PostgreSQL

18 de abril de 2007

Sumário

Parte I

Sobre essa Apostila

Conteúdo

O conteúdo dessa apostila é fruto da compilação de diversos materiais livres publicados na internet, disponíveis em diversos sites ou originalmente produzido no CDTC em *http://www.cdtc.org.br*.

O formato original deste material bem como sua atualização está disponível dentro da licença *GNU Free Documentation License*, cujo teor integral encontra-se aqui reproduzido na seção de mesmo nome, tendo inclusive uma versão traduzida (não oficial).

A revisão e alteração vem sendo realizada pelo CDTC (*suporte@cdtc.org.br*) desde outubro de 2006. Críticas e sugestões construtivas são bem-vindas a qualquer tempo.

Autores

A autoria deste é de responsabilidade de Fabio Hitsuki Nitto .

O texto original faz parte do projeto Centro de Difusão de Tecnologia e Conhecimento, que vem sendo realizado pelo ITI (Instituto Nacional de Tecnologia da Informação) em conjunto com outros parceiros institucionais, atuando em conjunto com as universidades federais brasileiras que tem produzido e utilizado Software Livre, apoiando inclusive a comunidade Free Software junto a outras entidades no país.

Informações adicionais podem ser obtidas através do email *ouvidoria@cdtc.org.br*, ou da *home page* da entidade, através da URL **http://www.cdtc.org.br**.

Garantias

O material contido nesta apostila é isento de garantias e o seu uso é de inteira responsabilidade do usuário/leitor. Os autores, bem como o ITI e seus parceiros, não se responsabilizam direta ou indiretamente por qualquer prejuízo oriundo da utilização do material aqui contido.

Licença

Parte II

Informações Básicas

Sobre o CDTC

Objetivo Geral

O Projeto CDTC visa a promoção e o desenvolvimento de ações que incentivem a disseminação de soluções que utilizem padrões abertos e não proprietários de tecnologia, em proveito do desenvolvimento social, cultural, político, tecnológico e econômico da sociedade brasileira.

Objetivo Específico

Auxiliar o Governo Federal na implantação do plano nacional de software não-proprietário e de código fonte aberto, identificando e mobilizando grupos de formadores de opinião dentre os servidores públicos e agentes políticos da União Federal, estimulando e incentivando o mercado nacional a adotar novos modelos de negócio da tecnologia da informação e de novos negócios de comunicação com base em software não-proprietário e de código fonte aberto, oferecendo treinamento específico para técnicos, profissionais de suporte e funcionários públicos usuários, criando grupos de funcionários públicos que irão treinar outros funcionários públicos e atuar como incentivadores e defensores de produtos de software não proprietários e código fonte aberto, oferecendo conteúdo técnico on-line para serviços de suporte, ferramentas para desenvolvimento de produtos de software não proprietários e de seu código fonte livre, articulando redes de terceiros (dentro e fora do governo) fornecedoras de educação, pesquisa, desenvolvimento e teste de produtos de software livre.

Guia do aluno

Neste guia, você terá reunidas uma série de informações importantes para que você comece seu curso. São elas:

- Licenças para cópia de material disponível
- Os 10 mandamentos do aluno de Educação a Distância
- Como participar dos foruns e da wikipédia
- Primeiros passos

É muito importante que você entre em contato com TODAS estas informações, seguindo o roteiro acima.

Licença

Os 10 mandamentos do aluno de educação online

- 1. Acesso à Internet: ter endereço eletrônico, um provedor e um equipamento adequado é pré-requisito para a participação nos cursos a distância.
- 2. Habilidade e disposição para operar programas: ter conhecimentos básicos de Informática é necessário para poder executar as tarefas.
- 3. Vontade para aprender colaborativamente: interagir, ser participativo no ensino a distância conta muitos pontos, pois irá colaborar para o processo ensino-aprendizagem pessoal, dos colegas e dos professores.
- 4. Comportamentos compatíveis com a etiqueta: mostrar-se interessado em conhecer seus colegas de turma respeitando-os e fazendo ser respeitado pelo mesmo.
- 5. Organização pessoal: planejar e organizar tudo é fundamental para facilitar a sua revisão e a sua recuperação de materiais.
- 6. Vontade para realizar as atividades no tempo correto: anotar todas as suas obrigações e realizá-las em tempo real.
- 7. Curiosidade e abertura para inovações: aceitar novas idéias e inovar sempre.
- 8. Flexibilidade e adaptação: requisitos necessário à mudança tecnológica, aprendizagens e descobertas.
- 9. Objetividade em sua comunicação: comunicar-se de forma clara, breve e transparente é ponto - chave na comunicação pela Internet.
- 10. Responsabilidade: ser responsável por seu próprio aprendizado. O ambiente virtual não controla a sua dedicação, mas reflete os resultados do seu esforço e da sua colaboração.

Como participar dos fóruns e Wikipédia

Você tem um problema e precisa de ajuda?

Podemos te ajudar de 2 formas:

A primeira é o uso dos fóruns de notícias e de dúvidas gerais que se distinguem pelo uso:

. O fórum de notícias tem por objetivo disponibilizar um meio de acesso rápido a informações que sejam pertinentes ao curso (avisos, notícias). As mensagens postadas nele são enviadas a

todos participantes. Assim, se o monitor ou algum outro participante tiver uma informação que interesse ao grupo, favor postá-la aqui.
Porém, se o que você deseja é resolver alguma dúvida ou discutir algum tópico específico do curso. É recomendado que você faça uso do Forum de dúvidas gerais que lhe dá recursos mais efetivos para esta prática.

. O fórum de dúvidas gerais tem por objetivo disponibilizar um meio fácil, rápido e interativo para solucionar suas dúvidas e trocar experiências. As mensagens postadas nele são enviadas a todos participantes do curso. Assim, fica muito mais fácil obter respostas, já que todos podem ajudar.
Se você receber uma mensagem com algum tópico que saiba responder, não se preocupe com a formalização ou a gramática. Responda! E não se esqueça de que antes de abrir um novo tópico é recomendável ver se a sua pergunta já foi feita por outro participante.

A segunda forma se dá pelas Wikis:

. Uma wiki é uma página web que pode ser editada colaborativamente, ou seja, qualquer participante pode inserir, editar, apagar textos. As versões antigas vão sendo arquivadas e podem ser recuperadas a qualquer momento que um dos participantes o desejar. Assim, ela oferece um ótimo suporte a processos de aprendizagem colaborativa. A maior wiki na web é o site "Wikipédia", uma experiência grandiosa de construção de uma enciclopédia de forma colaborativa, por pessoas de todas as partes do mundo. Acesse-a em português pelos links:

- Página principal da Wiki - http://pt.wikipedia.org/wiki/

Agradecemos antecipadamente a sua colaboração com a aprendizagem do grupo!

Primeiros Passos

Para uma melhor aprendizagem é recomendável que você siga os seguintes passos:

- Ler o Plano de Ensino e entender a que seu curso se dispõe a ensinar;
- Ler a Ambientação do Moodle para aprender a navegar neste ambiente e se utilizar das ferramentas básicas do mesmo;
- Entrar nas lições seguindo a seqüência descrita no Plano de Ensino;
- Qualquer dúvida, reporte ao Fórum de Dúvidas Gerais.

Perfil do Tutor

Segue-se uma descrição do tutor ideal, baseada no feedback de alunos e de tutores.

O tutor ideal é um modelo de excelência: é consistente, justo e profissional nos respectivos valores e atitudes, incentiva mas é honesto, imparcial, amável, positivo, respeitador, aceita as idéias dos estudantes, é paciente, pessoal, tolerante, apreciativo, compreensivo e pronto a ajudar.

A classificação por um tutor desta natureza proporciona o melhor feedback possível, é crucial, e, para a maior parte dos alunos, constitui o ponto central do processo de aprendizagem.' Este tutor ou instrutor:

- fornece explicações claras acerca do que ele espera, e do estilo de classificação que irá utilizar;
- gosta que lhe façam perguntas adicionais;
- identifica as nossas falhas, mas corrige-as amavelmente', diz um estudante, 'e explica porque motivo a classificação foi ou não foi atribuída';
- tece comentários completos e construtivos, mas de forma agradável (em contraste com um reparo de um estudante: 'os comentários deixam-nos com uma sensação de crítica, de ameaça e de nervossismo')
- dá uma ajuda complementar para encorajar um estudante em dificuldade;
- esclarece pontos que não foram entendidos, ou corretamente aprendidos anteriormente;
- ajuda o estudante a alcançar os seus objetivos;
- é flexível quando necessário;
- mostra um interesse genuíno em motivar os alunos (mesmo os principiantes e, por isso, talvez numa fase menos interessante para o tutor);
- escreve todas as correções de forma legível e com um nível de pormenorização adequado;
- acima de tudo, devolve os trabalhos rapidamente;

Parte III

PostgreSQL

Capítulo 1

O que é o PostgreSQL

O PostgreSQL é um sistema para gerenciamento de banco de dados relacional, ou SGBD, e é desenvolvido sob uma licença nos padrões BSD, servindo como alternativa a outros gerenciadores de banco de dados, livres (MySQL, FireBird) ou proprietários(Oracle, SyBase, Microsoft SQL Server). A licença garante o livre uso, a cópia, distribuição ou modificação do software e de toda a sua documentação, desde que mantida a licença. Leia a licença completa em http://www.postgresql.org/about/licence.
O curso, com base na distribuição Debian possui duas semanas, começa na Segunda-Feira da primeira semana e termina no Domingo da ultima. Todo o conteúdo do curso estará visível somente a partir da data de início. Para começar o curso você deve ler o Guia do aluno a seguir.

Capítulo 2

Plano de ensino

2.1 Objetivo

Qualificar técnicos e programadores na utilização do banco de dados PostgreSQL.

2.2 Público Alvo

Técnicos e Programadores que desejam trabalhar com o banco de dados PostgreSQL.

2.3 Pré-requisitos

Os usuários deverão ser, necessariamente, indicados por empresas públicas e ter conhecimento básico acerca de bancos de dados.

2.4 Descrição

O curso de PostgreSQL será realizado na modalidade EAD e utilizará a plataforma Moodle como ferramenta de aprendizagem. O material didático estará disponível on-line de acordo com as datas pré-estabelecidas no calendário. A versão utilizada para o PostgreSQL será a 8.1

2.5 Metodologia

O curso está dividido da seguinte maneira:

2.6 Cronograma

- **Semana 1**
- Introdução, instalação e configuração;
- **Semana 2**
- Manipulação de tabelas e gerenciamento do banco de dados

As lições, disponíveis em cada módulo, contém o contéudo principal. Elas poderão ser acessadas quantas vezes forem necessárias, desde que esteja dentro da semana programada. Ao final de uma lição, você receberá uma nota de acordo com o seu desempenho. Caso sua nota numa determinada lição for menor do que 6.0, sugerimos que você faça novamente esta lição. // Ao final do curso serão disponibilizadas as avaliações referentes aos módulos estudados anteriormente. Somente as notas das avaliações serão consideradas para a nota final. Todos os módulos ficarão visíveis para que possam ser consultados durante a avaliação final. // Para conhecer as demais atividades de cada módulo leia o tópico seguinte: "Ambientação do Moodle". // Os instrutores estarão a sua disposição ao longo de todo curso. Qualquer dúvida deve ser enviada ao fórum correspondente. Diariamente os monitores darão respostas e esclarecimentos.

2.7 Programa

O curso oferecerá o seguinte conteúdo:

- Semana 1
- Visão Geral
- Instalação e configuração
- Estrutura do PostgreSQL
- Criando, removendo e acessando bancos de dados
- Criando, removendo e manipulando tabelas
- Semana 2
- Acessando dados de tabelas
- Conceitos Avançados
- Administração do servidor

2.8 Avaliação

Toda a avaliação será feita on-line.
Aspectos a serem considerados na avaliação:

- Iniciativa e autonomia no processo de aprendizagem e de produção de conhecimento;
- Capacidade de pesquisa e abordagem criativa na solução dos problemas apresentados.

Instrumentos de avaliação:

- Participação ativa nas atividades programadas.
- Avaliação ao final do curso.
- O participante fará várias avaliações referente ao conteúdo do curso. Para a aprovação e obtenção do certificado o participante deverá obter nota final maior ou igual a 6.0 de acordo com a fórmula abaixo:

- Nota Final = ((ML x 7) + (AF x 3)) / 10 = Média aritmética das lições
- AF = Avaliações

2.9 Bibliografia

- Site official: http://www.postgresql.org
- Guia em Português: http://www.postgresql.org.br

Capítulo 3

Módulo I - Introdução, instalação e configuração

3.1 Visão geral

Hoje o PostgreSQL é considerado um dos SGBD(Sistema de Gerenciamento de Banco de Dados) de código aberto mais avançado. Vejamos suas características.

3.1.1 Um pouco da história do PostgreSQL

A origem do PostgreSQL está ligada ao projeto Ingres da Universidade de Berkeley, na Califórnia. O líder desse projeto era Michael Stonebreaker, um dos pioneiros das bases de dados relacionais. Em 1985, na tentativa de solucionar muitos dos problemas que haviam na época acerca de bancos de dados relacionais, Stonebreaker deu início, na própria Universidade, a um novo projeto post-Ingres, ou em bom português, ?pós-Ingres?, que viria a ser chamado de Postgres.

Pouco tempo depois, em 1993, a Universidade de Berkeley abandonou o projeto Postgres, e este, apoiado pela sua licença BSD, passou a ser sustentado pela comunidade. Em 1995, foi adicionado suporte à linguagem SQL, substituindo a antiga QUEL (linguagem utilizada no Ingres) e o projeto passou a se chamar, inicialmente, Postgres95 e logo depois de PostgreSQL.

Assim, hoje, como muitos outros softwares livres, o PostgreSQL não é desenvolvido por uma só empresa, mas sim mantido por uma comunidade global de desenvolvedores e de empresas.

3.1.2 Características do PostgreSQL

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (SGBDOR),baseado no POSTGRES versão 4.2 desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O POSTGRES foi pioneiro em vários conceitos que somente se tornaram disponíveis muito mais tarde em alguns sistemas de banco de dados comerciais.

O PostgreSQL é um descendente de código fonte aberto deste código original de Berkeley. É suportada grande parte do padrão SQL:2003, além de serem oferecidas muitas funcionalidades modernas, como:

- comandos complexos
- chaves estrangeiras
- gatilhos
- visões
- integridade transacional
- controle de simultaneidade multiversão (MVCC)

Além disso, o PostgreSQL pode ser estendido pelo usuário de muitas maneiras como, por exemplo, adicionando novos:

- tipos de dado
- funções
- operadores
- funções de agregação
- métodos de índice
- linguagens procedurais

Devido à sua licença liberal, o PostgreSQL pode ser utilizado, modificado e distribuído por qualquer pessoa para qualquer finalidade, seja privada, comercial ou acadêmica, livre de encargos.
Atualmente na versão 8.1, o PostgreSQL possui suporte nativo para Microsoft Windows e Sistemas Unix e Linux.

3.2 Instalação e configuração

Aprenda aqui como instalar e configurar corretamente o servidor postgreSQL, deixando-o pronto para o uso. Faremos uma instalação com pacotes Debian e outra a partir do código-fonte, compilando todo o programa.

3.2.1 Instalando com o apt-get

Utilizando a ferramenta apt-get do Debian, instalaremos a mais nova versão do PostgreSQL, a versão 8.1. Porém esta versão ainda não está presente nos repositórios oficiais stable do Debian. Vamos então utilizar repositórios unstable. Adicionamos os endereços desses repositórios no arquivo de repositórios do apt-get. O nome do arquivo é "sources.list"e está localizado na pasta /etc/apt. Apenas adicione as seguintes linhas no arquivo sources.list e salve.

deb http://ftp.br.debian.org/debian/ unstable main contrib non-free
deb-src http://ftp.br.debian.org/debian/ unstable main contrib
deb ftp://ftp.nerim.net/debian-marillat/ sid main

O apt-get deverá ser capaz de reconhecer sem problemas esses repositórios.A seguir devemos atualizar a lista de pacotes disponíveis. Para isso digite, como root:
 # apt-get update

Com as listas atualizadas, podemos agora baixar e instalar o PostgreSQL. Entre com o seguinte comando, no terminal, novamente como root:
 # apt-get install postgresql-8.1

Isto deverá instalar os seguintes pacotes:

- postgresql-8.1
- postgresql-client-8.1
- postgresql-common

Baixando um total de mais ou menos 17MB. É possível que o apt-get precise instalar outros pacotes que são pré-requisitos para o postgreSQL, no entanto, não se preocupe com isso. Apenas prossiga com a instalação. Quando se instala utilizando a ferramenta apt-get, toda configuração do postgreSQL é feita automaticamente. Logo, ao terminar de instalar já deve ser possível testar o postgreSQL. Digite os comandos abaixo:
$ su
 # su postgres

Durante a configuração do postgreSQL, o super-usuário postgres é automaticamente criado. Podemos utilizá-lo aqui para testar a nossa instalação. Após logar como postgres. Inicializamos o terminal 'psql', que já vem instalado com o postgreSQL, e indicamos o banco de dados que será utilizado: 'template1':
$ psql template1

Esse banco de dados 'template1' é um banco de dados padrão utilizado como base para se criar outros bancos de dados.

Para sair, apenas digite \q e tecle ENTER.

3.2.2 Instalando a partir do código-fonte

Caso desejar, você pode compilar inteiramente o programa, ao invés de instalar os binários com o apt-get. Para isso, baixe o código na seção de downloads na página do postgreSQL http://www.postgresql.org/ftp/source/). Na página, escolha a versão a baixar (no nosso caso, 8.1.3) e baixe o arquivo postgresql-8.1.3.tar.bz2. Salve o arquivo em qualquer pasta de sua preferência. No terminal, entre na pasta onde você salvou o arquivo e descompacte-o:
$ tar xjvf postgresql-8.1.3.tar.bz2

Isto criará uma nova pasta chamada postgresql-8.1.3/ e todos os arquivos serão descompactados para dentro dessa pasta. Agora devemos iniciar o script configure, que irá configurar os parametros necessários para a compilação, de acordo com a sua máquina.
$ cd postgresql-8.1.3
$ ./configure

Vale lembrar que a Biblioteca GNU Readline (utilizada para edição da linha de comando e historico de comandos) é utilizada por padrão durante a execução do script configure. Se você não possui esta biblioteca instalada, ou por algum motivo ela não foi reconhecida durante a configuração, voce pode desabilitá-la com a opção –without-readline. O comando ficaria assim:
$ ./configure –without-readline

Em caso de sucesso ao terminar o script, podemos compilar o programa:
$ make

E instalar o programa no sistema (este comando deve ser executado como root):
$ su
 # make install

Quando se instala a partir do código-fonte, é preciso configurar o postgreSQL manualmente. Primeiro devemos criar o usuário postgres, que é usado pelo servidor postgreSQL.
 # adduser postgres

O nome postgres é comumente usado, porém não é obrigatório, podendo ser mudado para outro qualquer. A seguir devemos escolher um espaço em disco no qual o postgreSQL irá armazenar os dados. A escolha é totalmente livre, porém a pasta /usr/local/pgsql/data é comumente usada. É ela que iremos utilizar aqui. O usuário criado acima(postgres) precisa ser o dono desta pasta. Primeiro devemos criá-la:
 # mkdir /usr/local/pgsql/data

E alteramos o dono da pasta para postgres:
 # chown postgres /usr/local/pgsql/data

Feito isso, devemos informar ao postgreSQL que esta pasta será usada como local de armazenamento dos dados. Isto deve ser feito como postgres:
 # su postgres
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

Agora podemos iniciar o servidor sem maiores problemas com o comando:
$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &

E testá-lo:
$ psql template1 A saída é a mesma da página anterior.

3.3 Estrutura do PostgreSQL

Veremos como funciona o postgreSQL, aprendendo comandos básicos do cliente psql e de SQL. Esta seção contém 3 Lições onde você verá tudo sobre como criar bancos de dados e tabelas.

3.3.1 Estrutura do postgreSQL

Antes de tudo devemos entender o funcionamento da estrutura do postgreSQL. O postgreSQL adota um modelo conhecido como cliente/servidor. Ao se acessar um banco de dados utilizando o postgreSQL, faz-se uso de no mínimo dois programas:

- O servidor. É ele quem gerencia os bancos de dados, recebe pedidos dos seus clientes, acessando e retornando os devidos dados. No postgreSQL, o servidor se chama postmaster.
- O cliente(front-end). É o programa do usuário. Através dele que é possível realizar operações nos bancos de dados. O psql é um front-end que já vem instalado com o postgreSQL e será usado aqui para fins de aprendizado.

Este modelo permite o acesso remoto ao servidor através de uma rede.

3.3.2 Entendendo o psql

O psql é um front-end (programa cliente) do postgreSQL, e já foi utilizado por nós. Através dele é que fazemos ?pedidos? ao servidor, requisitando todo o tipo de operações relacionadas aos bancos de dados.

O psql possui um conjunto de comandos próprios que são bem úteis para a utilização do programa, esses comandos são chamados de meta-comandos.

Abaixo está uma lista reduzida de alguns dos meta-comandos mais usados:

- \q ? sai do psql.
- \c [NOME] ? conecta a outro banco de dados. Por exemplo:
- \c template0 : conecta ao banco de dados template0.
- \l ? lista todos os bancos de dados disponibilizados.
- \d [NOME] ? descreve uma tabela, índice, visão ou sequência. Por exemplo:
- \d tabela : mostra todas as informações sobre as colunas da tabela 'tabela'
- \du ? lista todos os usuários cadastrados no servidor.
- \dg ? lista todos os grupos cadastrados no servidor.
- \dT ? lista todos os tipos de dados.
- \h [NOME] ? Mostra a sintaxe do comando SQL. Por exemplo:
- \h SELECT : mostra a sintaxe do comando SELECT.
- \? ? lista todos os comandos disponíveis.

Recomendamos a testar os meta-comandos acima na medida do possível, familiarizando-se com eles. Como você pode notar, todos os comandos do psql começam com a barra invertida \. Logo, todo comando que começar com \daqui para frente será um meta-comando do psql. Para ver a lista de todos os meta-comandos apenas digite '\?' no prompt do psql.

Observe que meta-comandos do psql não necessitam do delimitador 'ponto-e-vírgula' ;

Capítulo 4

Módulo II - Acesso e manipulação de tabelas e administração do banco de dados

4.1 Criando, removendo e acessando bancos de dados

4.1.1 Criando bancos de dados

Podemos criar novos bancos de dados utilizando o prompt de comando do sistema ou o prompt do psql. Utilizando o prompt do sistema, rodamos o script createdb para criar um novo banco de dados. Fazemos o seguinte:

$ createdb nome_do_banco

É necessário salientar que só é possível criar um banco de dados com um usuário com tais permissões (Veremos mais adiante como adicionar ou alterar usuários.) No nosso caso, ainda não vimos como criar novos usuários e lidar com suas permissões, portanto crie um novo banco de dados com o usuário postgres.Para criar novos bancos de dados através do psql, fazemos uso da linguagem SQL:

CREATE DATABASE nome_do_banco;

Note a presença do delimitador ; no fim da query. A saída, em caso de sucesso e em ambos os casos, será:
CREATE DATABASE

4.1.2 Acessando bancos de dados

Após criar o banco, obviamente você vai querer usá-lo. Para usar um banco de dados, deve-se iniciar o psql já indicando qual banco deverá ser usado. Lembram do comando psql template1? Esse comando iniciava o psql utilizando o banco de dados padrão template1. Com um mínimo de esforço, pode-se deduzir o comando para iniciar o psql com o banco desejado:

$ psql nome_do_banco

Se você já inicializou o psql, é possível mudar o banco de dados que está sendo utilizado. Para fazer isso digite, no prompt do psql o seguinte comando:
\c nome_do_banco

Como já vimos anteriormente, este é um comando próprio do psql. Caso ocorra como esperado, a seguinte saída será mostrada:

Você está conectado ao banco de dados ?nome_do_banco? agora.

4.1.3 Removendo bancos de dados

Depois de usar um banco de dados, você pode não querer mais usá-lo. É possível portanto removê-lo. Novamente é possível fazer isso pela linha de comando do sistema, ou do psql. Pelo prompt do sistema:
$ dropdb nome_do_banco

Pelo prompt do psql:

DROP DATABASE nome_do_banco;

A saída, em caso de sucesso e em ambos os casos, será:
DROP DATABASE

O comando dropdb apaga todos os arquivos relacionados ao banco de dados em questão, e não é reversível.

4.2 Criando, removendo e manipulando tabelas

4.2.1 Criando tabelas

Em um banco de dados relacional as informações são armazenadas em tabelas, que podem ou não ter relação entre si. Vamos criar tabelas simples, somente para aprendermos a sintaxe. Mais a frente veremos como criar tabelas um pouco mais complexas. Para se criar uma tabela, usamos o comando CREATE TABLE. Vejamos como fazer isso no psql:

```
CREATE TABLE tabela_de_teste (
valor1 int,
valor2 varchar(80),
valor3 int
);
```

Este comando cria uma tabela de nome 'tabela_de_teste' com as seguintes colunas: valor1, valor2 e valor3; cada uma com um tipo de dado específico. A saída deverá ser:
CREATE TABLE

A sintaxe do comando é:
CREATE TABLE nome_da_tabela (nome_da_coluna tipo_da_coluna);

Claro que normalmente se tem mais de uma coluna, portanto, separa-se as colunas com vírgulas, especificando o tipo de cada uma delas:
CREATE TABLE nome_da_tabela (
nome_da_coluna1 tipo_da_coluna1,
nome_da_coluna2 tipo_da_coluna2,
.
.
.
nome_da_colunaM tipo_da_colunaM,
nome_da_colunaN tipo_da_colunaN
);
Podemos ver se a tabela foi criada da forma desejada com o comando do psql. Apenas digite:
\d tabela_de_teste

Isto retornará uma tabela com as colunas de 'tabela_de_teste' e os respectivos tipos de dados e modificadores.

Uma observação importante: Ao digitar os comandos no psql, não é necessário que esteja tudo em uma só linha. É totalmente possível pular linhas em um mesmo comando, isso se deve ao fato de ser necessária a presença do delimitador ; em comandos SQL. O psql não executará o comando enquanto não encontrar o delimitador ; .

É possível ainda atribuir um valor padrão para qualquer uma das colunas criadas, fazemos isto na hora de criar uma tabela:

CREATE TABLE numeros (valor1 integer,
valor2 integer DEFAULT 0);

Isto faz com que o valor 0 seja automaticamente inserido na coluna valor2, caso nenhum valor seja especificado.

4.2.2 Tipos de dados

Cada coluna de uma tabela guarda uma informação. Pode ser por exemplo uma data, ou ainda um nome ou um numero. Podemos querer armazenar valores em dinheiro, ou valores booleanos como Verdadeiro ou Falso. Para tratar os diferentes tipos de valores, existem diferentes tipos de dados. Estes tipos de dados informam o postgreSQL qual o tipo de informação está armazenada em uma dada coluna. Logo é necessário especificar qual o tipo de dado de cada coluna em uma tabela. Vimos isso na página anterior, com o comando:

CREATE TABLE tabela_de_teste (valor1 int, valor2 varchar(80),
valor3 int);

No postgreSQL podemos separar os tipos de dados disponíveis em certos grupos lógicos, veremos a seguir alguns dos grupos mais usados:

- Tipos Numéricos:

nome	nome inteiro	descrição
smallint	int2	inteiro. Abrange valores de -32678 a +32678
integer	int4	inteiro. Abrange valores de -2147483648 a +2147483647
bigint	int8	inteiro. Abrange valores de -9223372036854775808 a 922337203685477
numeric	numeric	precisão definida pelo usuário
real	float4	Número em vírgula flutuante com precisão simples
double precision	float8	Número em vírgula flutuante de dupla precisão.
serial	serial	Inteiro com auto-incrementação. Abrange de 1 a 2147483647
bigserial	bigserial	Inteiro com auto-incrementação. Abrange de 1 a 9223372036854775807

Cabe dizer aqui que os tipos serial e bigserial não são realmente tipos de dados. Mas apenas uma notação conveniente para criar colunas identificadoras com valores únicos e auto-incrementação (Outros bancos de dados utilizam a propriedade AUTO_INCREMENT para criar colunas com auto-incrementação).

- Tipos de Data e hora:

nome	nome interno	descrição
timestamp without time zone	timestamp	Data e hora. Abrange de 4713AC até 5874897DC
timestamp with time zone	timestampz	Data e hora com zona horária. Abrange de 4713AC até 587
interval	interval	Intervalo de tempo. Abrange de -178000000 a 178000000 a
date	date	Apenas datas. Abrange de 4713AC até 5874897DC
time without time zone	time	Apenas hora. Abrange de 00:00:00.00 a 23:59:59.99
time with time zone	timez	Apenas hora com zona horária. Abrange de 00:00:00.00+12

- Tipos de Cadeia(String):

nome	nome interno	descrição
character varying(tamanho)	varchar(tamanho)	Cadeia de caracteres com comprimento variável. Possui
character(tamanho)	char(tamanho)	Cadeia de caracteres com tamanho fixo. Completado co
text	text	Cadeia de caracteres de tamanho variável. Sem limite de

Para a declaração dos tipos pode-se usar tanto o nome, como o nome interno do tipo. Isto fica à sua escolha. Há ainda tipos binários, tipos geométricos, tipos para endereços de rede entre outros, que não foram explicitados aqui. Para um lista completa dos tipos de dados suportados, digite o comando:
\dT+

Já vimos este comando, porém agora entramos com um sinal de +. Este sinal serve apenas para listar os tipos de dados com mais informações. Caso deseje saber mais sobre tipos de dados visite: http://pgdocptbr.sourceforge.net/pg80/datatype.html

4.2.3 Restrições (Constraints)

Ao criar uma tabela podemos impor algum tipo de restrição, por exemplo, ao criar uma coluna que represente alguma quantidade é desejável que não se entre com valores menores do que

zero. Talvez queiramos que um dado valor seja único, ou seja, não se repita em outras linhas da tabela. Tudo isso é possível através do uso de restrições.

Restrições de verificação:
Podemos fazer a verificação de um dado antes que ele seja inserido na tabela. Essa verificação será uma propriedade da tabela em questão e sempre será checada. Para fazer isso utilizamos a palavra-chave CHECK:

CREATE TABLE coleção (quantidade integer CHECK (quantidade >= 0));

Com esse comando criamos uma tabela com a coluna quantidade, aonde a quantidade não poderá ser menor do que zero.
Podemos ainda fazer referencia a outros valores da tabela na hora da verificação:

CREATE TABLE produtos(preço numeric, preço_desconto numeric, CHECK (preço > preço_desconto)
);

Onde o preço entrado deve ser maior do que o preço com desconto.

Restrição de não-nulo:
A palavra-chave NOT NULL especifica que uma coluna não pode assumir um valor nulo. Não se deve confundir um valor nulo com o valor 0. Uma coluna com valor nulo é simplismente uma coluna vazia.
CREATE TABLE produtos (identificador varchar(4) NOT NULL,
nome varchar(80) NOT NULL,
preço numeric);

Este comando cria uma tabela onde todas as linhas devem possuir um identificador e um nome. Não é possível inserir um dado sem um identificador e sem um nome, é possível porém inserir um dado sem preço.

Restrição de unicidade:
Este tipo de restrição garante que o valor de uma linha é único na coluna, ou seja, não há valores repetidos na coluna que possui esta propriedade. Vejamos um exemplo para facilitar o entendimento:
CREATE TABLE unicidade (identificador integer UNIQUE,
nome varchar(80)
);
Este comando cria uma tabela aonde a coluna identificador possui uma propriedade de unicidade. Logo, não é possível inserir duas linhas nesta tabela com o mesmo identificador. Cada linha terá um campo identificador diferente das outras. É possível explicitar as colunas que terão propriedade UNIQUE fazendo da seguinte forma:
CREATE TABLE unicidade (identificador integer,
nome varchar(80),
UNIQUE (identificador)
);

e é claro, fazer isso com várias colunas:

```
CREATE TABLE exemplo (A integer,
B integer,
C integer,
D integer,
UNIQUE (A, C)
);
```

Chaves primárias e estrangeiras: O conceito de chave primária é uma coluna que identifique cada linha de uma tabela, logo esta coluna deve ser única e não pode ter valores nulos. No postgreSQL, PRIMARY KEY é uma junção de UNIQUE com NOT NULL, logo:

```
CREATE TABLE exemplo (
chave integer PRIMARY KEY
);
```

é, em termos, igual a (possuem as mesmas restrições)

```
CREATE TABLE exemplo (
chave integer NOT NULL UNIQUE
);
```

Já uma chave estrangeira é uma referência a uma linha de outra tabela, ou seja, o valor de uma coluna de chave estrangeira é exatamente o valor da coluna que ela referencia. Vamos a um exemplo para facilitar o entendimento: Suponha que possuímos um programa de agendamento, que suporta vários usuários. Existiria uma tabela para todos os usuários cadastrados no programa:

```
CREATE TABLE tabela_usuarios (
cod_usuario integer PRIMARY KEY,
nome varchar(80),
senha varchar(10)
);
```

e poderíamos ter também uma outra tabela para agendar os compromissos dos usuários:

```
CREATE TABLE tabela_compromissos (
cod_compromisso integer PRIMARY KEY,
usuario integer REFERENCES tabela_usuarios(cod_usuario),
data date
);
```

Com esta montagem de tabelas, não é possível inserir um compromisso com usuario que não existe na tabela tabela_usuarios. Ou seja, no contexto do nosso programa, somente usuários cadastrados podem inserir compromissos. Dizemos que a tabela tabela_compromissos é a tabela que faz referência, enquanto que a tabela tabela_usuarios é a tabela referenciada.
A tabela tabela_compromissos poderia ser simplificada para:

```
CREATE TABLE tabela_compromissos (
cod_compromisso integer PRIMARY KEY,
usuario integer REFERENCES tabela_usuarios,
data date
```

);
Isto porque tabela_usuarios possui uma chave primária, e a palavra-chave REFERENCES toma como referência uma coluna de chave primária, caso uma coluna não seja especificada.

A partir daqui poderíamos imaginar: ?E se fosse removido um usuário da tabela 'tabela_usuarios', mas este possuísse compromissos na tabela 'tabela_compromissos'??
Poderíamos ou impedir que tal usuário fosse apagado do programa ou então que todos os seus compromissos fossem apagados também. Para isso, utilizamos ON DELETE, CASCADE e RESTRICT:
Faríamos, ao invés do comando anterior:

CREATE TABLE tabela_compromissos (
cod_compromisso integer PRIMARY KEY,
usuario integer REFERENCES tabela_usuarios ON DELETE CASCADE,
data date
);

Isto apagaria todos os compromissos do usuário, caso o usuário seja apagado da tabela tabela_usuarios. Para impedir que o usuário seja apagado se ainda houver compromissos, simplesmente trocamos a palavra CASCADE por RESTRICT. Ficaria assim: ON DELETE RESTRICT.

Além de DELETE e RESTRICT existem ainda,
NO ACTION: gera um erro.
SET DEFAULT: muda o valor da coluna que faz referência para o valor padrão definido.
SET NULL: muda o valor da coluna que faz referência para um valor nulo.

4.2.4 Modificando tabelas

Ao criar uma tabela, é possível que a mesma não fique exatamente da forma como era desejada, seja por falta de especificação ou esquecimento na hora de contruir a tabela. Por causa disso, é necessário que seja possível modificar uma tabela já existente, já que destruir e criar a tabela novamente talvez não seja uma opção muito viável.
Com o postgreSQL é possível adicionar novas colunas a uma tabela, remover colunas, alterar o tipo de dado de uma coluna, renomear a coluna e a tabela. Pode-se ainda modificar constraints e valores padrao de uma coluna.

Todas essas alterações listadas são feitas através do comando SQL, ALTER TABLE. Veremos a seguir:

Adicionando colunas: Usamos ADD COLUMN para adicionar novas colunas. Ao adicionar novas colunas, a sintaxe usada é a mesma de quando criamos uma nova tabela, ou seja, informamos o nome da coluna e depois o tipo:

ALTER TABLE nome_da_tabela ADD COLUMN nova_coluna tipo_da_coluna;

Pode-se ainda definir restrições com a sintaxe usual de restrições ou com um valor padrão.

Removendo colunas:
Para remover uma coluna de uma tabela usamos DROP COLUMN seguido do nome da coluna a ser apagada:

ALTER TABLE nome_da_tabela DROP COLUMN coluna_a_remover;

Todos os dados e restrições desta coluna são apagados, porém se esta é uma coluna referenciada uma mensagem de erro surgirá.
Para apagar uma coluna referenciada e todos os objetos que dependem dela usamos:

ALTER TABLE nome_da_tabela DROP COLUMN coluna_a_remover CASCADE;

Adicionar restrições:
Para adicionar restrições, fazemos:

ALTER TABLE nome_da_tabela ADD restrição;

Por exemplo:
ALTER TABLE nome_da_tabela ADD CHECK(coluna > 0);
ALTER TABLE nome_da_tabela ADD UNIQUE(coluna);
ALTER TABLE nome_da_tabela ADD PRIMARY KEY(nome_da_coluna);
ALTER TABLE nome_da_tabela ADD FOREIGN KEY(nome_da_coluna) REFERENCES tabela_referenciada;

Para o caso da restrição de não-nulo, a sintaxe é um pouco diferente:

ALTER TABLE nome_da_tabela ALTER COLUMN nome_da_coluna SET NOT NULL;

Remover restrições:
Para remover uma restrição é necessário saber o seu nome. Para isso utilize o comando \d nome_da_tabela. Isto mostrará a tabela em questão incluindo todas as suas restrições com seus respectivos nomes.
De posse do nome da restrição a ser retirada, simplismente faça:

ALTER TABLE nome_da_tabela DROP CONSTRAINT nome_da_restrição;

Da mesma forma que adicionar uma restrição não-nulo, remover uma restrição de não-nulo é um pouco diferente:

ALTER TABLE nome_da_tabela ATER COLUMN nome_da_coluna DROP NOT NULL;

Alterar o valor padrão:
Simplesmente faça:

ALTER TABLE nome_da_tabela ALTER COLUMN nome_da_coluna SET DEFAULT valor_padrão;

Alterando o tipo:
Para alterar o tipo de dado de uma coluna usamos:

ALTER TABLE nome_da_tabela ALTER COLUMN nome_da_coluna TYPE novo_tipo;

Este comando só funciona se for possível converter todos os dados desta coluna para o novo tipo de dado.

Renomear uma coluna:
Para alterar o nome de uma coluna, simplismente faça:

ALTER TABLE nome_da_tabela RENAME COLUMN nome_da_coluna TO novo_nome;

Renomear a tabela:
E finalmente, para alterar o nome da tabela, faça:

ALTER TABLE nome_da_tabela RENAME TO novo_nome;

4.2.5 Removendo tabelas

Depois de todo o trabalho de criar as tabelas, eventualmente você vai precisar apagar algumas delas. Fazer isso é bem simples, veja a seguir:

DROP TABLE nome_da_tabela;

Caso existam objetos que referenciem esta tabela, a mesma não será apagada. Para removê-la e também os objetos que a referenciam faça:

DROP TABLE nome_da_tabela CASCADE;

4.3 Acessando dados de tabelas

Veja aqui como povoar as tabelas e como acessar os seus dados posteriormente.

4.3.1 Inserindo dados na tabela

Depois de criar sua tabela, você precisa povoá-la. As instruções COPY e INSERT são úteis para isso. Suponha que criamos a seguinte tabela:

CREATE TABLE tempo (
cidade varchar(80),

```
maximo numeric DEFAULT 40,
minimo numeric,
data date
);
```

Como você está começando com uma tabela vazia, uma forma simples de povoá-la é criar um arquivo texto contendo uma linha para cada um de seus animais, e depois carregar o conteúdo do arquivo para a tabela com uma simples instrução.

Você pode criar um arquivo texto 'dados.txt' contendo um registro por linha, com valores separado por tabulações e na mesma ordem em que as colunas foram listadas na instrução CREATE TABLE. Para valores em falta você pode usar valores NULL. Para representá-lo em seu arquivo texto, use \N (barra invertidam N maíusculo).
Para carregar o arquivo texto 'dados.txt' na tabela você deve explicitar o caminho completo para o arquivo. Use este comando:

```
COPY nome_da_tabela FROM 'caminho_do_arquivo';
```

Por exemplo:

```
COPY tempo FROM '/home/usuario/Desktop/dados.txt';
```

Quando você desejar adicionar novos registros um a um, a instrução INSERT é usada. Na sua forma mais simples, você fornece valores para cada coluna, na ordem em que as colunas foram listadas na instrução CREATE TABLE.
Você pode adicionar um registro utilizando uma instrução INSERT desta forma:

```
INSERT INTO tempo VALUES('Brasilia', 32, 26, 'Mar-06-2006');
```

Que representam as temperaturas máximas e mínimas em Brasília, no dia 06 de março de 2006. Observe como o formato da data é utilizado. Você pode não estar acostumado com este formato, mas este não é o único formato que pode ser utilizado. Veremos mais adiante que formatos utilizar para inserir os diferentes tipos de dados em uma tabela de forma correta.
Veja também que o formato para a inserção de cadeias (strings) deve ser feito com aspas simples, enquanto valores numéricos não.

É possível também inserir dados em colunas específicas da tabela, bastando para isso identificá-la:

```
INSERT INTO tempo(cidade, data) VALUES ('Goiania', 'Jan-02-2006');
```

Quando se faz isso, deve-se ficar atento a propriedade NOT NULL das colunas. Se existir uma coluna NOT NULL um valor deve obrigatoriamente ser inserido. Portanto a instrução acima não funcionaria caso as colunas maximo e minimo tivessem a restrição NOT NULL.

Para contornar isso pode-se definir um valor padrão para a coluna. Assim, se não for especificado um valor a ser inserido, o valor padrão será automaticamente inserido naquela coluna.

Podemos ainda utilizar as palavras-chave default e NULL, para inserir os valores padrão e nulo respectivamente:
INSERT INTO tempo VALUES ('Rio', default, 34, NULL);

A instrução faria com que fosse inserido na tabela seguinte registro:

cidade	maximo	minimo	data
Rio	40	34	

Como foi dito, existem diferentes formas de inserir uma data. Abaixo está uma lista dos possíveis formatos de data e hora:
Formato de datas:

Exemplo	Descrição
May 3, 2006	não-ambíguo em qualquer modo de entrada em datestyle
2006-05-03	ISO 8601; 3 de maio de 2006 em qualquer modo (formato recomendado)
3/8/2006	8 de março no modo MDY; 3 de agosto no modo DMY
1/18/1999	18 de janeiro no modo MDY; rejeitado nos demais modos
01/02/03	2 de janeiro de 2003 no modo MDY; 1 de fevereiro de 2003 no modo DMY; 3 de fevereiro d
2006-May-03	3 de maio em qualquer modo
May-03-2006	3 de maio em qualquer modo
03-May-2006	3 de maio em qualquer modo
99-May-03	3 de maio no modo YMD, caso contrário errado
03-May-99	3 de maio, porém errado no modo YMD
May-03-99	3 de maio, porém errado no modo YMD
20060503	ISO 8601; 3 de maio de 2006 em qualquer modo
060503	ISO 8601; 3 de maio de 2006 em qualquer modo
2006.123	ano e dia do ano
J2453859	dia juliano
May 3, 99 BC	ano 99 antes da era comum

Formato de horas:

Exemplo	Descrição
14:15:10.156	ISO 8601
14:15:10	ISO 8601
14:15	ISO 8601
141510	ISO 8601
02:15 AM	o mesmo que 02:15; AM não afeta o valor
02:15 PM	o mesmo que 14:15; a hora deve ser menor do que 12
14:15:10.156-8	ISO 8601
14:15:10-08:00	ISO 8601
14:15-08:00	ISO 8601
141510-08	ISO 8601
14:15:10 PST	zona horária especificada pelo nome

Insira alguns registros na tabela tempo, iremos utilizá-la novamente em lições seguintes. Use a sua imaginação!

4.3.2 A instrução SELECT

Agora que possuímos uma tabela povoada, vamos precisar recuperar informações dessa tabela. Fazemos isto com a instrução SELECT. Eis a sua sintaxe:

SELECT colunas FROM nome_da_tabela WHERE condições;

onde colunas é uma lista das colunas que se deseja ver, nome_da_tabela é a tabela da qual se deseja obter as informações e condições são condições que as informações devem satisfazer para que sejam selecionadas. A parte 'WHERE condições' da instrução acima não é obrigatória. Vamos utilizar a tabela tempo criada na lição anterior. Para selecionar todas as colunas da tabela tempo fazemos o seguinte:

SELECT * FROM tempo;

O retorno é uma tabela com todas as linhas e colunas da tabela tempo.

Podemos selecionar apenas as colunas que são relevantes para nós, fazendo:

SELECT cidade, maximo FROM tempo;

Esta instrução retorna apenas as colunas cidade e maximo da tabela tempo. Podemos fazer isso com qualquer número de colunas.

Este tipo de seleção não está restrito apenas as colunas na tabela, pode-se utilizar expressões que envolvem as colunas. Por exemplo:

SELECT cidade, (maximo + minimo)/2 AS media, data FROM tempo;

Aqui introduzimos também a cláusula AS. A cláusula AS apenas atribui um nome a esta coluna, e pode ser usada com qualquer coluna.
Teremos como resultado então 3 colunas: cidade, media e data.

Até agora os comandos acima selecionam todas as linhas da tabela, restringindo sua escolha apenas as colunas desejadas. Mas e se precisarmos obter apenas determinadas linhas? Isso mesmo! Podemos fazer isso com o uso da cláusula WHERE:

SELECT * FROM tempo WHERE cidade = 'Brasilia';

A instrução seleciona todas as colunas da tabelas, mas apenas as linhas onde o campo cidade é igual a 'Brasilia'.
Na verdade a cláusula WHERE é um verificador booleano. Ela avalia a expressão e retorna apenas as linhas para as quais a expressão é verdadeira. Podemos utilizar os operadores AND, OR e NOT para ampliar as expressões de WHERE. Por exemplo:

SELECT * FROM tempo WHERE cidade = 'Brasilia' AND data = '2006-04-06';

Isto nos mostra apenas as linhas da tabela em que cidade é igual a Brasília e data é igual a

6 de abril de 2006.

Quando se faz uma busca, pode ser que existam linhas com valores iguais. Por exemplo, podem existir várias linhas na nossa tabela tempo em que o campo cidade seja Brasilia. Podemos fazer com que seja exibido apenas uma dessas linhas com a instrução DISTINCT:

SELECT DISTINCT cidade FROM tempo;

Isto mostraria todas as cidades existentes na tabela, sem repetição.

Podemos ainda ordenar os resultados de acordo com alguma coluna:

SELECT * FROM tempo ORDER BY cidade;

Isto retornaria o resultado com as cidades em ordem alfabética.

4.3.3 Junções (Joins)

Você pode estar imaginando, ?Mas só é possível ver dados de apenas uma tabela por vez?? A resposta é não. É possível fazer junções de tabelas para obter dados cruzados. Veremos como isso é possível. Imagine que temos uma nova tabela chamada cidades. Esta tabela guarda informações sobre a localização de uma determinada cidade e possui colunas como: nome, latitude, longitude. Vamos fazer uma pesquisa com o tempo das cidades juntamente com a localização de cada cidade.

SELECT * FROM tempo, cidades WHERE cidade = nome;

Existem duas colunas contendo o nome da cidade, o que está correto porque a lista de colunas das tabelas tempo e cidades estão concatenadas. Na prática isto não é desejado, sendo preferível, portanto, escrever a lista das colunas de saída explicitamente em vez de utilizar o *:

SELECT cidade, maximo, minimo, data, latitude, longitude
FROM tempo, cidades WHERE cidade = nome;

Como todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence, mas é um bom estilo qualificar completamente os nomes das colunas nas consultas de junção:
SELECT tempo.cidade, tempo.maximo, tempo.minimo,
tempo.data, cidades.latitude, cidades.longitude
FROM tempo, cidades
WHERE cidades.nome = tempo.cidade;

As consultas de junção do tipo visto até agora também poderiam ser escritas da seguinte forma alternativa:

SELECT *
FROM tempo INNER JOIN cidades ON (tempo.cidade = cidades.nome);

A utilização desta sintaxe não é tão comum quanto a usada acima, mas é mostrada para ajudar a entender os próximos tópicos.
Com esta sintaxe é possível cruzar os dados de maneira mais específica. Imagine que desejamos o seguinte: que a consulta varra a tabela tempo e, para cada uma de suas linhas, encontre a linha correspondente na tabela cidades. Se não for encontrada nenhuma linha correspondente, desejamos que sejam colocados "valores vazios"nas colunas da tabela cidades. Este tipo de consulta é chamada de junção externa (outer join). As consultas vistas até agora são junções internas (inner join). O comando então fica assim:

```
SELECT *
FROM tempo LEFT OUTER JOIN cidades ON (tempo.cidade = cidades.nome);
```

Esta consulta é chamada de junção externa esquerda (left outer join), porque a tabela mencionada à esquerda do operador de junção terá cada uma de suas linhas aparecendo na saída pelo menos uma vez, enquanto a tabela à direita terá somente as linhas correspondendo a alguma linha da tabela à esquerda aparecendo na saída. Ao listar uma linha da tabela à esquerda, para a qual não existe nenhuma linha correspondente na tabela à direita, são colocados valores vazios (null) nas colunas da tabela à direita.

4.3.4 Visões (Views)

Supondo que a consulta combinando os registros de tempo e de localização das cidades seja de particular interesse para um aplicativo, mas que não se deseja digitar esta consulta toda vez que for necessária, então é possível criar uma visão baseada na consulta, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_visao AS
SELECT cidade, maximo, minimo, data, localizacao
FROM tempo, cidades
WHERE cidade = nome;

SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL. As visões permitem encapsular, atrás de interfaces que não mudam, os detalhes da estrutura das tabelas, que podem mudar na medida em que os aplicativos evoluem.

As visões podem ser utilizadas em praticamente todos os lugares onde uma tabela real pode ser utilizada. Construir visões baseadas em visões não é raro.

4.3.5 Funções de agregação

Como a maioria dos produtos de banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação computa um único resultado para várias linhas de entrada. Por exemplo, existem funções de agregação para contar (count), somar (sum), calcular a média (avg), o valor máximo (max) e o valor mínimo (min) para um conjunto de linhas.

Para servir de exemplo, é possível encontrar a maior temperatura mínima ocorrida em qualquer

lugar usando:

SELECT max(minimo) FROM tempo;

Se for desejado saber a cidade (ou cidades) onde esta temperatura ocorreu pode-se tentar usar:

SELECT cidade FROM tempo WHERE minimo = max(minimo);

mas não vai funcionar, porque a função de agregação max não pode ser usada na cláusula WHERE (Esta restrição existe porque a cláusula WHERE determina as linhas que vão passar para o estágio de agregação e, portanto, precisa ser avaliada antes das funções de agregação serem computadas). Entretanto, como é geralmente o caso, a consulta pode ser reformulada para obter o resultado pretendido, o que será feito por meio de uma subconsulta:

SELECT cidade FROM tempo
WHERE minimo = (SELECT max(minimo) FROM tempo);

Isto está correto porque a subconsulta é uma ação independente, que calcula sua agregação em separado do que está acontecendo na consulta externa. As agregações também são muito úteis em combinação com a cláusula GROUP BY. Por exemplo, pode ser obtida a maior temperatura mínima observada em cada cidade usando:

SELECT cidade, max(minimo)
FROM tempo
GROUP BY cidade;

A saída será algo do tipo:

cidade | max
———-+——
Brasilia | 25
Goiania | 27
(2 linhas)

produzindo uma linha de saída para cada cidade. Cada resultado da agregação é computado sobre as linhas da tabela correspondendo a uma cidade. As linhas agrupadas podem ser filtradas utilizando a cláusula HAVING:

SELECT cidade, max(minimo)
FROM tempo
GROUP BY cidade
HAVING max(minimo) < 26;

cidade | max
———+——
Brasilia | 25
(1 linha)

que mostra os mesmos resultados, mas apenas para as cidades que possuem todos os valores de minimo abaixo de 26. Para concluir, se desejarmos somente as cidades com nome começando pela letra "B"podemos escrever:

```
SELECT cidade, max(minimo)
FROM tempo
WHERE cidade LIKE 'B%'
GROUP BY cidade
HAVING max(minimo) < 26;
```

É importante compreender a interação entre as agregações e as cláusulas WHERE e HAVING do SQL. A diferença fundamental entre WHERE e HAVING é esta: WHERE seleciona as linhas de entrada antes dos grupos e agregações serem computados (portanto, controla quais linhas irão para o computo da agregação), enquanto HAVING seleciona linhas de grupo após os grupos e agregações serem computados.
Portanto, a cláusula WHERE não pode conter funções de agregação; não faz sentido tentar utilizar uma agregação para determinar quais linhas serão a entrada da agregação. Por outro lado, a cláusula HAVING sempre contém funções de agregação (A rigor, é permitido escrever uma cláusula HAVING que não possua agregação, mas é desperdício: A mesma condição poderia ser utilizada de forma mais eficiente no estágio do WHERE).

No exemplo anterior, a restrição do nome da cidade pode ser aplicada na cláusula WHERE, porque não necessita de nenhuma agregação, sendo mais eficiente que colocar a restrição na cláusula HAVING, porque evita realizar os procedimentos de agrupamento e agregação em todas as linhas que não atendem a cláusula WHERE.

4.3.6 Modificando dados

Atualizamos os dados de uma tabela com o comando UPDATE. Vejamos sua sintaxe:

```
UPDATE nome_da_tabela SET nome_da_coluna = 'valor' WHERE condição;
```

Este comando altera o valor da coluna 'nome_da_coluna' para 'valor' em todas as linhas que satisfizerem a condição.
Por exemplo:

```
UPDATE tempo SET maximo = 29 WHERE cidade = 'Brasilia';
```

Atualiza o valor maximo para 29 em todas as linhas em que cidade é igual a 'Brasilia'.

Nota: Sem o uso da cláusula WHERE, todas as linhas da tabela são alteradas.

4.3.7 Removendo dados

Removemos os dados de uma tabela com o comando DELETE. Sua sintaxe é a seguinte:

```
DELETE FROM nome_da_tabela WHERE condição;
```

Isto paga da tabela todas as linhas que satisfazem a condição. Por exemplo:

DELETE FROM tempo WHERE cidade = 'Brasilia';

Isto apagaria todas as linhas da tabela tempo onde o campo cidade fosse igual a 'Brasilia';
Deve-se tomar um especial cuidado com este comando, analisando bem a condição a ser usada, para que não se apague dados importantes. Observe que:

DELETE FROM tempo;

Apaga TODOS os registros da tabela tempo, já que não existe nenhuma condição. E, a não ser que este seja seu objetivo, não deve ser usado o comando de tal forma.

4.4 Conceitos avançados

Alguns conceitos mais avançados sobre bancos de dados relacionais que são suportados pelo PostgreSQL

4.4.1 Transações

Transação é um conceito fundamental de todo sistema de banco de dados. O ponto essencial da transação é englobar vários passos em uma única operação de tudo ou nada. Os estados intermediários entre os passos não são vistos pelas demais transações simultâneas e, se ocorrer alguma falha que impeça a transação chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de forma alguma.

Por exemplo, considere um banco de dados de uma instituição financeira contendo o saldo da conta corrente de vários clientes, assim como o saldo total dos depósitos de cada agência. Suponha que se deseje transferir $100.00 da conta da Alice para a conta do Bob. Simplificando barbaramente, os comandos SQL para esta operação seriam:
UPDATE conta_corrente SET saldo = saldo - 100.00
WHERE nome = 'Alice';
UPDATE filiais SET saldo = saldo - 100.00
WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Alice');
UPDATE conta_corrente SET saldo = saldo + 100.00
WHERE nome = 'Bob';
UPDATE filiais SET saldo = saldo + 100.00
WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Bob');

Os detalhes destes comandos não são importantes aqui; o importante é o fato de existirem várias atualizações distintas envolvidas para realizar uma operação bem simples. A contabilidade quer ter certeza que todas as atualizações são realizadas, ou que nenhuma delas é realizada. Não é interessante uma falha no sistema fazer com que Bob receba $100.00 que não foi debitado da Alice.

Também a Alice não continuará sendo uma cliente satisfeita se o dinheiro for debitado da conta

dela e não for creditado na de Bob. É necessário garantir que, caso aconteça algo errado no meio da operação, nenhum dos passos executados até este ponto produza efeito. Agrupar as atualizações em uma transação dá esta garantia. Uma transação é dita como sendo atômica: do ponto de vista das outras transações, ou a transação acontece completamente ou nada acontece.

Desejamos, também, ter a garantia de estando a transação completa e aceita pelo sistema de banco de dados, que esta fique permanentemente gravada, e não seja perdida mesmo no caso de acontecer uma pane logo em seguida. Por exemplo, se estiver sendo registrado saque em dinheiro pelo Bob não se deseja, de forma alguma, que o débito em sua conta corrente desapareça por causa de uma pane ocorrida logo depois dele sair da agência. Um banco de dados transacional garante que todas as atualizações realizadas por uma transação ficam registradas em meio de armazenamento permanente (ou seja, em disco), antes da transação ser considerada completa.

Outra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando simultaneamente, cada uma delas não deve enxergar as alterações incompletas efetuadas pelas outras. Por exemplo, se uma transação está ocupada totalizando o saldo de todas as agências, não pode ser visto o débito efetuado na agência da Alice mas ainda não creditado na agência do Bob, nem o contrário. Portanto, as transações devem ser tudo ou nada não apenas em termos do efeito permanente no banco de dados, mas também em termos de visibilidade durante o processamento. As atualizações feitas por uma transação em andamento não podem ser vistas pelas outras transações enquanto não terminar, quando todas as atualizações se tornam visíveis ao mesmo tempo.

No PostgreSQL a transação é definida envolvendo os comandos SQL da transação pelos comandos BEGIN e COMMIT . Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
WHERE nome = 'Alice';
– etc etc
COMMIT;
```

Se no meio da transação for decidido que esta não deve ser efetivada (talvez porque tenha sido visto que o saldo da Alice ficou negativo), pode ser executado o comando ROLLBACK em vez do COMMIT para fazer com que todas as atualizações sejam canceladas.

O PostgreSQL, na verdade, trata todo comando SQL como sendo executado dentro de uma transação. Se não for utilizado o comando BEGIN, então cada comando possui um BEGIN e, se der tudo certo, um COMMIT individual envolvendo-o. Um grupo de comandos envolvidos por um BEGIN e um COMMIT é algumas vezes chamado de bloco de transação.

É possível controlar os comandos na transação de uma forma mais granular utilizando os pontos de salvamento (savepoints). Os pontos de salvamento permitem cancelar partes da transação seletivamente, e efetivar as demais partes. Após definir o ponto de salvamento, através da instrução SAVEPOINT, é possível cancelar a transação até o ponto de salvamento, se for necessário, usando ROLLBACK TO. Todas as alterações no banco de dados efetuadas entre a definição do ponto de salvamento e o cancelamento são desprezadas, mas as alterações efetuadas antes do

ponto de salvamento são mantidas.

Após cancelar até o ponto de salvamento este ponto de salvamento continua definido e, portanto, é possível cancelar várias vezes. Ao contrário, havendo certeza que não vai ser mais necessário cancelar até o ponto de salvamento, o ponto de salvamento pode ser liberado, para que o sistema possa liberar alguns recursos. Deve-se ter em mente que liberar ou cancelar até um ponto de salvamento libera, automaticamente, todos os ponto de salvamento definidos após o mesmo.

Tudo isto acontece dentro do bloco de transação e, portanto, nada disso é visto pelas outras sessões do banco de dados. Quando o bloco de transação é efetivado, as ações efetivadas se tornam visíveis como uma unidade para as outras sessões, enquanto as ações canceladas nunca se tornam visíveis.

Recordando o banco de dados da instituição financeira, suponha que devesse ser debitado $100.00 da conta da Alice e creditado na conta do Bob, mas que foi descoberto em seguida que era para ser creditado na conta do Wally. Isso poderia ser feito utilizando pontos de salvamento como mostrado abaixo:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
WHERE nome = 'Alice';
SAVEPOINT meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
WHERE nome = 'Bob';
– uai ... o certo é na conta do Wally
ROLLBACK TO meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
WHERE nome = 'Wally';
COMMIT;
```

Obviamente este exemplo está simplificado ao extremo, mas é possível efetuar um grau elevado de controle sobre a transação através do uso de pontos de salvamento. Além disso, a instrução ROLLBACK TO é a única forma de obter novamente o controle sobre um bloco de transação colocado no estado interrompido devido a um erro, fora cancelar completamente e começar tudo de novo.

4.4.2 Heranças

Herança é um conceito de banco de dados orientado a objeto, que abre novas possibilidades interessantes ao projeto de banco de dados. Vamos criar duas tabelas: a tabela cidades e a tabela capitais. Como é natural, as capitais também são cidades e, portanto, deve existir alguma maneira para mostrar implicitamente as capitais quando todas as cidades são mostradas. Se formos bastante perspicazes, poderemos criar um esquema como este:

```
CREATE TABLE capitais (
nome text,
populacao real,
altitude int,
```

```
estado char(2)
);

CREATE TABLE interior (
nome text,
populacao real,
altitude int
);

CREATE VIEW cidades AS
SELECT nome, populacao, altitude FROM capitais
UNION
SELECT nome, populacao, altitude FROM interior;
```

Não se preocupe aqui com a instrução UNION, apenas saiba que a visão acima seleciona todas as cidades, seja da tabela capitais seja da interior.

Este esquema funciona bem para as consultas, mas não é bom quando é necessário atualizar várias linhas, entre outras coisas.
Esta é uma solução melhor:

```
CREATE TABLE cidades (
nome text,
populacao real,
altitude int – (em pés)
);

CREATE TABLE capitais (
estado char(2)
) INHERITS (cidades);
```

Neste caso, as linhas da tabela capitais herdam todas as colunas (nome, populacao e altitude) da sua tabela ancestral cidades. O tipo da coluna nome é text, um tipo nativo do PostgreSQL para cadeias de caracteres de tamanho variável. As capitais dos estados possuem uma coluna a mais chamada estado, que armazena a sigla do estado. No PostgreSQL uma tabela pode herdar de nenhuma, uma, ou de várias tabelas.

Por exemplo, a consulta abaixo retorna os nomes de todas as cidades, incluindo as capitais dos estados, localizadas a uma altitude superior a 500 metros:

```
SELECT nome, altitude
FROM cidades
WHERE altitude > 500;
```

Por outro lado, a consulta abaixo traz todas as cidades que não são capitais de estado e estão situadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
```

FROM ONLY cidades
WHERE altitude > 500;

Nesta consulta a palavra chave ONLY antes de cidades indica que a consulta deve ser efetuada apenas na tabela cidades, sem incluir as tabelas abaixo de cidades na hierarquia de herança. Muitos comandos mostrados até agora ? SELECT, UPDATE e DELETE ? permitem usar a notação ONLY.

4.5 Administração do Servidor

Veja como autenticar usuários, adicionar e remover estes usuários, conceder privilégios. E ao final, como criar e recuperar cópias de segurança de seus bancos.

4.5.1 Autenticação de clientes

4.5.2 Autenticação de clientes

Quando um aplicativo cliente se conecta ao servidor de banco de dados especifica o nome de usuário do PostgreSQL a ser usado na conexão, de forma semelhante à feita pelo usuário para acessar o sistema operacional Unix. Dentro do ambiente SQL, o nome de usuário do banco de dados determina os privilégios de acesso aos objetos do banco de dados . Portanto, é essencial controlar como os usuários de banco de dados podem se conectar.

A autenticação é o processo pelo qual o servidor de banco de dados estabelece a identidade do cliente e, por extensão, determina se o aplicativo cliente (ou o usuário executando o aplicativo cliente) tem permissão para se conectar com o nome de usuário que foi informado.

O PostgreSQL possui vários métodos diferentes para autenticação de clientes. O método utilizado para autenticar uma determinada conexão cliente pode ser selecionado tomando por base o endereço de hospedeiro (do cliente), o banco de dados ou o usuário.

Os nomes de usuário do PostgreSQL são logicamente distintos dos nomes de usuário do sistema operacional onde o servidor executa. Se todos os usuários de um determinado servidor de banco de dados também possuem conta no sistema operacional do servidor, é razoável atribuir nomes de usuário do banco de dados correspondendo aos nomes de usuário do sistema operacional.

Entretanto, um servidor que aceita conexões remotas pode possuir muitos usuários de banco de dados que não possuem conta no sistema operacional local e, nestes casos, a associação entre os nomes de usuário do banco de dados e os nomes de usuário do sistema operacional não é necessária.

O arquivo pg_hba.conf:
A autenticação do cliente é controlada pelo arquivo que por tradição se chama pg_hba.conf e é armazenado no diretório de dados do agrupamento de bancos de dados. HBA significa autenticação baseada no hospedeiro (host-based authentication). É instalado um arquivo pg_hba.conf padrão quando o diretório de dados é inicializado pelo utilitário initdb.

O formato geral do arquivo pg_hba.conf é um conjunto de registros, sendo um por linha. As linhas em branco são ignoradas, da mesma forma que qualquer texto após o caractere de comentário #. Um registro é formado por vários campos separados por espaços ou tabulações. Os campos podem conter espaços em branco se o valor do campo estiver entre aspas. Os registros não podem ocupar mais de uma linha.

Cada registro especifica um tipo de conexão, uma faixa de endereços de IP de cliente (se for relevante para o tipo de conexão), um nome de banco de dados, um nome de usuário e o método de autenticação a ser utilizado nas conexões que correspondem a estes parâmetros. O primeiro registro com o tipo de conexão, endereço do cliente, banco de dados solicitado e nome de usuário que corresponder é utilizado para realizar a autenticação. Não existe fall-through (procura exaustiva) ou backup: se um registro for escolhido e a autenticação não for bem-sucedida, os próximos registros não serão levados em consideração. Se não houver correspondência com nenhum registro, então o acesso é negado.

O registro pode ter um dos sete formatos a seguir:

- local banco_de_dados usuário método_de_autenticação [opção_de_autenticação]
- host banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autenticação]
- hostssl banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autenticaçã
- hostnossl banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autentica
- host banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação [opção_de_autenticação]
- hostssl banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação [opção_de_autenticação]
- hostnossl banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação [opção_de_autenticação]

O significado de cada campo está descrito abaixo:

- local

Este registro corresponde às tentativas de conexão feitas utilizando soquete do domínio Unix. Sem um registro deste tipo não são permitidas conexões através de soquete do domínio Unix.

- host
  Este registro corresponde às tentativas de conexão feitas utilizando o protocolo TCP/IP. Os registros host correspondem tanto às conexões SSL (Secure Socket Layer), quanto às não SSL.

  Nota: Não serão possíveis conexões TCP/IP remotas, a menos que o servidor seja inicializado com o valor apropriado para o parâmetro de configuração listen_addresses , uma vez que o comportamento padrão é aceitar conexões TCP/IP apenas no endereço retornante (loopback) localhost.

- hostssl
  Este registro corresponde às tentativas de conexão feitas utilizando o protocolo TCP/IP, mas somente quando a conexão é feita com a criptografia SSL. Para esta opção poder ser utilizada o servidor deve ter sido construído com o suporte a SSL habilitado. Além disso, o SSL deve ser habilitado na inicialização do servidor através do parâmetro de configuração ssl.

- hostnossl
  Este registro é o oposto lógico de hostssl: só corresponde a tentativas de conexão feitas através do protocolo TCP/IP que não utilizam SSL.

- banco_de_dados
  Especifica quais bancos de dados este registro corresponde. O valor all especifica que corresponde a todos os bancos de dados. O valor sameuser especifica que o registro corresponde ao banco de dados com o mesmo nome do usuário fazendo o pedido de conexão. O valor samegroup especifica que o usuário deve ser membro do grupo com o mesmo nome do banco de dados do pedido de conexão. Senão, é o nome de um banco de dados específico do PostgreSQL. Podem ser fornecidos vários nomes de banco de dados separados por vírgula. Pode ser especificado um arquivo contendo nomes de banco de dados, precedendo o nome do arquivo por @.

- usuário

  Especifica quais usuários do PostgreSQL este registro corresponde. O valor all especifica que corresponde a todos os usuários. Senão, é o nome de um usuário específico do PostgreSQL. Podem ser fornecidos vários nomes de usuário separados por vírgula. Podem ser especificados nomes de grupo precedendo o nome do grupo por +. Pode ser especificado um arquivo contendo nomes de usuário precedendo o nome do arquivo por @.

- endereço_de_CIDR
  Especifica a faixa de endereços de IP da máquina cliente que este registro corresponde. Contém um endereço de IP na notação padrão decimal com pontos, e o comprimento da máscara de CIDR (Os endereços de IP somente podem ser especificados numericamente, e não como domínios ou nomes de hospedeiro). O comprimento da máscara indica o número de bits de mais alta ordem que o endereço de IP do cliente deve corresponder. Os bits à direita devem ser zero em um determinado endereço de IP. Não pode haver espaços em branco entre os endereços de IP, a / e o comprimento da máscara de CIDR. Um endereço_de_CIDR típico seria 172.20.143.89/32 para um único hospedeiro, ou 172.20.143.0/24 para uma rede. Para especificar um único hospedeiro deve ser utilizada uma máscara de CIDR igual a 32 para o IPv4, ou igual a 128 para o IPv6. Um endereço especificado no formato IPv4 corresponde às conexões IPv6 que possuem o endereço correspondente como, por exemplo, 127.0.0.1 corresponde ao endereço de IPv6 ::ffff:127.0.0.1. Uma entrada especificada no formato IPv6 corresponde apenas às conexões IPv6, mesmo que represente um endereço na faixa IPv4-em-IPv6. Deve ser observado que as entradas no formato IPv6 serão rejeitadas se a biblioteca C do sistema não possuir suporte a endereços IPv6. Este campo se aplica apenas aos registros host, hostssl e hostnossl.

- endereço_de_IP e máscara_de_IP
  Estes campos podem ser utilizados como uma alternativa à notação endereço_de_CIDR.

Em vez de especificar o comprimento da máscara, a máscara é especificada como uma coluna em separado. Por exemplo, 255.0.0.0 representa uma máscara de CIDR para endereços de IPv4 com comprimento igual a 8, e 255.255.255.255 representa uma máscara de CIDR com comprimento igual a 32. Estes campos se aplicam apenas aos registros host, hostssl e hostnossl.

- método_de_autenticação
  Especifica o método de autenticação a ser utilizado para se conectar através deste registro. Abaixo está mostrado um resumo das escolhas possíveis: trust : A conexão é permitida incondicionalmente. Este método permite a qualquer um que possa se conectar ao servidor de banco de dados PostgreSQL se autenticar como o usuário do PostgreSQL que for desejado, sem necessidade de senha.
  reject : A conexão é rejeitada incondicionalmente. É útil para "eliminar por filtragem"certos hospedeiros de um grupo.
  md5 : Requer que o cliente forneça uma senha criptografada pelo método md5 para autenticação.
  crypt : Requer que o cliente forneça uma senha criptografada através de crypt() para autenticação. Deve-se dar preferência ao método md5 para os clientes com versão 7.2 ou posterior, mas os clientes com versão anterior a 7.2 somente suportam crypt.
  password : Requer que o cliente forneça uma senha não criptografada para autenticação. Uma vez que a senha é enviada em texto puro pela rede, não deve ser utilizado em redes não confiáveis.
  krb4 : É utilizado Kerberos V4 para autenticar o usuário. Somente disponível para conexões TCP/IP.
  krb5 : É utilizado Kerberos V5 para autenticar o usuário. Somente disponível para conexões TCP/IP.
  ident : Obtém o nome de usuário do sistema operacional do cliente (para conexões TCP/IP fazendo contato com o servidor de identificação no cliente, para conexões locais obtendo a partir do sistema operacional) e verifica se o usuário possui permissão para se conectar como o usuário de banco de dados solicitado consultando o mapa especificado após a palavra chave ident.
  pam : Autenticação utilizando o serviço Pluggable Authentication Modules (PAM) fornecido pelo sistema operacional.

- opção_de_autenticação
  O significado deste campo opcional depende do método de autenticação escolhido.

Os arquivos incluídos pela construção @ são lidos como nomes de listas, que podem ser separadas por espaços em branco ou vírgulas. Os comentários são iniciados por #, como no arquivo pg_hba.conf, sendo permitidas construções @ aninhadas. A menos que o nome do arquivo que segue a @ seja um caminho absoluto, é considerado como sendo relativo ao diretório que contém o arquivo que faz referência.

Uma vez que os registros de pg_hba.conf são examinados seqüencialmente a cada tentativa de conexão, a ordem dos registros possui significado. Normalmente, os primeiros registros possuem parâmetros de correspondência de conexão mais exigentes e métodos de autenticação menos exigentes, enquanto os últimos registros possuem parâmetros de correspondência menos exigentes e métodos de autenticação mais exigentes. Por exemplo, pode-se desejar utilizar a autenticação trust para conexões TCP/IP locais, mas requerer o uso de senha para conexões

TCP/IP remotas. Neste caso, o registro especificando a autenticação trust para conexões a partir de 127.0.0.1 deve aparecer antes do registro especificando autenticação por senha para uma faixa mais ampla de endereços de IP de cliente permitidos.

O arquivo pg_hba.conf é lido durante a inicialização e quando o processo servidor principal (postmaster) recebe um sinal SIGHUP. Se o arquivo for editado enquanto o sistema estiver ativo, será necessário enviar um sinal para o postmaster (utilizando pg_ctl reload ou kill -HUP) para fazer com que o arquivo seja lido novamente.

4.5.3 Administrando usuários

Conceitualmente, os usuários de banco de dados são completamente distintos dos usuários de sistema operacional. Na prática, pode ser conveniente manter correspondência, mas não é requerido. Os nomes dos usuários de banco de dados são globais para todo o agrupamento de bancos de dados (e não próprio de cada banco de dados). Para criar um usuário deve ser utilizado o comando SQL CREATE USER :

CREATE USER nome_do_usuário;

Onde nome_do_usuário segue as regras dos identificadores SQL: ou não contém caracteres especiais, ou está entre aspas. Para remover um usuário existente deve ser utilizado o comando DROP USER :
DROP USER nome_do_usuário;

Para facilitar, são fornecidos os programas createuser e dropuser que incorporam estes comandos SQL, e que podem ser executados a partir do interpretador de comandos:
$ createuser nome_do_usuário

$ dropuser nome_do_usuário

Para conhecer o conjunto de usuários existentes deve ser consultado o catálogo do sistema pg_user como, por exemplo:

SELECT usename FROM pg_user;

Também pode ser utilizado o meta-comando \du do programa psql para listar os usuários existentes.

Para ser possível ativar o sistema de banco de dados, um sistema recém criado sempre contém um usuário pré-definido. É atribuído o valor 1 para o identificador deste usuário e, por padrão (a menos que seja alterado ao executar o utilitário initdb), possui o mesmo nome do usuário de sistema operacional que inicializou o agrupamento de bancos de dados. Geralmente este usuário se chama postgres. Para poder criar mais usuários, primeiro é necessário se conectar como este usuário inicial.

Em uma conexão com o servidor de banco de dados, está ativa a identidade de exatamente um usuário. O nome de usuário a ser utilizado em uma determinada conexão com o banco de dados é indicado pelo cliente ao fazer o pedido de conexão, de uma forma específica do aplica-

tivo. Por exemplo, o programa psql utiliza a opção -U na linha de comando para indicar o usuário a ser utilizado na conexão. Muitos aplicativos assumem, por padrão, o nome do usuário corrente do sistema operacional (inclusive o createuser e o psql). Portanto, é conveniente manter uma correspondência de nomes entre estes dois conjuntos de usuários.

4.5.4 Privilégios

Quando um objeto do banco de dados é criado, é atribuído um dono ao mesmo. O dono é o usuário que executou o comando de criação. Para mudar o dono de uma tabela, índice, seqüência ou visão deve ser utilizado o comando ALTER TABLE . Por padrão, somente o dono (ou um superusuário) pode fazer qualquer coisa com o objeto. Para permitir o uso por outros usuários, devem ser concedidos privilégios.

Existem vários privilégios distintos: SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, TRIGGER, CREATE, TEMPORARY, EXECUTE, USAGE e ALL PRIVILEGES. Para obter mais informações sobre os diferentes tipos de privilégio suportados pelo PostgreSQL deve ser consultada a página de referência do comando GRANT . O direito de modificar ou remover um objeto é sempre um privilégio apenas de seu dono. É utilizado o comando GRANT para conceder privilégios. Portanto, se joel for um usuário existente, e tbl_contas for uma tabela existente, o privilégio de atualizar a tabela pode ser concedido pelo comando:

GRANT UPDATE ON tbl_contas TO joel;

Este comando deve ser executado pelo usuário dono da tabela. Para conceder privilégios para um grupo deve ser utilizado o comando:

GRANT SELECT ON tbl_contas TO GROUP grp_financas;

O nome especial de "usuário"PUBLIC pode ser utilizado para conceder o privilégio para todos os usuários do sistema. Escrevendo ALL no lugar do privilégio especifica a concessão de todos os privilégios.
Para revogar um privilégio deve ser utilizado o comando REVOKE :

REVOKE ALL ON tbl_contas FROM PUBLIC;

Os privilégios especiais do dono da tabela (ou seja, o direito de DROP (remover), GRANT (conceder), REVOKE (revogar), etc.) são sempre implícitos ao fato de ser o dono, não podendo ser concedidos ou revogados. Mas o dono da tabela pode decidir revogar seus próprios privilégios comuns como, por exemplo, tornando uma tabela somente para leitura para o próprio e para os outros.