhos arquivos de log poderiam ser removidos. A implementação desse método irá necessitar do log dos arquivos de dados e da criação e deleção de índices; isso irá também necessitar o desenvolvimento de um método para a cópia dos arquivos de dados (os comandos de cópia do sistema operacional não serviriam para isso).

Uma dificuldade que fica no caminho desses benefícios é que eles requerem o arquivamento das entradas do WAL por consideraveis períodos de tempo.

O presente formato do WAL é extremamente volumoso já que ele inclui muitos snapshots das páginas de dados em disco. Isso não é uma séria preocupação no momento, já que esses snapshots precisam ser mantidos por um ou dois intervalos de checkpoints; mas para se atingir todos esses futuros benefícios algum tipo de WAL compactado será necessário.

Implementação

O WAL é automaticamente habilitado apartir do release 7.1 em diante. Nenhuma ação pe necessária por parte do administrador com a exceção de garantir espaço adicional em disco para cumprir com os requirementos dos logs do WAL, e que qualquer tipo de tuning seja feito caso necessário (veja a seção de configuração).

Os logs do WAL são gravados no diretório *$PGDATA*/pg_xlog, em um conunto de arquivos segmentados, cada um com 16 MB de tamanho.

Cada segmento é dividido em páginas de 8 kB. Os headers do registro de log são descritos no access/xlog.h; o conteúdo de um registro é dependente do tipo de evento que está sendo logado. Os nomes dos segmento são dados em números crescentes, começando com 0000000000000000. Os números não reiniciam novamente, pois isso levaria muito tempo até ter esse limite alcançado.

Os buffers do WAL e sua estrutura de controle estão em memória compartilhada, e são manuseados pelos processos backend; eles são protegidos por leves locks.

O tamanho default de um buffer do WAL é de 8 buffers de 8 kB cada, ou 64 kB no total.

Seria uma vantagem se os logs fossem localizados em outro disco. Isso pode ser feito movendo-se o diretório pg_xlog para outra localização (faça isso com o postmaster em shutdown) e criando-se um link simbólico partindo da sua localização original no *$PGDATA* para a sua nova localização.

A meta do WAL, para garantir que o log seja escrito antes que os registros do banco de dados sejam alterados, pode ser subvertida por drives de discos que falsamente reportam uma gravação bem sucedida para o kernel, quando, em fato, eles permanecem ainda em cache e ainda não foram realmente armazenados no disco. Uma falha de energia numa situação dessas poderia levar o banco de dados a uma irrecuperavel corrupção de dados. Os administradores deveriam tentar garantir que os discos dos seus servidores não geram reports falsos como esses.

Recovery do Banco de Dados com o WAL

Após um checkpoint ter sido feito e o log descarregado, a posição do checkpoint é salva no arquivo `pg_control`. Portanto, quando um recovery for necessário, o backend primeiro lê o `pg_control` e depois o registro do checkpoint; então ele executa a operação de REDO escaneando a posição indicada pelo registro de checkpoint dali para a frente. Porque o conteúdo inteiro das páginas de dados está salva no log na primeira modificação de página feita após um checkpoint, todas as páginas modificadas desde aquele checkpoint serão restauradas para um estado consistente.

Usando o `pg_control` para pegar a posição do checkpoint acelera o processo de recovery, mas para manusear a possível corrupção do `pg_control`, nós na verdade implementaríamos uma leitura dos logs existentes numa ordem reversa – mais novo até o mais velho – em ordem de descobrirmos o íltimo checkpoint. Isto não está implementado, ainda.

Configuração do WAL

Existem vários parâmetros relativos ao WAL que afetam a performance do banco de dados. Existem duas funções comumente usadas do WAL: `LogInsert` e `LogFlush`. `LogInsert` é usado para colocar um novo registro dentro dos buffers do WAL na memória compartilhada. Se não há espaço para um novo registro, `LogInsert` terá que escrever (mover para o cache do kernel) alguns buffers já preenchidos. Isso não é desejável porque o `LogInsert` é usado em cada modificação de baixo nível do banco de dados (por exemplo, a inserção de uma tupla) de cada vez quando um lock exclusivo é seguro nas páginas de dados afetadas, então essa operação necessita ser a mais rápida possível. O que é pior, escrever buffers do WAL pode também forçar a criação de um novo segmento de log, o qual leva mais tempo ainda. Normalmente, os buffers do WAL deveriam ser escritos e descarregados por uma requisição de `LogFlush`, a qual é feita, na sua maioria, na hora do commit da transação para garantir que os registros são descarregados para uma área permanente de armazenamento.

Em sistemas com uma alta saída de logs, as requisições `LogFlush` podem não ocorrer em frequência suficiente para prevenir que os WAL buffers sejam escritos pelo `LogInsert`. Nesse tipo de sistemas alguém deveria aumentar o número de WAL buffers modificando o parâmetro `WAL_BUFFERS`. O número default de WAL buffers é 8. O aumento desse valor irá também aumentar o uso de memória compartilhada.

Os c*heckpoints* são pontos na sequência das transações no qual é garantido que os arquivos de dados foram alterados com toda a informação também salva em logs antes do checkpoint. Na hora do checkpoint, todas as páginas de dados "sujas" são descarregadas no disco e um registro especial de checkpoint é escrito no arquivo de log. Como resultado, no evento de uma falha, o recuperador sabe de que registro do log (conhecido como registro redo) ele deveria começar a operação de REDO, já que todas as mudanças feitas nos arquivos de dados antes daquele registro já estão armazenadas em disco. Após um checkpoint ter sido feito, qualquer segmento de log escrito antes dos registros de undo não são mais necessários e podem ser reciclados ou removidos.

O processo responsável pelos checkpoints é capaz de criar novos segmentos de log para uso futuro, isso para evitar que o `LogInsert` ou o `LogFlush` percam tempo fazendo isso. Alguém pode instruir o servidor a pré-criar até 64 segmentos de logs modificando o parâmetro de configuração `WAL_FILES`.

O postmaster cria um processo backend especial para criaar o próximo checkpoint. Um checkpoint é criado a cada `CHECKPOINT_SEGMENTS`, ou a cada `CHECKPOINT_TIMEOUT`, qualquer um que aconteça antes. O default é 3 segmentos e 300 segundos respectivamente. É possível também forçar um checkpoint usando o comando SQL **CHECKPOINT**.

Reduzindo-se os parâmetros `CHECKPOINT_SEGMENTS` e/ou `CHECKPOINT_TIMEOUT` causará checkpoints feitos mais frequentemente. Isso permite maior rapidez para uma recovery após uma falha (já que menos trabalho necessitará ser refeito). Uma redução do intervalo do checkpoint aumentará o volume de saída para os logs, causando com isso mais I/O em disco.

O parâmetro `COMMIT_DELAY` define em quantos microsegundos o backend irá dormir após escrever um registro commitado no log com `LogInsert` antes de executar um `LogFlush`. Essa demora permite aos outros backends adicionar os seus registros commitados no log também, os fazendo todos serem descarregados em um simples log sync.

Resumo da Unidade

- O WAL é fundamental para o futuro do PostgreSQL.
- O recovery automático já está implementdo.

Revisão da Unidade

1. Qual vai ser a maior finalidade do WAL ?

Laboratório 9:
Write-Ahead Logging

Acompanhe o instrutor na execução do laboratório.

Unidade 10:

Falhas de Bancos de Dados

Objetivos da Unidade

Após completar essa unidade, você será capaz de:

- Advertir o administrador do PostgreSQL da possibilidade de falhas de bancos de dados e suas possíveis causas.

Tópicos da Unidade

- Verificando o espaço em disco
- A perda de um disco pode ainda comprometer toda a sua empresa
- A futura aplicação de logs armazenados

Considerações Gerais

Falhas de bancos de dados (ou sua possibilidade) devem ser vistas como uma emboscada, pronta para nos atacar em algum lugar no futuro. Um administrador de bancos de dados prudente irá se preparar para a inevitabilidade de falhas de todos os tipos, e terá planos e procedimentos apropriados *antes* que a falha ocorra.

O recovery do banco de dados é necessário em casos de falhas de hardware ou de software. Existem várias categorias de falhas; algumas delas requerem relativamente poucos ajustes no banco de dados, enquanto outras podem depender da existencia de dumps previamente preparados. O importante a ser enfatizado é que os seus dados são importantes e/ou difíceis de regerar, então você deve realmente estar preparado para vários cenários de falhas.

Disco cheio

Um disco de dados cheio pode resultar uma subsequente corrupção de índices, mas não das tabelas de dados fundamentais. Se os arquivo WAL estão no mesmo disco (como é a configuração default) então um disco cheio durante a inicialização do banco de dados pode resultar em arquivos WAL corruptos ou incompletos. Essa condição de falha é detectada e o banco de dados se recusará a startar.Você deve liberar algum espaço adicional no disco (ou mover a área WAL para outro disco; veja a seção de configuração do WAL) e então restartar o postmaster para se recuperar dessa condição.

Falha de disco

A falha de qualquer disco (ou a de um dispositivo lógico de armazenamento, tipo um subsistema RAID) envolvido com o banco de dados ativo irá requerer que o banco seja recuperado apartir de um dump previamente preparado. Esse dump deve ter sido preparado usando-se o pg_dumpall, e saiba que os updates no banco de dados ocorridos após esse dump serão perdidos.

Resumo da Unidade

- O uso de RAID de segurança é fundamental no uso do PostgreSQL.
- O DBA deve sempre acompanhar e dimensionar os discos do servidor de banco de dados.
- Uma máquina mal dimensionada pode comprometer o dia a dia de sua empresa.

Revisão da Unidade

1. Que tipo de RAID é totalmente inviável para uso com PostgreSQL ?

2. Como verificar o uso dos discos no sistema operacional ?

Laboratório 10:
Falhas de Bancos de Dados

Acompanhe o instrutor na execução do laboratório.

Unidade 11:

Noções básicas de otimização

Objetivos da Unidade

Após completar esta unidade, você será capaz de:

- Identificar os principais parâmetros de otimização .
- Definir valores para os parâmetros de memória .
- Analisar o plano de execução de uma consulta.

Tópicos da Unidade

- Parâmetros fundamentais
- Usando o comando EXPLAIN em uma consulta

Parâmetros de otimização

No PostgreSQL existem alguns parâmetros que devem ser alterados para que seu banco de dados tenha a performace desejável. Assim como outros sistemas de banco de dados a segurança da informação anda na contra mão da performace.

O arquivo que define o valor dos parâmetros é o postgresql.conf e vai ser interpretado no momento em que colocamos o banco de dados no ar ou quando o DBA executar um reload no banco de dados.

Os 3 principais parâmetros de otimização do PostgreSQL são shared_buffers, sort_mem e fsync. O parâmetro sort_mem não necessita da reinicializacao do banco de dados , somente de um reload, já os outros dois parâmetros necessitam que o banco seja parado e iniciado novamente.

Abaixo segue a definição dos mesmos.

Shared Buffers

Este parâmetro define o volume de memória que vai ser compartilhada por todas as seções abertas no banco de dados. Por exemplo, você executou uma consulta que leu 200 blocos do disco e sua memória compartilhada era de 400 blocos. Agora sua memória foi populada com os 200 blocos lidos do disco. Se um colega seu necessitar em uma outra consulta dos mesmos blocos que você selecionou, o PostgreSQL já possui os blocos em memória e não vai ao disco novamente.

Um banco com baixo valor de blocos compartilhados vai ter sempre de ler a informação do disco e assim terá uma baixa performace.

O valor deste parâmetro vai sempre ser definido em função do volume de memória RAM disponível em seu servidor de banco de dados.

A unidade de definição do parâmetro deve ser em blocos de 8KB. Quando o seu volume de memória compartilhada poder ser elevado, você deve configurar seu sistema operacional para que permita que o processo POSTMATER tenha acesso a essa memória.

Sort Mem

Este parâmetro define o volume de memória que uma seção vai ter para poder fazer sort (ordenação) de dados. Quando executamos uma consulta e pedimos para o banco de dados ordenar a mesma (order by,group by) o banco necessita de memória para fazer esta ordenação. Se definirmos um sort_mem de 1024 KB e nossa consulta tiver 2048Kb de tamanho, o banco então faz a ordenação em disco pois a memória não era suficiente. Se tivermos 50 seções abertas no banco de dados e um sort_mem de 1048 KB, o postgres estará usando 50MB de memória no servidor. Este parâmetro vai ser definido em função

da memória RAM disponível no servidor e do numero de seções concorrentes no banco de dados.

Fsync

Este parâmetro forca ou não a atualização dos arquivos de dados no disco. Se trabalharmos com fsync=false, teremos uma melhor performace pois o disco vai ser atualizado de tempo em tempo e não no momento do final da transação. Mas por outro lado corremos risco de perder informação em uma eventual queda de luz por exemplo.

O comando EXPLAIN

Sempre que tivermos de colocar código SQL em nossas aplicações , devemos sempre verificar qual o plano de execução que o banco de dados vai utilizar. Um código SQL mal escrito pode comprometer a performace do banco de dados.

O PostgreSQL tem no comando EXPLAIN o aliado para o programador. Com ele o programador pode analisar como o banco de dados vai se comportar quando o código for executado.

Uma leitura seqüencial em uma tabela grande pode levar muito tempo para ser executada e deixar o servidor muito ocupado com esse processamento. No exemplo abaixo o comando EXPLAIN nos mostra que a leitura da tabela de notas_fiscais sera executado em modo seqüencial. Para evitar este tipo de acesso um índice pode ser necessário.

curso=# explain select count(*) from notas_fiscais where oid_lote=123::char

QUERY PLAN

Aggregate (cost=12769.84..12769.84 rows=1 width=0)

-> **Seq Scan** on notas_fiscais (cost=0.00..12766.42 rows=1369 width=0)

Filter: ((oid_lote)::text = '1'::text)

Cuidado com a conversão do tipo de dado, no exemplo acima a coluna oid_lote possui um índice, mas o mesmo não foi utilizado em função do tipo de conversão definida.

Se executarmos a mesma consulta e fizermos a conversão corretamente o banco então faz uso do índice definido na coluna.

curso=# explain select count(*) from notas_fiscais where oid_lote=123::numeric

QUERY PLAN

Aggregate (cost=5117.12..5117.12 rows=1 width=0)

-> **Index Scan** using oid_lote_nota_fiscal_idx on notas_fiscais (cost=0.00..5113.70 rows=1369 width=0)

Index Cond: (oid_lote = 123::numeric)

Resumo da Unidade

- Os parâmetros definem a forma como o banco de dados deve trabalhar.
- O uso de memória é fundamental para uma excelente performace.
- Código SQL deve ser sempre analisado antes de fazer parte de uma aplicação.Você pode facilmente converter qualquer objeto em símbolo para uso em um botão.

Revisão da Unidade

1. Qual parâmetro define o volume de memória compartilhada ?

2. Qual a função do sort_mem ?

3. Como analiso o plano de execução de uma consulta ?

Laboratório 11:
Noções básicas de otimização

Acompanhe o instrutor na execução do laboratório.

Unidade 12: Tablespaces

Objetivos da Unidade

Após completar esta unidade, você será capaz de:

• Criar bases e objetos de bases de dados em discos/diretórios diferentes

Tablespaces são estruturas que permitem direcionar a gravação de dados em volumes ou diretórios diferentes do padrão do servidor PostgreSQL.

Para criar uma tablespace, utiliza-se o comando **CREATE TABLESPACE**, como exemplificado abaixo:

CREATE TABLESPACE *nome_da_tablespace* [OWNER *usuario*] LOCATION '*diretório*';

Para utilizar uma tablespace específica, utilizamos o parâmetro **TABLESPACE** ao criarmos nossos objetos, como no exemplo abaixo:

CREATE TABLE *nome_da_tabela* (*definições de colunas*) TABLESPACE *nome_da_tablespace*;

Resumo da Unidade

• Tablespaces definem onde os dados de uma base serão gravados fisicamente.
• Para criar uma tablespace utilizamos o comando **CREATE TABLESPACE**, enquanto que para utilizá-la usamos o parâmetro **TABLESPACE**.

Manual de Referência do PostgreSQL 7.4

The PostgreSQL Global Development Group