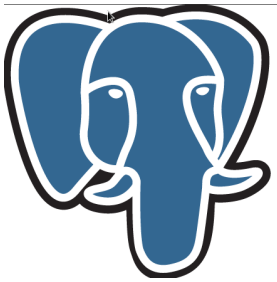

PostgreSQL

the world's most advanced open source database

Curso de PostgreSQL

Agnaldo Neto Marinho

PostgreSQL

Sumário

Capítulo 1

Introdução ao curso

Seja bem-vindo ao **Curso de PostgreSQL**. Este curso esta sendo fomentado pela *Centro de Tecnologia da Informação e Comunicação - CTIC* da *Universidade Federal do Pará* e ministrado por *Agnaldo Neto Marinho*. Realiza(ou)-se de *03/09/2015 a 09/09/2015*.

Os procedimentos descritos neste material foram validados sob a distribuição Debian GNU/Linux Jessie, todavia a base teórica ministrada é o conhecimento fundamental para a aplicabilidade dos procedimentos técnicos sob qualquer sistema operacional.

Apesar das peculiaridades de cada sistema operacional o conteúdo será abordado de forma isenta, para que o participante tenha a possibilidade de utilizar o conhecimento adquirido no ambiente que lhe for mais adequado.

Neste capítulo, serão abordados os seguintes temas: licenciamento deste material, origem do *software* utilizado (executável ou do código fonte) e configurações essenciais da distribuição Debian GNU/Linux.

1.1 Licença do material

1.2 Compilação de *software* X *software* da distribuição/sistema operacional

O acesso ao código fonte do *software* e sua compilação, é uma das liberdades propiciadas pelo *software livre*. Entretanto, o *software* também pode ser obtido em forma executável (compilada), e de forma integrada ao sistema operacional (empacotado), já estando pronto para utilização. Cada uma destas opções possui vantagens e desvantagens que serão enumeradas a seguir:

Características do *software* obtido na forma de executável (previamente compilado):

- **V:** Instalação rápida que requer menos espaço em disco; evita a compilação do *software*, assim como, a instalação de *software* de compilação (make, gcc, etc) e cabeçalhos de bibliotecas (libc6-dev, etc)
- **V:** Instalação automatizada de *software* e de bibliotecas necessárias (dependências) para o funcionamento do *software* principal.
- **V:** Versão testada pelo distribuidor do *software* (em geral o distribuidor do sistema operacional), e possivelmente livre de erros.
- **V:** Possibilita atualizações e correções de falhas de segurança de forma automática, e fornecida pelo distribuidor do sistema operacional.
- **V:** Facilita suporte externo devido ao método de instalação padronizado e utilização de versões invariantes do *software*.
- **V/D:** Pode não ser a versão mais nova do *software*, e não possuir funcionalidades mais recentes. Todavia, a utilização de versões maduras, tende a fornecer maior estabilidade.

Características do *software* obtido a partir do código fonte:

- **D:** Instalação mais complexa e demorada, demanda instalação manual de bibliotecas externas.
- **D:** Atualizações e correções são manuais, exigindo atenção diária às atualizações necessárias para correções de falhas de segurança.
- **D:** Dificulta suporte externo pois não é um método de instalação padronizado.
- **V/D:** Permite utilizar a última versão do *software*, com os novos recursos, mas trata-se de código menos testado podendo possuir falhas não detectadas.
- **V:** Pode permitir um ganho de performace com a compilação com otimizações do processador, e também com o desligamento de recursos não utilizados do *software*.

Após o levantamento destas características, é notável que em ambientes corporativos a utilização de *software* fornecido por um distribuidor é essencial para continuidade da disponibilidade dos sistemas.

Diminui-se o esforço empregado para manter o parque tecnológico atualizado e livre de falhas. Dessa forma, o treinamento utilizará os pacotes fornecidos pelo distribuidor do sistema operacional escolhido.

1.3 Debian GNU/Linux

Os sistemas operacionais baseados em tecnologias livres tendem a fornecer *software* que realizam instalações automatizadas. O Debian fornece os utilitários **apt-get** e **aptitude** para esta funcionalidade, sendo o segundo, sucessor e atualmente de uso recomendado.

O Debian fornece repositórios web que contém os *software* disponíveis para instalação, e estes são distribuídos em forma de pacotes: arquivos compactados com rotinas de pré/pós instalação e remoção, e informações sobre dependências, recomendações e sugestões de software adicionais.

O comportamento padrão do utilitário *aptitude* ao instalar um *software* é realizar a instalação das **dependências**, e também daqueles especificados como **recomendações**. Entretanto, este comportamento induz a instalação de *software* não requeridos, e demanda a utilização de espaço em disco adicional.

A instalação automática de *software* recomendado pode ser desabilitada através da adição da configuração abaixo ao arquivo **/etc/apt/apt.conf**:

- Debian Jessie (aptitude 0.6.11):

```
Apt::Install-Recommends "false";
```

O utilitário *aptitude* também requer a configuração da fonte dos *software* a serem instalados, e isto é realizado no arquivo **/etc/apt/sources.list**, conforme indicado no quadro abaixo.

```
deb http://ftp.br.debian.org/debian jessie main contrib non-free
deb http://security.debian.org/ jessie/updates main
```

Caso a conectividade seja fornecida por um proxy via http, a seguinte configuração deve ser adicionada ao arquivo **/etc/apt/apt.conf**, com a devida adequação ao endereço IP do proxy:

```
Acquire::http::Proxy "http://172.16.0.1:3128/";
```

Após a definição das fontes, é necessário o *download* da lista de *software* disponíveis, que é formada por informações como versão e descrição de cada *software*. Esse *download* deve ser realizado através do comando:

```
  # aptitude update
```

A lista de *software* disponíveis pode ser consultada, como indicado no exemplo abaixo:

- Pesquisar pelo nome do *software*:

```
  # aptitude search postgresql
```

- Pesquisar nas descrições do *software*, equivalente ao *apt-cache search openldap*:

```
  # aptitude search ~d'postgresql'
```

Maiores informações sobre um determinado *software* podem ser obtidas como indicado a seguir:

```
  # aptitude show postgresql
```

Capítulo 2

Entendendo um Banco de Dados

2.1 Bancos de Dados Relacionais

Um banco de dados é uma aplicação que lhe permite armazenar e obter de volta dados com eficiência. O que o torna *relacional* é a maniera como os dados são armazenados e organizados no banco de dados.

Quando falamos em banco de dados, aqui, nos referimos a um banco de dados relacional - RDBMS *Relational Database Management System.*

Em um banco de dados relacional, todos os dados são guardados em tabelas. Estas têm uma estrutura que se repete a cada linha, como você pode observar um uma planilha. São os relacionamento entre as tabelas que as tornam "relacionais"

- O modelo relacional surgiu devido às seguintes necessidades: - Aumentar a independência de dados nos sistemas gerenciadores de bancos de dados;

  - Prover um conjunto de funções apoiadas em álgebra relacional para armazenamento e recuperação de dados;

- A estrutura fundamental do modelo relacional é a relação. Uma relação é constituída por um ou mais atributos (campos), que traduzem o tipo de dados armazenados. Cada instância do esquema (linha), designa-se por tupla (registro). - O modelo implementa estruturas de dados organizados em relações (tabelas).

2.2 Banco de Dados Objeto-Relacional

- O PostgreSQL é normalmente considerado um sistema gerenciador de banco de dados relacional (SGBD-R, ou RDBMS, em inglês.) Entretanto, o PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (SGBD-OR).

- Por ser objeto-relacional, o PostgreSQL suporta recursos inexistentes a um banco de dados puramente ralacional, tais como: herança entre tabelas, arrays em colunas e sobrecarga de funções.

:

Capítulo 3

Introdução ao Postgresql

O PostgreSQL é um SGBD (Sistema Gerenciador de Banco de Dados) objeto relacional de código aberto, com mais de 15 anos de desenvolvimento. é extremamente robusto e confiável, além de ser extremante flexivel e rico em recursos. Ele é considerado objeto relacional por implementar, além das características de um SGBD relacional, algumas características de orientação a objetos, como herança e tipos personalizados.

3.1 O que é o PostgreSQL?

- O PostgreSQL é um dos bancos de dados abertos mais utilizados atualmente, possui recursos avançados e compete igualmente com muitos bancos de dados comerciais.
- O banco de dados PostgreSQL nasceu na Universidade de Berkeley, em 1986, como um projeto acadêmico e se encontra hoje na versão 9.1, sendo um projeto mantido pela comunidade de *Software Livre*.
- A coordernação de desenvolvimento do PostgreSQL é executado pelo *PostgreSQL Global Development Group* que conta com um grande número de desenvolvimento ao redor do mundo.
- Ele é um SGBD muito adequado para o estudo universitário do modelo relacional, além de ser uma ótima opção para empresas implatarem soluções de alta confiabilidade sem altos custos de licenciamento.
- É um programa distribuido sob a licença BSD, o que torna o seu código fonte disponível e o seu uso livre para aplicações comercias ou não.
- O PostgreSQL foi implementado em diversos ambientes de produção no mundo, entre eles, um bom exemplo do seu pontencial é o banco de dados que armazena os registro de domínio .org, mantido pela empresa Afilias.

3.2 Principais Funcionalidades

- Banco de dados objeto-relacional
  - Herança entre as tabelas
  - Sobrecarga de funções
  - Colunas do tipo array
- Suporte a transações (padrão ACID)

- Lock por registro (row level locking)
- Integridade referencial
- Sub-consultas.
- Controle de concorrência multi-versão (MVCC);
- Funções armazenadas (Stored Procedures), que podem ser escritas em várias linguagens de programação (PL/PgSQL, Perl, Python, Ruby, e outras);
- Gatilhos (Triggers);
- Tipos definidos pelo usuário;
- Esquemas (Schemas);
- Conexões SSL.
- Áreas de armazenamento (Tablespaces)
- Pontos de salvamento (Savepoints)
- Commit em duas fases
- Arquivamento e restauração do banco a partir de logs de transação
- Diversas ferramentas de replicação
- Extensões para dados geoespaciais, indexação de textos, xml e várias outras.
- Acesso via drivers ODBC e JDBC, além do suporte nativo em várias linguagens
- Suporte ao armazenamento de BLOBs (binary large objects)
- Sub-queries e queries na cláusula FROM
- Sofisticado mecanismo de tuning
- Suporte a conexão de banco de dados seguras (criptografia)
- Modelo de segurança para o acesso aos objetos do banco de dados por roles
- Triggers views e functions (PL/pgSQL, Perl, Python e Tcl
- Mecanismos próprio de logs

3.3 Plataforma Suportadas

- IBM AIX
- FreeBSD< OpenBSD, NetBSD
- HP-UX
- Irix
- Linux

- MarcOS X
- Microsoft Windows (suporte nativo desde a versão 8.0)
- SCO Open Server
- Sun Solaris
- Tru64 Unix
- Unix Ware

Capítulo 4

Intefaces de Acesso ao PostgreSQL

No jargão de banco de dados, o PostgreSQL utiliza o modelo cliente-servidor. Uma sessão do PostgreSQL consiste nos seguintes processos (programas) cooperando entre si:

4.1 Interfaces de Acesso ao Banco de Dados

- O PostgreSQL pode ser acessado a partir de várias linguagens, entre elas estão:
  - C, C++
  - Java (JDBC)
  - PHP, JSP, ColdFusion
  - TCL/Tk
  - Perl
  - Python
  - ODBC (ASP, Delphi ou qualquer linguagem que suporte ODBC)

4.2 Conexão JDBC

- Abaixo um exeomplo de conexção utilizando drive JDBC:

```
1        public Connection connect() {
2            driver = "org.postgresql.Driver";
3            url = "jdbc:postgresql://172.16.128.13:5432/teste?user=
4                postgres";
5            try{
6               Class.forName(drive).newInstance();
7               con = DriverManager.getConnection(url);
8            }
9            catch (Exception e){
10               System.out.println("Error");
11               e.printStackTrace();
12           }
13           return con;
14       }
```

- Para o URL de conexão temos as opções:
  - **User**: usuário para conexão.
  - **Password**: senha para a conexão.
  - **Database**: nome do banco de dados a se acessado.
  - **Port**: porta de conexão.
  - **IP**: endereço IP do servidor.

4.3 Configuração ao PostgreSQL

Depois de baixar e instalar é hora de configurar. O usuário root do nosso banco de dados é o **postgres**. No processo de instalação foi criado um usuário chamdo postgres também no sistema. Então, nos logaremos com este usuário.

```
# Su postgres
```

Por padrão, **o usuário de banco de dados 'postgres'** não tem senha, então agora nos logaremos no shell do PostgreSQL para alterar a senha do usuário postgres.

Primeiro, logando no shell...

```
$ psql
```

Agora, já no shell do PostgreSQL, vamos alterar a senha do usuário postgres

```
postgres=# ALTER USER postgres WITH PASSWORD 'qualquersenha';
```

Esse cara que a gente acabou de configurar ai é o root do banco de dados... A gente não vai ficar usando esse usuário nas nossas aplicações, né? Então! vamos criar um novo usuário.

```
postgres=# CREATE USER usuario NOCREATEDB NOSUPERUSER NOCREATEROLE PASSWORD
        'senha';
```

Agora, vamos criar uma tabela também

```
postgres=# CREATE DATABASE minhabase;
```

4.4 Introdução ao psql

- O psql é o modo interativo do PostgreSQL para acesso e manipulação dos bancod de dados.

  ```
  psql [-h hostname -p port -U user -W] [database]
  ```

- Onde:
  - **hostname**: nome ou IP do servidor (padrão é localhost)
  - **port**: porta de conexão (padrão ŕ 5432)
  - **user**: usuário postgresql (padrão é o usuário de sistema operacional)
  - **database**: nome do banco de dados.
- A opção -w força a entrada da senha do usuário.

o

Capítulo 5

Liguagem SQL

5.1 A Linguagem SQL

- SQL (Structured Query Languagem) é uma linguagem declarativa de acesso à banco de dados.
- Por ser uma linguagem padronizada, a migração para o PostgreSQL é fácilitada para aqueles que conhecem SQL.
- O PostgreSQL está em conformidade com a maior parte das espeficições SQL92 e SQL99.
- A linguagem SQL não considera a caixa dos comandos (*xase insensitive*). Entretanto, a caixa faz diferença para os leitores entre *aspas*.

5.2 Introdução

Este capítulo fornece uma visão geral sobre como utilizar a linguagem SQL para realizar operações simples. O propósito deste tutorial é apenas fazer uma introdução e, de forma alguma, ser um tutorial completo sobre a linguagem SQL. É preciso estar ciente que algumas funcionalidades da linguagem SQL do PostgreSQL são extensões ao padrão.

Conforme criando o usuario e banco de dados, conforme descrito no capítulo anterior, e que o psql esteja ativo.

5.3 Criação de Tabelas

Pode-se criar uma tabela especificando o seu nome juntamente com os nomes das colunas e seus tipos de dado:

```
CREATE TABLE clima (
    ciade           char(80),
    tem_min         int,            -- temperatura mínima
    temp_max        int,            -- temperatura máxima
    prcp            real,           -- precipitação
    data            date
);
```

Este comando pode ser digitado no psql com quebras de linha. O psql reconhece que o comando só termina quando é encontrado o ponto-e-vírgula.

Espaços em branco (ou seja, espaços, tabulações e novas linhas) podem ser utilizados livremente nos comandos SQL. Isto significa que o comando pode ser digitado com um alinhamento diferente do

mostrado acima, ou mesmo tudo em uma única linha. Dois hífens (--") iniciam um comentário; tudo que vem depois é ignorado até o final da linha. A linguagem SQL não diferencia letras maiúsculas e minúsculas nas palavras chave e nos identificadores, a não ser que os identificadores sejam delimitados por aspas (") para preservar letras maiúsculas e minúsculas, o que não foi feito acima.

No comando, varchar(80) especifica um tipo de dado que pode armazenar cadeias de caracteres arbitrárias com comprimento até 80 caracteres; int é o tipo inteiro normal; real é o tipo para armazenar números de ponto flutuante de precisão simples; date é o tipo para armazenar data e hora (a coluna do tipo date pode se chamar date, o que tanto pode ser conveniente quanto pode causar confusão).

O PostgreSQL suporta os tipos de dado SQL padrão int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp e interval, assim como outros tipos de utilidade geral, e um conjunto diversificado de tipos geométricos. O PostgreSQL pode ser personalizado com um número arbitrário de tipos de dado definidos pelo usuário. Como conseqüência, sintaticamente os nomes dos tipos não são palavras chave, exceto onde for requerido para suportar casos especiais do padrão SQL.

No segundo exemplo são armazenadas cidades e suas localizações geográficas associadas:

```
CREATE TABLE cidade(
    nome              varchar(80),
    localizacao       pooint
);
```

O tipo point é um exemplo de tipo de dado específico do PostgreSQL.

Para terminar deve ser mencionado que, quando a tabela não é mais necessária, ou se deseja recriá-la de uma forma diferente, é possível removê-la por meio do comando:

```
DROP TABLE nome_da_tabela;
```

5.4 Inserção de linhas em tablea

É utilizado o comando INSERT para inserir linhas nas tabelas:

```
INSERT INTO clima VALUES ('São Francisco', 46, 50, 0.25, '1994-11-27');
```

Repare que todos os tipos de dado possuem formato de entrada de dados bastante óbvios. As constantes, que não são valores numéricos simples, geralmente devem estar entre apóstrofos ('), como no exemplo acima. O tipo date é, na verdade, muito flexível em relação aos dados que aceita, mas para este tutorial vamos nos fixar no formato sem ambigüidade mostrado acima.

O tipo point requer um par de coordenadas como entrada, como mostrado abaixo:

```
INSERT INTO cidades VALUES ('São Francisco', '(-194.0, 53.0)');
```

A sintaxe usada até agora requer que seja lembrada a ordem das colunas. Uma sintaxe alternativa permite declarar as colunas explicitamente:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, data)
    VALUES ('São Francisco', 43, 57, 0.0, '2015-09-29');
```

Se for desejado, pode-se declarar as colunas em uma ordem diferente, ou mesmo, omitir algumas colunas. Por exemplo, se a precipitação não for conhecida:

```
INSERT INTO clima (data, cidade, temp_max, temp_min)
    VALUES ('2015-11-29', 'Mosqueiro', 54, 37);
```

Muitos desenvolvedores consideram que declarar explicitamente as colunas é um estilo melhor que confiar na ordem implícita.

Também pode ser utilizado o comando COPY para carregar uma grande quantidade de dados a partir de arquivos texto puro. Geralmente é mais rápido, porque o comando COPY é otimizado para esta finalidade, embora possua menos flexibilidade que o comando INSERT. Como exemplo poderíamos ter

```
COPY clima FROM '/home/user/clima.txt';
```

onde o arquivo contendo os dados deve estar disponível para a máquina servidora, e não para a estação cliente, porque o servidor lê o arquivo diretamente. Podem ser obtidas mais informações sobre o comando COPY em COPY.

5.5 Consultar tabelas

Para trazer os dados de uma tabela, a tabela deve ser consultada. Para esta finalidade é utilizado o comando SELECT do SQL. Este comando é dividido em lista de seleção (a parte que especifica as colunas a serem trazidas), lista de tabelas (a parte que especifica as tabelas de onde os dados vão ser trazidos), e uma qualificação opcional (a parte onde são especificadas as restrições). Por exemplo, para trazer todas as linhas da tabela clima digite:

```
SELECT * FROM clima;
```

Aqui o * é uma forma abreviada de "todas as colunas". [1] Seriam obtidos os mesmos resultados usando:

```
SELECT cidade, temp_min, temp_max, prcp, data FROM clima;
```

A saída deve ser:

```
cidade         | temp_min | temp_max | prcp |    data
_______________|__________|__________|______|___________
São Francisco  |      46  |       50 | 0.25 | 1994-11-27
São Francisco  |      43  |       57 |   0  | 1994-11-29
Hayward        |      37  |       54 |      | 1994-11-29
(3 linhas)
```

Na lista de seleção podem ser especificadas expressões, e não apenas referências a colunas. Por exemplo, pode ser escrito

```
SELECT cidade, (temp_max+temp_min)/2 AS temp_media, data FROM clima;
```

devendo produzir:

```
        cidade   | temp_media |   data
-----------------|------------|-----------
São Francisco    |         48 | 1994-11-27
São Francisco    |         50 | 1994-11-29
Hayward          |         45 | 1994-11-29
(3 linhas)
```

Perceba que a cláusula AS foi utilizada para mudar o nome da coluna de saída (a cláusula AS é opcional).

A consulta pode ser "qualificada", adicionando a cláusula WHERE para especificar as linhas desejadas. A cláusula WHERE contém expressões booleanas (valor verdade), e somente são retornadas as linhas para as quais o resultado da expressão booleana for verdade. São permitidos os operadores booleanos usuais (AND, OR e NOT) na qualificação. Por exemplo, o comando abaixo mostra o clima de São Francisco nos dias de chuva:

```
SELECT * FROM clima
    WHERE cidade = 'São Francisco' AND prcp > 0.0;
```

Resultado:

```
   cidade       | temp_min | temp_max | prcp |    data
----------------|----------|----------|------|------------
Heyward         |      37  |      54  |      | 1994-11-29
São Francisco   |      43  |      57  |    0 | 1994-11-29
São Francisco   |      46  |      50  | 0.25 | 1994-11-27
```

Neste exemplo a ordem de classificação não está totalmente especificada e, portanto, as linhas de São Francisco podem retornar em qualquer ordem. Mas sempre seriam obtidos os resultados mostrados acima se fosse executado:

```
SELECT * FROM clima
    ORDER BY cidade, temp_min;
```

Pode ser solicitado que as linhas duplicadas sejam removidas do resultado da consulta

```
SELECT DISTINCT cidade
    FROM clima;


    cidade
---------------
 Heyward
 São Francisco
 (2 linhas)
```

Novamente, neste exemplo a ordem das linhas pode variar. Pode-se garantir resultados consistentes utilizando DISTINCT e ORDER BY juntos:

```
SELECT DISTINCT cidade
    FROM clima
        ORDER BY cidade;
```

Notas

- Embora o SELECT * seja útil para consultas improvisadas, geralmente é considerado um estilo ruim para código em produção, uma vez que a adição de uma coluna à tabela mudaria os resultados.
- Em alguns sistemas de banco de dados, incluindo as verdadesões antigas do PostgreSQL, a implementação do DISTINCT ordena automaticamente as linhas e, por isso, o ORDER BY não é necessário. Mas isto não é requerido pelo padrão SQL, e o PostgreSQL corrente não garante que DISTINCT faça com que as linhas sejam ordenadas.

5.6 Juncões entre tabelas

A consulta que acessa várias linhas da mesma tabela, ou de tabelas diferentes, de uma vez, é chamada de consulta de junção. Como exemplo, suponha que se queira listar todas as linhas de clima junto com a localização da cidade associada. Para fazer isto, é necessário comparar a coluna cidade de cada linha da tabela clima com a coluna nome de todas as linhas da tabela cidades, e selecionar os pares de linha onde estes valores são correspondentes.

Nota:Este é apenas um modelo conceitual, a junção geralmente é realizada de uma maneira mais eficiente que realmente comparar cada par de linhas possível, mas isto não é visível para o usuário.

Esta operação pode ser efetuada por meio da seguinte consulta:

```
SELECT *
    FROM clima, cidades
    WHERE cidade = nome;



    cidade   |temp_min|temp_max|prcp|   data   |     nome    | Localização
-------------|--------|--------|----|----------|-------------|------------
São Francisco|      46|      50|0.25|1994-11-27|São Francisco|(-194, 53)
São Francisco|      43|      57|  0 |1994-11-29|São Francisco|(-194, 53)
(2 linhas)
```

Duas coisas devem ser observadas no conjunto de resultados produzido:

- Não existe nenhuma linha de resultado para a cidade Hayward. Isto acontece porque não existe entrada correspondente na tabela cidades para Hayward, e a junção ignora as linhas da tabela clima sem correspondência. Veremos em breve como isto pode ser corrigido. Existem duas colunas contendo o nome da cidade, o que está correto porque a lista de colunas das tabelas clima 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, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
```

Como todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence. Se existissem nomes de colunas duplicados nas duas tabelas, seria necessário qualificar os nomes das colunas para mostrar qual delas está sendo referenciada, como em:

```
SELECT clima.cidade, clima.temp_min, clima.temp_max,
       clima.prcp, clima.data, cidades.localizacao
   FROM clima, cidades
   WHERE cidades.nome = clima.cidade;
```

Muitos consideram um bom estilo qualificar todos os nomes de colunas nas consultas de junção, para que a consulta não falhe ao se adicionar posteriormente um nome de coluna duplicado a uma das tabelas.

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

```
SELECT *
    FROM clima INNER JOIN cidades ON (clima.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.

Agora vamos descobrir como se faz para obter as linhas de Hayward. Desejamos o seguinte: que a consulta varra a tabela clima 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 junções vistas até agora foram junções internas (inner join). O comando então fica assim:

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


cidade        |temp_min |temp_max | prcp |    data    |    nome      | localizacao
______________|_________|_________|______|____________|______________|____________
Hayward       |      37 |      54 |      | 1994-11-29 |              |
São Francisco |      46 |      50 | 0.25 | 1994-11-27 |São Francisco | (-194,53)
São Francisco |      43 |      57 |    0 | 1994-11-29 |São Francisco | (-194,53)
(3 linhas)
```

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 (nulos) nas colunas da tabela à direita.

Também é possível fazer a junção da tabela consigo mesma. Isto é chamado de autojunção (self join). Como exemplo, suponha que desejamos descobrir todas as linhas de clima que estão no intervalo de temperatura de outros registros de clima. Para isso é necessário comparar as colunas temp_min e temp_max de cada linha da tabela clima com as colunas temp_min e temp_max de todos as outras linhas da mesma tabela clima, o que pode ser feito utilizando a seguinte consulta:

```
SELECT C1.cidade, C1.temp_min AS menor, C1.temp_max AS maior,
    C2.cidade, C2.temp_min AS menor, C2.temp_max AS maior
    FROM clima C1, clima C2
    WHERE C1.temp_min < C2.temp_min
```

```
    AND C1.temp_max > C2.temp_max;



 cidade          | menor | maior |     cidade    | menor | maior
   --------------+-------+-------+---------------+-------+-------
 São Francisco   |    43 |    57 | São Francisco |    46 |    50
 Hayward         |    37 |    54 | São Francisco |    46 |    50
 (2 linhas)
```

A tabela clima teve seu nome mudado para C1 e C2, para ser possível distinguir o lado esquerdo e o lado direito da junção. Estes tipos de "aliases"também podem ser utilizados em outras consultas para reduzir a digitação como, por exemplo:

```
SELECT *
    FROM clima w, cidades c
    WHERE w.cidade = c.nome;
```

Será encontrada esta forma de abreviar com bastante freqüência.

5.7 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 observada em qualquer lugar usando

```
SELECT max(temp_min) FROM clima;

 max
 -----
   46
 (1 linha)
```

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

```
SELECT cidade FROM clima WHERE temp_min = max(temp_min);     ERRADO
```

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 quais linhas serão incluídas no cálculo da agregação e, neste caso, teria que 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 clima
    WHERE temp_min = (SELECT max(temp_min) FROM clima);

    cidade
```

```
---------------
 São Francisco
 (1 linha)
```

Isto está correto porque a subconsulta é uma ação independente, que calcula sua agregação isoladamente 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(temp_min)
    FROM clima
    GROUP BY cidade;

    cidade      | max
----------------+-----
Hayward         |  37
São Francisco   |  46
(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(temp_min)
    FROM clima
    GROUP BY cidade
    HAVING max(temp_min) < 40;


  cidade   | max
-----------+-----
 Hayward   |  37
(1 linha)
```

que mostra os mesmos resultados, mas apenas para as cidades que possuem os valores de max(temp_min) abaixo de 40. Para concluir, se desejarmos somente as cidades com nome começando pela letra "S"podemos escrever:

```
SELECT cidade, max(temp_min)
    FROM clima
    WHERE cidade LIKE 'S%'(1)
    GROUP BY cidade
    HAVING max(temp_min) < 40;
```

5.8 Atualizações

As linhas existentes podem ser atualizadas utilizando o comando UPDATE. Suponha ter sido descoberto que as leituras de temperatura realizadas após 28 de novembro de 1994 estão todas 2 graus mais altas. Os dados podem ser atualizados da seguinte maneira:

```
UPDATE clima
    SET temp_max = temp_max - 2,  temp_min = temp_min - 2
    WHERE data > '1994-11-28';
```

Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;

     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
 São Francisco   |       41 |       55 |    0 | 1994-11-29
 Hayward         |       35 |       52 |      | 1994-11-29
(3 linhas)
```

5.9 Exclusões

As linhas podem ser excluídas da tabela através do comando DELETE. Suponha que não estamos mais interessados no clima de Hayward. Então podemos executar comando a seguir para excluir estas linhas da tabela

```
DELETE FROM clima WHERE cidade = 'Hayward';
```

e todos os registros de clima pertencentes a Hayward serão removidos. Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;

   cidade     | temp_min | temp_max | prcp |    data
--------------+----------+----------+------+------------
São Francisco |       46 |       50 | 0.25 | 1994-11-27
São Francisco |       41 |       55 |    0 | 1994-11-29
(2 linhas)
```

Deve-se tomar cuidado com comandos na forma

```
DELETE FROM nome_da_tabela;
```

porque, sem uma qualificação, o comando DELETE remove todas as linhas da tabela, deixando-a vazia. O sistema não solicita confirmação antes de realizar esta operação!

Capítulo 6

Funcionalidades avançadas

6.1 Introdução

Neste capítulo serão mostradas algumas funcionalidades mais avançadas da linguagem SQL que simplificam a gerência, e evitam que os dados sejam perdidos ou danificados. No final serão vistas algumas extensões do PostgreSQL.

6.2 Visões

Supondo que a consulta combinando os registros de clima com a localização das cidades seja de particular interesse para um projeto, 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, temp_min, temp_max, prcp, data, localizacao
    FROM clima, 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 detalhes das estruturas das tabelas, que podem mudar à medida que os aplicativos evoluem, atrás de interfaces consistentes.

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

6.3 Chaves estrangeiras

Considere o seguinte problema: Desejamos ter certeza que não serão inseridas linhas na tabela clima sem que haja um registro correspondente na tabela cidades. Isto é chamado de manter a integridade referencial dos dados. Em sistemas de banco de dados muito simples poderia ser implementado (caso fosse) olhando primeiro a tabela cidades para verificar se existe a linha correspondente e, depois, inserir ou rejeitar a nova linha de clima. Esta abordagem possui vários problemas, e é muito inconveniente, por isso o PostgreSQL pode realizar esta operação por você.

A nova declaração das tabelas ficaria assim:

```
CREATE TABLE cidades (
       cidade      varchar(80) PRIMARY KEY,
       localizacao point
);

CREATE TABLE clima (
        cidade     varchar(80) REFERENCES cidades(cidade),
        temp_min   int,
        temp_max   int,
        prcp       real,
        data       date
);
```

Agora, ao se tentar inserir uma linha inválida:

```
INSERT INTO clima VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "clima" violates foreign key constraint "clima_cidade
DETAIL:  Key (cidade)=(Berkeley) is not present in table "cidades".

-- Tradução da mensagem

ERRO:  inserção ou atualização na tabela "clima" viola a restrição de chave estrangeira
DETALHE:  Chave (cidade)=(Berkeley) não está presente na tabela "cidades".
```

O comportamento das chaves estrangeiras pode receber ajuste fino no aplicativo. Não iremos além deste exemplo simples neste tutorial. Com certeza o uso correto de chaves estrangeiras melhora a qualidade dos aplicativos de banco de dados, portanto incentivamos muito que se aprenda a usá-las.

6.4 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 ao extremo, 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');
```

Existirem várias atualizações distintas envolvidas para realizar esta operação tão simples. A contabilidade do banco quer ter certeza que todas estas atualizações foram realizadas, ou que nenhuma delas foi realizada. Com certeza não é interessante que uma falha no sistema faça com que Bob receba $100.00 que não foi debitado da Alice. É necessário garantir que, caso aconteça algo errado no meio da operação, nenhum dos passos executados até este ponto irá valer. 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 por inteiro, ou nada acontece.

utra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando ao mesmo tempo, nenhuma delas deve enxergar as alterações incompletas efetuadas pelas outras.

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 usado 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 emitido o comando BEGIN, então cada comando possuirá um BEGIN implícito e, se der tudo certo, um COMMIT, envolvendo-o. Um grupo de comandos envolvidos por um BEGIN e um COMMIT é algumas vezes chamado de bloco de transação.

possível controlar os comandos na transação de uma forma mais granular utilizando os pontos de salvamento (savepoints). Os pontos de salvamento permitem descartar 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 pela transação entre o estabelecimento do ponto de salvamento e o cancelamento são descartadas, 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 poderá 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 pontos 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 desfeitas nunca se tornam visíveis.

Recordando o banco de dados da instituição financeira, suponha que tivesse sido debitado $100.00 da conta da Alice e creditado na conta do Bob, e descoberto em seguida que era para ser creditado na conta do Wally. Isso poderia ser feito utilizando um ponto de salvamento, conforme mostrado abaixo:

```
EGIN;
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 pelo sistema devido a um erro, fora cancelar completamente e começar tudo de novo.

6.5 Herança

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,    -- (em pés)
   estado     char(2)
);

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

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

```
CCREATE 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, de 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 pés:

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


  nome    | altitude
----------+----------
Las Vegas |     2174
Mariposa  |     1953
Madison   |      845
(3 linhas)
```

Por outro lado, a consulta abaixo retorna 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;


  nome    | altitude
----------+----------
Las Vegas |     2174
Mariposa  |     1953
(2 linhas)
```

a consulta acima 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.

Nota: Embora a hierarquia seja útil com freqüência, sua utilidade é limitada porque não está integrada com as restrições de unicidade e de chave estrangeira.

6.6 Conclusão

O PostgreSQL possui muitas funcionalidades não abordadas neste tutorial introdutório, o qual está orientado para os usuários com pouca experiência na linguagem SQL. Estas funcionalidades são mostradas com mais detalhes no restante desta documentação.

Se sentir necessidade de mais material introdutório, por favor visite o sítio do PostgreSQL na Web e o sítio PostgreSQLBrasil para obter indicações sobre onde encontrar este material.

Apêndice A

Licença